

MPI: A Message-Passing Interface Standard

Version **3.0**

(Draft, with MPI 3 Nonblocking Collectives

and new Fortran 2008 Interface)

Unofficial, for comment only

Message Passing Interface Forum

December 16, 2011

ticket0.

ticket229-A.

16.2.1	Overview	552	
16.2.2	Fortran Support Through the <code>mpi_f08</code> Module	553	ticket230-B.
16.2.3	Fortran Support Through the <code>mpi</code> Module	556	ticket230-B.
16.2.4	Fortran Support Through the <code>mpif.h</code> Include File	558	ticket230-B.
16.2.5	Interface Specifications, Linker Names and the Profiling Interface	560	ticket230-B.
16.2.6	MPI for Different Fortran Standard Versions	564	ticket230-B.
16.2.7	Requirements on Fortran Compilers	568	ticket247-S.
16.2.8	Additional Support for Fortran Register-Memory-Synchronization	569	ticket247-S.
16.2.9	Additional Support for Fortran Numeric Intrinsic Types	570	ticket230-B.
	Parameterized Datatypes with Specified Precision and Exponent Range	571	ticket238-J.
	Support for Size-specific MPI Datatypes	575	
	Communication With Size-specific Types	577	
16.2.10	Problems With Fortran Bindings for MPI	579	
16.2.11	Problems Due to Strong Typing	580	
16.2.12	Problems Due to Data Copying and Sequence Association with Subscript Triplets	581	ticket236-H.
16.2.13	Problems Due to Data Copying and Sequence Association with Vector Subscripts	584	ticket236-H.
16.2.14	Special Constants	584	
16.2.15	Fortran Derived Types	584	
16.2.16	Optimization Problems, an Overview	586	
16.2.17	Problems with Code Movement and Register Optimization	587	ticket238-J.
	Nonblocking operations	587	
	One-sided communication	588	
	MPI_BOTTOM and combining independent variables in datatypes	588	
	Solutions	589	ticket238-J.
	The Fortran ASYNCHRONOUS attribute	591	ticket238-J.
	Calling MPI_F_SYNC_REG	592	ticket238-J.
	A user defined routine instead of MPI_F_SYNC_REG	593	
	Module variables and COMMON blocks	594	
	The (poorly performing) Fortran VOLATILE attribute	594	
	The Fortran TARGET attribute	594	ticket238-J.
16.2.18	Temporary Data Movement and Temporary Memory Modification	594	ticket238-J.
16.2.19	Permanent Data Movement	597	ticket238-J.
16.2.20	Comparison with C	597	
16.3	Language Interoperability	600	
16.3.1	Introduction	600	
16.3.2	Assumptions	600	
16.3.3	Initialization	600	
16.3.4	Transfer of Handles	601	
16.3.5	Status	604	
16.3.6	MPI Opaque Objects	607	
	Datatypes	607	
	Callback Functions	609	
	Error Handlers	609	
	Reduce Operations	609	
	Addresses	609	
16.3.7	Attributes	610	

A.4.2	Datatypes Fortran Bindings	701
A.4.3	Collective Communication Fortran Bindings	704
A.4.4	Groups, Contexts, Communicators, and Caching Fortran Bindings	707
A.4.5	Process Topologies Fortran Bindings	711
A.4.6	MPI Environmental Management Fortran Bindings	713
A.4.7	The Info Object Fortran Bindings	715
A.4.8	Process Creation and Management Fortran Bindings	715
A.4.9	One-Sided Communications Fortran Bindings	716
A.4.10	External Interfaces Fortran Bindings	717
A.4.11	I/O Fortran Bindings	718
A.4.12	Language Bindings Fortran Bindings	722
A.4.13	Profiling Interface Fortran Bindings	723
A.4.14	Deprecated Fortran Bindings	723
A.5	C++ Bindings (deprecated)	725
A.5.1	Point-to-Point Communication C++ Bindings	725
A.5.2	Datatypes C++ Bindings	728
A.5.3	Collective Communication C++ Bindings	730
A.5.4	Groups, Contexts, Communicators, and Caching C++ Bindings	733
A.5.5	Process Topologies C++ Bindings	736
A.5.6	MPI Environmental Management C++ Bindings	737
A.5.7	The Info Object C++ Bindings	738
A.5.8	Process Creation and Management C++ Bindings	739
A.5.9	One-Sided Communications C++ Bindings	740
A.5.10	External Interfaces C++ Bindings	741
A.5.11	I/O C++ Bindings	741
A.5.12	Language Bindings C++ Bindings	745
A.5.13	Profiling Interface C++ Bindings	746
A.5.14	C++ Bindings on all MPI Classes	746
A.5.15	Construction / Destruction	746
A.5.16	Copy / Assignment	746
A.5.17	Comparison	746
A.5.18	Inter-language Operability	747
B	Change-Log	749
B.1	Changes from Version 2.2 to Version 3.0	749
B.2	Changes from Version 2.1 to Version 2.2	751
B.3	Changes from Version 2.0 to Version 2.1	754
B.4	Other New Fortran Interfaces for New MPI-3 Routines	758
B.4.1	Matched Probe & Matched Receive, ticket #38 and #274	758
B.4.2	Nonblocking Collective Communication, ticket #109	759
B.4.3	Neighborhood Collective Communication, ticket #258	764
B.4.4	MPI_Count, ticket #265	767
B.4.5	MPI_T_..., ticket #266	768
B.4.6	RMA, ticket #270	768
B.4.7	Fault Tolerance, ticket #276	774
B.4.8	Hybrid Programming, ticket #284	774
	Bibliography	775

Acknowledgments

This document is the product of a number of distinct efforts in three distinct phases: one for each of MPI-1, MPI-2, and MPI-3. This section describes these in historical order, starting with MPI-1. Some efforts, particularly parts of MPI-2, had distinct groups of individuals associated with them, and these efforts are detailed separately.

This document represents the work of many people who have served on the MPI Forum. The meetings have been attended by dozens of people from many parts of the world. It is the hard and dedicated work of this group that has led to the MPI standard.

The technical development was carried out by subgroups, whose work was reviewed by the full committee. During the period of development of the Message-Passing Interface (MPI), many people helped with this effort.

Those who served as primary coordinators in MPI-1.0 and MPI-1.1 are:

- Jack Dongarra, David Walker, Conveners and Meeting Chairs
- Ewing Lusk, Bob Knighten, Minutes
- Marc Snir, William Gropp, Ewing Lusk, Point-to-Point Communication
- Al Geist, Marc Snir, Steve Otto, Collective Communication
- Steve Otto, Editor
- Rolf Hempel, Process Topologies
- Ewing Lusk, Language Binding
- William Gropp, Environmental Management
- James Cownie, Profiling
- Tony Skjellum, Lyndon Clarke, Marc Snir, Richard Littlefield, Mark Sears, Groups, Contexts, and Communicators
- Steven Huss-Lederman, Initial Implementation Subset

The following list includes some of the active participants in the MPI-1.0 and MPI-1.1 process not mentioned above.

Oak Ridge National Laboratory
Ohio State University
Pacific Northwest National Laboratory
QLogic Corporation
RunTime Computing Solutions, LLC
Sandia National Laboratories
SiCortex, Inc.
Silicon Graphics Inc.
Sun Microsystems, Inc.
Tokyo Institute of Technology
University of Alabama at Birmingham
University of Houston
University of Illinois at Urbana-Champaign
University of Stuttgart, High Performance Computing Center Stuttgart (HLRS)
University of Tennessee, Knoxville
University of Tokyo
University of Wisconsin

Funding for the MPI Forum meetings was partially supported by award #CCF-0816909 from the National Science Foundation. In addition, the HDF Group provided travel support for one U.S. academic.

MPI-3:

MPI-3 is a significant effort to extend and modernize the MPI Standard.

The editors and organizers of the MPI-3 have been: *Taken from MPI-2.2 with minor corrections. Need to separate the working groups list (which is currently reviewers) from the primary authors . Also, did I miss active steering committee members?*

- William Gropp, Steering committee, Frontmatter, Introduction, Groups, Contexts, and Communicators, One-Sided Communications, and Bibliography
- Richard Graham, Steering committee, Point-to-Point Communication
- Adam Moody, Collective Communication
- Torsten Hoefer, Collective Communication and Process Topologies
- George Bosilca, Datatypes and Environmental Management
- David Solt, Process Creation and Management
- Bronis R. de Supinski, External Interfaces, and Profiling
- Rajeev Thakur, I/O and One-Sided Communications
- Darius Buntinas, Info Object
- Jeffrey M. Squyres, Language Bindings and MPI 3.0 Secretary
- Rolf Rabenseifner, Steering committee, Terms and Definitions, Fortran Bindings, Deprecated Functions, Annex Change-Log, and Annex Language Bindings

- Craig Rasmussen, Fortran Bindings

The following list includes some of the active participants who attended MPI-3 Forum meetings and in the e-mail discussions of the errata items and are not mentioned above.

THIS IS THE MPI-2.2 LIST: PLEASE UPDATE

Pavan Balaji	Purushotham V. Bangalore	Brian Barrett
Richard Barrett	Christian Bell	Robert Blackmore
Gil Bloch	Ron Brightwell	Greg Bronevetsky
Jeff Brown	Darius Buntinas	Jonathan Carter
Nathan DeBardeleben	Terry Dontje	Gabor Dozsa
Edric Ellis	Karl Feind	Edgar Gabriel
Patrick Geoffray	Johann George	David Gingold
David Goodell	Erez Haba	Robert Harrison
Thomas Herault	Marc-André Hermanns	Steve Hodson
Joshua Hursey	Yutaka Ishikawa	Bin Jia
Hideyuki Jitsumoto	Terry Jones	Yann Kalemkarian
Ranier Keller	Matthew Koop	Quincey Koziol
Manojkumar Krishnan	Sameer Kumar	Miron Livny
Andrew Lumsdaine	Miao Luo	Ewing Lusk
Timothy I. Mattox	Kathryn Mohror	
	Kannan Narasimhan	Mark Pagel
Avneesh Pant	Steve Poole	Howard Pritchard
[ticket229.2.] Hubert Ritzdorf	Rob Ross	
Martin Schulz	Pavel Shamis	Galen Shipman
Christian Siebert	Anthony Skjellum	Brian Smith
Naoki Sueyasu	Vinod Tipparaju	Keith Underwood
Rolf Vandevaart	Abhinav Vishnu	Weikuan Yu

The MPI Forum also acknowledges and appreciates the valuable input from people via e-mail and in person.

THIS IS TAKEN FROM MPI-2.2. PLEASE CORRECT. The following institutions supported the MPI-3 effort through time and travel support for the people listed above.

Argonne National Laboratory
 Auburn University
 Bull
 Cisco Systems, Inc.
 Cray Inc.
 CSCS
 Forschungszentrum Jülich
 Fujitsu
 German Research School for Simulation Sciences
 The HDF Group
 Hewlett-Packard
 International Business Machines
 Indiana University
 Institut National de Recherche en Informatique et Automatique (INRIA)

Chapter 1

Introduction to MPI

1.1 Overview and Goals

MPI (Message-Passing Interface) is a *message-passing library interface specification*. All parts of this definition are significant. MPI addresses primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process. Extensions to the “classical” message-passing model are provided in collective operations, remote-memory access operations, dynamic process creation, and parallel I/O. MPI is a *specification*, not an implementation; there are multiple implementations of MPI. This specification is for a *library interface*; MPI is not a language, and all MPI operations are expressed as functions, subroutines, or methods, according to the appropriate language bindings, which for C, C++, and Fortran, are part of the MPI standard. The standard has been defined through an open process by a community of parallel computing vendors, computer scientists, and application developers. The next few sections provide an overview of the history of MPI’s development.

The main advantages of establishing a message-passing standard are portability and ease of use. In a distributed memory communication environment in which the higher level routines and/or abstractions are built upon lower level message-passing routines the benefits of standardization are particularly apparent. Furthermore, the definition of a message-passing standard, such as that proposed here, provides vendors with a clearly defined base set of routines that they can implement efficiently, or in some cases for which they can provide hardware support, thereby enhancing scalability.

The goal of the Message-Passing Interface simply stated is to develop a widely used standard for writing message-passing programs. As such the interface should establish a practical, portable, efficient, and flexible standard for message passing.

A complete list of goals follows.

- Design an application programming interface (not necessarily for compilers or a system implementation library).
- Allow efficient communication: Avoid memory-to-memory copying, allow overlap of computation and communication, and offload to communication co-processor, where available.
- Allow for implementations that can be used in a heterogeneous environment.

- Allow convenient C, C++, and Fortran bindings for the interface.
- Assume a reliable communication interface: the user need not cope with communication failures. Such failures are dealt with by the underlying communication subsystem.
- Define an interface that can be implemented on many vendor's platforms, with no significant changes in the underlying communication and system software.
- Semantics of the interface should be language independent.
- The interface should be designed to allow for thread safety.

1.2 Background of MPI-1.0

MPI sought to make use of the most attractive features of a number of existing message-passing systems, rather than selecting one of them and adopting it as the standard. Thus, MPI was strongly influenced by work at the IBM T. J. Watson Research Center [1, 2], Intel's NX/2 [45], Express [12], nCUBE's Vertex [41], p4 [7, 8], and PARMACS [5, 9]. Other important contributions have come from Zipcode [48, 49], Chimp [16, 17], PVM [4, 14], Chameleon [25], and PICL [24].

The MPI standardization effort involved about 60 people from 40 organizations mainly from the United States and Europe. Most of the major vendors of concurrent computers were involved in MPI, along with researchers from universities, government laboratories, and industry. The standardization process began with the Workshop on Standards for Message-Passing in a Distributed Memory Environment, sponsored by the Center for Research on Parallel Computing, held April 29-30, 1992, in Williamsburg, Virginia [56]. At this workshop the basic features essential to a standard message-passing interface were discussed, and a working group established to continue the standardization process.

A preliminary draft proposal, known as MPI1, was put forward by Dongarra, Hempel, Hey, and Walker in November 1992, and a revised version was completed in February 1993 [15]. MPI1 embodied the main features that were identified at the Williamsburg workshop as being necessary in a message passing standard. Since MPI1 was primarily intended to promote discussion and "get the ball rolling," it focused mainly on point-to-point communications. MPI1 brought to the forefront a number of important standardization issues, but did not include any collective communication routines and was not thread-safe.

In November 1992, a meeting of the MPI working group was held in Minneapolis, at which it was decided to place the standardization process on a more formal footing, and to generally adopt the procedures and organization of the High Performance Fortran Forum. Subcommittees were formed for the major component areas of the standard, and an email discussion service established for each. In addition, the goal of producing a draft MPI standard by the Fall of 1993 was set. To achieve this goal the MPI working group met every 6 weeks for two days throughout the first 9 months of 1993, and presented the draft MPI standard at the Supercomputing 93 conference in November 1993. These meetings and the email discussion together constituted the MPI Forum, membership of which has been open to all members of the high performance computing community.

MPI-2.0 was released, and a second ballot was voted on May 22, 2002. Both votes were done electronically. Both ballots were combined into one document: “Errata for MPI-2”, May 15, 2002. This errata process was then interrupted, but the Forum and its e-mail reflectors kept working on new requests for clarification.

Restarting regular work of the MPI Forum was initiated in three meetings, at EuroPVM/MPI’06 in Bonn, at EuroPVM/MPI’07 in Paris, and at SC’07 in Reno. In December 2007, a steering committee started the organization of new MPI Forum meetings at regular 8-weeks intervals. At the January 14-16, 2008 meeting in Chicago, the MPI Forum decided to combine the existing and future MPI documents to one document for each version of the MPI standard. For technical and historical reasons, this series was started with MPI-1.3. Additional Ballots 3 and 4 solved old questions from the errata list started in 1995 up to new questions from the last years. After all documents (MPI-1.1, MPI-2, Errata for MPI-1.1 (Oct. 12, 1998), and MPI-2.1 Ballots 1-4) were combined into one draft document, for each chapter, a chapter author and review team were defined. They cleaned up the document to achieve a consistent MPI-2.1 document. The final MPI-2.1 standard document was finished in June 2008, and finally released with a second vote in September 2008 in the meeting at Dublin, just before EuroPVM/MPI’08. The major work of the current MPI Forum is the preparation of MPI-3.

1.5 Background of MPI-2.2

MPI-2.2 is a minor update to the MPI-2.1 standard. This version addresses additional errors and ambiguities that were not corrected in the MPI-2.1 standard as well as a small number of extensions to MPI-2.1 that met the following criteria:

- Any correct MPI-2.1 program is a correct MPI-2.2 program.
- Any extension must have significant benefit for users.
- Any extension must not require significant implementation effort. To that end, all such changes are accompanied by an open source implementation.

The discussions of MPI-2.2 proceeded concurrently with the MPI-3 discussions; in some cases, extensions were proposed for MPI-2.2 but were later moved to MPI-3.

1.6 Background of MPI-3.0

MPI-3.0 is a major update to the MPI standard. Areas of particular interest are the extension of collective operations to include nonblocking and sparse-group routines and more flexible and powerful one-sided operations. This *draft* contains the MPI Forum’s current draft of nonblocking collective routines.

A new Fortran `mpi_f08` module is introduced to provide extended compile-time argument checking and buffer handling in nonblocking routines. This new Fortran support method provides protection against the optimization problems with asynchronous accesses to the buffers of nonblocking calls. The existing `mpi` module is enhanced to provide basic compile-time argument checking for MPI calls. The use of `mpif.h` is strongly discouraged.

and is marked as such, although, semantically, it is not used in one call both for input and for output on a single process.

Another frequent situation arises when an argument value is needed only by a subset of the processes. When an argument is not significant at a process then an arbitrary value can be passed as an argument.

Unless specified otherwise, an argument of type OUT or type INOUT cannot be aliased with any other argument passed to an MPI procedure. An example of argument aliasing in C appears below. If we define a C procedure like this,

```
void copyIntBuffer( int *pin, int *pout, int len )
{
    int i;
    for (i=0; i<len; ++i) *pout++ = *pin++;
}
```

then a call to it in the following code fragment has aliased arguments.

```
int a[10];
copyIntBuffer( a, a+3, 7);
```

Although the C language allows this, such usage of MPI procedures is forbidden unless otherwise specified. Note that Fortran prohibits aliasing of arguments.

All MPI functions are first specified in the language-independent notation. Immediately below this, **language dependent bindings follow**:

- The ISO C version of the function.
- The Fortran version used with `USE mpi_f08`.
- The Fortran version of the same function used with `USE mpi` or `INCLUDE 'mpif.h'`
- The C++ binding (which is deprecated).

“Fortran” in this document refers to Fortran 90 and higher; see Section 2.6.

2.4 Semantic Terms

When discussing MPI procedures the following semantic terms are used.

nonblocking A procedure is nonblocking if the procedure may return before the operation completes, and before the user is allowed to reuse resources (such as buffers) specified in the call. A nonblocking request is **started** by the call that initiates it, e.g., `MPI_ISEND`. The word complete is used with respect to operations, requests, and communications. An **operation completes** when the user is allowed to reuse resources, and any output buffers have been updated; i.e. a call to `MPI_TEST` will return `flag = true`. A **request is completed** by a call to wait, which returns, or a test or get status call which returns `flag = true`. This completing call has two effects: the status is extracted from the request; in the case of test and wait, if the request was nonpersistent, it is **freed**, and becomes **inactive** if it was persistent. A **communication completes** when all participating operations complete.

blocking A procedure is blocking if return from the procedure indicates the user is allowed to reuse resources specified in the call.

arguments to access these objects. In addition to their use by MPI calls for object access, handles can participate in assignments and comparisons.

In Fortran with `USE mpi` or `INCLUDE 'mpif.h'`, all handles have type `INTEGER`. In Fortran with `USE mpi_f08`, and in C and C++, a different handle type is defined for each category of objects. With Fortran `USE mpi_f08`, the handles are defined as Fortran `BIND(C)` derived types that consist of only one element `INTEGER :: MPI_VAL`. The internal handle value is identical to the Fortran `INTEGER` value used in the `mpi` module and `mpif.h`. The type names are identical to the names in C, except that they are not case sensitive. For example:

```
TYPE, BIND(C) :: MPI_Comm
  INTEGER      :: MPI_VAL
END TYPE MPI_Comm
```

In addition, handles themselves are distinct objects in C++. The C and C++ types must support the use of the assignment and equality operators.

Advice to implementors. In Fortran, the handle can be an index into a table of opaque objects in a system table; in C it can be such an index or a pointer to the object. C++ handles can simply “wrap up” a table index or pointer. (*End of advice to implementors.*)

Rationale. Since the Fortran integer values are equivalent, applications can easily convert MPI handles between all three supported Fortran methods. For example, an integer communicator handle `COMM` can be converted directly into an exactly equivalent `mpi_f08` communicator handle named `comm_f08` by `comm_f08%MPI_VAL=COMM`, and vice versa. The use of the `INTEGER` defined handles and the `BIND(C)` derived type handles is different: Fortran 2003 (and later) define that `BIND(C)` derived types can be used within user defined common blocks, but it is up to the rules of the companion C compiler how many numerical storage units are used for these `BIND(C)` derived type handles. Most compilers use one unit for both, the `INTEGER` handles and the handles defined as `BIND(C)` derived types. (*End of rationale.*)

Advice to users. If a user wants to substitute `mpif.h` or the `mpi` module by the `mpi_f08` module and the application program stores a handle in a Fortran common block then it is necessary to change the Fortran support method in all application routines that use this common block, because the number of numerical storage units of such a handle can be different in the two modules. (*End of advice to users.*)

Opaque objects are allocated and deallocated by calls that are specific to each object type. These are listed in the sections where the objects are described. The calls accept a handle argument of matching type. In an allocate call this is an OUT argument that returns a valid reference to the object. In a call to deallocate this is an INOUT argument which returns with an “invalid handle” value. MPI provides an “invalid handle” constant for each object type. Comparisons to this constant are used to test for validity of the handle.

A call to a deallocate routine invalidates the handle and marks the object for deallocation. The object is not accessible to the user after the call. However, MPI need not deallocate the object immediately. Any operation pending (at the time of the deallocate) that involves this object will complete normally; the object will be deallocated afterwards.

MPI_SUBARRAYS_SUPPORTED (Fortran only)

MPI_ASYNC_PROTECTS_NONBLOCKING (Fortran only)

and their C++ counterparts where appropriate.

The constants that cannot be used in initialization expressions or assignments in Fortran are:

MPI_BOTTOM

MPI_STATUS_IGNORE

MPI_STATUSES_IGNORE

MPI_ERRCODES_IGNORE

MPI_IN_PLACE

MPI_ARGV_NULL

MPI_ARGVS_NULL

MPI_UNWEIGHTED

Advice to implementors. In Fortran the implementation of these special constants may require the use of language constructs that are outside the Fortran standard. Using special values for the constants (e.g., by defining them through **PARAMETER** statements) is not possible because an implementation cannot distinguish these values from legal data. Typically, these constants are implemented as predefined static variables (e.g., a variable in an **MPI**-declared **COMMON** block), relying on the fact that the target compiler passes data by address. Inside the subroutine, this address can be extracted by some mechanism outside the Fortran standard (e.g., by Fortran extensions or by implementing the function in C). (*End of advice to implementors.*)

2.5.5 Choice

MPI functions sometimes use arguments with a *choice* (or union) data type. Distinct calls to the same routine may pass by reference actual arguments of different types. The mechanism for providing such arguments will differ from language to language. For Fortran **with the include file `mpif.h` or the `mpi` module**, the document uses `<type>` to represent a choice variable; **with the Fortran `mpi_f08` module**, such arguments are declared **with the Fortran 2008 + TR 29113 syntax `TYPE(*), DIMENSION(..)`**; for C and C++, we use `void *`.

Advice to implementors. Implementors can freely choose how to implement choice arguments in the `mpi` module, e.g., with a non-standard compiler-dependent method that has the quality of the call mechanism in the implicit Fortran interfaces, or with the method defined for the `mpi_f08` module. See details in Section 16.2.1 on page 552. (*End of advice to implementors.*)

2.5.6 Addresses

Some **MPI** procedures use *address* arguments that represent an absolute address in the calling program. The datatype of such an argument is **MPI_Aint** in C, **MPI::Aint** in C++ and **INTEGER (KIND=MPI_ADDRESS_KIND)** in Fortran. These types must have the same width and encode address values in the same manner such that address values in one language may be passed directly to another language without conversion. There is the **MPI** constant **MPI_BOTTOM** to indicate the start of the address range.

2.5.7 File Offsets

For I/O there is a need to give the size, displacement, and offset into a file. These quantities can easily be larger than 32 bits which can be the default size of a Fortran integer. To overcome this, these quantities are declared to be `INTEGER (KIND=MPI_OFFSET_KIND)` in Fortran. In C one uses `MPI_Offset` whereas in C++ one uses `MPI::Offset`. These types must have the same width and encode address values in the same manner such that offset values in one language may be passed directly to another language without conversion.

2.6 Language Binding

This section defines the rules for MPI language binding in general and for Fortran, ISO C, and C++, in particular. (Note that ANSI C has been replaced by ISO C.) The C++ language bindings have been deprecated. Defined here are various object representations, as well as the naming conventions used for expressing this standard. The actual calling sequences are defined elsewhere.

MPI bindings are for Fortran 90 or later, though they were originally designed to be usable in Fortran 77 environments. With the `mpi_f08` module, two new Fortran features, *assumed type* and *assumed rank*, are also required, see Section 2.5.5 on page 16.

Since the word `PARAMETER` is a keyword in the Fortran language, we use the word “argument” to denote the arguments to a subroutine. These are normally referred to as parameters in C and C++, however, we expect that C and C++ programmers will understand the word “argument” (which has no specific meaning in C/C++), thus allowing us to avoid unnecessary confusion for Fortran programmers.

Since Fortran is case insensitive, linkers may use either lower case or upper case when resolving Fortran names. Users of case sensitive languages should avoid the “`mpi_`” and “`pmpi_`” prefixes.

2.6.1 Deprecated Names and Functions

A number of chapters refer to deprecated or replaced MPI-1 constructs. These are constructs that continue to be part of the MPI standard, as documented in Chapter 15, but that users are recommended not to continue using, since better solutions were provided with MPI-2. For example, the Fortran binding for MPI-1 functions that have address arguments uses `INTEGER`. This is not consistent with the C binding, and causes problems on machines with 32 bit `INTEGER`s and 64 bit addresses. In MPI-2, these functions were given new names with new bindings for the address arguments. The use of the old functions is deprecated. For consistency, here and in a few other cases, new C functions are also provided, even though the new functions are equivalent to the old functions. The old names are deprecated. Another example is provided by the MPI-1 predefined datatypes `MPI_UB` and `MPI_LB`. They are deprecated, since their use is awkward and error-prone. The MPI-2 function `MPI_TYPE_CREATE_RESIZED` provides a more convenient mechanism to achieve the same effect.

Table 2.1 shows a list of all of the deprecated constructs. Note that the constants `MPI_LB` and `MPI_UB` are replaced by the function `MPI_TYPE_CREATE_RESIZED`; this is because their principal use was as input datatypes to `MPI_TYPE_STRUCT` to create resized datatypes. Also note that some C typedefs and Fortran subroutine names are included in this list; they are the types of callback functions.

Deprecated	MPI-2 Replacement
MPI_ADDRESS	MPI_GET_ADDRESS
MPI_TYPE_HINDEXED	MPI_TYPE_CREATE_HINDEXED
MPI_TYPE_HVECTOR	MPI_TYPE_CREATE_HVECTOR
MPI_TYPE_STRUCT	MPI_TYPE_CREATE_STRUCT
MPI_TYPE_EXTENT	MPI_TYPE_GET_EXTENT
MPI_TYPE_UB	MPI_TYPE_GET_EXTENT
MPI_TYPE_LB	MPI_TYPE_GET_EXTENT
MPI_LB	MPI_TYPE_CREATE_RESIZED
MPI_UB	MPI_TYPE_CREATE_RESIZED
MPI_ERRHANDLER_CREATE	MPI_COMM_CREATE_ERRHANDLER
MPI_ERRHANDLER_GET	MPI_COMM_GET_ERRHANDLER
MPI_ERRHANDLER_SET	MPI_COMM_SET_ERRHANDLER
MPI_Handler_function	MPI_Comm_errhandler_function
MPI_KEYVAL_CREATE	MPI_COMM_CREATE_KEYVAL
MPI_KEYVAL_FREE	MPI_COMM_FREE_KEYVAL
MPI_DUP_FN	MPI_COMM_DUP_FN
MPI_NULL_COPY_FN	MPI_COMM_NULL_COPY_FN
MPI_NULL_DELETE_FN	MPI_COMM_NULL_DELETE_FN
MPI_Copy_function	MPI_Comm_copy_attr_function
COPY_FUNCTION	COMM_COPY_ATTR_[ticket250-V.] FUNCTION
MPI_Delete_function	MPI_Comm_delete_attr_function
DELETE_FUNCTION	COMM_DELETE_ATTR_[ticket250-V.] FUNCTION
MPI_ATTR_DELETE	MPI_COMM_DELETE_ATTR
MPI_ATTR_GET	MPI_COMM_GET_ATTR
MPI_ATTR_PUT	MPI_COMM_SET_ATTR

Table 2.1: Deprecated constructs

2.6.2 Fortran Binding Issues

Originally, MPI-1.1 provided bindings for Fortran 77. These bindings are retained, but they are now interpreted in the context of the Fortran 90 standard. MPI can still be used with most Fortran 77 compilers, as noted below. When the term “Fortran” is used it means Fortran 90 or later; it means Fortran 2008 + TR 29113 and later if the `mpi_f08` module is used.

All MPI names have an `MPI_` prefix, and all characters are capitals. Programs must not declare variables, parameters, or functions with names beginning with the prefix `MPI_`. To avoid conflicting with the profiling interface, programs should also avoid functions with the prefix `PMPI_`. This is mandated to avoid possible name collisions.

All MPI Fortran subroutines have a return code in the last argument. With `USE mpi_f08`, this last argument is declared as `OPTIONAL`, except for user-defined callback functions (e.g., `COMM_COPY_ATTR_FUNCTION`) and their predefined callbacks (e.g., `MPI_NULL_COPY_FN`). A few MPI operations which are functions do not have the return code argument. The return code value for successful completion is `MPI_SUCCESS`. Other error codes are implementation dependent; see the error codes in Chapter 8 and Annex A.

Constants representing the maximum length of a string are one smaller in Fortran than

in C and C++ as discussed in Section 16.3.9.

Handles are represented in Fortran as `INTEGER`s, or as a `BIND(C)` derived type with the `mpi_f08` module; see Section 2.5.1 on page 12. Binary-valued variables are of type `LOGICAL`.

Array arguments are indexed from one.

The older MPI Fortran bindings (`mpif.h` and `use mpi`) are inconsistent with the Fortran standard in several respects. These inconsistencies, such as register optimization problems, have implications for user codes that are discussed in detail in Section 16.2.16.

2.6.3 C Binding Issues

We use the ISO C declaration format. All MPI names have an `MPI_` prefix, defined constants are in all capital letters, and defined types and functions have one capital letter after the prefix. Programs must not declare variables or functions with names beginning with the prefix `MPI_`. To support the profiling interface, programs should not declare functions with names beginning with the prefix `PMPI_`.

The definition of named constants, function prototypes, and type definitions must be supplied in an include file `mpi.h`.

Almost all C functions return an error code. The successful return code will be `MPI_SUCCESS`, but failure return codes are implementation dependent.

Type declarations are provided for handles to each category of opaque objects.

Array arguments are indexed from zero.

Logical flags are integers with value 0 meaning “false” and a non-zero value meaning “true.”

Choice arguments are pointers of type `void *`.

Address arguments are of MPI defined type `MPI_Aint`. File displacements are of type `MPI_Offset`. `MPI_Aint` is defined to be an integer of the size needed to hold any valid address on the target architecture. `MPI_Offset` is defined to be an integer of the size needed to hold any valid file size on the target architecture.

2.6.4 C++ Binding Issues

The C++ language bindings have been deprecated. There are places in the standard that give rules for C and not for C++. In these cases, the C rule should be applied to the C++ case, as appropriate. In particular, the values of constants given in the text are the ones for C and Fortran. A cross index of these with the C++ names is given in Annex A.

We use the ISO C++ declaration format. All MPI names are declared within the scope of a namespace called `MPI` and therefore are referenced with an `MPI::` prefix. Defined constants are in all capital letters, and class names, defined types, and functions have only their first letter capitalized. Programs must not declare variables or functions in the `MPI` namespace. This is mandated to avoid possible name collisions.

The definition of named constants, function prototypes, and type definitions must be supplied in an include file `mpi.h`.

Advice to implementors. The file `mpi.h` may contain both the C and C++ definitions. Usually one can simply use the defined value (generally `__cplusplus`, but not required) to see if one is using C++ to protect the C++ definitions. It is possible that a C compiler will require that the source protected this way be legal C code. In this case, all the C++ definitions can be placed in a different include file and the

operation to select a particular message. The last three parameters of the send operation, along with the rank of the sender, specify the envelope for the message sent. Process one (`myrank = 1`) receives this message with the **receive** operation `MPI_RECV`. The message to be received is selected according to the value of its envelope, and the message data is stored into the **receive buffer**. In the example above, the receive buffer consists of the storage containing the string `message` in the memory of process one. The first three parameters of the receive operation specify the location, size and type of the receive buffer. The next three parameters are used for selecting the incoming message. The last parameter is used to return information on the message just received.

The next sections describe the blocking send and receive operations. We discuss send, receive, blocking communication semantics, type matching requirements, type conversion in heterogeneous environments, and more general communication modes. Nonblocking communication is addressed next, followed by channel-like constructs and send-receive operations. Nonblocking communication is addressed next, followed by channel-like constructs and send-receive operations, ending with a description of the “dummy” process, `MPI_PROC_NULL`.

3.2 Blocking Send and Receive Operations

3.2.1 Blocking Send

The syntax of the blocking send operation is given below.

`MPI_SEND(buf, count, datatype, dest, tag, comm)`

IN	buf	initial address of send buffer (choice)
IN	count	number of elements in send buffer (non-negative integer)
IN	datatype	datatype of each send buffer element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)

```
int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
```

```
MPI_Send(buf, count, datatype, dest, tag, comm, ierror) BIND(C)
  TYPE(*), DIMENSION(..), INTENT(IN) :: buf
  INTEGER, INTENT(IN) :: count, dest, tag
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
  <type> BUF(*)
  INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```


A predefined communicator `MPI_COMM_WORLD` is provided by MPI. It allows communication with all processes that are accessible after MPI initialization and processes are identified by their rank in the group of `MPI_COMM_WORLD`.

Advice to users. Users that are comfortable with the notion of a flat name space for processes, and a single communication context, as offered by most existing communication libraries, need only use the predefined variable `MPI_COMM_WORLD` as the `comm` argument. This will allow communication with all the processes available at initialization time.

Users may define new communicators, as explained in Chapter 6. Communicators provide an important encapsulation mechanism for libraries and modules. They allow modules to have their own disjoint communication universe and their own process numbering scheme. (*End of advice to users.*)

Advice to implementors. The message envelope would normally be encoded by a fixed-length message header. However, the actual encoding is implementation dependent. Some of the information (e.g., source or destination) may be implicit, and need not be explicitly carried by messages. Also, processes may be identified by relative ranks, or absolute ids, etc. (*End of advice to implementors.*)

3.2.4 Blocking Receive

The syntax of the blocking receive operation is given below.

`MPI_RECV (buf, count, datatype, source, tag, comm, status)`

OUT	buf	initial address of receive buffer (choice)
IN	count	number of elements in receive buffer (non-negative integer)
IN	datatype	datatype of each receive buffer element (handle)
IN	source	rank of source or <code>MPI_ANY_SOURCE</code> (integer)
IN	tag	message tag or <code>MPI_ANY_TAG</code> (integer)
IN	comm	communicator (handle)
OUT	status	status object (Status)

```
int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source,
            int tag, MPI_Comm comm, MPI_Status *status)
```

```
MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror) BIND(C)
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count, source, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
```

a unique receiver. This matches a “push” communication mechanism, where data transfer is effected by the sender (rather than a “pull” mechanism, where data transfer is effected by the receiver).

Source = destination is allowed, that is, a process can send a message to itself. (However, it is unsafe to do so with the blocking send and receive operations described above, since this may lead to deadlock. See Section 3.5.)

Advice to implementors. Message context and other communicator information can be implemented as an additional tag field. It differs from the regular message tag in that wild card matching is not allowed on this field, and that value setting for this field is controlled by communicator manipulation functions. (*End of advice to implementors.*)

3.2.5 Return Status

The source or tag of a received message may not be known if wildcard values were used in the receive operation. Also, if multiple requests are completed by a single MPI function (see Section 3.7.5), a distinct error code may need to be returned for each request. The information is returned by the `status` argument of `MPI_RECV`. The type of `status` is MPI-defined. Status variables need to be explicitly allocated by the user, that is, they are not system objects.

In C, `status` is a structure that contains three fields named `MPI_SOURCE`, `MPI_TAG`, and `MPI_ERROR`; the structure may contain additional fields. Thus, `status.MPI_SOURCE`, `status.MPI_TAG` and `status.MPI_ERROR` contain the source, tag, and error code, respectively, of the received message.

In Fortran with `USE mpi` or `INCLUDE 'mpif.h'`, `status` is an array of `INTEGER`s of size `MPI_STATUS_SIZE`. The constants `MPI_SOURCE`, `MPI_TAG` and `MPI_ERROR` are the indices of the entries that store the source, tag and error fields. Thus, `status(MPI_SOURCE)`, `status(MPI_TAG)` and `status(MPI_ERROR)` contain, respectively, the source, tag and error code of the received message.

With Fortran `USE mpi_f08`, `status` is defined as the Fortran `BIND(C)` derived type `TYPE(MPI_Status)` containing three public fields named `MPI_SOURCE`, `MPI_TAG`, and `MPI_ERROR`. `TYPE(MPI_Status)` may contain additional, implementation-specific fields. Thus, `status%MPI_SOURCE`, `status%MPI_TAG` and `status%MPI_ERROR` contain the source, tag, and error code of a received message respectively. Additionally, within both the `mpi` and the `mpi_f08` modules, the constants `MPI_STATUS_SIZE`, `MPI_SOURCE`, `MPI_TAG`, `MPI_ERROR`, and `TYPE(MPI_Status)` are defined to allow conversion between both `status` representations. Conversion routines are provided in Section 16.3.5 on page 604.

Rationale. The Fortran `TYPE(MPI_Status)` is defined as a `BIND(C)` derived type so that it can be used at any location where the `status` integer array representation can be used, e.g., in user defined common blocks. (*End of rationale.*)

Rationale. It is allowed to have the same name (e.g., `MPI_SOURCE`) defined as a constant (e.g., Fortran parameter) and as a field of a derived type. (*End of rationale.*)

In C++, the `status` object is handled through the following methods:

```
{int MPI::Status::Get_source() const(binding deprecated, see Section 15.2) }
```

messages and return the actual envelope values of the received message. The use of a separate status argument prevents errors that are often attached with INOUT argument (e.g., using the MPI_ANY_TAG constant as the tag in a receive). Some libraries use calls that refer implicitly to the “last message received.” This is not thread safe.

The datatype argument is passed to MPI_GET_COUNT so as to improve performance. A message might be received without counting the number of elements it contains, and the count value is often not needed. Also, this allows the same function to be used after a call to MPI_PROBE or MPI_IPROBE. With a status from MPI_PROBE or MPI_IPROBE, the same datatypes are allowed as in a call to MPI_RECV to receive this message. (*End of rationale.*)

The value returned as the count argument of MPI_GET_COUNT for a datatype of length zero where zero bytes have been transferred is zero. If the number of bytes transferred is greater than zero, MPI_UNDEFINED is returned.

Rationale. Zero-length datatypes may be created in a number of cases. An important case is MPI_TYPE_CREATE_DARRAY, where the definition of the particular darray results in an empty block on some MPI process. Programs written in an SPMD style will not check for this special case and may want to use MPI_GET_COUNT to check the status. (*End of rationale.*)

Advice to users. The buffer size required for the receive can be affected by data conversions and by the stride of the receive datatype. In most cases, the safest approach is to use the same datatype with MPI_GET_COUNT and the receive. (*End of advice to users.*)

All send and receive operations use the buf, count, datatype, source, dest, tag, comm and status arguments in the same way as the blocking MPI_SEND and MPI_RECV operations described in this section.

3.2.6 Passing MPI_STATUS_IGNORE for Status

Every call to MPI_RECV includes a status argument, wherein the system can return details about the message received. There are also a number of other MPI calls where status is returned. An object of type MPI_STATUS is not an MPI opaque object; its structure is declared in mpi.h and mpif.h, and it exists in the user’s program. In many cases, application programs are constructed so that it is unnecessary for them to examine the status fields. In these cases, it is a waste for the user to allocate a status object, and it is particularly wasteful for the MPI implementation to fill in fields in this object.

To cope with this problem, there are two predefined constants, MPI_STATUS_IGNORE and MPI_STATUSES_IGNORE, which when passed to a receive, probe, wait, or test function, inform the implementation that the status fields are not to be filled in. Note that MPI_STATUS_IGNORE is not a special type of MPI_STATUS object; rather, it is a special value for the argument. In C one would expect it to be NULL, not the address of a special MPI_STATUS.

MPI_STATUS_IGNORE, and the array version MPI_STATUSES_IGNORE, can be used everywhere a status argument is passed to a receive, wait, or test function. MPI_STATUS_IGNORE cannot be used when status is an IN argument. Note that in Fortran MPI_STATUS_IGNORE

and `MPI_STATUSES_IGNORE` are objects like `MPI_BOTTOM` (not usable for initialization or assignment). See Section 2.5.4.

In general, this optimization can apply to all functions for which `status` or an array of `statuses` is an OUT argument. Note that this converts `status` into an INOUT argument. The functions that can be passed `MPI_STATUS_IGNORE` are all the various forms of `MPI_RECV`, `MPI_PROBE`, `MPI_TEST`, and `MPI_WAIT`, as well as `MPI_REQUEST_GET_STATUS`. When an array is passed, as in the `MPI_{TEST|WAIT}{ALL|SOME}` functions, a separate constant, `MPI_STATUSES_IGNORE`, is passed for the array argument. It is possible for an MPI function to return `MPI_ERR_IN_STATUS` even when `MPI_STATUS_IGNORE` or `MPI_STATUSES_IGNORE` has been passed to that function.

`MPI_STATUS_IGNORE` and `MPI_STATUSES_IGNORE` are not required to have the same values in C and Fortran.

It is not allowed to have some of the statuses in an array of statuses for `MPI_{TEST|WAIT}{ALL|SOME}` functions set to `MPI_STATUS_IGNORE`; one either specifies ignoring *all* of the statuses in such a call with `MPI_STATUSES_IGNORE`, or *none* of them by passing normal statuses in all positions in the array of statuses.

There are no C++ bindings for `MPI_STATUS_IGNORE` or `MPI_STATUSES_IGNORE`. To allow an OUT or INOUT `MPI::Status` argument to be ignored, all MPI C++ bindings that have OUT or INOUT `MPI::Status` parameters are overloaded with a second version that omits the OUT or INOUT `MPI::Status` parameter.

Example 3.1 The C++ bindings for `MPI_PROBE` are:

```
void MPI::Comm::Probe(int source, int tag, MPI::Status& status) const
void MPI::Comm::Probe(int source, int tag) const
```

3.3 Data Type Matching and Data Conversion

3.3.1 Type Matching Rules

One can think of message transfer as consisting of the following three phases.

1. Data is pulled out of the send buffer and a message is assembled.
2. A message is transferred from sender to receiver.
3. Data is pulled from the incoming message and disassembled into the receive buffer.

Type matching has to be observed at each of these three phases: The type of each variable in the sender buffer has to match the type specified for that entry by the send operation; the type specified by the send operation has to match the type specified by the receive operation; and the type of each variable in the receive buffer has to match the type specified for that entry by the receive operation. A program that fails to observe these three rules is erroneous.

To define type matching more precisely, we need to deal with two issues: matching of types of the host language with types specified in communication operations; and matching of types at sender and receiver.

The types of a send and receive match (phase two) if both operations use identical names. That is, `MPI_INTEGER` matches `MPI_INTEGER`, `MPI_REAL` matches `MPI_REAL`, and so on. There is one exception to this rule, discussed in Section 4.2, the type `MPI_PACKED` can match any other type.

might also be used for debugging purposes, as buffer overflow conditions are easier to diagnose than deadlock conditions.

Nonblocking message-passing operations, as described in Section 3.7, can be used to avoid the need for buffering outgoing messages. This prevents deadlocks due to lack of buffer space, and improves performance, by allowing overlap of computation and communication, and avoiding the overheads of allocating buffers and copying messages into buffers. (*End of advice to users.*)

3.6 Buffer Allocation and Usage

A user may specify a buffer to be used for buffering messages sent in buffered mode. Buffering is done by the sender.

`MPI_BUFFER_ATTACH(buffer, size)`

IN	buffer	initial buffer address (choice)
IN	size	buffer size, in bytes (non-negative integer)

`int MPI_Buffer_attach(void* buffer, int size)`

```
MPI_Buffer_attach(buffer, size, ierror) BIND(C)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buffer
    INTEGER, INTENT(IN) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

`MPI_BUFFER_ATTACH(BUFFER, SIZE, IERROR)`

```
<type> BUFFER(*)
    INTEGER SIZE, IERROR
```

```
{void MPI::Attach_buffer(void* buffer, int size) (binding deprecated, see
    Section 15.2) }
```

Provides to MPI a buffer in the user's memory to be used for buffering outgoing messages. The buffer is used only by messages sent in buffered mode. Only one buffer can be attached to a process at a time. In C and C++, *buffer* is the starting address of a memory region. In Fortran, one can pass the first element of a memory region or a whole array, which must be 'simply contiguous' (for 'simply contiguous', see also Section 16.2.12 on page 581).

`MPI_BUFFER_DETACH(buffer_addr, size)`

OUT	buffer_addr	initial buffer address (choice)
OUT	size	buffer size, in bytes (non-negative integer)

`int MPI_Buffer_detach(void* buffer_addr, int* size)`

```
MPI_Buffer_detach(buffer_addr, size, ierror) BIND(C)
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
```

```

1      TYPE(C_PTR), INTENT(OUT) :: buffer_addr
2      INTEGER, INTENT(OUT) :: size
3      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
5      MPI_BUFFER_DETACH(BUFFER_ADDR, SIZE, IERROR)
6      <type> BUFFER_ADDR(*)
7      INTEGER SIZE, IERROR
8
9      {int MPI::Detach_buffer(void*& buffer) (binding deprecated, see Section 15.2) }

```

Detach the buffer currently associated with MPI. The call returns the address and the size of the detached buffer. This operation will block until all messages currently in the buffer have been transmitted. Upon return of this function, the user may reuse or deallocate the space taken by the buffer.

Example 3.11 Calls to attach and detach buffers.

```

16      #define BUFFSIZE 10000
17      int size;
18      char *buff;
19      MPI_Buffer_attach( malloc(BUFFSIZE), BUFFSIZE);
20      /* a buffer of 10000 bytes can now be used by MPI_Bsend */
21      MPI_Buffer_detach( &buff, &size);
22      /* Buffer size reduced to zero */
23      MPI_Buffer_attach( buff, size);
24      /* Buffer of 10000 bytes available again */

```

Advice to users. Even though the C functions `MPI_Buffer_attach` and `MPI_Buffer_detach` both have a first argument of type `void*`, these arguments are used differently: A pointer to the buffer is passed to `MPI_Buffer_attach`; the address of the pointer is passed to `MPI_Buffer_detach`, so that this call can return the pointer value. In Fortran with the `mpi` module or `mpif.h`, the type of the `buffer_addr` argument is wrongly defined and the argument is therefore unused. In Fortran with the `mpi_f08` module, the address of the buffer is returned as `TYPE(C_PTR)`, see also Example 8.1 on page 330 about the use of `C_PTR` pointers. (*End of advice to users.*)

Rationale. Both arguments are defined to be of type `void*` (rather than `void*` and `void**`, respectively), so as to avoid complex type casts. E.g., in the last example, `&buff`, which is of type `char**`, can be passed as argument to `MPI_Buffer_detach` without type casting. If the formal parameter had type `void**` then we would need a type cast before and after the call. (*End of rationale.*)

The statements made in this section describe the behavior of MPI for buffered-mode sends. When no buffer is currently associated, MPI behaves as if a zero-sized buffer is associated with the process.

MPI must provide as much buffering for outgoing messages as if outgoing message data were buffered by the sending process, in the specified buffer space, using a circular, contiguous-space allocation policy. We outline below a model implementation that defines this policy. MPI may provide more buffering, and may use a better buffer allocation algorithm than described below. On the other hand, MPI may signal an error whenever the

```

1 MPI_IRSEND(buf, count, datatype, dest, tag, comm, request)
2     IN      buf                initial address of send buffer (choice)
3
4     IN      count              number of elements in send buffer (non-negative inte-
5                                ger)
6
7     IN      datatype           datatype of each send buffer element (handle)
8
9     IN      dest               rank of destination (integer)
10
11    IN      tag                 message tag (integer)
12
13    IN      comm                communicator (handle)
14
15    OUT     request             communication request (handle)

```

```

16 int MPI_Irsend(void* buf, int count, MPI_Datatype datatype, int dest,
17               int tag, MPI_Comm comm, MPI_Request *request)

```

```

18 MPI_Irsend(buf, count, datatype, dest, tag, comm, request, ierror) BIND(C)
19     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
20     INTEGER, INTENT(IN) :: count, dest, tag
21     TYPE(MPI_Datatype), INTENT(IN) :: datatype
22     TYPE(MPI_Comm), INTENT(IN) :: comm
23     TYPE(MPI_Request), INTENT(OUT) :: request
24     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

25 MPI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
26     <type> BUF(*)
27     INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

```

```

28 {MPI::Request MPI::Comm::Irsend(const void* buf, int count, const
29                               MPI::Datatype& datatype, int dest, int tag) const(binding
30                               deprecated, see Section 15.2) }

```

Start a ready mode nonblocking send.

```

33 MPI_IRECV (buf, count, datatype, source, tag, comm, request)
34
35     OUT     buf                initial address of receive buffer (choice)
36
37     IN      count              number of elements in receive buffer (non-negative in-
38                                te-ger)
39
40     IN      datatype           datatype of each receive buffer element (handle)
41
42     IN      source              rank of source or MPI_ANY_SOURCE (integer)
43
44     IN      tag                 message tag or MPI_ANY_TAG (integer)
45
46     IN      comm                communicator (handle)
47
48     OUT     request             communication request (handle)

```

```

49 int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source,
50               int tag, MPI_Comm comm, MPI_Request *request)

```



```

MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror) BIND(C)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count, source, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR

{MPI::Request MPI::Comm::Irecv(void* buf, int count, const
    MPI::Datatype& datatype, int source, int tag) const(binding
    deprecated, see Section 15.2) }

```

Start a nonblocking receive.

These calls allocate a communication request object and associate it with the request handle (the argument `request`). The request can be used later to query the status of the communication or wait for its completion.

A nonblocking send call indicates that the system may start copying data out of the send buffer. The sender should not modify any part of the send buffer after a nonblocking send operation is called, until the send completes.

A nonblocking receive call indicates that the system may start writing data into the receive buffer. The receiver should not access any part of the receive buffer after a nonblocking receive operation is called, until the receive completes.

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 16.2.10-16.2.20, especially in Sections 16.2.12 and 16.2.13 on pages 581-584 about “Problems Due to Data Copying and Sequence Association with Subscript Triplets” and “Vector Subscripts”, and in Sections 16.2.16 to 16.2.19 on pages 586 to 597 about “Optimization Problems”, “Code Movements and Register Optimization”, “Temporary Data Movements” and “Permanent Data Movements”. (*End of advice to users.*)

ticket238-J.
ticket238-J.
ticket236-H.
ticket238-J.

3.7.3 Communication Completion

The functions `MPI_WAIT` and `MPI_TEST` are used to complete a nonblocking communication. The completion of a send operation indicates that the sender is now free to update the locations in the send buffer (the send operation itself leaves the content of the send buffer unchanged). It does not indicate that the message has been received, rather, it may have been buffered by the communication subsystem. However, if a **synchronous** mode send was used, the completion of the send operation indicates that a matching receive was initiated, and that the message will eventually be received by this matching receive.

The completion of a receive operation indicates that the receive buffer contains the received message, the receiver is now free to access it, and that the status object is set. It does not indicate that the matching send operation has completed (but indicates, of course, that the send was initiated).

We shall use the following terminology: A **null** handle is a handle with value `MPI_REQUEST_NULL`. A persistent request and the handle to it are **inactive** if the request

the user send buffer can be reused — i.e., data has been sent out or copied into a buffer attached with `MPI_BUFFER_ATTACH`. Note that, at this point, we can no longer cancel the send (see Section 3.8). If a matching receive is never posted, then the buffer cannot be freed. This runs somewhat counter to the stated goal of `MPI_CANCEL` (always being able to free program space that was committed to the communication subsystem). (*End of advice to users.*)

Advice to implementors. In a multi-threaded environment, a call to `MPI_WAIT` should block only the calling thread, allowing the thread scheduler to schedule another thread for execution. (*End of advice to implementors.*)

`MPI_TEST(request, flag, status)`

INOUT	request	communication request (handle)
OUT	flag	true if operation completed (logical)
OUT	status	status object (Status)

`int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)`

```

MPI_Test(request, flag, status, ierror) BIND(C)
  TYPE(MPI_Request), INTENT(INOUT) :: request
  LOGICAL, INTENT(OUT) :: flag
  TYPE(MPI_Status) :: status
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

MPI_TEST(REQUEST, FLAG, STATUS, IERROR)
  LOGICAL FLAG
  INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR

```

```

{bool MPI::Request::Test(MPI::Status& status) (binding deprecated, see
  Section 15.2) }

```

```

{bool MPI::Request::Test() (binding deprecated, see Section 15.2) }

```

A call to `MPI_TEST` returns `flag = true` if the operation identified by `request` is complete. In such a case, the status object is set to contain information on the completed operation; if the communication object was created by a nonblocking send or receive, then it is deallocated and the request handle is set to `MPI_REQUEST_NULL`. The call returns `flag = false`, otherwise. In this case, the value of the status object is undefined. `MPI_TEST` is a local operation.

The return status object for a receive operation carries information that can be accessed as described in Section 3.2.5. The status object for a send operation carries information that can be accessed by a call to `MPI_TEST_CANCELLED` (see Section 3.8).

One is allowed to call `MPI_TEST` with a null or inactive `request` argument. In such a case the operation returns with `flag = true` and empty `status`.

The functions `MPI_WAIT` and `MPI_TEST` can be used to complete both sends and receives.

One is allowed to call `MPI_REQUEST_GET_STATUS` with a null or inactive request argument. In such a case the operation returns with `flag=true` and empty status.

3.8 Probe and Cancel

The `MPI_PROBE` and `MPI_IPROBE` operations allow incoming messages to be checked for, without actually receiving them. The user can then decide how to receive them, based on the information returned by the probe (basically, the information returned by `status`). In particular, the user may allocate memory for the receive buffer, according to the length of the probed message.

The `MPI_CANCEL` operation allows pending communications to be canceled. This is required for cleanup. Posting a send or a receive ties up user resources (send or receive buffers), and a cancel may be needed to free these resources gracefully.

`MPI_IPROBE(source, tag, comm, flag, status)`

IN	source	rank of source or <code>MPI_ANY_SOURCE</code> (integer)
IN	tag	message tag or <code>MPI_ANY_TAG</code> (integer)
IN	comm	communicator (handle)
OUT	flag	(logical)
OUT	status	status object (<code>Status</code>)

```
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,
               MPI_Status *status)
```

```
MPI_Iprobe(source, tag, comm, flag, status, ierror) BIND(C)
  INTEGER, INTENT(IN) :: source, tag
  TYPE(MPI_Comm), INTENT(IN) :: comm
  LOGICAL, INTENT(OUT) :: flag
  TYPE(MPI_Status) :: status
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR)
  LOGICAL FLAG
  INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
```

```
{bool MPI::Comm::Iprobe(int source, int tag, MPI::Status& status)
  const(binding deprecated, see Section 15.2) }
```

```
{bool MPI::Comm::Iprobe(int source, int tag) const(binding deprecated, see
  Section 15.2) }
```

`MPI_IPROBE(source, tag, comm, flag, status)` returns `flag = true` if there is a message that can be received and that matches the pattern specified by the arguments `source`, `tag`, and `comm`. The call matches the same message that would have been received by a call to `MPI_RECV(..., source, tag, comm, status)` executed at the same point in the program, and returns in `status` the same value that would have been returned by `MPI_RECV()`. Otherwise, the call returns `flag = false`, and leaves `status` undefined.

If `MPI_Iprobe` returns `flag = true`, then the content of the status object can be subsequently accessed as described in Section 3.2.5 to find the source, tag and length of the probed message.

A subsequent receive executed with the same communicator, and the source and tag returned in status by `MPI_Iprobe` will receive the message that was matched by the probe, if no other intervening receive occurs after the probe, and the send is not successfully cancelled before the receive. If the receiving process is multi-threaded, it is the user's responsibility to ensure that the last condition holds.

The source argument of `MPI_Probe` can be `MPI_ANY_SOURCE`, and the tag argument can be `MPI_ANY_TAG`, so that one can probe for messages from an arbitrary source and/or with an arbitrary tag. However, a specific communication context must be provided with the `comm` argument.

It is not necessary to receive a message immediately after it has been probed for, and the same message may be probed for several times before it is received.

`MPI_Probe(source, tag, comm, status)`

IN	source	rank of source or <code>MPI_ANY_SOURCE</code> (integer)
IN	tag	message tag or <code>MPI_ANY_TAG</code> (integer)
IN	comm	communicator (handle)
OUT	status	status object (Status)

`int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)`

```
MPI_Probe(source, tag, comm, status, ierror) BIND(C)
    INTEGER, INTENT(IN) :: source, tag
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

`MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)`

`INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR`

```
{void MPI::Comm::Probe(int source, int tag, MPI::Status& status)
    const(binding deprecated, see Section 15.2) }
```

```
{void MPI::Comm::Probe(int source, int tag) const(binding deprecated, see
    Section 15.2) }
```

`MPI_PROBE` behaves like `MPI_Iprobe` except that it is a blocking call that returns only after a matching message has been found.

The MPI implementation of `MPI_PROBE` and `MPI_Iprobe` needs to guarantee progress: if a call to `MPI_PROBE` has been issued by a process, and a send that matches the probe has been initiated by some process, then the call to `MPI_PROBE` will return, unless the message is received by another concurrent receive operation (that is executed by another thread at the probing process). Similarly, if a process busy waits with `MPI_Iprobe` and a matching message has been issued, then the call to `MPI_Iprobe` will eventually return `flag = true` unless the message is received by another concurrent receive operation.

3.9 Persistent Communication Requests

Often a communication with the same argument list is repeatedly executed within the inner loop of a parallel computation. In such a situation, it may be possible to optimize the communication by binding the list of communication arguments to a **persistent** communication request once and, then, repeatedly using the request to initiate and complete messages. The persistent request thus created can be thought of as a communication port or a “half-channel.” It does not provide the full functionality of a conventional channel, since there is no binding of the send port to the receive port. This construct allows reduction of the overhead for communication between the process and communication controller, but not of the overhead for communication between one communication controller and another. It is not necessary that messages sent with a persistent request be received by a receive operation using a persistent request, or vice versa.

A persistent communication request is created using one of the five following calls. These calls involve no communication.

`MPI_SEND_INIT(buf, count, datatype, dest, tag, comm, request)`

IN	buf	initial address of send buffer (choice)
IN	count	number of elements sent (non-negative integer)
IN	datatype	type of each element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

`int MPI_Send_init(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)`

`MPI_Send_init(buf, count, datatype, dest, tag, comm, request, ierror)`

`BIND(C)`

```

TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
INTEGER, INTENT(IN) :: count, dest, tag
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

`MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)`

`<type> BUF(*)`

`INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR`

```

{MPI::Prequest MPI::Comm::Send_init(const void* buf, int count, const
MPI::Datatype& datatype, int dest, int tag) const(binding
deprecated, see Section 15.2) }

```

Creates a persistent communication request for a standard mode send operation, and binds to it all the arguments of a send operation.

MPI_BSEND_INIT(buf, count, datatype, dest, tag, comm, request)

IN	buf	initial address of send buffer (choice)
IN	count	number of elements sent (non-negative integer)
IN	datatype	type of each element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

```
int MPI_Bsend_init(void* buf, int count, MPI_Datatype datatype, int dest,
                  int tag, MPI_Comm comm, MPI_Request *request)
```

```
MPI_Bsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
    BIND(C)
```

```
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_BSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

```
{MPI::Prequest MPI::Comm::Bsend_init(const void* buf, int count, const
    MPI::Datatype& datatype, int dest, int tag) const(binding
    deprecated, see Section 15.2) }
```

Creates a persistent communication request for a buffered mode send.

MPI_SSEND_INIT(buf, count, datatype, dest, tag, comm, request)

IN	buf	initial address of send buffer (choice)
IN	count	number of elements sent (non-negative integer)
IN	datatype	type of each element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

```
int MPI_Ssend_init(void* buf, int count, MPI_Datatype datatype, int dest,
                  int tag, MPI_Comm comm, MPI_Request *request)
```

```
MPI_Ssend_init(buf, count, datatype, dest, tag, comm, request, ierror)
    BIND(C)
```

```

1      TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
2      INTEGER, INTENT(IN) :: count, dest, tag
3      TYPE(MPI_Datatype), INTENT(IN) :: datatype
4      TYPE(MPI_Comm), INTENT(IN) :: comm
5      TYPE(MPI_Request), INTENT(OUT) :: request
6      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
8      MPI_SSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
9          <type> BUF(*)
10         INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
11     {MPI::Prequest MPI::Comm::Ssend_init(const void* buf, int count, const
12         MPI::Datatype& datatype, int dest, int tag) const(binding
13         deprecated, see Section 15.2) }
14
15     Creates a persistent communication object for a synchronous mode send operation.

```

```

17      MPI_RSEND_INIT(buf, count, datatype, dest, tag, comm, request)
18
19      IN          buf                initial address of send buffer (choice)
20      IN          count              number of elements sent (non-negative integer)
21      IN          datatype            type of each element (handle)
22      IN          dest                rank of destination (integer)
23      IN          tag                 message tag (integer)
24      IN          comm                communicator (handle)
25      OUT         request              communication request (handle)
26
27
28
29      int MPI_Rsend_init(void* buf, int count, MPI_Datatype datatype, int dest,
30          int tag, MPI_Comm comm, MPI_Request *request)

```

ticket229.2.
ticket-248T.

```

31      MPI_Rsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
32          BIND(C)
33      TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
34      INTEGER, INTENT(IN) :: count, dest, tag
35      TYPE(MPI_Datatype), INTENT(IN) :: datatype
36      TYPE(MPI_Comm), INTENT(IN) :: comm
37      TYPE(MPI_Request), INTENT(OUT) :: request
38      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40      MPI_RSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
41          <type> BUF(*)
42         INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
43
44     {MPI::Prequest MPI::Comm::Rsend_init(const void* buf, int count, const
45         MPI::Datatype& datatype, int dest, int tag) const(binding
46         deprecated, see Section 15.2) }

```

Creates a persistent communication object for a ready mode send operation.

Create (Start Complete)* Free

where * indicates zero or more repetitions. If the same communication object is used in several concurrent threads, it is the user's responsibility to coordinate calls so that the correct sequence is obeyed.

A send operation initiated with `MPI_START` can be matched with any receive operation and, likewise, a receive operation initiated with `MPI_START` can receive messages generated by any send operation.

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in [Sections 16.2.10-16.2.20](#), especially in [Sections 16.2.12 and 16.2.13](#) on pages 581-584 about “Problems Due to Data Copying and Sequence Association with Subscript Triplets” and “Vector Subscripts”, and in [Sections 16.2.16 to 16.2.19](#) on pages 586 to 597 about “Optimization Problems”, “Code Movements and Register Optimization”, “Temporary Data Movements” and “Permanent Data Movements”. (*End of advice to users.*)

3.10 Send-Receive

The **send-receive** operations combine in one call the sending of a message to one destination and the receiving of another message, from another process. The two (source and destination) are possibly the same. A send-receive operation is very useful for executing a shift operation across a chain of processes. If blocking sends and receives are used for such a shift, then one needs to order the sends and receives correctly (for example, even processes send, then receive, odd processes receive first, then send) so as to prevent cyclic dependencies that may lead to deadlock. When a send-receive operation is used, the communication subsystem takes care of these issues. The send-receive operation can be used in conjunction with the functions described in Chapter 7 in order to perform shifts on various logical topologies. Also, a send-receive operation is useful for implementing remote procedure calls.

A message sent by a send-receive operation can be received by a regular receive operation or probed by a probe operation; a send-receive operation can receive a message sent by a regular send operation.

Chapter 4

Datatypes

Basic datatypes were introduced in Section 3.2.2 Message Data on page 29 and in Section 3.3 Data Type Matching and Data Conversion on page 37. In this chapter, this model is extended to describe any data layout. We consider general datatypes that allow one to transfer efficiently heterogeneous and noncontiguous data. We conclude with the description of calls for explicit packing and unpacking of messages.

4.1 Derived Datatypes

Up to here, all point to point communication have involved only buffers containing a sequence of identical basic datatypes. This is too constraining on two accounts. One often wants to pass messages that contain values with different datatypes (e.g., an integer count, followed by a sequence of real numbers); and one often wants to send noncontiguous data (e.g., a sub-block of a matrix). One solution is to pack noncontiguous data into a contiguous buffer at the sender site and unpack it at the receiver site. This has the disadvantage of requiring additional memory-to-memory copy operations at both sites, even when the communication subsystem has scatter-gather capabilities. Instead, MPI provides mechanisms to specify more general, mixed, and noncontiguous communication buffers. It is up to the implementation to decide whether data should be first packed in a contiguous buffer before being transmitted, or whether it can be collected directly from where it resides.

The general mechanisms provided here allow one to transfer directly, without copying, objects of various shape and size. It is not assumed that the MPI library is cognizant of the objects declared in the host language. Thus, if one wants to transfer a structure, or an array section, it will be necessary to provide in MPI a definition of a communication buffer that mimics the definition of the structure or array section in question. These facilities can be used by library designers to define communication functions that can transfer objects defined in the host language — by decoding their definitions as available in a symbol table or a dope vector. Such higher-level communication functions are not part of MPI.

More general communication buffers are specified by replacing the basic datatypes that have been used so far with derived datatypes that are constructed from basic datatypes using the constructors described in this section. These methods of constructing derived datatypes can be applied recursively.

A **general datatype** is an opaque object that specifies two things:

- A sequence of basic datatypes

- A sequence of integer (byte) displacements

The displacements are not required to be positive, distinct, or in increasing order. Therefore, the order of items need not coincide with their order in store, and an item may appear more than once. We call such a pair of sequences (or sequence of pairs) a **type map**. The sequence of basic datatypes (displacements ignored) is the **type signature** of the datatype.

Let

$$Typemap = \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\},$$

be such a type map, where $type_i$ are basic types, and $disp_i$ are displacements. Let

$$Typesig = \{type_0, \dots, type_{n-1}\}$$

be the associated type signature. This type map, together with a base address buf , specifies a communication buffer: the communication buffer that consists of n entries, where the i -th entry is at address $buf + disp_i$ and has type $type_i$. A message assembled from such a communication buffer will consist of n values, of the types defined by $Typesig$.

Most datatype constructors have replication count or block length arguments. Allowed values are non-negative integers. If the value is zero, no elements are generated in the type map and there is no effect on datatype bounds or extent.

We can use a handle to a general datatype as an argument in a send or receive operation, instead of a basic datatype argument. The operation `MPI_SEND(buf, 1, datatype,...)` will use the send buffer defined by the base address `buf` and the general datatype associated with `datatype`; it will generate a message with the type signature determined by the `datatype` argument. `MPI_RECV(buf, 1, datatype,...)` will use the receive buffer defined by the base address `buf` and the general datatype associated with `datatype`.

General datatypes can be used in all send and receive operations. We discuss, in Section 4.1.11, the case where the second argument `count` has value > 1 .

The basic datatypes presented in Section 3.2.2 are particular cases of a general datatype, and are predefined. Thus, `MPI_INT` is a predefined handle to a datatype with type map $\{(int, 0)\}$, with one entry of type `int` and displacement zero. The other basic datatypes are similar.

The **extent** of a datatype is defined to be the span from the first byte to the last byte occupied by entries in this datatype, rounded up to satisfy alignment requirements. That is, if

$$Typemap = \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\},$$

then

$$\begin{aligned} lb(Typemap) &= \min_j disp_j, \\ ub(Typemap) &= \max_j (disp_j + sizeof(type_j)) + \epsilon, \text{ and} \\ extent(Typemap) &= ub(Typemap) - lb(Typemap). \end{aligned} \tag{4.1}$$

If $type_i$ requires alignment to a byte address that is a multiple of k_i , then ϵ is the least non-negative increment needed to round $extent(Typemap)$ to the next multiple of $\max_i k_i$.

In Fortran, whether the alignments k_i are computed according to the alignments used by the compiler in common blocks, `SEQUENCE` derived types, `BIND(C)` derived types, or derived

types that are neither **SEQUENCE** nor **BIND(C)**, is implementation-dependent. The complete definition of **extent** is given in Section 4.1.6 on page 106.

Example 4.1 Assume that $Type = \{(\text{double}, 0), (\text{char}, 8)\}$ (a **double** at displacement zero, followed by a **char** at displacement eight). Assume, furthermore, that doubles have to be strictly aligned at addresses that are multiples of eight. Then, the extent of this datatype is 16 (9 rounded to the next multiple of 8). A datatype that consists of a character immediately followed by a double will also have an extent of 16.

Rationale. The definition of extent is motivated by the assumption that the amount of padding added at the end of each structure in an array of structures is the least needed to fulfill alignment constraints. More explicit control of the extent is provided in Section 4.1.6. Such explicit control is needed in cases where the assumption does not hold, for example, where union types are used. In Fortran, structures can be expressed with several language features, e.g., common blocks, **SEQUENCE** derived types, or **BIND(C)** derived types. The compiler may use different alignments, and therefore, it is recommended to use **MPI_TYPE_CREATE_RESIZED** for arrays of structures if an alignment may cause an alignment-gap at the end of a structure as described in Section 4.1.6 on page 106 and in Section 16.2.15 on page 584. (*End of rationale.*)

4.1.1 Type Constructors with Explicit Addresses

In Fortran, the functions **MPI_TYPE_CREATE_HVECTOR**, **MPI_TYPE_CREATE_HINDEXED**, **MPI_TYPE_CREATE_STRUCT**, and **MPI_GET_ADDRESS** accept arguments of type **INTEGER(KIND=MPI_ADDRESS_KIND)**, wherever arguments of type **MPI_Aint** and **MPI::Aint** are used in C and C++. On Fortran 77 systems that do not support the Fortran 90 **KIND** notation, and where addresses are 64 bits whereas default **INTEGER**s are 32 bits, these arguments will be of type **INTEGER*8**.

4.1.2 Datatype Constructors

Contiguous The simplest datatype constructor is **MPI_TYPE_CONTIGUOUS** which allows replication of a datatype into contiguous locations.

MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)

IN	count	replication count (non-negative integer)
IN	oldtype	old datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
                        MPI_Datatype *newtype)
```

```
MPI_Type_contiguous(count, oldtype, newtype, ierror) BIND(C)
  INTEGER, INTENT(IN) :: count
  TYPE(MPI_Datatype), INTENT(IN) :: oldtype
  TYPE(MPI_Datatype), INTENT(OUT) :: newtype
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```

1      <oldtype> FILEARRAY(100, 200, 300)
2      !HPF$ PROCESSORS PROCESSES(2, 3)
3      !HPF$ DISTRIBUTE FILEARRAY(CYCLIC(10), *, BLOCK) ONTO PROCESSES

```

This can be achieved by the following Fortran code, assuming there will be six processes attached to the run:

```

7      ndims = 3
8      array_of_gsizes(1) = 100
9      array_of_distribs(1) = MPI_DISTRIBUTE_CYCLIC
10     array_of_dargs(1) = 10
11     array_of_gsizes(2) = 200
12     array_of_distribs(2) = MPI_DISTRIBUTE_NONE
13     array_of_dargs(2) = 0
14     array_of_gsizes(3) = 300
15     array_of_distribs(3) = MPI_DISTRIBUTE_BLOCK
16     array_of_dargs(3) = MPI_DISTRIBUTE_DFLT_DARG
17     array_of_psizes(1) = 2
18     array_of_psizes(2) = 1
19     array_of_psizes(3) = 3
20     call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
21     call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
22     call MPI_TYPE_CREATE_DARRAY(size, rank, ndims, array_of_gsizes, &
23                                array_of_distribs, array_of_dargs, array_of_psizes, &
24                                MPI_ORDER_FORTRAN, oldtype, newtype, ierr)

```

4.1.5 Address and Size Functions

The displacements in a general datatype are relative to some initial buffer address. **Absolute addresses** can be substituted for these displacements: we treat them as displacements relative to “address zero,” the start of the address space. This initial address zero is indicated by the constant `MPI_BOTTOM`. Thus, a datatype can specify the absolute address of the entries in the communication buffer, in which case the `buf` argument is passed the value `MPI_BOTTOM`.

The address of a location in memory can be found by invoking the function `MPI_GET_ADDRESS`.

```

38 MPI_GET_ADDRESS(location, address)

```

IN	location	location in caller memory (choice)
OUT	address	address of location (integer)

```

42 int MPI_Get_address(void *location, MPI_Aint *address)

```

```

43 MPI_Get_address(location, address, ierror) BIND(C)
44     TYPE(*), DIMENSION(..) :: location
45     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: address
46     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

MPI_GET_ADDRESS(LOCATION, ADDRESS, IERROR)
  <type> LOCATION(*)
  INTEGER IERROR
  INTEGER(KIND=MPI_ADDRESS_KIND) ADDRESS

```

```

{MPI::Aint MPI::Get_address(void* location) (binding deprecated, see Section 15.2)
  }

```

This function replaces MPI_ADDRESS, whose use is deprecated. See also Chapter 15. Returns the (byte) address of location.

Advice to users. Current Fortran MPI codes will run unmodified, and will port to any system. However, they may fail if addresses larger than $2^{32} - 1$ are used in the program. New codes should be written so that they use the new functions. This provides compatibility with C/C++ and avoids errors on 64 bit architectures. However, such newly written codes may need to be (slightly) rewritten to port to old Fortran 77 environments that do not support KIND declarations. (*End of advice to users.*)

Rationale. In the mpi_f08 module, the location argument is not defined with INTENT(IN) because existing applications may use MPI_GET_ADDRESS (and the deprecated MPI_ADDRESS) as a substitute for MPI_F_SYNC_REG that was not defined before MPI-3.0. (*End of rationale.*)

Example 4.8 Using MPI_GET_ADDRESS for an array.

```

REAL A(100,100)
INTEGER(KIND=MPI_ADDRESS_KIND) I1, I2, DIFF
CALL MPI_GET_ADDRESS(A(1,1), I1, IERROR)
CALL MPI_GET_ADDRESS(A(10,10), I2, IERROR)
DIFF = I2 - I1
! The value of DIFF is 909*sizeofreal; the values of I1 and I2 are
! implementation dependent.

```

Advice to users. C users may be tempted to avoid the usage of MPI_GET_ADDRESS and rely on the availability of the address operator &. Note, however, that & *cast-expression* is a pointer, not an address. ISO C does not require that the value of a pointer (or the pointer cast to int) be the absolute address of the object pointed at — although this is commonly the case. Furthermore, referencing may not have a unique definition on machines with a segmented address space. The use of MPI_GET_ADDRESS to “reference” C variables guarantees portability to such machines as well. (*End of advice to users.*)

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 16.2.10-16.2.20, especially in Sections 16.2.12 and 16.2.13 on pages 581-584 about “Problems Due to Data Copying and Sequence Association with Subscript Triplets” and “Vector Subscripts”, and in Sections 16.2.16 to 16.2.19 on pages 586 to 597 about “Optimization Problems”, “Code Movements and Register Optimization”, “Temporary Data Movements” and “Permanent Data Movements”. (*End of advice to users.*)

The following auxiliary function provides useful information on derived datatypes.

`MPI_TYPE_SIZE(datatype, size)`

IN	datatype	datatype (handle)
OUT	size	datatype size (integer)

`int MPI_Type_size(MPI_Datatype datatype, int *size)`

```
MPI_Type_size(datatype, size, ierror) BIND(C)
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, INTENT(OUT) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR)
    INTEGER DATATYPE, SIZE, IERROR
```

```
{int MPI::Datatype::Get_size() const(binding deprecated, see Section 15.2) }
```

`MPI_TYPE_SIZE` returns the total size, in bytes, of the entries in the type signature associated with `datatype`; i.e., the total size of the data in a message that would be created with this datatype. Entries that occur multiple times in the datatype are counted with their multiplicity.

4.1.6 Lower-Bound and Upper-Bound Markers

It is often convenient to define explicitly the lower bound and upper bound of a type map, and override the definition given on page 107. This allows one to define a datatype that has “holes” at its beginning or its end, or a datatype with entries that extend above the upper bound or below the lower bound. Examples of such usage are provided in Section 4.1.14. Also, the user may want to override the alignment rules that are used to compute upper bounds and extents. E.g., a C compiler may allow the user to override default alignment rules for some of the structures within a program. The user has to specify explicitly the bounds of the datatypes that match these structures.

To achieve this, we add two additional “pseudo-datatypes,” `MPI_LB` and `MPI_UB`, that can be used, respectively, to mark the lower bound or the upper bound of a datatype. These pseudo-datatypes occupy no space ($\text{extent}(\text{MPI_LB}) = \text{extent}(\text{MPI_UB}) = 0$). They do not affect the size or count of a datatype, and do not affect the content of a message created with this datatype. However, they do affect the definition of the extent of a datatype and, therefore, affect the outcome of a replication of this datatype by a datatype constructor.

Example 4.9 Let $D = (-3, 0, 6)$; $T = (\text{MPI_LB}, \text{MPI_INT}, \text{MPI_UB})$, and $B = (1, 1, 1)$. Then a call to `MPI_TYPE_STRUCT(3, B, D, T, type1)` creates a new datatype that has an extent of 9 (from -3 to 5, 5 included), and contains an integer at displacement 0. This is the datatype defined by the sequence $\{(\text{lb}, -3), (\text{int}, 0), (\text{ub}, 6)\}$. If this type is replicated twice by a call to `MPI_TYPE_CONTIGUOUS(2, type1, type2)` then the newly created type can be described by the sequence $\{(\text{lb}, -3), (\text{int}, 0), (\text{int}, 9), (\text{ub}, 15)\}$. (An entry of type `ub` can be deleted if there is another entry of type `ub` with a higher displacement; an entry of type `lb` can be deleted if there is another entry of type `lb` with a lower displacement.)

In general, if

$$Typemap = \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\},$$

then the **lower bound** of $Typemap$ is defined to be

$$lb(Typemap) = \begin{cases} \min_j disp_j & \text{if no entry has basic type lb} \\ \min_j \{disp_j \text{ such that } type_j = \text{lb}\} & \text{otherwise} \end{cases}$$

Similarly, the **upper bound** of $Typemap$ is defined to be

$$ub(Typemap) = \begin{cases} \max_j disp_j + sizeof(type_j) + \epsilon & \text{if no entry has basic type ub} \\ \max_j \{disp_j \text{ such that } type_j = \text{ub}\} & \text{otherwise} \end{cases}$$

Then

$$extent(Typemap) = ub(Typemap) - lb(Typemap)$$

If $type_i$ requires alignment to a byte address that is a multiple of k_i , then ϵ is the least non-negative increment needed to round $extent(Typemap)$ to the next multiple of $\max_i k_i$. In Fortran, whether the alignments k_i are computed according to the alignments used by the compiler in common blocks, SEQUENCE derived types, BIND(C) derived types, or derived types that are neither SEQUENCE nor BIND(C), is implementation-dependent.

The formal definitions given for the various datatype constructors apply now, with the amended definition of **extent**.

Rationale. Before Fortran 2003, MPI_TYPE_CREATE_STRUCT could be applied to Fortran common blocks and SEQUENCE derived types. With Fortran 2003, this list was extended by BIND(C) derived types and MPI implementors have implemented the alignments k_i differently, i.e., some based on the alignments used in SEQUENCE derived types, and others according to BIND(C) derived types. (*End of rationale.*)

Advice to implementors. In Fortran, it is generally recommended to use BIND(C) derived types instead of common blocks or SEQUENCE derived types. Therefore it is recommended to calculate the alignments k_i based on BIND(C) derived types. (*End of advice to implementors.*)

Advice to users. Structures combining different basic datatypes should be defined so that there will be no gaps based on alignment rules, and if used as an array of structures, then also without such an alignment-gap at the end of the structure. In MPI communication, the content of such gaps would not be communicated into the receiver's buffer. For example, such an alignment-gap may occur between an odd number of floats or REALs before a double or DOUBLE PRECISION data. Such gaps may be added explicitly to both the structure and the MPI derived datatype handle because the communication of a contiguous derived datatype may be significantly faster than the communication of one that is non-contiguous due to such alignment-gaps.

Example: Instead of

```

1      TYPE, BIND(C) :: my_data
2          REAL, DIMENSION(3) :: x
3          ! there may be a gap of the size of one REAL
4          ! if the alignment of a DOUBLE PRECISION is
5          ! two times the size of a REAL
6          DOUBLE PRECISION: p
7      END TYPE

```

one should define

```

11     TYPE, BIND(C) :: my_data
12         REAL, DIMENSION(3) :: x
13         REAL :: gap1
14         DOUBLE PRECISION: p
15     END TYPE

```

and also including `gap1` in the matching MPI derived datatype. It is required that all processes in a communication add the same gaps, i.e., defined with the same basic datatype. Both, the original and the modified structures are portable, but may have different performance implications for the communication and memory accesses during computation on systems with different alignment values.

In principle, a compiler may define an additional alignment rule for structures, e.g., to use at least 4 or 8 byte alignment although the content may have a $max_i k_i$ alignment less than this structure alignment. To keep an application portable, it is therefore recommended to always resize structure derived datatype handles if used in an array of structures, see the Example in Section 16.2.15 on page 584. (*End of advice to users.*)

4.1.7 Extent and Bounds of Datatypes

The following function replaces the three functions `MPI_TYPE_UB`, `MPI_TYPE_LB` and `MPI_TYPE_EXTENT`. It also returns address sized integers, in the Fortran binding. The use of `MPI_TYPE_UB`, `MPI_TYPE_LB` and `MPI_TYPE_EXTENT` is deprecated.

`MPI_TYPE_GET_EXTENT(datatype, lb, extent)`

IN	<code>datatype</code>	datatype to get information on (handle)
OUT	<code>lb</code>	lower bound of datatype (integer)
OUT	<code>extent</code>	extent of datatype (integer)

```

int MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *lb,
                        MPI_Aint *extent)

```

```

MPI_Type_get_extent(datatype, lb, extent, ierror) BIND(C)
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: lb, extent
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```



```

1 CALL MPI_TYPE_COMMIT(type1, ierr)
2     ! now type1 can be used anew for communication
3
4

```

```

5 MPI_TYPE_FREE(datatype)
6
7     INOUT    datatype                datatype that is freed (handle)
8

```

```

9 int MPI_Type_free(MPI_Datatype *datatype)

```

```

10 MPI_Type_free(datatype, ierror) BIND(C)
11     TYPE(MPI_Datatype), INTENT(INOUT) :: datatype
12     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13

```

```

14 MPI_TYPE_FREE(DATATYPE, IERROR)
15     INTEGER DATATYPE, IERROR
16

```

```

17 {void MPI::Datatype::Free() (binding deprecated, see Section 15.2) }

```

Marks the datatype object associated with **datatype** for deallocation and sets **datatype** to `MPI_DATATYPE_NULL`. Any communication that is currently using this datatype will complete normally. Freeing a datatype does not affect any other datatype that was built from the freed datatype. The system behaves as if input datatype arguments to derived datatype constructors are passed by value.

Advice to implementors. The implementation may keep a reference count of active communications that use the datatype, in order to decide when to free it. Also, one may implement constructors of derived datatypes so that they keep pointers to their datatype arguments, rather than copying them. In this case, one needs to keep track of active datatype definition references in order to know when a datatype object can be freed. (*End of advice to implementors.*)

4.1.10 Duplicating a Datatype

```

34 MPI_TYPE_DUP(oldtype, newtype)

```

```

35     IN      [ticket252-W.]oldtype        datatype (handle)
36
37     OUT     newtype                      copy of oldtype (handle)
38

```

```

39 int MPI_Type_dup(MPI_Datatype oldtype, MPI_Datatype *newtype)

```

```

40 MPI_Type_dup(oldtype, newtype, ierror) BIND(C)
41     TYPE(MPI_Datatype), INTENT(IN) :: oldtype
42     TYPE(MPI_Datatype), INTENT(OUT) :: newtype
43     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44

```

```

45 MPI_TYPE_DUP(OLDTYPE, NEWTYPE, IERROR)
46     INTEGER OLDTYPE, NEWTYPE, IERROR
47

```

```

48 {MPI::Datatype MPI::Datatype::Dup() const (binding deprecated, see Section 15.2) }

```


MPI_TYPE_DUP is a type constructor which duplicates the existing type with associated key values. For each key value, the respective copy callback function determines the attribute value associated with this key in the new communicator; one particular action that a copy callback may take is to delete the attribute from the new datatype. Returns in *newtype* a new datatype with exactly the same properties as *oldtype* and any copied cached information, see Section 6.7.4 on page 282. The new datatype has identical upper bound and lower bound and yields the same net result when fully decoded with the functions in Section 4.1.13. The *newtype* has the same committed state as the old *oldtype*.

4.1.11 Use of General Datatypes in Communication

Handles to derived datatypes can be passed to a communication call wherever a datatype argument is required. A call of the form MPI_SEND(buf, count, datatype, ...), where count > 1, is interpreted as if the call was passed a new datatype which is the concatenation of count copies of datatype. Thus, MPI_SEND(buf, count, datatype, dest, tag, comm) is equivalent to,

```
MPI_TYPE_CONTIGUOUS(count, datatype, newtype)
MPI_TYPE_COMMIT(newtype)
MPI_SEND(buf, 1, newtype, dest, tag, comm).
```

Similar statements apply to all other communication functions that have a count and datatype argument.

Suppose that a send operation MPI_SEND(buf, count, datatype, dest, tag, comm) is executed, where datatype has type map,

$$\{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\},$$

and extent *extent*. (Empty entries of “pseudo-type” MPI_UB and MPI_LB are not listed in the type map, but they affect the value of *extent*.) The send operation sends $n \cdot \text{count}$ entries, where entry $i \cdot n + j$ is at location $addr_{i,j} = \text{buf} + \text{extent} \cdot i + disp_j$ and has type $type_j$, for $i = 0, \dots, \text{count} - 1$ and $j = 0, \dots, n - 1$. These entries need not be contiguous, nor distinct; their order can be arbitrary.

The variable stored at address $addr_{i,j}$ in the calling program should be of a type that matches $type_j$, where type matching is defined as in Section 3.3.1. The message sent contains $n \cdot \text{count}$ entries, where entry $i \cdot n + j$ has type $type_j$.

Similarly, suppose that a receive operation MPI_RECV(buf, count, datatype, source, tag, comm, status) is executed, where datatype has type map,

$$\{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\},$$

with extent *extent*. (Again, empty entries of “pseudo-type” MPI_UB and MPI_LB are not listed in the type map, but they affect the value of *extent*.) This receive operation receives $n \cdot \text{count}$ entries, where entry $i \cdot n + j$ is at location $\text{buf} + \text{extent} \cdot i + disp_j$ and has type $type_j$. If the incoming message consists of k elements, then we must have $k \leq n \cdot \text{count}$; the $i \cdot n + j$ -th element of the message should have a type that matches $type_j$.

Type matching is defined according to the type signature of the corresponding datatypes, that is, the sequence of basic type components. Type matching does not depend on some aspects of the datatype definition, such as the displacements (layout in memory) or the intermediate types used.

If `comm` is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument `root`, which is the rank of the root in group A. The root passes the value `MPI_ROOT` in `root`. All other processes in group A pass the value `MPI_PROC_NULL` in `root`. Data is broadcast from the root to all processes in group B. The buffer arguments of the processes in group B must be consistent with the buffer argument of the root.

5.4.1 Example using `MPI_BCAST`

The examples in this section use intracommunicators.

Example 5.1

Broadcast 100 ints from process 0 to every process in the group.

```
MPI_Comm comm;
int array[100];
int root=0;
...
MPI_Bcast(array, 100, MPI_INT, root, comm);
```

As in many of our example code fragments, we assume that some of the variables (such as `comm` in the above) have been assigned appropriate values.

5.5 Gather

`MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)`

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice, significant only at root)
IN	recvcount	number of elements for any single receive (non-negative integer, significant only at root)
IN	recvtype	data type of recv buffer elements (significant only at root) (handle)
IN	root	rank of receiving process (integer)
IN	comm	communicator (handle)

```
int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
              void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
              MPI_Comm comm)
```

ticket229.1.
ticket-248T.

```

1 MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
2           root, comm, ierror) BIND(C)
3     TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
4     TYPE(*), DIMENSION(..) :: recvbuf
5     INTEGER, INTENT(IN) :: sendcount, recvcount, root
6     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
7     TYPE(MPI_Comm), INTENT(IN) :: comm
8     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
10 MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
11            ROOT, COMM, IERROR)
12     <type> SENDBUF(*), RECVBUF(*)
13     INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
14
15 {void MPI::Comm::Gather(const void* sendbuf, int sendcount, const
16                        MPI::Datatype& sendtype, void* recvbuf, int recvcount,
17                        const MPI::Datatype& recvtype, int root) const = 0(binding
18                        deprecated, see Section 15.2) }

```

If `comm` is an intracommunicator, each process (root process included) sends the contents of its send buffer to the root process. The root process receives the messages and stores them in rank order. The outcome is *as if* each of the `n` processes in the group (including the root process) had executed a call to

```
MPI_Send(sendbuf, sendcount, sendtype, root, ...),
```

and the root had executed `n` calls to

```
MPI_Recv(recvbuf + i · recvcount · extent(recvtype), recvcount, recvtype, i, ...),
```

where `extent(recvtype)` is the type extent obtained from a call to `MPI_Type_get_extent()`.

An alternative description is that the `n` messages sent by the processes in the group are concatenated in rank order, and the resulting message is received by the root as if by a call to `MPI_RECV(recvbuf, recvcount·n, recvtype, ...)`.

The receive buffer is ignored for all non-root processes.

General, derived datatypes are allowed for both `sendtype` and `recvtype`. The type signature of `sendcount`, `sendtype` on each process must be equal to the type signature of `recvcount`, `recvtype` at the root. This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed.

All arguments to the function are significant on process `root`, while on other processes, only arguments `sendbuf`, `sendcount`, `sendtype`, `root`, and `comm` are significant. The arguments `root` and `comm` must have identical values on all processes.

The specification of counts and types should not cause any location on the root to be written more than once. Such a call is erroneous.

Note that the `recvcount` argument at the root indicates the number of items it receives from *each* process, not the total number of items it receives.

The “in place” option for intracommunicators is specified by passing `MPI_IN_PLACE` as the value of `sendbuf` at the root. In such a case, `sendcount` and `sendtype` are ignored, and the contribution of the root to the gathered vector is assumed to be already in the correct place in the receive buffer.

```

    */
    rbuf = (int *)malloc(gsize*(displs[gsize-1]+rcounts[gsize-1])
                        *sizeof(int));
    /* Create datatype for one int, with extent of entire row
    */
    disp[0] = 0;      disp[1] = 150*sizeof(int);
    type[0] = MPI_INT; type[1] = MPI_UB;
    blocklen[0] = 1;  blocklen[1] = 1;
    MPI_Type_create_struct( 2, blocklen, disp, type, &stype );
    MPI_Type_commit(&stype);
    sptr = &sendarray[0][myrank];
    MPI_Gatherv(sptr, num, stype, rbuf, rcounts, displs, MPI_INT,
                root, comm);

```

5.6 Scatter

```

MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

```

IN	sendbuf	address of send buffer (choice, significant only at root)
IN	sendcount	number of elements sent to each process (non-negative integer, significant only at root)
IN	sendtype	data type of send buffer elements (significant only at root) (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements in receive buffer (non-negative integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	root	rank of sending process (integer)
IN	comm	communicator (handle)

```

int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype,
                void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
                MPI_Comm comm)
MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
            root, comm, ierror) BIND(C)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  sendbuf
    TYPE(*), DIMENSION(..)  ::  recvbuf
    INTEGER, INTENT(IN) ::  sendcount, recvcount, root
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
            ROOT, COMM, IERROR)

```

ticket229.1.
ticket-248T.

`MPI_SCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm)`

IN	sendbuf	address of send buffer (choice, significant only at root)
IN	sendcounts	non-negative integer array (of length group size) specifying the number of elements to send to each processor
IN	displs	integer array (of length group size). Entry <i>i</i> specifies the displacement (relative to <code>sendbuf</code>) from which to take the outgoing data to process <i>i</i>
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements in receive buffer (non-negative integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	root	rank of sending process (integer)
IN	comm	communicator (handle)

```
int MPI_Scatterv(void* sendbuf, int sendcounts[], int displs[],
                MPI_Datatype sendtype, void* recvbuf, int recvcount,
                MPI_Datatype recvtype, int root, MPI_Comm comm)
```

```
MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
             recvtype, root, comm, ierror) BIND(C)
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
    TYPE(*), DIMENSION(..) :: recvbuf
    INTEGER, INTENT(IN) :: sendcounts(*), displs(*), recvcount, root
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, REVCOUNT,
             RECVTYPE, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, REVCOUNT, RECVTYPE, ROOT,
    COMM, IERROR
```

```
{void MPI::Comm::Scatterv(const void* sendbuf, const int sendcounts[],
                        const int displs[], const MPI::Datatype& sendtype,
                        void* recvbuf, int recvcount, const MPI::Datatype& recvtype,
                        int root) const = 0(binding deprecated, see Section 15.2) }
```

`MPI_SCATTERV` is the inverse operation to `MPI_GATHERV`.

`MPI_SCATTERV` extends the functionality of `MPI_SCATTER` by allowing a varying count of data to be sent to each process, since `sendcounts` is now an array. It also allows more flexibility as to where the data is taken from on the root, by providing an additional argument, `displs`.



Figure 5.11: The root scatters blocks of $100-i$ ints into column i of a 100×150 C array. At the sending side, the blocks are `stride[i]` ints apart.

5.7 Gather-to-all

`MPI_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)`

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements received from any process (non-negative integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)

```
int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
                 void* recvbuf, int recvcount, MPI_Datatype recvtype,
                 MPI_Comm comm)
```

```
MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
              comm, ierror) BIND(C)
```

```
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
```

```
TYPE(*), DIMENSION(..) :: recvbuf
```

```
INTEGER, INTENT(IN) :: sendcount, recvcount
```

```
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
```

```
TYPE(MPI_Comm), INTENT(IN) :: comm
```

```
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, REVCOUNT, RECVTYPE,
              COMM, IERROR)
```

```
<type> SENDBUF(*), RECVBUF(*)
```

```
INTEGER SENDCOUNT, SENDTYPE, REVCOUNT, RECVTYPE, COMM, IERROR
```

`MPI_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcunts, displs, recvtype, comm)`

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcunts	non-negative integer array (of length group size) containing the number of elements that are received from each process
IN	displs	integer array (of length group size). Entry <i>i</i> specifies the displacement (relative to <code>recvbuf</code>) at which to place the incoming data from process <i>i</i>
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)

```
int MPI_Allgatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype,
                  void* recvbuf, int recvcunts[], int displs[],
                  MPI_Datatype recvtype, MPI_Comm comm)
```

```
MPI_Allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcunts, displs,
               recvtype, comm, ierror) BIND(C)
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
    TYPE(*), DIMENSION(..) :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcunts(*), displs(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
               RECVTYPE, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, REVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
    IERROR
```

```
{void MPI::Comm::Allgatherv(const void* sendbuf, int sendcount, const
    MPI::Datatype& sendtype, void* recvbuf,
    const int recvcunts[], const int displs[],
    const MPI::Datatype& recvtype) const = 0(binding deprecated, see
    Section 15.2) }
```

`MPI_ALLGATHERV` can be thought of as `MPI_GATHERV`, but where all processes receive the result, instead of just the root. The block of data sent from the *j*-th process is received by every process and placed in the *j*-th block of the buffer `recvbuf`. These blocks need not all be the same size.

The type signature associated with `sendcount`, `sendtype`, at process *j* must be equal to the type signature associated with `recvcunts[j]`, `recvtype` at any other process.

5.8 All-to-All Scatter/Gather

`MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)`

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements sent to each process (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements received from any process (non-negative integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)

```
int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype,
                void* recvbuf, int recvcount, MPI_Datatype recvtype,
                MPI_Comm comm)
```

```
MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
             comm, ierror) BIND(C)
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
    TYPE(*), DIMENSION(..) :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
             COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
```

```
{void MPI::Comm::Alltoall(const void* sendbuf, int sendcount, const
    MPI::Datatype& sendtype, void* recvbuf, int recvcount,
    const MPI::Datatype& recvtype) const = 0 (binding deprecated, see
    Section 15.2) }
```

`MPI_ALLTOALL` is an extension of `MPI_ALLGATHER` to the case where each process sends distinct data to each of the receivers. The *j*-th block sent from process *i* is received by process *j* and is placed in the *i*-th block of `recvbuf`.

The type signature associated with `sendcount`, `sendtype`, at a process must be equal to the type signature associated with `recvcount`, `recvtype` at any other process. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of processes. As usual, however, the type maps may be different.

If `comm` is an intracommunicator, the outcome is as if each process executed a send to each process (itself included) with a call to,

```
MPI_Send(sendbuf + i · sendcount · extent(sendtype), sendcount, sendtype, i, ...),
```

MPI_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcoun-
ts, rdispls, recv-
type, comm)

	IN	sendbuf	starting address of send buffer (choice)
	IN	sendcounts	non-negative integer array (of length group size) spec- ifying the number of elements to send to each processor
ticket109.	IN	sdispls	integer array (of length group size). Entry j specifies the displacement (relative to sendbuf) from which to take the outgoing data destined for process j
	IN	sendtype	data type of send buffer elements (handle)
	OUT	recvbuf	address of receive buffer (choice)
	IN	recvcoun- ts	non-negative integer array (of length group size) spec- ifying the number of elements that can be received from each processor
ticket109.	IN	rdispls	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i
	IN	recvtype	data type of receive buffer elements (handle)
	IN	comm	communicator (handle)

```
int MPI_Alltoallv(void* sendbuf, int sendcounts[], int sdispls[],  
MPI_Datatype sendtype, void* recvbuf, int recvcoun-  
ts[], int rdispls[], MPI_Datatype recvtype, MPI_Comm comm)
```

```
MPI_Alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcoun-  
ts, rdispls, recvtype, comm, ierror) BIND(C)  
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf  
TYPE(*), DIMENSION(..) :: recvbuf  
INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcoun-  
ts(*), rdispls(*)  
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype  
TYPE(MPI_Comm), INTENT(IN) :: comm  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS,  
RDISPLS, RECVTYPE, COMM, IERROR)  
<type> SENDBUF(*), RECVBUF(*)  
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, REVCOUNTS(*), RDISPLS(*),  
RECVTYPE, COMM, IERROR
```

```
{void MPI::Comm::Alltoallv(const void* sendbuf, const int sendcounts[],  
const int sdispls[], const MPI::Datatype& sendtype,  
void* recvbuf, const int recvcoun-  
ts[], const int rdispls[],  
const MPI::Datatype& recvtype) const = 0(binding deprecated, see  
Section 15.2) }
```

MPI_ALLTOALLV adds flexibility to MPI_ALLTOALL in that the location of data for
the send is specified by sdispls and the location of the placement of the data on the receive

Rationale. The definition of MPI_MINLOC and MPI_MAXLOC given here has the advantage that it does not require any special-case handling of these two operations: they are handled like any other reduce operation. A programmer can provide his or her own definition of MPI_MAXLOC and MPI_MINLOC, if so desired. The disadvantage is that values and indices have to be first interleaved, and that indices and values have to be coerced to the same type, in Fortran. (*End of rationale.*)

5.9.5 User-Defined Reduction Operations

MPI_OP_CREATE(*user_fn*, *commute*, *op*)

IN	[ticket252-W.] <i>user_fn</i>	user defined function (function)
IN	<i>commute</i>	true if commutative; false otherwise.
OUT	<i>op</i>	operation (handle)

int MPI_Op_create(MPI_User_function* *user_fn*, int *commute*, MPI_Op* *op*)

```

MPI_Op_create(user_fn, commute, op, ierror) BIND(C)
  PROCEDURE(MPI_User_function) :: user_fn
  LOGICAL, INTENT(IN) :: commute
  TYPE(MPI_Op), INTENT(OUT) :: op
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

MPI_OP_CREATE(*USER_FN*, *COMMUTE*, *OP*, *IERROR*)

```

EXTERNAL USER_FN
LOGICAL COMMUTE
INTEGER OP, IERROR

```

```

{void MPI::Op::Init(MPI::User_function* user_fn, bool commute) (binding
  deprecated, see Section 15.2) }

```

MPI_OP_CREATE binds a user-defined reduction operation to an *op* handle that can subsequently be used in MPI_REDUCE, MPI_ALLREDUCE, MPI_REDUCE_SCATTER, MPI_SCAN, and MPI_EXSCAN. The user-defined operation is assumed to be associative. If *commute* = true, then the operation should be both commutative and associative. If *commute* = false, then the order of operands is fixed and is defined to be in ascending, process rank order, beginning with process zero. The order of evaluation can be changed, taking advantage of the associativity of the operation. If *commute* = true then the order of evaluation can be changed, taking advantage of commutativity and associativity.

The argument *user_fn* is the user-defined function, which must have the following four arguments: *invec*, *inoutvec*, *len* and *datatype*.

The ISO C prototype for the function is the following.

```

typedef void MPI_User_function(void* invec, void* inoutvec, int* len,
  MPI_Datatype* datatype);

```

The Fortran declarations of the user-defined function *user_fn* appear below.

```

ABSTRACT INTERFACE
  SUBROUTINE MPI_User_function(invec, inoutvec, len, datatype) BIND(C)

```

```

USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
TYPE(C_PTR), VALUE :: invec, inoutvec
INTEGER :: len
TYPE(MPI_Datatype) :: datatype

SUBROUTINE USER_FUNCTION(INVEC, INOUTVEC, LEN, DATATYPE)
  <type> INVEC(LEN), INOUTVEC(LEN)
  INTEGER LEN, DATATYPE

```

The C++ declaration of the user-defined function appears below.

```

{typedef void MPI::User_function(const void* invec, void* inoutvec, int
    len, const Datatype& datatype); (binding deprecated, see
    Section 15.2)}

```

The `datatype` argument is a handle to the data type that was passed into the call to `MPI_REDUCE`. The user reduce function should be written such that the following holds: Let $u[0], \dots, u[\text{len}-1]$ be the `len` elements in the communication buffer described by the arguments `invec`, `len` and `datatype` when the function is invoked; let $v[0], \dots, v[\text{len}-1]$ be `len` elements in the communication buffer described by the arguments `inoutvec`, `len` and `datatype` when the function is invoked; let $w[0], \dots, w[\text{len}-1]$ be `len` elements in the communication buffer described by the arguments `inoutvec`, `len` and `datatype` when the function returns; then $w[i] = u[i] \circ v[i]$, for $i=0, \dots, \text{len}-1$, where \circ is the reduce operation that the function computes.

Informally, we can think of `invec` and `inoutvec` as arrays of `len` elements that `user_fn` is combining. The result of the reduction over-writes values in `inoutvec`, hence the name. Each invocation of the function results in the pointwise evaluation of the reduce operator on `len` elements: i.e., the function returns in `inoutvec[i]` the value `invec[i] \circ inoutvec[i]`, for $i = 0, \dots, \text{count} - 1$, where \circ is the combining operation computed by the function.

Rationale. The `len` argument allows `MPI_REDUCE` to avoid calling the function for each element in the input buffer. Rather, the system can choose to apply the function to chunks of input. In C, it is passed in as a reference for reasons of compatibility with Fortran.

By internally comparing the value of the `datatype` argument to known, global handles, it is possible to overload the use of a single user-defined function for several, different data types. (*End of rationale.*)

General datatypes may be passed to the user function. However, use of datatypes that are not contiguous is likely to lead to inefficiencies.

No MPI communication function may be called inside the user function. `MPI_ABORT` may be called inside the function in case of an error.

Advice to users. Suppose one defines a library of user-defined reduce functions that are overloaded: the `datatype` argument is used to select the right execution path at each invocation, according to the types of the operands. The user-defined reduce function cannot “decode” the `datatype` argument that it is passed, and cannot identify, by itself, the correspondence between the datatype handles and the datatype they represent. This correspondence was established when the datatypes were created. Before the library is used, a library initialization preamble must be executed. This preamble

```

1      Complex a[100], answer[100];
2      MPI_Op myOp;
3      MPI_Datatype ctype;
4
5      /* explain to MPI how type Complex is defined
6       */
7      MPI_Type_contiguous(2, MPI_DOUBLE, &ctype);
8      MPI_Type_commit(&ctype);
9      /* create the complex-product user-op
10     */
11     MPI_Op_create( myProd, 1, &myOp );
12
13     MPI_Reduce(a, answer, 100, ctype, myOp, root, comm);
14
15     /* At this point, the answer, which consists of 100 Complexes,
16      * resides on process root
17     */
18
19

```

Example 5.21 How to use the `mpi_f08` interface of the Fortran `MPI_User_function`.

```

20
21     subroutine my_user_function( invec, inoutvec, len, type )    bind(c)
22     use, intrinsic :: iso_c_binding, only : c_ptr, c_f_pointer
23     type(c_ptr), value :: invec, inoutvec
24     integer :: len
25     type(MPI_Datatype) :: type
26     real, pointer :: invec_r(:), inoutvec_r(:)
27     if (type%MPI_VAL == MPI_REAL%MPI_VAL) then
28         call c_f_pointer(invec, invec_r, (/ len /) )
29         call c_f_pointer(inoutvec, inoutvec_r, (/ len /) )
30         inoutvec_r = invec_r + inoutvec_r
31     end if
32 end subroutine
33
34

```

5.9.6 All-Reduce

MPI includes a variant of the reduce operations where the result is returned to all processes in a group. MPI requires that all processes from the same group participating in these operations receive identical results.

```
MPI_REDUCE_LOCAL(INBUF, INOUTBUF, COUNT, DATATYPE, OP, IERROR)
```

```
<type> INBUF(*), INOUTBUF(*)
```

```
INTEGER COUNT, DATATYPE, OP, IERROR
```

```
{void MPI::Op::Reduce_local(const void* inbuf, void* inoutbuf, int count,
    const MPI::Datatype& datatype) const(binding deprecated, see
    Section 15.2) }
```

The function applies the operation given by `op` element-wise to the elements of `inbuf` and `inoutbuf` with the result stored element-wise in `inoutbuf`, as explained for user-defined operations in Section 5.9.5. Both `inbuf` and `inoutbuf` (input as well as result) have the same number of elements given by `count` and the same datatype given by `datatype`. The `MPI_IN_PLACE` option is not allowed.

Reduction operations can be queried for their commutativity.

```
MPI_OP_COMMUTATIVE( op, commute)
```

```
IN      op      operation (handle)
```

```
OUT     commute  true if op is commutative, false otherwise (logical)
```

```
int MPI_Op_commutative(MPI_Op op, int *commute)
```

```
MPI_Op_commutative(op, commute, ierror) BIND(C)
```

```
TYPE(MPI_Op), INTENT(IN) :: op
```

```
LOGICAL, INTENT(OUT) :: commute
```

```
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_OP_COMMUTATIVE(OP, COMMUTE, IERROR)
```

```
LOGICAL COMMUTE
```

```
INTEGER OP, IERROR
```

```
{bool MPI::Op::Is_commutative() const(binding deprecated, see Section 15.2) }
```

5.10 Reduce-Scatter

MPI includes variants of the reduce operations where the result is scattered to all processes in a group on return. One variant scatters equal-sized blocks to all processes, while another variant scatters blocks that may vary in size for each process.

```

MPI_IGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, REVCOUNT, RECVTYPE,
            ROOT, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, REVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,
    IERROR

```

```

{MPI::Request MPI::Comm::Igather(const void* sendbuf, int sendcount, const
    MPI::Datatype& sendtype, void* recvbuf, int recvcount,
    const MPI::Datatype& recvtype, int root) const = 0(binding
    deprecated, see Section 15.2) }

```

This call starts a nonblocking variant of MPI_GATHER (see Section 5.5).

```

MPI_IGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcnts, displs, recvtype, root,
            comm, request)

```

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice, significant only at root)
IN	recvcnts	non-negative integer array (of length group size) containing the number of elements that are received from each process (significant only at root)
IN	displs	integer array (of length group size). Entry <i>i</i> specifies the displacement relative to <i>recvbuf</i> at which to place the incoming data from process <i>i</i> (significant only at root)
IN	recvtype	data type of recv buffer elements (significant only at root) (handle)
IN	root	rank of receiving process (integer)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

```

int MPI_Igatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype,
    void* recvbuf, int recvcnts[], int displs[],
    MPI_Datatype recvtype, int root, MPI_Comm comm,
    MPI_Request* request)

```

```

MPI_Igatherv(sendbuf, sendcount, sendtype, recvbuf, recvcnts, displs,
    recvtype, root, comm, request, ierror) BIND(C)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, root
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcnts(*), displs(*)

```

ticket0.
ticket0.
ticket0.
ticket0.
ticket229.1.
ticket-248T.


```

    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_ISCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
             ROOT, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,
    IERROR

{MPI::Request MPI::Comm::Iscatter(const void* sendbuf, int sendcount, const
    MPI::Datatype& sendtype, void* recvbuf, int recvcount,
    const MPI::Datatype& recvtype, int root) const = 0(binding
    deprecated, see Section 15.2) }

```

This call starts a nonblocking variant of MPI_SCATTER (see Section 5.6).

```

MPI_ISCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root,
              comm, request)

```

IN	sendbuf	address of send buffer (choice, significant only at root)
IN	sendcounts	non-negative integer array (of length group size) specifying the number of elements to send to each processor
IN	displs	integer array (of length group size). Entry <i>i</i> specifies the displacement (relative to <i>sendbuf</i>) from which to take the outgoing data to process <i>i</i>
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements in receive buffer (non-negative integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	root	rank of sending process (integer)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

```

int MPI_Iscatterv(void* sendbuf, int sendcounts[], int displs[],
    MPI_Datatype sendtype, void* recvbuf, int recvcount,
    MPI_Datatype recvtype, int root, MPI_Comm comm,
    MPI_Request* request)

```

```

MPI_Iscatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
              recvtype, root, comm, request, ierror) BIND(C)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), displs(*)

```

```

    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_IALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
               COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR

{MPI::Request MPI::Comm::Iallgather(const void* sendbuf, int sendcount,
    const MPI::Datatype& sendtype, void* recvbuf, int recvcount,
    const MPI::Datatype& recvtype) const = 0 (binding deprecated, see
    Section 15.2) }
```

This call starts a nonblocking variant of MPI_ALLGATHER (see Section 5.7).

```

MPI_IALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcnts, displs, recvtype, comm,
                request)
```

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcnts	non-negative integer array (of length group size) containing the number of elements that are received from each process
IN	displs	integer array (of length group size). Entry <i>i</i> specifies the displacement (relative to <i>recvbuf</i>) at which to place the incoming data from process <i>i</i>
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

```

int MPI_Iallgatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype,
    void* recvbuf, int recvcnts[], int displs[],
    MPI_Datatype recvtype, MPI_Comm comm, MPI_Request* request)
```

```

MPI_Iallgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcnts, displs,
    recvtype, comm, request, ierror) BIND(C)
```

```

    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
```

```

    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
```

```

    INTEGER, INTENT(IN) :: sendcount
```

```

    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcnts(*), displs(*)
```

```

    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
```

```

    TYPE(MPI_Comm), INTENT(IN) :: comm
```

```

    TYPE(MPI_Request), INTENT(OUT) :: request
```

ticket0.
ticket0.
ticket0.
ticket0.
ticket229.1.
ticket-248T.

```

MPI_IALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, REVCOUNT, RECVTYPE,
              COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, REVCOUNT, RECVTYPE, COMM, REQUEST, IERROR

{MPI::Request MPI::Comm::Ialltoall(const void* sendbuf, int sendcount,
    const MPI::Datatype& sendtype, void* recvbuf, int recvcount,
    const MPI::Datatype& recvtype) const = 0(binding deprecated, see
    Section 15.2) }

```

This call starts a nonblocking variant of MPI_ALLTOALL (see Section 5.8).

```

MPI_IALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcoun-
    ts, rdispls, recv-
    type, comm, request)

```

IN	sendbuf	starting address of send buffer (choice)
IN	sendcounts	non-negative integer array (of length group size) specifying the number of elements to send to each processor
IN	sdispls	integer array (of length group size). Entry j specifies the displacement (relative to sendbuf) from which to take the outgoing data destined for process j
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcoun-	non-negative integer array (of length group size) specifying the number of elements that can be received from each processor
IN	rdispls	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

```

int MPI_Ialltoallv(void* sendbuf, int sendcounts[], int sdispls[],
    MPI_Datatype sendtype, void* recvbuf, int recvcoun-
    ts[], MPI_Datatype recvtype, MPI_Comm comm,
    MPI_Request* request)

```

```

MPI_Ialltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcoun-
    ts, rdispls, recvtype, comm, request, ierror) BIND(C)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
    recvcoun-
    ts(*), rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm

```

`MPI_IALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm, request)`

IN	sendbuf	starting address of send buffer (choice)
IN	sendcounts	integer array (of length group size) specifying the number of elements to send to each processor (array of non-negative integers)
IN	sdispls	integer array (of length group size). Entry <i>j</i> specifies the displacement in bytes (relative to <code>sendbuf</code>) from which to take the outgoing data destined for process <i>j</i> (array of integers)
IN	sendtypes	array of datatypes (of length group size). Entry <i>j</i> specifies the type of data to send to process <i>j</i> (array of handles)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcounts	integer array (of length group size) specifying the number of elements that can be received from each processor (array of non-negative integers)
IN	rdispls	integer array (of length group size). Entry <i>i</i> specifies the displacement in bytes (relative to <code>recvbuf</code>) at which to place the incoming data from process <i>i</i> (array of integers)
IN	recvtypes	array of datatypes (of length group size). Entry <i>i</i> specifies the type of data received from process <i>i</i> (array of handles)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

```
int MPI_Ialltoallw(void* sendbuf, int sendcounts[], int sdispls[],
    MPI_Datatype sendtypes[], void* recvbuf, int recvcounts[],
    int rdispls[], MPI_Datatype recvtypes[], MPI_Comm comm,
    MPI_Request* request)
```

```
MPI_Ialltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
    recvcounts, rdispls, recvtypes, comm, request, ierror) BIND(C)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
    recvcounts(*), rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
    recvtypes(*)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

ticket229.1.
ticket-248T.

```
{MPI::Request MPI::Comm::Ireduce_scatter_block(const void* sendbuf,
        void* recvbuf, int recvcnt, const MPI::Datatype& datatype,
        const MPI::Op& op) const = 0(binding deprecated, see Section 15.2) }
```

This call starts a nonblocking variant of MPI_REDUCE_SCATTER_BLOCK (see Section 5.10.1).

5.12.10 Nonblocking Reduce-Scatter

MPI_IREDUCE_SCATTER(sendbuf, recvbuf, recvcnts, datatype, op, comm, request)

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcnts	non-negative integer array specifying the number of elements in result distributed to each process. Array must be identical on all calling processes.
IN	datatype	data type of elements of input buffer (handle)
IN	op	operation (handle)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

```
int MPI_Ireduce_scatter(void* sendbuf, void* recvbuf, int recvcnts[],
        MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
        MPI_Request* request)
```

```
MPI_Ireduce_scatter(sendbuf, recvbuf, recvcnts, datatype, op, comm,
        request, ierror) BIND(C)
```

```
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcnts(*)
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Op), INTENT(IN) :: op
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_IREDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
        REQUEST, IERROR)
```

```
<type> SENDBUF(*), RECVBUF(*)
INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, REQUEST, IERROR
```

```
{MPI::Request MPI::Comm::Ireduce_scatter(const void* sendbuf,
        void* recvbuf, int recvcnts[],
        const MPI::Datatype& datatype, const MPI::Op& op)
        const = 0(binding deprecated, see Section 15.2) }
```

This call starts a nonblocking variant of MPI_REDUCE_SCATTER (see Section 5.10.2).

```

1 MPI_Group_difference(group1, group2, newgroup, ierror) BIND(C)
2     TYPE(MPI_Group), INTENT(IN) :: group1, group2
3     TYPE(MPI_Group), INTENT(OUT) :: newgroup
4     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
6 MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR)
7     INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
8
9 {static MPI::Group MPI::Group::Difference(const MPI::Group& group1,
10     const MPI::Group& group2) (binding deprecated, see Section 15.2) }

```

The set-like operations are defined as follows:

union All elements of the first group (group1), followed by all elements of second group (group2) not in first.

intersect all elements of the first group that are also in the second group, ordered as in first group.

difference all elements of the first group that are not in the second group, ordered as in the first group.

Note that for these operations the order of processes in the output group is determined primarily by order in the first group (if possible) and then, if necessary, by order in the second group. Neither union nor intersection are commutative, but both are associative.

The new group can be empty, that is, equal to MPI_GROUP_EMPTY.

```

26 MPI_GROUP_INCL(group, n, ranks, newgroup)
27
28     IN      group      group (handle)
29     IN      n          number of elements in array ranks (and size of
30                      newgroup) (integer)
31
32     IN      ranks      ranks of processes in group to appear in
33                      newgroup (array of integers)
34
35     OUT     newgroup    new group derived from above, in the order defined by
36                      ranks (handle)

```

```

37 int MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)

```

```

38 MPI_Group_incl(group, n, ranks, newgroup, ierror) BIND(C)
39     TYPE(MPI_Group), INTENT(IN) :: group
40     INTEGER, INTENT(IN) :: n, ranks(n)
41     TYPE(MPI_Group), INTENT(OUT) :: newgroup
42     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
44 MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR)
45     INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR
46
47 {MPI::Group MPI::Group::Incl(int n, const int ranks[]) const (binding
48     deprecated, see Section 15.2) }

```

indicates the rank of the process that calls it in the range from 0...size-1, where size is the return value of MPI_COMM_SIZE. (*End of advice to users.*)

MPI_COMM_RANK(comm, rank)

IN	comm	communicator (handle)
OUT	rank	rank of the calling process in group of comm (integer)

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

```
MPI_Comm_rank(comm, rank, ierror) BIND(C)
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, INTENT(OUT) :: rank
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_COMM_RANK(COMM, RANK, IERROR)
  INTEGER COMM, RANK, IERROR
```

```
{int MPI::Comm::Get_rank() const(binding deprecated, see Section 15.2) }
```

Rationale. This function is equivalent to accessing the communicator's group with MPI_COMM_GROUP (see above), computing the rank using MPI_GROUP_RANK, and then freeing the temporary group via MPI_GROUP_FREE. However, this function is so commonly used, that this shortcut was introduced. (*End of rationale.*)

Advice to users. This function gives the rank of the process in the particular communicator's group. It is useful, as noted above, in conjunction with MPI_COMM_SIZE.

Many programs will be written with the master-slave model, where one process (such as the rank-zero process) will play a supervisory role, and the other processes will serve as compute nodes. In this framework, the two preceding calls are useful for determining the roles of the various processes of a communicator. (*End of advice to users.*)

MPI_COMM_COMPARE(comm1, comm2, result)

IN	comm1	first communicator (handle)
IN	comm2	second communicator (handle)
OUT	result	result (integer)

```
int MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int *result)
```

```
MPI_Comm_compare(comm1, comm2, result, ierror) BIND(C)
  TYPE(MPI_Comm), INTENT(IN) :: comm1, comm2
  INTEGER, INTENT(OUT) :: result
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```


no pending communication on `peer_comm` that could interfere with this communication.

Advice to users. We recommend using a dedicated peer communicator, such as a duplicate of `MPI_COMM_WORLD`, to avoid trouble with peer communicators. (*End of advice to users.*)

`MPI_INTERCOMM_MERGE(intercomm, high, newintracomm)`

IN	intercomm	Inter-Communicator (handle)
IN	high	(logical)
OUT	newintracomm	new intra-communicator (handle)

```
int MPI_Intercomm_merge(MPI_Comm intercomm, int high,
                        MPI_Comm *newintracomm)
```

```
MPI_Intercomm_merge(intercomm, high, newintracomm, ierror) BIND(C)
    TYPE(MPI_Comm), INTENT(IN) :: intercomm
    LOGICAL, INTENT(IN) :: high
    TYPE(MPI_Comm), INTENT(OUT) :: newintracomm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, NEWINTRACOMM, IERROR)
    INTEGER INTERCOMM, NEWINTRACOMM, IERROR
    LOGICAL HIGH
```

```
{MPI::Intracomm MPI::Intercomm::Merge(bool high) const(binding deprecated, see
    Section 15.2) }
```

This function creates an intra-communicator from the union of the two groups that are associated with `intercomm`. All processes should provide the same `high` value within each of the two groups. If processes in one group provided the value `high = false` and processes in the other group provided the value `high = true` then the union orders the “low” group before the “high” group. If all processes provided the same `high` argument then the order of the union is arbitrary. This call is blocking and collective within the union of the two groups.

The error handler on the new intercommunicator in each process is inherited from the communicator that contributes the local group. Note that this can result in different processes in the same communicator having different error handlers.

Advice to implementors. The implementation of `MPI_INTERCOMM_MERGE`, `MPI_COMM_FREE` and `MPI_COMM_DUP` are similar to the implementation of `MPI_INTERCOMM_CREATE`, except that contexts private to the input inter-communicator are used for communication between group leaders rather than contexts inside a bridge communicator. (*End of advice to implementors.*)

- obtain a key value (used to identify an attribute); the user specifies “callback” functions by which MPI informs the application when the communicator is destroyed or copied.
- store and retrieve the value of an attribute;

Advice to implementors. Caching and callback functions are only called synchronously, in response to explicit application requests. This avoids problems that result from repeated crossings between user and system space. (This synchronous calling rule is a general property of MPI.)

The choice of key values is under control of MPI. This allows MPI to optimize its implementation of attribute sets. It also avoids conflict between independent modules caching information on the same communicators.

A much smaller interface, consisting of just a callback facility, would allow the entire caching facility to be implemented by portable code. However, with the minimal callback interface, some form of table searching is implied by the need to handle arbitrary communicators. In contrast, the more complete interface defined here permits rapid access to attributes through the use of pointers in communicators (to find the attribute table) and cleverly chosen key values (to retrieve individual attributes). In light of the efficiency “hit” inherent in the minimal interface, the more complete interface defined here is seen to be superior. (*End of advice to implementors.*)

MPI provides the following services related to caching. They are all process local.

6.7.2 Communicators

Functions for caching on communicators are:

```
MPI_COMM_CREATE_KEYVAL(comm_copy_attr_fn, comm_delete_attr_fn, comm_keyval,
                        extra_state)
```

IN	comm_copy_attr_fn	copy callback function for comm_keyval (function)
IN	comm_delete_attr_fn	delete callback function for comm_keyval (function)
OUT	comm_keyval	key value for future access (integer)
IN	extra_state	extra state for callback functions

```
int MPI_Comm_create_keyval(MPI_Comm_copy_attr_function *comm_copy_attr_fn,
                          MPI_Comm_delete_attr_function *comm_delete_attr_fn,
                          int *comm_keyval, void *extra_state)
```

```
MPI_Comm_create_keyval(comm_copy_attr_fn, comm_delete_attr_fn, comm_keyval,
                      extra_state, ierror) BIND(C)
PROCEDURE(MPI_Comm_copy_attr_function) :: comm_copy_attr_fn
PROCEDURE(MPI_Comm_delete_attr_function) :: comm_delete_attr_fn
INTEGER, INTENT(OUT) :: comm_keyval
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```

MPI_COMM_CREATE_KEYVAL(COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN, COMM_KEYVAL,
                        EXTRA_STATE, IERROR)
EXTERNAL COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN
INTEGER COMM_KEYVAL, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE

{static int MPI::Comm::Create_keyval(MPI::Comm::Copy_attr_function*
                                     comm_copy_attr_fn,
                                     MPI::Comm::Delete_attr_function* comm_delete_attr_fn,
                                     void* extra_state) (binding deprecated, see Section 15.2) }
```

Generates a new attribute key. Keys are locally unique in a process, and opaque to user, though they are explicitly stored in integers. Once allocated, the key value can be used to associate attributes and access them on any locally defined communicator.

This function replaces `MPI_KEYVAL_CREATE`, whose use is deprecated. The C binding is identical. The Fortran binding differs in that `extra_state` is an address-sized integer. Also, the copy and delete callback functions have Fortran bindings that are consistent with address-sized attributes.

The C callback functions are:

```

typedef int MPI_Comm_copy_attr_function(MPI_Comm oldcomm, int comm_keyval,
                                         void *extra_state, void *attribute_val_in,
                                         void *attribute_val_out, int *flag);
```

and

```

typedef int MPI_Comm_delete_attr_function(MPI_Comm comm, int comm_keyval,
                                         void *attribute_val, void *extra_state);
```

which are the same as the MPI-1.1 calls but with a new name. The old names are deprecated.

With the `mpi_f08` module, the Fortran callback functions are:

ABSTRACT INTERFACE

```

SUBROUTINE MPI_Comm_copy_attr_function(oldcomm, comm_keyval, extra_state,
attribute_val_in, attribute_val_out, flag, ierror) BIND(C)
  TYPE(MPI_Comm) :: oldcomm
  INTEGER :: comm_keyval, ierror
  INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
attribute_val_out
  LOGICAL :: flag
```

and

ABSTRACT INTERFACE

```

SUBROUTINE MPI_Comm_delete_attr_function(comm, comm_keyval,
attribute_val, extra_state, ierror) BIND(C)
  TYPE(MPI_Comm) :: comm
  INTEGER :: comm_keyval, ierror
  INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
```

With the `mpi` module and `mpif.h`, the Fortran callback functions are:

```

SUBROUTINE COMM_COPY_ATTR_FUNCTION(OLDCOMM, COMM_KEYVAL, EXTRA_STATE,
```

```

1      ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
2      INTEGER OLDCOMM, COMM_KEYVAL, IERROR
3      INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
4      ATTRIBUTE_VAL_OUT
5      LOGICAL FLAG

```

and

```

6
7      SUBROUTINE COMM_DELETE_ATTR_FUNCTION(COMM, COMM_KEYVAL, ATTRIBUTE_VAL,
8      EXTRA_STATE, IERROR)
9      INTEGER COMM, COMM_KEYVAL, IERROR
10     INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
11

```

The C++ callbacks are:

```

12
13
14     {typedef int MPI::Comm::Copy_attr_function(const MPI::Comm& oldcomm,
15         int comm_keyval, void* extra_state, void* attribute_val_in,
16         void* attribute_val_out, bool& flag); (binding deprecated, see
17         Section 15.2)}
18

```

and

```

19     {typedef int MPI::Comm::Delete_attr_function(MPI::Comm& comm,
20         int comm_keyval, void* attribute_val, void* extra_state);
21         (binding deprecated, see Section 15.2)}
22

```

The `comm_copy_attr_fn` function is invoked when a communicator is duplicated by `MPI_COMM_DUP`. `comm_copy_attr_fn` should be of type `MPI_Comm_copy_attr_function`. The copy callback function is invoked for each key value in `oldcomm` in arbitrary order. Each call to the copy callback is made with a key value and its corresponding attribute. If it returns `flag = 0`, then the attribute is deleted in the duplicated communicator. Otherwise (`flag = 1`), the new attribute value is set to the value returned in `attribute_val_out`. The function returns `MPI_SUCCESS` on success and an error code on failure (in which case `MPI_COMM_DUP` will fail).

The argument `comm_copy_attr_fn` may be specified as `MPI_COMM_NULL_COPY_FN` or `MPI_COMM_DUP_FN` from either C, C++, or Fortran. `MPI_COMM_NULL_COPY_FN` is a function that does nothing other than returning `flag = 0` and `MPI_SUCCESS`.

`MPI_COMM_DUP_FN` is a simple-minded copy function that sets `flag = 1`, returns the value of `attribute_val_in` in `attribute_val_out`, and returns `MPI_SUCCESS`. These replace the MPI-1 predefined callbacks `MPI_NULL_COPY_FN` and `MPI_DUP_FN`, whose use is deprecated.

Advice to users. Even though both formal arguments `attribute_val_in` and `attribute_val_out` are of type `void *`, their usage differs. The C copy function is passed by MPI in `attribute_val_in` the *value* of the attribute, and in `attribute_val_out` the *address* of the attribute, so as to allow the function to return the (new) attribute value. The use of type `void *` for both is to avoid messy type casts.

A valid copy function is one that completely duplicates the information by making a full duplicate copy of the data structures implied by an attribute; another might just make another reference to that data structure, while using a reference-count mechanism. Other types of attributes might not copy at all (they might be specific to `oldcomm` only). (*End of advice to users.*)

Advice to implementors. A C interface should be assumed for copy and delete functions associated with key values created in C; a Fortran calling interface should be assumed for key values created in Fortran. (*End of advice to implementors.*)

Analogous to `comm_copy_attr_fn` is a callback deletion function, defined as follows. The `comm_delete_attr_fn` function is invoked when a communicator is deleted by `MPI_COMM_FREE` or when a call is made explicitly to `MPI_COMM_DELETE_ATTR`. `comm_delete_attr_fn` should be of type `MPI_Comm_delete_attr_function`.

This function is called by `MPI_COMM_FREE`, `MPI_COMM_DELETE_ATTR`, and `MPI_COMM_SET_ATTR` to do whatever is needed to remove an attribute. The function returns `MPI_SUCCESS` on success and an error code on failure (in which case `MPI_COMM_FREE` will fail).

The argument `comm_delete_attr_fn` may be specified as `MPI_COMM_NULL_DELETE_FN` from either C, C++, or Fortran. `MPI_COMM_NULL_DELETE_FN` is a function that does nothing, other than returning `MPI_SUCCESS`. `MPI_COMM_NULL_DELETE_FN` replaces `MPI_NULL_DELETE_FN`, whose use is deprecated.

If an attribute copy function or attribute delete function returns other than `MPI_SUCCESS`, then the call that caused it to be invoked (for example, `MPI_COMM_FREE`), is erroneous.

The special key value `MPI_KEYVAL_INVALID` is never returned by `MPI_KEYVAL_CREATE`. Therefore, it can be used for static initialization of key values.

Advice to implementors. To be able to use the predefined C functions `MPI_COMM_NULL_COPY_FN` or `MPI_COMM_DUP_FN` as `comm_copy_attr_fn` argument and/or `MPI_COMM_NULL_DELETE_FN` as the `comm_delete_attr_fn` argument in a call to the C++ routine `MPI::Comm::Create_keyval`, this routine may be overloaded with 3 additional routines that accept the C functions as the first, the second, or both input arguments (instead of an argument that matches the C++ prototype). (*End of advice to implementors.*)

Advice to users. If a user wants to write a “wrapper” routine that internally calls `MPI::Comm::Create_keyval` and `comm_copy_attr_fn` and/or `comm_delete_attr_fn` are arguments of this wrapper routine, and if this wrapper routine should be callable with both user-defined C++ copy and delete functions and with the predefined C functions, then the same overloading as described above in the advice to implementors may be necessary. (*End of advice to users.*)

Advice to implementors. The predefined Fortran functions `MPI_COMM_NULL_COPY_FN`, `MPI_COMM_DUP_FN`, and `MPI_COMM_NULL_DELETE_FN` are defined in the `mpi` module (and `mpif.h`) and the `mpi_f08` module with the same name, but with different interfaces. Each function can coexist twice with the same name in the same MPI library, one routine as an implicit interface outside of the `mpi` module, i.e., declared as `EXTERNAL`, and the other routine within `mpi_f08` declared with `CONTAINS`. These routines have different link names, which are also different to the link names used for the routines used in C and C++. (*End of advice to implementors.*)

Advice to users. Callbacks, including the predefined Fortran functions `MPI_COMM_NULL_COPY_FN`, `MPI_COMM_DUP_FN`, and `MPI_COMM_NULL_DELETE_FN` should not be passed from one application routine that uses the `mpi_f08` module to another application routine that uses the `mpi` module or `mpif.h`, and vice versa, see also the advice to users on page 609. (*End of advice to users.*)

MPI_COMM_FREE_KEYVAL(comm_keyval)

INOUT	comm_keyval	key value (integer)
-------	-------------	---------------------

```
int MPI_Comm_free_keyval(int *comm_keyval)
```

```
MPI_Comm_free_keyval(comm_keyval, ierror) BIND(C)
```

```
INTEGER, INTENT(INOUT) :: comm_keyval
```

```
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

MPI_COMM_FREE_KEYVAL(COMM_KEYVAL, IERROR)

```
INTEGER COMM_KEYVAL, IERROR
```

```
{static void MPI::Comm::Free_keyval(int& comm_keyval) (binding deprecated, see  
Section 15.2) }
```

Frees an extant attribute key. This function sets the value of `keyval` to `MPI_KEYVAL_INVALID`. Note that it is not erroneous to free an attribute key that is in use, because the actual free does not transpire until after all references (in other communicators on the process) to the key have been freed. These references need to be explicitly freed by the program, either via calls to `MPI_COMM_DELETE_ATTR` that free one attribute instance, or by calls to `MPI_COMM_FREE` that free all attribute instances associated with the freed communicator.

This call is identical to the MPI-1 call `MPI_KEYVAL_FREE` but is needed to match the new communicator-specific creation function. The use of `MPI_KEYVAL_FREE` is deprecated.

MPI_COMM_SET_ATTR(comm, comm_keyval, attribute_val)

INOUT	comm	communicator from which attribute will be attached (handle)
-------	------	--

IN	comm_keyval	key value (integer)
----	-------------	---------------------

IN	attribute_val	attribute value
----	---------------	-----------------

```
int MPI_Comm_set_attr(MPI_Comm comm, int comm_keyval, void *attribute_val)
```

```
MPI_Comm_set_attr(comm, comm_keyval, attribute_val, ierror) BIND(C)
```

```
TYPE(MPI_Comm), INTENT(IN) :: comm
```

```
INTEGER, INTENT(IN) :: comm_keyval
```

```
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
```

```
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

MPI_COMM_SET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, IERROR)

MPI_WIN_CREATE_KEYVAL(win_copy_attr_fn, win_delete_attr_fn, win_keyval, extra_state)

IN	win_copy_attr_fn	copy callback function for win_keyval (function)
IN	win_delete_attr_fn	delete callback function for win_keyval (function)
OUT	win_keyval	key value for future access (integer)
IN	extra_state	extra state for callback functions

```
int MPI_Win_create_keyval(MPI_Win_copy_attr_function *win_copy_attr_fn,
    MPI_Win_delete_attr_function *win_delete_attr_fn,
    int *win_keyval, void *extra_state)
```

```
MPI_Win_create_keyval(win_copy_attr_fn, win_delete_attr_fn, win_keyval,
    extra_state, ierror) BIND(C)
    PROCEDURE(MPI_Win_copy_attr_function) :: win_copy_attr_fn
    PROCEDURE(MPI_Win_delete_attr_function) :: win_delete_attr_fn
    INTEGER, INTENT(OUT) :: win_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_WIN_CREATE_KEYVAL(WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN, WIN_KEYVAL,
    EXTRA_STATE, IERROR)
    EXTERNAL WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN
    INTEGER WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

```
{static int MPI::Win::Create_keyval(MPI::Win::Copy_attr_function*
    win_copy_attr_fn,
    MPI::Win::Delete_attr_function* win_delete_attr_fn,
    void* extra_state) (binding deprecated, see Section 15.2) }
```

The argument win_copy_attr_fn may be specified as MPI_WIN_NULL_COPY_FN or MPI_WIN_DUP_FN from either C, C++, or Fortran. MPI_WIN_NULL_COPY_FN is a function that does nothing other than returning flag = 0 and MPI_SUCCESS. MPI_WIN_DUP_FN is a simple-minded copy function that sets flag = 1, returns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS.

The argument win_delete_attr_fn may be specified as MPI_WIN_NULL_DELETE_FN from either C, C++, or Fortran. MPI_WIN_NULL_DELETE_FN is a function that does nothing, other than returning MPI_SUCCESS.

The C callback functions are:

```
typedef int MPI_Win_copy_attr_function(MPI_Win oldwin, int win_keyval,
    void *extra_state, void *attribute_val_in,
    void *attribute_val_out, int *flag);
```

and

```
typedef int MPI_Win_delete_attr_function(MPI_Win win, int win_keyval,
    void *attribute_val, void *extra_state);
```

With the mpi_f08 module, the Fortran callback functions are:

ticket-248T.

```

1  ABSTRACT INTERFACE
2
3      SUBROUTINE MPI_Win_copy_attr_function(oldwin, win_keyval, extra_state,
4      attribute_val_in, attribute_val_out, flag, ierror) BIND(C)
5          TYPE(MPI_Win) :: oldwin
6          INTEGER :: win_keyval, ierror
7          INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
8      attribute_val_out
9          LOGICAL :: flag

```

and

```

11 ABSTRACT INTERFACE
12
13     SUBROUTINE MPI_Win_delete_attr_function(win, win_keyval, attribute_val,
14     extra_state, ierror) BIND(C)
15         TYPE(MPI_Win) :: win
16         INTEGER :: win_keyval, ierror
17         INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state

```

With the `mpi` module and `mpif.h`, the Fortran callback functions are:

```

20 SUBROUTINE WIN_COPY_ATTR_FUNCTION(OLDWIN, WIN_KEYVAL, EXTRA_STATE,
21     ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
22     INTEGER OLDWIN, WIN_KEYVAL, IERROR
23     INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
24     ATTRIBUTE_VAL_OUT
25     LOGICAL FLAG

```

and

```

27 SUBROUTINE WIN_DELETE_ATTR_FUNCTION(WIN, WIN_KEYVAL, ATTRIBUTE_VAL,
28     EXTRA_STATE, IERROR)
29     INTEGER WIN, WIN_KEYVAL, IERROR
30     INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE

```

The C++ callbacks are:

```

34 {typedef int MPI::Win::Copy_attr_function(const MPI::Win& oldwin,
35     int win_keyval, void* extra_state, void* attribute_val_in,
36     void* attribute_val_out, bool& flag); (binding deprecated, see
37     Section 15.2)}
```

and

```

39 {typedef int MPI::Win::Delete_attr_function(MPI::Win& win, int win_keyval,
40     void* attribute_val, void* extra_state); (binding deprecated, see
41     Section 15.2)}
```

If an attribute copy function or attribute delete function returns other than `MPI_SUCCESS`, then the call that caused it to be invoked (for example, `MPI_WIN_FREE`), is erroneous.

```

MPI_Type_create_keyval(type_copy_attr_fn, type_delete_attr_fn, type_keyval,
    extra_state, ierror) BIND(C)
    PROCEDURE(MPI_Type_copy_attr_function) :: type_copy_attr_fn
    PROCEDURE(MPI_Type_delete_attr_function) :: type_delete_attr_fn
    INTEGER, INTENT(OUT) :: type_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_TYPE_CREATE_KEYVAL(TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN, TYPE_KEYVAL,
    EXTRA_STATE, IERROR)
    EXTERNAL TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN
    INTEGER TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE

{static int MPI::Datatype::Create_keyval(MPI::Datatype::Copy_attr_function*
    type_copy_attr_fn, MPI::Datatype::Delete_attr_function*
    type_delete_attr_fn, void* extra_state) (binding deprecated, see
    Section 15.2) }

```

The argument `type_copy_attr_fn` may be specified as `MPI_TYPE_NULL_COPY_FN` or `MPI_TYPE_DUP_FN` from either C, C++, or Fortran. `MPI_TYPE_NULL_COPY_FN` is a function that does nothing other than returning `flag = 0` and `MPI_SUCCESS`. `MPI_TYPE_DUP_FN` is a simple-minded copy function that sets `flag = 1`, returns the value of `attribute_val_in` in `attribute_val_out`, and returns `MPI_SUCCESS`.

The argument `type_delete_attr_fn` may be specified as `MPI_TYPE_NULL_DELETE_FN` from either C, C++, or Fortran. `MPI_TYPE_NULL_DELETE_FN` is a function that does nothing, other than returning `MPI_SUCCESS`.

The C callback functions are:

```

typedef int MPI_Type_copy_attr_function(MPI_Datatype oldtype,
    int type_keyval, void *extra_state, void *attribute_val_in,
    void *attribute_val_out, int *flag);

```

and

```

typedef int MPI_Type_delete_attr_function(MPI_Datatype datatype,
    int type_keyval, void *attribute_val, void *extra_state);

```

With the `mpi_f08` module, the Fortran callback functions are:

```

ABSTRACT INTERFACE
    SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_state,
        attribute_val_in, attribute_val_out, flag, ierror) BIND(C)
        TYPE(MPI_Datatype) :: oldtype
        INTEGER :: type_keyval, ierror
        INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
            attribute_val_out
        LOGICAL :: flag

```

and

```

ABSTRACT INTERFACE

```

```

1      SUBROUTINE MPI_Type_delete_attr_function(datatype, type_keyval,
2      attribute_val, extra_state, ierror) BIND(C)
3          TYPE(MPI_Datatype) :: datatype
4          INTEGER :: type_keyval, ierror
5          INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
6

```

ticket230-B. 7 With the `mpi` module and `mpif.h`, the Fortran callback functions are:

```

8
9      SUBROUTINE TYPE_COPY_ATTR_FUNCTION(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE,
10      ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
11          INTEGER OLDTYPE, TYPE_KEYVAL, IERROR
12          INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE,
13      ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT
14          LOGICAL FLAG
15

```

and

```

16      SUBROUTINE TYPE_DELETE_ATTR_FUNCTION(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL,
17      EXTRA_STATE, IERROR)
18          INTEGER DATATYPE, TYPE_KEYVAL, IERROR
19          INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
20

```

The C++ callbacks are:

```

22      {typedef int
23          MPI::Datatype::Copy_attr_function(const MPI::Datatype& oldtype,
24          int type_keyval, void* extra_state,
25          const void* attribute_val_in, void* attribute_val_out,
26          bool& flag); (binding deprecated, see Section 15.2)}
27

```

and

```

29      {typedef int MPI::Datatype::Delete_attr_function(MPI::Datatype& datatype,
30      int type_keyval, void* attribute_val, void* extra_state);
31      (binding deprecated, see Section 15.2)}
32

```

If an attribute copy function or attribute delete function returns other than `MPI_SUCCESS`, then the call that caused it to be invoked (for example, `MPI_TYPE_FREE`), is erroneous.

`MPI_TYPE_FREE_KEYVAL(type_keyval)`

INOUT `type_keyval` key value (integer)

`int MPI_Type_free_keyval(int *type_keyval)`

```

41
42      MPI_Type_free_keyval(type_keyval, ierror) BIND(C)
43          INTEGER, INTENT(INOUT) :: type_keyval
44          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46      MPI_TYPE_FREE_KEYVAL(TYPE_KEYVAL, IERROR)
47          INTEGER TYPE_KEYVAL, IERROR
48

```

```
{static void MPI::Datatype::Free_keyval(int& type_keyval) (binding deprecated,  

see Section 15.2) }
```

```
MPI_TYPE_SET_ATTR(datatype, type_keyval, attribute_val)
```

INOUT	[ticket252-W.] datatype	datatype to which attribute will be attached (handle)
IN	type_keyval	key value (integer)
IN	attribute_val	attribute value

```
int MPI_Type_set_attr(MPI_Datatype datatype, int type_keyval,  

void *attribute_val)
```

```
MPI_Type_set_attr(datatype, type_keyval, attribute_val, ierror) BIND(C)  

  TYPE(MPI_Datatype), INTENT(IN) :: datatype  

  INTEGER, INTENT(IN) :: type_keyval  

  INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val  

  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_TYPE_SET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR)  

  INTEGER DATATYPE, TYPE_KEYVAL, IERROR  

  INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
```

```
{void MPI::Datatype::Set_attr(int type_keyval, const void*  

attribute_val) (binding deprecated, see Section 15.2) }
```

```
MPI_TYPE_GET_ATTR(datatype, type_keyval, attribute_val, flag)
```

IN	[ticket252-W.] datatype	datatype to which the attribute is attached (handle)
IN	type_keyval	key value (integer)
OUT	attribute_val	attribute value, unless flag = false
OUT	flag	false if no attribute is associated with the key (logical)

```
int MPI_Type_get_attr(MPI_Datatype datatype, int type_keyval, void  

*attribute_val, int *flag)
```

```
MPI_Type_get_attr(datatype, type_keyval, attribute_val, flag, ierror)  

  BIND(C)  

  TYPE(MPI_Datatype), INTENT(IN) :: datatype  

  INTEGER, INTENT(IN) :: type_keyval  

  INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val  

  LOGICAL, INTENT(OUT) :: flag  

  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_TYPE_GET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)  

  INTEGER DATATYPE, TYPE_KEYVAL, IERROR  

  INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL  

  LOGICAL FLAG
```

```

1  {bool MPI::Datatype::Get_attr(int type_keyval, void* attribute_val)
2      const(binding deprecated, see Section 15.2) }
3
4
5  MPI_TYPE_DELETE_ATTR(datatype, type_keyval)
6
7      INOUT    [ticket252-W.]datatype    datatype from which the attribute is deleted (handle)
8
9      IN       type_keyval              key value (integer)
10
11  int MPI_Type_delete_attr(MPI_Datatype datatype, int type_keyval)
12
13  MPI_Type_delete_attr(datatype, type_keyval, ierror) BIND(C)
14      TYPE(MPI_Datatype), INTENT(IN) :: datatype
15      INTEGER, INTENT(IN) :: type_keyval
16      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
18  MPI_TYPE_DELETE_ATTR(DATATYPE, TYPE_KEYVAL, IERROR)
19
20  INTEGER DATATYPE, TYPE_KEYVAL, IERROR
21
22  {void MPI::Datatype::Delete_attr(int type_keyval)(binding deprecated, see
23      Section 15.2) }
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48

```

6.7.5 Error Class for Invalid Keyval

Key values for attributes are system-allocated, by MPI_{TYPE,COMM,WIN}_CREATE_KEYVAL. Only such values can be passed to the functions that use key values as input arguments. In order to signal that an erroneous key value has been passed to one of these functions, there is a new MPI error class: MPI_ERR_KEYVAL. It can be returned by MPI_ATTR_PUT, MPI_ATTR_GET, MPI_ATTR_DELETE, MPI_KEYVAL_FREE, MPI_{TYPE,COMM,WIN}_DELETE_ATTR, MPI_{TYPE,COMM,WIN}_SET_ATTR, MPI_{TYPE,COMM,WIN}_GET_ATTR, MPI_{TYPE,COMM,WIN}_FREE_KEYVAL, MPI_COMM_DUP, MPI_COMM_DISCONNECT, and MPI_COMM_FREE. The last three are included because keyval is an argument to the copy and delete functions for attributes.

6.7.6 Attributes Example

Advice to users. This example shows how to write a collective communication operation that uses caching to be more efficient after the first call. The coding style assumes that MPI function results return only error statuses. (*End of advice to users.*)

```

/* key for this module's stuff: */
static int gop_key = MPI_KEYVAL_INVALID;

typedef struct
{
    int ref_count;          /* reference count */
    /* other stuff, whatever else we want */
} gop_stuff_type;

```

- The Fortran binding is not trivial to write (it will depend on details of the Fortran compilation system), and will not be portable. Therefore it should be in the library rather than in user code.

(End of rationale.)

Advice to users. The above definition means that it is safe simply to print the string returned by `MPI_COMM_GET_NAME`, as it is always a valid string even if there was no name.

Note that associating a name with a communicator has no effect on the semantics of an MPI program, and will (necessarily) increase the store requirement of the program, since the names must be saved. Therefore there is no requirement that users use these functions to associate names with communicators. However debugging and profiling MPI applications may be made easier if names are associated with communicators, since the debugger or profiler should then be able to present information in a less cryptic manner. *(End of advice to users.)*

The following functions are used for setting and getting names of datatypes.

`MPI_TYPE_SET_NAME` (`datatype`, `type_name`)

INOUT	[ticket252-W.] <code>datatype</code>	datatype whose identifier is to be set (handle)
IN	<code>type_name</code>	the character string which is remembered as the name (string)

`int MPI_Type_set_name(MPI_Datatype datatype, char *type_name)`

```
MPI_Type_set_name(datatype, type_name, ierror) BIND(C)
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  CHARACTER(LEN=*), INTENT(IN) :: type_name
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_TYPE_SET_NAME(DATATYPE, TYPE_NAME, IERROR)
  INTEGER DATATYPE, IERROR
  CHARACTER*(*) TYPE_NAME
```

```
{void MPI::Datatype::Set_name(const char* type_name) (binding deprecated, see
  Section 15.2) }
```

`MPI_TYPE_GET_NAME` (`datatype`, `type_name`, `resultlen`)

IN	[ticket252-W.] <code>datatype</code>	datatype whose name is to be returned (handle)
OUT	<code>type_name</code>	the name previously stored on the datatype, or a empty string if no such name exists (string)
OUT	<code>resultlen</code>	length of returned name (integer)

```
int MPI_Type_get_name(MPI_Datatype datatype, char *type_name, int
  *resultlen)
```

ticket-248T.

```

1 MPI_Type_get_name(datatype, type_name, resultlen, ierror) BIND(C)
2   TYPE(MPI_Datatype), INTENT(IN) :: datatype
3   CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: type_name
4   INTEGER, INTENT(OUT) :: resultlen
5   INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

ticket252-W.

ticket252-W.

```

7 MPI_TYPE_GET_NAME(DATATYPE, TYPE_NAME, RESULTLEN, IERROR)
8   INTEGER DATATYPE, RESULTLEN, IERROR
9   CHARACTER*(*) TYPE_NAME

```

```

11 {void MPI::Datatype::Get_name(char* type_name, int& resultlen) const(binding
12     deprecated, see Section 15.2) }
```

Named predefined datatypes have the default names of the datatype name. For example, MPI_WCHAR has the default name of MPI_WCHAR.

The following functions are used for setting and getting names of windows.

```

18 MPI_WIN_SET_NAME (win, win_name)

```

19	INOUT	win	window whose identifier is to be set (handle)
21	IN	win_name	the character string which is remembered as the name (string)

ticket-248T.

```

24 int MPI_Win_set_name(MPI_Win win, char *win_name)

```

```

26 MPI_Win_set_name(win, win_name, ierror) BIND(C)
27   TYPE(MPI_Win), INTENT(IN) :: win
28   CHARACTER(LEN=*), INTENT(IN) :: win_name
29   INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

30 MPI_WIN_SET_NAME(WIN, WIN_NAME, IERROR)
31   INTEGER WIN, IERROR
32   CHARACTER*(*) WIN_NAME

```

```

34 {void MPI::Win::Set_name(const char* win_name) (binding deprecated, see
35     Section 15.2) }
```

```

38 MPI_WIN_GET_NAME (win, win_name, resultlen)

```

39	IN	win	window whose name is to be returned (handle)
41	OUT	win_name	the name previously stored on the window, or a empty string if no such name exists (string)
43	OUT	resultlen	length of returned name (integer)

ticket-248T.

```

46 int MPI_Win_get_name(MPI_Win win, char *win_name, int *resultlen)

```

```

47 MPI_Win_get_name(win, win_name, resultlen, ierror) BIND(C)
48   TYPE(MPI_Win), INTENT(IN) :: win

```

MPI_DIST_GRAPH_CREATE_ADJACENT(comm_old, indegree, sources, sourceweights, out-			1
degree, destinations, destweights, info, reorder, comm_dist_graph)			2
IN	comm_old	input communicator (handle)	3
IN	indegree	size of sources and sourceweights arrays (non-negative	4
		integer)	5
IN	sources	ranks of processes for which the calling process is a	6
		destination (array of non-negative integers)	7
IN	sourceweights	weights of the edges into the calling process (array of	8
		non-negative integers)	9
IN	outdegree	size of destinations and destweights arrays (non-negative	10
		integer)	11
IN	destinations	ranks of processes for which the calling process is a	12
		source (array of non-negative integers)	13
IN	destweights	weights of the edges out of the calling process (array	14
		of non-negative integers)	15
IN	info	hints on optimization and interpretation of weights	16
		(handle)	17
IN	reorder	the ranks may be reordered (true) or not (false) (logi-	18
		cal)	19
OUT	comm_dist_graph	communicator with distributed graph topology (han-	20
		dle)	21

```

int MPI_Dist_graph_create_adjacent(MPI_Comm comm_old, int indegree,
    int sources[], int sourceweights[], int outdegree,
    int destinations[], int destweights[], MPI_Info info,
    int reorder, MPI_Comm *comm_dist_graph)

```

```

MPI_Dist_graph_create_adjacent(comm_old, indegree, sources, sourceweights,
    outdegree, destinations, destweights, info, reorder,
    comm_dist_graph, ierror) BIND(C)
TYPE(MPI_Comm), INTENT(IN) :: comm_old
INTEGER, INTENT(IN) :: indegree, sources(indegree), outdegree,
destinations(outdegree)
INTEGER, INTENT(IN) :: sourceweights(*), destweights(*)
TYPE(MPI_Info), INTENT(IN) :: info
LOGICAL, INTENT(IN) :: reorder
TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

MPI_DIST_GRAPH_CREATE_ADJACENT(COMM_OLD, INDEGREE, SOURCES, SOURCEWEIGHTS,
    OUTDEGREE, DESTINATIONS, DESTWEIGHTS, INFO, REORDER,
    COMM_DIST_GRAPH, IERROR)
INTEGER COMM_OLD, INDEGREE, SOURCES(*), SOURCEWEIGHTS(*), OUTDEGREE,
    DESTINATIONS(*), DESTWEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR
LOGICAL REORDER

```

ticket229.2.
ticket-248T.


```

1      TYPE(MPI_Comm), INTENT(IN) :: comm
2      INTEGER, INTENT(OUT) :: indegree, outdegree
3      LOGICAL, INTENT(OUT) :: weighted
4      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
6 MPI_DIST_GRAPH_NEIGHBORS_COUNT(COMM, INDEGREE, OUTDEGREE, WEIGHTED, IERROR)
7     INTEGER COMM, INDEGREE, OUTDEGREE, IERROR
8     LOGICAL WEIGHTED
9
10 {void MPI::Distgraphcomm::Get_dist_neighbors_count(int rank,
11             int indegree[], int outdegree[], bool& weighted) const(binding
12             deprecated, see Section 15.2) }
13
14 MPI_DIST_GRAPH_NEIGHBORS(comm, maxindegree, sources, sourceweights, maxoutdegree,
15                          destinations, destweights)
16
17     IN      comm      communicator with distributed graph topology (han-
18                      dle)
19     IN      maxindegree      size of sources and sourceweights arrays (non-negative
20                              integer)
21     OUT     sources      processes for which the calling process is a destination
22                          (array of non-negative integers)
23     OUT     sourceweights      weights of the edges into the calling process (array of
24                              non-negative integers)
25
26     IN      maxoutdegree      size of destinations and destweights arrays (non-negative
27                              integer)
28     OUT     destinations      processes for which the calling process is a source (ar-
29                              ray of non-negative integers)
30
31     OUT     destweights      weights of the edges out of the calling process (array
32                              of non-negative integers)
33
34 int MPI_Dist_graph_neighbors(MPI_Comm comm, int maxindegree, int sources[],
35                             int sourceweights[], int maxoutdegree, int destinations[],
36                             int destweights[])
37
38 MPI_Dist_graph_neighbors(comm, maxindegree, sources, sourceweights,
39                          maxoutdegree, destinations, destweights, ierror) BIND(C)
40     TYPE(MPI_Comm), INTENT(IN) :: comm
41     INTEGER, INTENT(IN) :: maxindegree, maxoutdegree
42     INTEGER, INTENT(OUT) :: sources(maxindegree),
43                          destinations(maxoutdegree)
44     INTEGER :: sourceweights(*), destweights(*)
45     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47 MPI_DIST_GRAPH_NEIGHBORS(COMM, MAXINDEGREE, SOURCES, SOURCEWEIGHTS,
48                          MAXOUTDEGREE, DESTINATIONS, DESTWEIGHTS, IERROR)
49     INTEGER COMM, MAXINDEGREE, SOURCES(*), SOURCEWEIGHTS(*), MAXOUTDEGREE,

```

Chapter 8

MPI Environmental Management

This chapter discusses routines for getting and, where appropriate, setting various parameters that relate to the MPI implementation and the execution environment (such as error handling). The procedures for entering and leaving the MPI execution environment are also described here.

8.1 Implementation Information

8.1.1 Version Inquiries

In order to cope with changes to the MPI Standard, there are both compile-time and run-time ways to determine which version of the standard is in use in the environment one is using.

The “version” will be represented by two separate integers, for the version and subversion: In C and C++,

```
#define MPI_VERSION    2
#define MPI_SUBVERSION 2
```

in Fortran,

```
INTEGER [ticket240-L.] :: MPI_VERSION, MPI_SUBVERSION
PARAMETER (MPI_VERSION    = 2)
PARAMETER (MPI_SUBVERSION = 2)
```

For runtime determination,

`MPI_GET_VERSION(version, subversion)`

OUT	version	version number (integer)
OUT	subversion	subversion number (integer)

```
int MPI_Get_version(int *version, int *subversion)
```

```
MPI_Get_version(version, subversion, ierror) BIND(C)
```

```
    INTEGER, INTENT(OUT) :: version, subversion
```

```
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```

1 {void MPI::Get_processor_name(char* name, int& resultlen) (binding deprecated,
2 see Section 15.2) }
```

This routine returns the name of the processor on which it was called at the moment of the call. The name is a character string for maximum flexibility. From this value it must be possible to identify a specific piece of hardware; possible values include “processor 9 in rack 4 of mpp.cs.org” and “231” (where 231 is the actual processor number in the running homogeneous system). The argument `name` must represent storage that is at least `MPI_MAX_PROCESSOR_NAME` characters long. `MPI_GET_PROCESSOR_NAME` may write up to this many characters into `name`.

The number of characters actually written is returned in the output argument, `resultlen`. In C, a null character is additionally stored at `name[resultlen]`. The `resultlen` cannot be larger than `MPI_MAX_PROCESSOR_NAME-1`. In Fortran, `name` is padded on the right with blank characters. The `resultlen` cannot be larger than `MPI_MAX_PROCESSOR_NAME`.

Rationale. This function allows MPI implementations that do process migration to return the current processor. Note that nothing in MPI *requires* or defines process migration; this definition of `MPI_GET_PROCESSOR_NAME` simply allows such an implementation. (*End of rationale.*)

Advice to users. The user must provide at least `MPI_MAX_PROCESSOR_NAME` space to write the processor name — processor names can be this long. The user should examine the output argument, `resultlen`, to determine the actual length of the name. (*End of advice to users.*)

The constant `MPI_BSEND_OVERHEAD` provides an upper bound on the fixed overhead per message buffered by a call to `MPI_BSEND` (see Section 3.6.1).

8.2 Memory Allocation

In some systems, message-passing and remote-memory-access (RMA) operations run faster when accessing specially allocated memory (e.g., memory that is shared by the other processes in the communicating group on an SMP). MPI provides a mechanism for allocating and freeing such special memory. The use of such memory for message-passing or RMA is not mandatory, and this memory can be used without restrictions as any other dynamically allocated memory. However, implementations may restrict the use of the `MPI_WIN_LOCK` and `MPI_WIN_UNLOCK` functions to windows allocated in such memory (see Section 11.4.3.)

```

39 MPI_ALLOC_MEM(size, info, baseptr)
```

IN	size	size of memory segment in bytes (non-negative integer)
IN	info	info argument (handle)
OUT	baseptr	pointer to beginning of memory segment allocated

```

46 int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr)
```

```

47 MPI_Alloc_mem(size, info, baseptr, ierror) BIND(C)
```

```

USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
TYPE(MPI_Info), INTENT(IN) :: info
TYPE(C_PTR), INTENT(OUT) :: baseptr
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
  INTEGER INFO, IERROR
  INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR

{void* MPI::Alloc_mem(MPI::Aint size, const MPI::Info& info) (binding
  deprecated, see Section 15.2) }

```

If the Fortran compiler provides `TYPE(C_PTR)`, then the following interface must be provided in the `mpi` module and should be provided in `mpif.h` through overloading, i.e., with the same routine name as the routine with `INTEGER(KIND=MPI_ADDRESS_KIND) BASEPTR`, but with a different linker name:

```

INTERFACE MPI_ALLOC_MEM
  SUBROUTINE MPI_ALLOC_MEM_CPTR(SIZE, INFO, BASEPTR, IERROR)
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
    INTEGER :: INFO, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
    TYPE(C_PTR) :: BASEPTR
  END SUBROUTINE
END INTERFACE

```

The linker name base of this overloaded function is `MPI_ALLOC_MEM_CPTR`. The implied linker names are described in Section 16.2.5 on page 560.

The `info` argument can be used to provide directives that control the desired location of the allocated memory. Such a directive does not affect the semantics of the call. Valid `info` values are implementation-dependent; a null directive value of `info = MPI_INFO_NULL` is always valid.

The function `MPI_ALLOC_MEM` may return an error code of class `MPI_ERR_NO_MEM` to indicate it failed because memory is exhausted.

```

MPI_FREE_MEM(base)
  IN      base
          initial address of memory segment allocated by
          MPI_ALLOC_MEM (choice)

```

```

int MPI_Free_mem(void *base)

MPI_Free_mem(base, ierror) BIND(C)
  TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: base
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_FREE_MEM(BASE, IERROR)
  <type> BASE(*)
  INTEGER IERROR

```

```
1 {void MPI::Free_mem(void *base) (binding deprecated, see Section 15.2) }
```

2 The function `MPI_FREE_MEM` may return an error code of class `MPI_ERR_BASE` to
3 indicate an invalid base argument.
4

5 *Rationale.* The C and C++ bindings of `MPI_ALLOC_MEM` and `MPI_FREE_MEM`
6 are similar to the bindings for the `malloc` and `free` C library calls: a call to
7 `MPI_Alloc_mem(..., &base)` should be paired with a call to `MPI_Free_mem(base)` (one
8 less level of indirection). Both arguments are declared to be of same type `void*` so
9 as to facilitate type casting. The Fortran binding is consistent with the C and C++
10 bindings: the Fortran `MPI_ALLOC_MEM` call returns in `baseptr` the `TYPE(C_PTR)`
11 `pointer` or the (integer valued) address of the allocated memory. The `base` argument
12 of `MPI_FREE_MEM` is a choice argument, which passes (a reference to) the variable
13 stored at that location. (*End of rationale.*)
14

15 *Advice to implementors.* If `MPI_ALLOC_MEM` allocates special memory, then a
16 design similar to the design of C `malloc` and `free` functions has to be used, in order
17 to find out the size of a memory segment, when the segment is freed. If no special
18 memory is used, `MPI_ALLOC_MEM` simply invokes `malloc`, and `MPI_FREE_MEM`
19 invokes `free`.
20

21 A call to `MPI_ALLOC_MEM` can be used in shared memory systems to allocate mem-
22 ory in a shared memory segment. (*End of advice to implementors.*)
23

24 **Example 8.1** Example of use of `MPI_ALLOC_MEM`, in Fortran with
25 `TYPE(C_PTR)` pointers. We assume 4-byte REALs.
26

```
27 USE mpi_f08 ! or USE mpi (not guaranteed with INCLUDE 'mpif.h')
28 USE, INTRINSIC :: ISO_C_BINDING
29 TYPE(C_PTR) :: p
30 REAL, DIMENSION(:,:), POINTER :: a ! no memory is allocated
31 INTEGER, DIMENSION(2) :: shape
32 INTEGER(KIND=MPI_ADDRESS_KIND) :: size
33 shape = (/100,100/)
34 size = 4 * shape(1) * shape(2) ! assuming 4 bytes per REAL
35 CALL MPI_Alloc_mem(size,MPI_INFO_NULL,p,ierr) ! memory is allocated and
36 CALL C_F_POINTER(p, a, shape) ! intrinsic ! now accessible via a(i,j)
37 ... ! in ISO_C_BINDING
38 a(3,5) = 2.71;
39 ...
40 CALL MPI_Free_mem(a, ierr) ! memory is freed
41
```

42 **Example 8.2** Example of use of `MPI_ALLOC_MEM`, in Fortran with non-standard *Cray-*
43 *pointer*. We assume 4-byte REALs, and assume that *these* pointers are address-sized.
44
45
46
47
48

```

REAL A
POINTER (P, A(100,100)) ! no memory is allocated
[ticket245-Q.]INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
[ticket245-Q.]SIZE = 4*100*100
CALL MPI_ALLOC_MEM([ticket245-Q.]SIZE, MPI_INFO_NULL, P, IERR)
! memory is allocated
...
A(3,5) = 2.71;
...
CALL MPI_FREE_MEM(A, IERR) ! memory is freed

```

This code is not Fortran 77 or Fortran 90 code. Some compilers **may** not support this code **or need a special option**, e.g., the GNU gFortran compiler needs **-fcray-pointer**.

Advice to implementors. Some compilers map Cray-pointers to address-sized integers, some to `TYPE(C_PTR)` pointers (e.g., Cray Fortran, version 7.3.3). From the user's viewpoint, this mapping is irrelevant because Examples 8.2 should work correctly with an MPI-3.0 (or later) library if Cray-pointers are available. (*End of advice to implementors.*)

Example 8.3 Same example, in C

```

float (*f)[100][100] ;
/* no memory is allocated */
MPI_Alloc_mem(sizeof(float)*100*100, MPI_INFO_NULL, &f);
/* memory allocated */
...
(*f)[5][3] = 2.71;
...
MPI_Free_mem(f);

```

8.3 Error Handling

An MPI implementation cannot or may choose not to handle some errors that occur during MPI calls. These can include errors that generate exceptions or traps, such as floating point errors or access violations. The set of errors that are handled by MPI is implementation-dependent. Each such error generates an **MPI exception**.

The above text takes precedence over any text on error handling within this document. Specifically, text that states that errors *will* be handled should be read as *may* be handled.

A user can associate error handlers to three types of objects: communicators, windows, and files. The specified error handling routine will be used for any MPI exception that occurs during a call to MPI for the respective object. MPI calls that are not related to any objects are considered to be attached to the communicator `MPI_COMM_WORLD`. The attachment of error handlers to objects is purely local: different processes may attach different error handlers to corresponding objects.

Several predefined error handlers are available in MPI:

should be called with the error handler returned from `MPI_ERRHANDLER_GET` or `MPI_{COMM,WIN,FILE}_GET_ERRHANDLER` to mark the error handler for deallocation. This provides behavior similar to that of `MPI_COMM_GROUP` and `MPI_GROUP_FREE`.

Advice to implementors. High-quality implementation should raise an error when an error handler that was created by a call to `MPI_XXX_CREATE_ERRHANDLER` is attached to an object of the wrong type with a call to `MPI_YYY_SET_ERRHANDLER`. To do so, it is necessary to maintain, with each error handler, information on the typedef of the associated user function. (*End of advice to implementors.*)

The syntax for these calls is given below.

8.3.1 Error Handlers for Communicators

`MPI_COMM_CREATE_ERRHANDLER(comm_errhandler_fn, errhandler)`

IN [ticket252-W.] `comm_errhandler_fn` user defined error handling procedure (function)

OUT `errhandler` MPI error handler (handle)

```
int MPI_Comm_create_errhandler(MPI_Comm_errhandler_function
                               *comm_errhandler_fn, MPI_Errhandler *errhandler)

MPI_Comm_create_errhandler(comm_errhandler_fn, errhandler, ierror) BIND(C)
PROCEDURE(MPI_Comm_errhandler_function) :: comm_errhandler_fn
TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_COMM_CREATE_ERRHANDLER(COMM_ERRHANDLER_FN, ERRHANDLER, IERROR)
EXTERNAL COMM_ERRHANDLER_FN
INTEGER ERRHANDLER, IERROR
```

```
{static MPI::Errhandler
    MPI::Comm::Create_errhandler(MPI::Comm::Errhandler_function*
    comm_errhandler_fn) (binding deprecated, see Section 15.2) }
```

Creates an error handler that can be attached to communicators. This function is identical to `MPI_ERRHANDLER_CREATE`, whose use is deprecated.

The user routine should be, in C, a function of type `MPI_Comm_errhandler_function`, which is defined as

```
typedef void MPI_Comm_errhandler_function(MPI_Comm *, int *, ...);
```

The first argument is the communicator in use. The second is the error code to be returned by the MPI routine that raised the error. If the routine would have returned `MPI_ERR_IN_STATUS`, it is the error code returned in the status for the request that caused the error handler to be invoked. The remaining arguments are “`stdargs`” arguments whose number and meaning is implementation-dependent. An implementation should clearly document these arguments. Addresses are used so that the handler may be written in Fortran. This typedef replaces `MPI_Handler_function`, whose use is deprecated.

With the Fortran `mpi_f08` module, the user routine `comm_errhandler_fn` should be of the

form:

ABSTRACT INTERFACE

```
SUBROUTINE MPI_Comm_errhandler_function(comm, error_code) BIND(C)
```

```
    TYPE(MPI_Comm) :: comm
```

```
    INTEGER :: error_code
```

With the Fortran `mpi` module and `mpif.h`, the user routine `COMM_ERRHANDLER_FN` should be of the form:

```
SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE)
```

```
    INTEGER COMM, ERROR_CODE
```

In C++, the user routine should be of the form:

```
{typedef void MPI::Comm::Errhandler_function(MPI::Comm &, int *, ...);  
    (binding deprecated, see Section 15.2)}
```

Rationale. The variable argument list is provided because it provides an ISO-standard hook for providing additional information to the error handler; without this hook, ISO C prohibits additional arguments. (*End of rationale.*)

Advice to users. A newly created communicator inherits the error handler that is associated with the “parent” communicator. In particular, the user can specify a “global” error handler for all communicators by associating this handler with the communicator `MPI_COMM_WORLD` immediately after initialization. (*End of advice to users.*)

```
MPI_COMM_SET_ERRHANDLER(comm, errhandler)
```

INOUT	comm	communicator (handle)
-------	------	-----------------------

IN	errhandler	new error handler for communicator (handle)
----	------------	---

```
int MPI_Comm_set_errhandler(MPI_Comm comm, MPI_Errhandler errhandler)
```

```
MPI_Comm_set_errhandler(comm, errhandler, ierror) BIND(C)
```

```
    TYPE(MPI_Comm), INTENT(IN) :: comm
```

```
    TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
```

```
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_COMM_SET_ERRHANDLER(COMM, ERRHANDLER, IERROR)
```

```
    INTEGER COMM, ERRHANDLER, IERROR
```

```
{void MPI::Comm::Set_errhandler(const MPI::Errhandler& errhandler) (binding  
    deprecated, see Section 15.2) }
```

Attaches a new error handler to a communicator. The error handler must be either a predefined error handler, or an error handler created by a call to `MPI_COMM_CREATE_ERRHANDLER`. This call is identical to `MPI_ERRHANDLER_SET`, whose use is deprecated.

MPI_COMM_GET_ERRHANDLER(comm, errhandler)

IN	comm	communicator (handle)
OUT	errhandler	error handler currently associated with communicator (handle)

```
int MPI_Comm_get_errhandler(MPI_Comm comm, MPI_Errhandler *errhandler)
```

```
MPI_Comm_get_errhandler(comm, errhandler, ierror) BIND(C)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_COMM_GET_ERRHANDLER(COMM, ERRHANDLER, IERROR)
    INTEGER COMM, ERRHANDLER, IERROR
```

```
{MPI::Errhandler MPI::Comm::Get_errhandler() const(binding deprecated, see
    Section 15.2) }
```

Retrieves the error handler currently associated with a communicator. This call is identical to MPI_ERRHANDLER_GET, whose use is deprecated.

Example: A library function may register at its entry point the current error handler for a communicator, set its own private error handler for this communicator, and restore before exiting the previous error handler.

8.3.2 Error Handlers for Windows

MPI_WIN_CREATE_ERRHANDLER(win_errhandler_fn, errhandler)

IN	[ticket252-W.]win_errhandler_fn	user defined error handling procedure (function)
OUT	errhandler	MPI error handler (handle)

```
int MPI_Win_create_errhandler(MPI_Win_errhandler_function
    *win_errhandler_fn, MPI_Errhandler *errhandler)
```

```
MPI_Win_create_errhandler(win_errhandler_fn, errhandler, ierror) BIND(C)
    PROCEDURE(MPI_Win_errhandler_function) :: win_errhandler_fn
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_WIN_CREATE_ERRHANDLER(WIN_ERRHANDLER_FN, ERRHANDLER, IERROR)
    EXTERNAL WIN_ERRHANDLER_FN
    INTEGER ERRHANDLER, IERROR
```

```
{static MPI::Errhandler
    MPI::Win::Create_errhandler(MPI::Win::Errhandler_function*
    win_errhandler_fn)(binding deprecated, see Section 15.2) }
```

Creates an error handler that can be attached to a window object. The user routine should be, in C, a function of type MPI_Win_errhandler_function which is defined as

```
1 typedef void MPI_Win_errhandler_function(MPI_Win *, int *, ...);
```

2 The first argument is the window in use, the second is the error code to be returned.

3 With the Fortran `mpi_f08` module, the user routine `win_errhandler_fn` should be of the form:

```
4 ABSTRACT INTERFACE
```

```
5 SUBROUTINE MPI_Win_errhandler_function(win, error_code) BIND(C)
```

```
6 TYPE(MPI_Win) :: win
```

```
7 INTEGER :: error_code
```

8 With the Fortran `mpi` module and `mpif.h`, the user routine `WIN_ERRHANDLER_FN` should be of the form:

```
9 SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)
```

```
10 INTEGER WIN, ERROR_CODE
```

11 In C++, the user routine should be of the form:

```
12 {typedef void MPI::Win::Errhandler_function(MPI::Win &, int *, ...);  
13 (binding deprecated, see Section 15.2)}
```

```
14 MPI_WIN_SET_ERRHANDLER(win, errhandler)
```

```
15 INOUT win window (handle)
```

```
16 IN errhandler new error handler for window (handle)
```

```
17 int MPI_Win_set_errhandler(MPI_Win win, MPI_Errhandler errhandler)
```

```
18 MPI_Win_set_errhandler(win, errhandler, ierror) BIND(C)
```

```
19 TYPE(MPI_Win), INTENT(IN) :: win
```

```
20 TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
```

```
21 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
22 MPI_WIN_SET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
```

```
23 INTEGER WIN, ERRHANDLER, IERROR
```

```
24 {void MPI::Win::Set_errhandler(const MPI::Errhandler& errhandler) (binding  
25 deprecated, see Section 15.2) }
```

26 Attaches a new error handler to a window. The error handler must be either a pre-defined error handler, or an error handler created by a call to `MPI_WIN_CREATE_ERRHANDLER`.

```
27 MPI_WIN_GET_ERRHANDLER(win, errhandler)
```

```
28 IN win window (handle)
```

```
29 OUT errhandler error handler currently associated with window (handle)
```

```
30 int MPI_Win_get_errhandler(MPI_Win win, MPI_Errhandler *errhandler)
```

ticket-248T.

```

MPI_Win_get_errhandler(win, errhandler, ierror) BIND(C)
    TYPE(MPI_Win), INTENT(IN) :: win
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_WIN_GET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
    INTEGER WIN, ERRHANDLER, IERROR

{MPI::Errhandler MPI::Win::Get_errhandler() const(binding deprecated, see
    Section 15.2) }
```

Retrieves the error handler currently associated with a window.

8.3.3 Error Handlers for Files

```

MPI_FILE_CREATE_ERRHANDLER(file_errhandler_fn, errhandler)

IN      [ticket252-W.]file_errhandler_fn  user defined error handling procedure (function)
OUT     errhandler                        MPI error handler (handle)

int MPI_File_create_errhandler(MPI_File_errhandler_function
    *file_errhandler_fn, MPI_Errhandler *errhandler)

MPI_File_create_errhandler(file_errhandler_fn, errhandler, ierror) BIND(C)
    PROCEDURE(MPI_File_errhandler_function) :: file_errhandler_fn
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_FILE_CREATE_ERRHANDLER(FILE_ERRHANDLER_FN, ERRHANDLER, IERROR)
    EXTERNAL FILE_ERRHANDLER_FN
    INTEGER ERRHANDLER, IERROR

{static MPI::Errhandler
    MPI::File::Create_errhandler(MPI::File::Errhandler_function*
    file_errhandler_fn) (binding deprecated, see Section 15.2) }
```

Creates an error handler that can be attached to a file object. The user routine should be, in C, a function of type `MPI_File_errhandler_function`, which is defined as

```
typedef void MPI_File_errhandler_function(MPI_File *, int *, ...);
```

The first argument is the file in use, the second is the error code to be returned.

With the Fortran `mpi_f08` module, the user routine `file_errhandler_fn` should be of the form:

```

ABSTRACT INTERFACE
    SUBROUTINE MPI_File_errhandler_function(file, error_code) BIND(C)
        TYPE(MPI_File) :: file
        INTEGER :: error_code
```

With the Fortran `mpi` module and `mpif.h`, the user routine `FILE_ERRHANDLER_FN` should be of the form:

Chapter 9

The Info Object

Many of the routines in MPI take an argument `info`. `info` is an opaque object with a handle of type `MPI_Info` in C and Fortran with the `mpi_f08` module, `MPI::Info` in C++, and `INTEGER` in Fortran with the `mpi` module or the include file `mpif.h`. It stores an unordered set of (key,value) pairs (both key and value are strings). A key can have only one value. MPI reserves several keys and requires that if an implementation uses a reserved key, it must provide the specified functionality. An implementation is not required to support these keys and may support any others not reserved by MPI.

An implementation must support info objects as caches for arbitrary (key, value) pairs, regardless of whether it recognizes the key. Each function that takes hints in the form of an `MPI_Info` must be prepared to ignore any key it does not recognize. This description of info objects does not attempt to define how a particular function should react if it recognizes a key but not the associated value. `MPI_INFO_GET_NKEYS`, `MPI_INFO_GET_NTHKEY`, `MPI_INFO_GET_VALUELEN`, and `MPI_INFO_GET` must retain all (key,value) pairs so that layered functionality can also use the `Info` object.

Keys have an implementation-defined maximum length of `MPI_MAX_INFO_KEY`, which is at least 32 and at most 255. Values have an implementation-defined maximum length of `MPI_MAX_INFO_VAL`. In Fortran, leading and trailing spaces are stripped from both. Returned values will never be larger than these maximum lengths. Both key and value are case sensitive.

Rationale. Keys have a maximum length because the set of known keys will always be finite and known to the implementation and because there is no reason for keys to be complex. The small maximum size allows applications to declare keys of size `MPI_MAX_INFO_KEY`. The limitation on value sizes is so that an implementation is not forced to deal with arbitrarily long strings. (*End of rationale.*)

Advice to users. `MPI_MAX_INFO_VAL` might be very large, so it might not be wise to declare a string of that size. (*End of advice to users.*)

When it is an argument to a nonblocking routine, `info` is parsed before that routine returns, so that it may be modified or freed immediately after return.

When the descriptions refer to a key or value as being a boolean, an integer, or a list, they mean the string representation of these types. An implementation may define its own rules for how info value strings are converted to other types, but to ensure portability, every implementation must support the following representations. Legal values for a boolean must

Advice to users. By default, requests are hard and MPI errors are fatal. This means that by default there will be a fatal error if MPI cannot spawn all the requested processes. If you want the behavior “spawn as many processes as possible, up to N ,” you should do a soft spawn, where the set of allowed values $\{m_i\}$ is $\{0 \dots N\}$. However, this is not completely portable, as implementations are not required to support soft spawning. (*End of advice to users.*)

The info argument The info argument to all of the routines in this chapter is an opaque handle of type `MPI_Info` in C and Fortran with the `mpi_f08` module, `MPI::Info` in C++ and `INTEGER` in Fortran with the `mpi` module or the include file `mpif.h`. It is a container for a number of user-specified (key,value) pairs. key and value are strings (null-terminated `char*` in C, `character(*)` in Fortran). Routines to create and manipulate the info argument are described in Section 9 on page 357.

For the `SPAWN` calls, info provides additional (and possibly implementation-dependent) instructions to MPI and the runtime system on how to start processes. An application may pass `MPI_INFO_NULL` in C or Fortran. Portable programs not requiring detailed control over process locations should use `MPI_INFO_NULL`.

MPI does not specify the content of the info argument, except to reserve a number of special key values (see Section 10.3.4 on page 374). The info argument is quite flexible and could even be used, for example, to specify the executable and its command-line arguments. In this case the `command` argument to `MPI_COMM_SPAWN` could be empty. The ability to do this follows from the fact that MPI does not specify how an executable is found, and the info argument can tell the runtime system where to “find” the executable “” (empty string). Of course a program that does this will not be portable across MPI implementations.

The root argument All arguments before the root argument are examined only on the process whose rank in `comm` is equal to `root`. The value of these arguments on other processes is ignored.

The array_of_errcodes argument The `array_of_errcodes` is an array of length `maxprocs` in which MPI reports the status of each process that MPI was requested to start. If all `maxprocs` processes were spawned, `array_of_errcodes` is filled in with the value `MPI_SUCCESS`. If only m ($0 \leq m < \text{maxprocs}$) processes are spawned, m of the entries will contain `MPI_SUCCESS` and the rest will contain an implementation-specific error code indicating the reason MPI could not start the process. MPI does not specify which entries correspond to failed processes. An implementation may, for instance, fill in error codes in one-to-one correspondence with a detailed specification in the info argument. These error codes all belong to the error class `MPI_ERR_SPAWN` if there was no error in the argument list. In C or Fortran, an application may pass `MPI_ERRCODES_IGNORE` if it is not interested in the error codes. In C++ this constant does not exist, and the `array_of_errcodes` argument may be omitted from the argument list.

Advice to implementors. `MPI_ERRCODES_IGNORE` in Fortran is a special type of constant, like `MPI_BOTTOM`. See the discussion in Section 2.5.4 on page 15. (*End of advice to implementors.*)

```

{MPI::Intercomm MPI::Intracomm::Spawn_multiple(int count,
    const char* array_of_commands[], const char** array_of_argv[],
    const int array_of_maxprocs[],
    const MPI::Info array_of_info[], int root,
    int array_of_errcodes[]) (binding deprecated, see Section 15.2) }

{MPI::Intercomm MPI::Intracomm::Spawn_multiple(int count,
    const char* array_of_commands[], const char** array_of_argv[],
    const int array_of_maxprocs[],
    const MPI::Info array_of_info[], int root) (binding deprecated, see
    Section 15.2) }

```

MPI_COMM_SPAWN_MULTIPLE is identical to MPI_COMM_SPAWN except that there are multiple executable specifications. The first argument, `count`, gives the number of specifications. Each of the next four arguments are simply arrays of the corresponding arguments in MPI_COMM_SPAWN. For the Fortran version of `array_of_argv`, the element `array_of_argv(i,j)` is the j -th argument to command number i .

Rationale. This may seem backwards to Fortran programmers who are familiar with Fortran's column-major ordering. However, it is necessary to do it this way to allow MPI_COMM_SPAWN to sort out arguments. Note that the leading dimension of `array_of_argv` *must* be the same as `count`. (*End of rationale.*)

Advice to users. The argument `count` is interpreted by MPI only at the root, as is `array_of_argv`. Since the leading dimension of `array_of_argv` is `count`, a non-positive value of `count` at a non-root node could theoretically cause a runtime bounds check error, even though `array_of_argv` should be ignored by the subroutine. If this happens, you should explicitly supply a reasonable value of `count` on the non-root nodes. (*End of advice to users.*)

In any language, an application may use the constant MPI_ARGVS_NULL (which is likely to be `(char ***)0` in C) to specify that no arguments should be passed to any commands. The effect of setting individual elements of `array_of_argv` to MPI_ARGV_NULL is not defined. To specify arguments for some commands but not others, the commands without arguments should have a corresponding `argv` whose first element is null (`(char *)0` in C and empty string in Fortran). **In Fortran at non-root processes, the `count` argument must be set to a value that is consistent with the provided `array_of_argv` although the content of these arguments has no meaning for this operation.**

ticket229.1.

All of the spawned processes have the same MPI_COMM_WORLD. Their ranks in MPI_COMM_WORLD correspond directly to the order in which the commands are specified in MPI_COMM_SPAWN_MULTIPLE. Assume that m_1 processes are generated by the first command, m_2 by the second, etc. The processes corresponding to the first command have ranks $0, 1, \dots, m_1 - 1$. The processes in the second command have ranks $m_1, m_1 + 1, \dots, m_1 + m_2 - 1$. The processes in the third have ranks $m_1 + m_2, m_1 + m_2 + 1, \dots, m_1 + m_2 + m_3 - 1$, etc.

Advice to users. Calling MPI_COMM_SPAWN multiple times would create many sets of children with different MPI_COMM_WORLDS whereas MPI_COMM_SPAWN_MULTIPLE creates children with a single MPI_COMM_WORLD,

process in which the memory is accessed. Thus, in a put operation, source=origin and destination=target; in a get operation, source=target and destination=origin.

11.2 Initialization

11.2.1 Window Creation

The initialization operation allows each process in an intracommunicator group to specify, in a collective operation, a “window” in its memory that is made accessible to accesses by remote processes. The call returns an opaque object that represents the group of processes that own and access the set of windows, and the attributes of each window, as specified by the initialization call.

`MPI_WIN_CREATE(base, size, disp_unit, info, comm, win)`

IN	base	initial address of window (choice)
IN	size	size of window in bytes (non-negative integer)
IN	disp_unit	local unit size for displacements, in bytes (positive integer)
IN	info	info argument (handle)
IN	comm	communicator (handle)
OUT	win	window object returned by the call (handle)

`int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win *win)`

`MPI_Win_create(base, size, disp_unit, info, comm, win, ierror) BIND(C)`
`TYPE(*), DIMENSION(..), ASYNCHRONOUS :: base`
`INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size`
`INTEGER, INTENT(IN) :: disp_unit`
`TYPE(MPI_Info), INTENT(IN) :: info`
`TYPE(MPI_Comm), INTENT(IN) :: comm`
`TYPE(MPI_Win), INTENT(OUT) :: win`
`INTEGER, OPTIONAL, INTENT(OUT) :: ierror`

`MPI_WIN_CREATE(BASE, SIZE, DISP_UNIT, INFO, COMM, WIN, IERROR)`

`<type> BASE(*)`
`INTEGER(KIND=MPI_ADDRESS_KIND) SIZE`
`INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR`

`{static MPI::Win MPI::Win::Create(const void* base, MPI::Aint size, int disp_unit, const MPI::Info& info, const MPI::Intracomm& comm) (binding deprecated, see Section 15.2) }`

This is a collective call executed by all processes in the group of comm. It returns a window object that can be used by these processes to perform RMA operations. Each process specifies a window of existing memory that it exposes to RMA accesses by the processes in the group of comm. The window consists of size bytes, starting at address base.

In C and C++, **base** is the starting address of a memory region. In Fortran, one can pass the first element of a memory region or a whole array, which must be ‘simply contiguous’ (for ‘simply contiguous’, see also Section 16.2.12 on page 581). A process may elect to expose no memory by specifying `size = 0`.

The displacement unit argument is provided to facilitate address arithmetic in RMA operations: the target displacement argument of an RMA operation is scaled by the factor `disp_unit` specified by the target process, at window creation.

Rationale. The window size is specified using an address sized integer, so as to allow windows that span more than 4 GB of address space. (Even if the physical memory size is less than 4 GB, the address range may be larger than 4 GB, if addresses are not contiguous.) (*End of rationale.*)

Advice to users. Common choices for `disp_unit` are 1 (no scaling), and (in C syntax) `sizeof(type)`, for a window that consists of an array of elements of type `type`. The later choice will allow one to use array indices in RMA calls, and have those scaled correctly to byte displacements, even in a heterogeneous environment. (*End of advice to users.*)

The `info` argument provides optimization hints to the runtime about the expected usage pattern of the window. The following info key is predefined:

`no_locks` — if set to true, then the implementation may assume that the local window is never locked (by a call to `MPI_WIN_LOCK`). This implies that this window is not used for 3-party communication, and RMA can be implemented with no (less) asynchronous agent activity at this process.

The various processes in the group of `comm` may specify completely different target windows, in location, size, displacement units and info arguments. As long as all the get, put and accumulate accesses to a particular process fit their specific target window this should pose no problem. The same area in memory may appear in multiple windows, each associated with a different window object. However, concurrent communications to distinct, overlapping windows may lead to erroneous results.

Advice to users. A window can be created in any part of the process memory. However, on some systems, the performance of windows in memory allocated by `MPI_ALLOC_MEM` (Section 8.2, page 328) will be better. Also, on some systems, performance is improved when window boundaries are aligned at “natural” boundaries (word, double-word, cache line, page frame, etc.). (*End of advice to users.*)

Advice to implementors. In cases where RMA operations use different mechanisms in different memory areas (e.g., load/store in a shared memory segment, and an asynchronous handler in private memory), the `MPI_WIN_CREATE` call needs to figure out which type of memory is used for the window. To do so, MPI maintains, internally, the list of memory segments allocated by `MPI_ALLOC_MEM`, or by other, implementation specific, mechanisms, together with information on the type of memory segment allocated. When a call to `MPI_WIN_CREATE` occurs, then MPI checks which segment contains each window, and decides, accordingly, which mechanism to use for RMA operations.


```

MPI_Put(origin_addr, origin_count, origin_datatype, target_rank,
        target_disp, target_count, target_datatype, win, ierror)
        BIND(C)
        TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
        INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
        TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
        INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
        TYPE(MPI_Win), INTENT(IN) :: win
        INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_PUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
        TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
        <type> ORIGIN_ADDR(*)
        INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
        INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
        TARGET_DATATYPE, WIN, IERROR

{void MPI::Win::Put(const void* origin_addr, int origin_count, const
        MPI::Datatype& origin_datatype, int target_rank, MPI::Aint
        target_disp, int target_count, const MPI::Datatype&
        target_datatype) const(binding deprecated, see Section 15.2) }

```

Transfers `origin_count` successive entries of the type specified by the `origin_datatype`, starting at address `origin_addr` on the origin node to the target node specified by the `win`, `target_rank` pair. The data are written in the target buffer at address `target_addr = window_base + target_disp × disp_unit`, where `window_base` and `disp_unit` are the base address and window displacement unit specified at window initialization, by the target process.

The target buffer is specified by the arguments `target_count` and `target_datatype`.

The data transfer is the same as that which would occur if the origin process executed a send operation with arguments `origin_addr`, `origin_count`, `origin_datatype`, `target_rank`, `tag`, `comm`, and the target process executed a receive operation with arguments `target_addr`, `target_count`, `target_datatype`, `source`, `tag`, `comm`, where `target_addr` is the target buffer address computed as explained above, and `comm` is a communicator for the group of `win`.

The communication must satisfy the same constraints as for a similar message-passing communication. The `target_datatype` may not specify overlapping entries in the target buffer. The message sent must fit, without truncation, in the target buffer. Furthermore, the target buffer must fit in the target window.

The `target_datatype` argument is a handle to a datatype object defined at the origin process. However, this object is interpreted at the target process: the outcome is as if the target datatype object was defined at the target process, by the same sequence of calls used to define it at the origin process. The target datatype must contain only relative displacements, not absolute addresses. The same holds for `get` and `accumulate`.

Advice to users. The `target_datatype` argument is a handle to a datatype object that is defined at the origin process, even though it defines a data layout in the target process memory. This causes no problems in a homogeneous environment, or in a heterogeneous environment, if only portable datatypes are used (portable datatypes are defined in Section 2.4, page 11).

The performance of a `put` transfer can be significantly affected, on some systems, from

the choice of window location and the shape and location of the origin and target buffer: transfers to a target window in memory allocated by `MPI_ALLOC_MEM` may be much faster on shared memory systems; transfers from contiguous buffers will be faster on most, if not all, systems; the alignment of the communication buffers may also impact performance. (*End of advice to users.*)

Advice to implementors. A high-quality implementation will attempt to prevent remote accesses to memory outside the window that was exposed by the process. This, both for debugging purposes, and for protection with client-server codes that use RMA. I.e., a high-quality implementation will check, if possible, window bounds on each RMA call, and raise an MPI exception at the origin call if an out-of-bound situation occurred. Note that the condition can be checked at the origin. Of course, the added safety achieved by such checks has to be weighed against the added cost of such checks. (*End of advice to implementors.*)

11.3.2 Get

`MPI_GET(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win)`

OUT	<code>origin_addr</code>	initial address of origin buffer (choice)
IN	<code>origin_count</code>	number of entries in origin buffer (non-negative integer)
IN	<code>origin_datatype</code>	datatype of each entry in origin buffer (handle)
IN	<code>target_rank</code>	rank of target (non-negative integer)
IN	<code>target_disp</code>	displacement from window start to the beginning of the target buffer (non-negative integer)
IN	<code>target_count</code>	number of entries in target buffer (non-negative integer)
IN	<code>target_datatype</code>	datatype of each entry in target buffer (handle)
IN	<code>win</code>	window object used for communication (handle)

```
int MPI_Get(void *origin_addr, int origin_count, MPI_Datatype
            origin_datatype, int target_rank, MPI_Aint target_disp, int
            target_count, MPI_Datatype target_datatype, MPI_Win win)
```

```
MPI_Get(origin_addr, origin_count, origin_datatype, target_rank,
        target_disp, target_count, target_datatype, win, ierror)
        BIND(C)
        TYPE(*), DIMENSION(..), ASYNCHRONOUS :: origin_addr
        INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
        TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
        INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
        TYPE(MPI_Win), INTENT(IN) :: win
        INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```

1 MPI_ACCUMULATE(origin_addr, origin_count, origin_datatype, target_rank, target_disp,
2               target_count, target_datatype, op, win)

```

3	IN	origin_addr	initial address of buffer (choice)
4			
5	IN	origin_count	number of entries in buffer (non-negative integer)
6	IN	origin_datatype	datatype of each buffer entry (handle)
7	IN	target_rank	rank of target (non-negative integer)
8			
9	IN	target_disp	displacement from start of window to beginning of target buffer (non-negative integer)
10			
11	IN	target_count	number of entries in target buffer (non-negative integer)
12			
13	IN	target_datatype	datatype of each entry in target buffer (handle)
14			
15	IN	op	reduce operation (handle)
16	IN	win	window object (handle)

```

17
18 int MPI_Accumulate(void *origin_addr, int origin_count,
19                  MPI_Datatype origin_datatype, int target_rank,
20                  MPI_Aint target_disp, int target_count,
21                  MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)

```

```

ticket229.2. 22 MPI_Accumulate(origin_addr, origin_count, origin_datatype, target_rank,
ticket-248T. 23 target_disp, target_count, target_datatype, op, win, ierror)
24 BIND(C)
25 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
26 INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
27 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
28 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
29 TYPE(MPI_Op), INTENT(IN) :: op
30 TYPE(MPI_Win), INTENT(IN) :: win
31 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32

```

```

33 MPI_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
34               TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
35 <type> ORIGIN_ADDR(*)
36 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
37 INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
38 TARGET_DATATYPE, OP, WIN, IERROR
39

```

```

40 {void MPI::Win::Accumulate(const void* origin_addr, int origin_count, const
41                          MPI::Datatype& origin_datatype, int target_rank, MPI::Aint
42                          target_disp, int target_count, const MPI::Datatype&
43                          target_datatype, const MPI::Op& op) const(binding deprecated, see
44                          Section 15.2) }
45

```

Accumulate the contents of the origin buffer (as defined by `origin_addr`, `origin_count` and `origin_datatype`) to the buffer specified by arguments `target_count` and `target_datatype`, at offset `target_disp`, in the target window specified by `target_rank` and `win`, using the operation

MPI_WIN_TEST(win, flag)

IN	win	window object (handle)
OUT	flag	success flag (logical)

```
int MPI_Win_test(MPI_Win win, int *flag)
```

```
MPI_Win_test(win, flag, ierror) BIND(C)
  TYPE(MPI_Win), INTENT(IN) :: win
  LOGICAL, INTENT(OUT) :: flag
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_WIN_TEST(WIN, FLAG, IERROR)
  INTEGER WIN, IERROR
  LOGICAL FLAG
```

```
{bool MPI::Win::Test() const(binding deprecated, see Section 15.2) }
```

This is the nonblocking version of **MPI_WIN_WAIT**. It returns `flag = true` if all accesses to the local window by the group to which it was exposed by the corresponding **MPI_WIN_POST** call have been completed as signalled by matching **MPI_WIN_COMPLETE** calls, and `flag = false` otherwise. In the former case **MPI_WIN_WAIT** would have returned immediately. The effect of return of **MPI_WIN_TEST** with `flag = true` is the same as the effect of a return of **MPI_WIN_WAIT**. If `flag = false` is returned, then the call has no visible effect.

MPI_WIN_TEST should be invoked only where **MPI_WIN_WAIT** can be invoked. Once the call has returned `flag = true`, it must not be invoked anew, until the window is posted anew.

Assume that window `win` is associated with a “hidden” communicator `wincomm`, used for communication by the processes of `win`. The rules for matching of post and start calls and for matching complete and wait call can be derived from the rules for matching sends and receives, by considering the following (partial) model implementation.

MPI_WIN_POST(group,0,win) initiate a nonblocking send with tag `tag0` to each process in `group`, using `wincomm`. No need to wait for the completion of these sends.

MPI_WIN_START(group,0,win) initiate a nonblocking receive with tag `tag0` from each process in `group`, using `wincomm`. An RMA access to a window in target process `i` is delayed until the receive from `i` is completed.

MPI_WIN_COMPLETE(win) initiate a nonblocking send with tag `tag1` to each process in the group of the preceding start call. No need to wait for the completion of these sends.

MPI_WIN_WAIT(win) initiate a nonblocking receive with tag `tag1` from each process in the group of the preceding post call. Wait for the completion of all receives.

No races can occur in a correct program: each of the sends matches a unique receive, and vice-versa.

MPI standard, process 0 must return from the complete call after a bounded delay, even if process 1 does not reach any MPI call in this period of time. According to another interpretation, the complete call may block until process 1 reaches the wait call, or reaches another MPI call. The qualitative behavior is the same, under both interpretations, unless a process is caught in an infinite compute loop, in which case the difference may not matter. However, the quantitative expectations are different. Different MPI implementations reflect these different interpretations. While this ambiguity is unfortunate, it does not seem to affect many real codes. The MPI forum decided not to decide which interpretation of the standard is the correct one, since the issue is very contentious, and a decision would have much impact on implementors but less impact on users. (*End of rationale.*)

11.7.3 Registers and Compiler Optimizations

Advice to users. All the material in this section is an advice to users. (*End of advice to users.*)

A coherence problem exists between variables kept in registers and the memory value of these variables. An RMA call may access a variable in memory (or cache), while the up-to-date value of this variable is in register. A get will not return the latest variable value, and a put may be overwritten when the register is stored back in memory.

The problem is illustrated by the following code:

Source of Process 1	Source of Process 2	Executed in Process 2
bbbb = 777	buff = 999	reg_A:=999
call MPI_WIN_FENCE	call MPI_WIN_FENCE	
call MPI_PUT(bbbb		stop appl.thread
into buff of process 2)		buff:=777 in PUT handler
		continue appl.thread
call MPI_WIN_FENCE	call MPI_WIN_FENCE	
	ccc = buff	ccc:=reg_A

In this example, variable `buff` is allocated in the register `reg_A` and therefore `ccc` will have the old value of `buff` and not the new value 777.

This problem, which also afflicts in some cases send/receive communication, is discussed more at length in Section 16.2.16.

MPI implementations will avoid this problem for standard conforming C programs. Many Fortran compilers will avoid this problem, without disabling compiler optimizations. However, in order to avoid register coherence problems in a completely portable manner, users should restrict their use of RMA windows to variables stored in **in modules or COMMON blocks**. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 16.2.10-16.2.20, especially in Sections 16.2.12 and 16.2.13 on pages 581-584 about “Problems Due to Data Copying and Sequence Association with Subscript Triplets” and “Vector Subscripts”, and in Sections 16.2.16 to 16.2.19 on pages 586 to 597 about “Optimization Problems”, “Code Movements and Register Optimization”, “Temporary Data Movements” and “Permanent Data Movements”. Sections “Solutions” to “VOLATILE” on pages 589-594 discuss several solutions for the problem in this example.

For a generalized request, the operation associated with the request is performed by the application; therefore, the application must notify MPI when the operation completes. This is done by making a call to `MPI_GREQUEST_COMPLETE`. MPI maintains the “completion” status of generalized requests. Any other request state has to be maintained by the user.

A new generalized request is started with

`MPI_GREQUEST_START(query_fn, free_fn, cancel_fn, extra_state, request)`

IN	query_fn	callback function invoked when request status is queried (function)
IN	free_fn	callback function invoked when request is freed (function)
IN	cancel_fn	callback function invoked when request is cancelled (function)
IN	extra_state	extra state
OUT	request	generalized request (handle)

```
int MPI_Grequest_start(MPI_Grequest_query_function *query_fn,
                      MPI_Grequest_free_function *free_fn,
                      MPI_Grequest_cancel_function *cancel_fn, void *extra_state,
                      MPI_Request *request)
```

```
MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request,
                  ierror) BIND(C)
    PROCEDURE(MPI_Grequest_query_function) :: query_fn
    PROCEDURE(MPI_Grequest_free_function) :: free_fn
    PROCEDURE(MPI_Grequest_cancel_function) :: cancel_fn
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_GREQUEST_START(QUERY_FN, FREE_FN, CANCEL_FN, EXTRA_STATE, REQUEST,
                  IERROR)
    INTEGER REQUEST, IERROR
    EXTERNAL QUERY_FN, FREE_FN, CANCEL_FN
    INTEGER (KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

```
{static MPI::Grequest
    MPI::Grequest::Start(const MPI::Grequest::Query_function*
                        query_fn, const MPI::Grequest::Free_function* free_fn,
                        const MPI::Grequest::Cancel_function* cancel_fn,
                        void *extra_state) (binding deprecated, see Section 15.2) }
```

Advice to users. Note that a generalized request belongs, in C++, to the class `MPI::Grequest`, which is a derived class of `MPI::Request`. It is of the same type as regular requests, in C and Fortran. (*End of advice to users.*)

The call starts a generalized request and returns a handle to it in `request`.

The syntax and meaning of the callback functions are listed below. All callback functions are passed the `extra_state` argument that was associated with the request by the starting call `MPI_GREQUEST_START`; `extra_state` can be used to maintain user-defined state for the request.

In C, the query function is

```
typedef int MPI_Grequest_query_function(void *extra_state,
                                         MPI_Status *status);
```

in Fortran with the `mpi_f08` module

ABSTRACT INTERFACE

```
SUBROUTINE MPI_Grequest_query_function(extra_state, status, ierror)
BIND(C)
```

```
    TYPE(MPI_Status) :: status
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
    INTEGER :: ierror
```

in Fortran with the `mpi` module and `mpif.h`

```
SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

and in C++

```
{typedef int MPI::Grequest::Query_function(void* extra_state,
                                             MPI::Status& status); (binding deprecated, see Section 15.2)}
```

The `query_fn` function computes the status that should be returned for the generalized request. The status also includes information about successful/unsuccessful cancellation of the request (result to be returned by `MPI_TEST_CANCELLED`).

The `query_fn` callback is invoked by the `MPI_{WAIT|TEST}{ANY|SOME|ALL}` call that completed the generalized request associated with this callback. The callback function is also invoked by calls to `MPI_REQUEST_GET_STATUS`, if the request is complete when the call occurs. In both cases, the callback is passed a reference to the corresponding status variable passed by the user to the MPI call; the status set by the callback function is returned by the MPI call. If the user provided `MPI_STATUS_IGNORE` or `MPI_STATUSES_IGNORE` to the MPI function that causes `query_fn` to be called, then MPI will pass a valid status object to `query_fn`, and this status will be ignored upon return of the callback function. Note that `query_fn` is invoked only after `MPI_GREQUEST_COMPLETE` is called on the request; it may be invoked several times for the same generalized request, e.g., if the user calls `MPI_REQUEST_GET_STATUS` several times for this request. Note also that a call to `MPI_{WAIT|TEST}{SOME|ALL}` may cause multiple invocations of `query_fn` callback functions, one for each generalized request that is completed by the MPI call. The order of these invocations is not specified by MPI.

In C, the free function is

```
typedef int MPI_Grequest_free_function(void *extra_state);
```

in Fortran with the `mpi_f08` module

ABSTRACT INTERFACE

```
SUBROUTINE MPI_Grequest_free_function(extra_state, ierror) BIND(C)
```



```
1      INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
```

```
2      INTEGER :: ierror
```

in Fortran with the `mpi` module and `mpif.h`

```
5      SUBROUTINE GREQUEST_FREE_FUNCTION(EXTRA_STATE, IERROR)
```

```
6          INTEGER IERROR
```

```
7          INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

and in C++

```
10     {typedef int MPI::Grequest::Free_function(void* extra_state); (binding
11                                     deprecated, see Section 15.2)}
```

The `free_fn` function is invoked to clean up user-allocated resources when the generalized request is freed.

The `free_fn` callback is invoked by the `MPI_{WAIT|TEST}{ANY|SOME|ALL}` call that completed the generalized request associated with this callback. `free_fn` is invoked after the call to `query_fn` for the same request. However, if the MPI call completed multiple generalized requests, the order in which `free_fn` callback functions are invoked is not specified by MPI.

The `free_fn` callback is also invoked for generalized requests that are freed by a call to `MPI_REQUEST_FREE` (no call to `WAIT_{WAIT|TEST}{ANY|SOME|ALL}` will occur for such a request). In this case, the callback function will be called either in the MPI call `MPI_REQUEST_FREE(request)`, or in the MPI call `MPI_GREQUEST_COMPLETE(request)`, whichever happens last, i.e., in this case the actual freeing code is executed as soon as both calls `MPI_REQUEST_FREE` and `MPI_GREQUEST_COMPLETE` have occurred. The request is not deallocated until after `free_fn` completes. Note that `free_fn` will be invoked only once per request by a correct program.

Advice to users. Calling `MPI_REQUEST_FREE(request)` will cause the request handle to be set to `MPI_REQUEST_NULL`. This handle to the generalized request is no longer valid. However, user copies of this handle are valid until after `free_fn` completes since MPI does not deallocate the object until then. Since `free_fn` is not called until after `MPI_GREQUEST_COMPLETE`, the user copy of the handle can be used to make this call. Users should note that MPI will deallocate the object after `free_fn` executes. At this point, user copies of the request handle no longer point to a valid request. MPI will not set user copies to `MPI_REQUEST_NULL` in this case, so it is up to the user to avoid accessing this stale handle. This is a special case in which MPI defers deallocating the object until a later time that is known by the user. (*End of advice to users.*)

In C, the cancel function is

```
41     typedef int MPI_Grequest_cancel_function(void *extra_state, int complete);
```

in Fortran with the `mpi_f08` module

```
44     ABSTRACT INTERFACE
```

```
45         SUBROUTINE MPI_Grequest_cancel_function(extra_state, complete, ierror)
```

```
46         BIND(C)
```

```
47             INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
```

```
48             LOGICAL :: complete
```



```

    INTEGER :: ierror
in Fortran with the mpi module and mpif.h
SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR)
    INTEGER IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
    LOGICAL COMPLETE
and in C++
{typedef int MPI::Grequest::Cancel_function(void* extra_state,
    bool complete); (binding deprecated, see Section 15.2)}

```

The `cancel_fn` function is invoked to start the cancelation of a generalized request. It is called by `MPI_CANCEL(request)`. MPI passes `complete=true` to the callback function if `MPI_GREQUEST_COMPLETE` was already called on the request, and `complete=false` otherwise.

All callback functions return an error code. The code is passed back and dealt with as appropriate for the error code by the MPI function that invoked the callback function. For example, if error codes are returned then the error code returned by the callback function will be returned by the MPI function that invoked the callback function. In the case of an `MPI_{WAIT|TEST}{ANY}` call that invokes both `query_fn` and `free_fn`, the MPI call will return the error code returned by the last callback, namely `free_fn`. If one or more of the requests in a call to `MPI_{WAIT|TEST}{SOME|ALL}` failed, then the MPI call will return `MPI_ERR_IN_STATUS`. In such a case, if the MPI call was passed an array of statuses, then MPI will return in each of the statuses that correspond to a completed generalized request the error code returned by the corresponding invocation of its `free_fn` callback function. However, if the MPI function was passed `MPI_STATUSES_IGNORE`, then the individual error codes returned by each callback functions will be lost.

Advice to users. `query_fn` must **not** set the error field of `status` since `query_fn` may be called by `MPI_WAIT` or `MPI_TEST`, in which case the error field of `status` should not change. The MPI library knows the “context” in which `query_fn` is invoked and can decide correctly when to put in the error field of `status` the returned error code. (*End of advice to users.*)

```

MPI_GREQUEST_COMPLETE(request)
    INOUT    request                generalized request (handle)

int MPI_Grequest_complete(MPI_Request request)
MPI_Grequest_complete(request, ierror) BIND(C)
    TYPE(MPI_Request), INTENT(IN) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_GREQUEST_COMPLETE(REQUEST, IERROR)
    INTEGER REQUEST, IERROR

{void MPI::Grequest::Complete() (binding deprecated, see Section 15.2) }

```

num_io_nodes (integer) [SAME]: This hint specifies the number of I/O devices in the system. This hint is most relevant when the file is created.

striping_factor (integer) [SAME]: This hint specifies the number of I/O devices that the file should be striped across, and is relevant only when the file is created.

striping_unit (integer) [SAME]: This hint specifies the suggested striping unit to be used for this file. The striping unit is the amount of consecutive data assigned to one I/O device before progressing to the next device, when striping across a number of devices. It is expressed in bytes. This hint is relevant only when the file is created.

13.3 File Views

MPI_FILE_SET_VIEW(fh, disp, etype, filetype, datarep, info)

INOUT	fh	file handle (handle)
IN	disp	displacement (integer)
IN	etype	elementary datatype (handle)
IN	filetype	filetype (handle)
IN	datarep	data representation (string)
IN	info	info object (handle)

int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype, MPI_Datatype filetype, char *datarep, MPI_Info info)

```

MPI_File_set_view(fh, disp, etype, filetype, datarep, info, ierror) BIND(C)
  TYPE(MPI_File), INTENT(IN) :: fh
  INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: disp
  TYPE(MPI_Datatype), INTENT(IN) :: etype, filetype
  CHARACTER(LEN=*), INTENT(IN) :: datarep
  TYPE(MPI_Info), INTENT(IN) :: info
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR)
  INTEGER FH, ETYPE, FILETYPE, INFO, IERROR
  CHARACTER*(*) DATAREP
  INTEGER(KIND=MPI_OFFSET_KIND) DISP

```

```

{void MPI::File::Set_view(MPI::Offset disp, const MPI::Datatype& etype,
  const MPI::Datatype& filetype, const char* datarep,
  const MPI::Info& info) (binding deprecated, see Section 15.2) }

```

The **MPI_FILE_SET_VIEW** routine changes the process's view of the data in the file. The start of the view is set to **disp**; the type of data is set to **etype**; the distribution of data to processes is set to **filetype**; and the representation of data in the file is set to **datarep**. In addition, **MPI_FILE_SET_VIEW** resets the individual file pointers and the shared file pointer to zero. **MPI_FILE_SET_VIEW** is collective; the values for **datarep** and the extents

Advice to users. In order to ensure interoperability in a heterogeneous environment, additional restrictions must be observed when constructing the **etype** (see Section 13.5, page 496). (*End of advice to users.*)

A filetype is either a single etype or a derived MPI datatype constructed from multiple instances of the same etype. In addition, the extent of any hole in the filetype must be a multiple of the etype’s extent. These displacements are not required to be distinct, but they cannot be negative, and they must be monotonically nondecreasing.

If the file is opened for writing, neither the **etype** nor the **filetype** is permitted to contain overlapping regions. This restriction is equivalent to the “datatype used in a receive cannot specify overlapping regions” restriction for communication. Note that filetypes from different processes may still overlap each other.

If filetype has holes in it, then the data in the holes is inaccessible to the calling process. However, the **disp**, **etype** and **filetype** arguments can be changed via future calls to **MPI_FILE_SET_VIEW** to access a different part of the file.

It is erroneous to use absolute addresses in the construction of the **etype** and **filetype**.

The **info** argument is used to provide information regarding file access patterns and file system specifics to direct optimization (see Section 13.2.8, page 461). The constant **MPI_INFO_NULL** refers to the null info and can be used when no info needs to be specified.

The **datarep** argument is a string that specifies the representation of data in the file. See the file interoperability section (Section 13.5, page 496) for details and a discussion of valid values.

The user is responsible for ensuring that all nonblocking requests and split collective operations on **fh** have been completed before calling **MPI_FILE_SET_VIEW**—otherwise, the call to **MPI_FILE_SET_VIEW** is erroneous.

MPI_FILE_GET_VIEW(**fh**, **disp**, **etype**, **filetype**, **datarep**)

IN	fh	file handle (handle)
OUT	disp	displacement (integer)
OUT	etype	elementary datatype (handle)
OUT	filetype	filetype (handle)
OUT	datarep	data representation (string)

```
int MPI_File_get_view(MPI_File fh, MPI_Offset *disp, MPI_Datatype *etype,
                     MPI_Datatype *filetype, char *datarep)
```

```
MPI_File_get_view(fh, disp, etype, filetype, datarep, ierror) BIND(C)
  TYPE(MPI_File), INTENT(IN) :: fh
  INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp
  TYPE(MPI_Datatype), INTENT(OUT) :: etype, filetype
  CHARACTER(LEN=*), INTENT(OUT) :: datarep
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_FILE_GET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, IERROR)
  INTEGER FH, ETYPE, FILETYPE, IERROR
  CHARACTER*(*) DATAREP
```

The split collective routines support a restricted form of “nonblocking” operations for collective data access (see Section 13.4.5, page 488).

Coordination

Every noncollective data access routine `MPI_FILE_XXX` has a collective counterpart. For most routines, this counterpart is `MPI_FILE_XXX_ALL` or a pair of `MPI_FILE_XXX_BEGIN` and `MPI_FILE_XXX_END`. The counterparts to the `MPI_FILE_XXX_SHARED` routines are `MPI_FILE_XXX_ORDERED`.

The completion of a noncollective call only depends on the activity of the calling process. However, the completion of a collective call (which must be called by all members of the process group) may depend on the activity of the other processes participating in the collective call. See Section 13.6.4, page 510, for rules on semantics of collective calls.

Collective operations may perform much better than their noncollective counterparts, as global data accesses have significant potential for automatic optimization.

Data Access Conventions

Data is moved between files and processes by calling read and write routines. Read routines move data from a file into memory. Write routines move data from memory into a file. The file is designated by a file handle, `fh`. The location of the file data is specified by an offset into the current view. The data in memory is specified by a triple: `buf`, `count`, and `datatype`. Upon completion, the amount of data accessed by the calling process is returned in a `status`.

An offset designates the starting position in the file for an access. The offset is always in `etype` units relative to the current view. Explicit offset routines pass `offset` as an argument (negative values are erroneous). The file pointer routines use implicit offsets maintained by MPI.

A data access routine attempts to transfer (read or write) `count` data items of type `datatype` between the user’s buffer `buf` and the file. The `datatype` passed to the routine must be a committed datatype. The layout of data in memory corresponding to `buf`, `count`, `datatype` is interpreted the same way as in MPI communication functions; see Section 3.2.2 on page 29 and Section 4.1.11 on page 113. The data is accessed from those parts of the file specified by the current view (Section 13.3, page 464). The type signature of `datatype` must match the type signature of some number of contiguous copies of the `etype` of the current view. As in a receive, it is erroneous to specify a `datatype` for reading that contains overlapping regions (areas of memory which would be stored into more than once).

The nonblocking data access routines indicate that MPI can start a data access and associate a request handle, `request`, with the I/O operation. Nonblocking operations are completed via `MPI_TEST`, `MPI_WAIT`, or any of their variants.

Data access operations, when completed, return the amount of data accessed in `status`.

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in — Sections 16.2.10-16.2.20, especially in Sections 16.2.12 and 16.2.13 on pages 581-584 about “Problems Due to Data Copying and Sequence Association with Subscript Triplets” and “Vector Subscripts”, and in Sections 16.2.16 to 16.2.19 on pages 586 to 597 about “Optimization Problems”, “Code Movements and Register Optimization”, “Temporary Data Movements” and “Permanent Data Movements”. (*End of advice to users.*)

ticket229.1.
ticket-248T.

```

MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror)
    BIND(C)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    TYPE(*), DIMENSION(..), INTENT(IN) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_FILE_WRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

{void MPI::File::Write_at_all(MPI::Offset offset, const void* buf,
    int count, const MPI::Datatype& datatype,
    MPI::Status& status) (binding deprecated, see Section 15.2) }

{void MPI::File::Write_at_all(MPI::Offset offset, const void* buf,
    int count, const MPI::Datatype& datatype) (binding deprecated, see
    Section 15.2) }
```

MPI_FILE_WRITE_AT_ALL is a collective version of the blocking
MPI_FILE_WRITE_AT interface.

```

MPI_FILE_IREAD_AT(fh, offset, buf, count, datatype, request)

IN      fh                file handle (handle)
IN      offset            file offset (integer)
OUT     buf               initial address of buffer (choice)
IN      count             number of elements in buffer (integer)
IN      datatype          datatype of each buffer element (handle)
OUT     request           request object (handle)

int MPI_File_iread_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
    MPI_Datatype datatype, MPI_Request *request)
```

```

MPI_File_iread_at(fh, offset, buf, count, datatype, request, ierror)
    BIND(C)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

ticket-248T.

`MPI_FILE_IWRITE_SHARED(fh, buf, count, datatype, request)`

INOUT	fh	file handle (handle)
IN	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)
OUT	request	request object (handle)

```
int MPI_File_iwrite_shared(MPI_File fh, void *buf, int count,
                          MPI_Datatype datatype, MPI_Request *request)
```

```
MPI_File_iwrite_shared(fh, buf, count, datatype, request, ierror) BIND(C)
  TYPE(MPI_File), INTENT(IN) :: fh
  TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
  INTEGER, INTENT(IN) :: count
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Request), INTENT(OUT) :: request
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_FILE_IWRITE_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
  <type> BUF(*)
  INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
```

```
{MPI::Request MPI::File::Iwrite_shared(const void* buf, int count,
                                       const MPI::Datatype& datatype) (binding deprecated, see Section 15.2)
}
```

`MPI_FILE_IWRITE_SHARED` is a nonblocking version of the `MPI_FILE_WRITE_SHARED` interface.

Collective Operations

The semantics of a collective access using a shared file pointer is that the accesses to the file will be in the order determined by the ranks of the processes within the group. For each process, the location in the file at which data is accessed is the position at which the shared file pointer would be after all processes whose ranks within the group less than that of this process had accessed their data. In addition, in order to prevent subsequent shared offset accesses by the same processes from interfering with this collective access, the call might return only after all the processes within the group have initiated their accesses. When the call returns, the shared file pointer points to the next etype accessible, according to the file view used by all processes, after the last etype requested.

Advice to users. There may be some programs in which all processes in the group need to access the file using the shared file pointer, but the program may not *require* that data be accessed in order of process rank. In such programs, using the shared ordered routines (e.g., `MPI_FILE_WRITE_ORDERED` rather than `MPI_FILE_WRITE_SHARED`) may enable an implementation to optimize access, improving performance. (*End of advice to users.*)

an end routine. The begin routine begins the operation, much like a nonblocking data access (e.g., `MPI_FILE_IREAD`). The end routine completes the operation, much like the matching test or wait (e.g., `MPI_WAIT`). As with nonblocking data access operations, the user must not use the buffer passed to a begin routine while the routine is outstanding; the operation must be completed with an end routine before it is safe to free buffers, etc.

Split collective data access operations on a file handle `fh` are subject to the semantic rules given below.

- On any MPI process, each file handle may have at most one active split collective operation at any time.
- Begin calls are collective over the group of processes that participated in the collective open and follow the ordering rules for collective calls.
- End calls are collective over the group of processes that participated in the collective open and follow the ordering rules for collective calls. Each end call matches the preceding begin call for the same collective operation. When an “end” call is made, exactly one unmatched “begin” call for the same operation must precede it.
- An implementation is free to implement any split collective data access routine using the corresponding blocking collective routine when either the begin call (e.g., `MPI_FILE_READ_ALL_BEGIN`) or the end call (e.g., `MPI_FILE_READ_ALL_END`) is issued. The begin and end calls are provided to allow the user and MPI implementation to optimize the collective operation.
- Split collective operations do not match the corresponding regular collective operation. For example, in a single collective read operation, an `MPI_FILE_READ_ALL` on one process does not match an `MPI_FILE_READ_ALL_BEGIN`/`MPI_FILE_READ_ALL_END` pair on another process.
- Split collective routines must specify a buffer in both the begin and end routines. By specifying the buffer that receives data in the end routine, we can avoid the problems described in “A Problem with Code Movements and Register Optimization,” [Section 16.2.17 on page 587](#), but not all of the problems described in [Section 16.2.16 on page 586](#).
- No collective I/O operations are permitted on a file handle concurrently with a split collective access on that file handle (i.e., between the begin and end of the access). That is

```
MPI_File_read_all_begin(fh, ...);
...
MPI_File_read_all(fh, ...);
...
MPI_File_read_all_end(fh, ...);
```

is erroneous.

- In a multithreaded implementation, any split collective begin and end operation called by a process must be called from the same thread. This restriction is made to simplify

Advice to users. The type MPI_PACKED is treated as bytes and is not converted. The user should be aware that MPI_PACK has the option of placing a header in the beginning of the pack buffer. (*End of advice to users.*)

The size of the predefined datatypes returned from MPI_TYPE_CREATE_F90_REAL, MPI_TYPE_CREATE_F90_COMPLEX, and MPI_TYPE_CREATE_F90_INTEGER are defined in Section 16.2.9, page 574.

Advice to implementors. When converting a larger size integer to a smaller size integer, only the less significant bytes are moved. Care must be taken to preserve the sign bit value. This allows no conversion errors if the data range is within the range of the smaller size integer. (*End of advice to implementors.*)

Table 13.2 specifies the sizes of predefined datatypes in “external32” format.

13.5.3 User-Defined Data Representations

There are two situations that cannot be handled by the required representations:

1. a user wants to write a file in a representation unknown to the implementation, and
2. a user wants to read a file written in a representation unknown to the implementation.

User-defined data representations allow the user to insert a third party converter into the I/O stream to do the data representation conversion.

```
MPI_REGISTER_DATAREP(datarep, read_conversion_fn, write_conversion_fn,
                     dtype_file_extent_fn, extra_state)
```

IN	datarep	data representation identifier (string)
IN	read_conversion_fn	function invoked to convert from file representation to native representation (function)
IN	write_conversion_fn	function invoked to convert from native representation to file representation (function)
IN	dtype_file_extent_fn	function invoked to get the extent of a datatype as represented in the file (function)
IN	extra_state	extra state

```
int MPI_Register_datarep(char *datarep,
                        MPI_Datarep_conversion_function *read_conversion_fn,
                        MPI_Datarep_conversion_function *write_conversion_fn,
                        MPI_Datarep_extent_function *dtype_file_extent_fn,
                        void *extra_state)
```

```
MPI_Register_datarep(datarep, read_conversion_fn, write_conversion_fn,
                     dtype_file_extent_fn, extra_state, ierror) BIND(C)
CHARACTER(LEN=*) , INTENT(IN) :: datarep
PROCEDURE(MPI_Datarep_conversion_function) :: read_conversion_fn
PROCEDURE(MPI_Datarep_conversion_function) :: write_conversion_fn
```

ticket-248T.


```

PROCEDURE(MPI_Datarep_extent_function) :: dtype_file_extent_fn
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_REGISTER_DATAREP(DATAREP, READ_CONVERSION_FN, WRITE_CONVERSION_FN,
                     DTYPE_FILE_EXTENT_FN, EXTRA_STATE, IERROR)
CHARACTER*(*) DATAREP
EXTERNAL READ_CONVERSION_FN, WRITE_CONVERSION_FN, DTYPE_FILE_EXTENT_FN
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
INTEGER IERROR

{void MPI::Register_datarep(const char* datarep,
                           MPI::Datarep_conversion_function* read_conversion_fn,
                           MPI::Datarep_conversion_function* write_conversion_fn,
                           MPI::Datarep_extent_function* dtype_file_extent_fn,
                           void* extra_state) (binding deprecated, see Section 15.2) }
```

The call associates `read_conversion_fn`, `write_conversion_fn`, and `dtype_file_extent_fn` with the data representation identifier `datarep`. `datarep` can then be used as an argument to `MPI_FILE_SET_VIEW`, causing subsequent data access operations to call the conversion functions to convert all data items accessed between file data representation and native representation. `MPI_REGISTER_DATAREP` is a local operation and only registers the data representation for the calling MPI process. If `datarep` is already defined, an error in the error class `MPI_ERR_DUP_DATAREP` is raised using the default file error handler (see Section 13.7, page 516). The length of a data representation string is limited to the value of `MPI_MAX_DATAREP_STRING`. `MPI_MAX_DATAREP_STRING` must have a value of at least 64. No routines are provided to delete data representations and free the associated resources; it is not expected that an application will generate them in significant numbers.

Extent Callback

```

typedef int MPI_Datarep_extent_function(MPI_Datatype datatype,
                                       MPI_Aint *file_extent, void *extra_state);

ABSTRACT INTERFACE
  SUBROUTINE MPI_Datarep_extent_function(datatype, extent, extra_state,
    ierror) BIND(C)
    TYPE(MPI_Datatype) :: datatype
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extent, extra_state
    INTEGER :: ierror

  SUBROUTINE DATAREP_EXTENT_FUNCTION(DATATYPE, EXTENT, EXTRA_STATE, IERROR)
    INTEGER DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, EXTRA_STATE

  {typedef void MPI::Datarep_extent_function(const MPI::Datatype& datatype,
    MPI::Aint& file_extent, void* extra_state); (binding deprecated,
    see Section 15.2)}
```

The function `dtype_file_extent_fn` must return, in `file_extent`, the number of bytes required to store `datatype` in the file representation. The function is passed, in `extra_state`,

Collective file operations are collective over a dup of the communicator used to open the file—this duplicate communicator is implicitly specified via the file handle argument. Different processes can pass different values for other arguments of a collective routine unless specified otherwise.

13.6.5 Type Matching

The type matching rules for I/O mimic the type matching rules for communication with one exception: if `etype` is `MPI_BYTE`, then this matches any `datatype` in a data access operation. In general, the `etype` of data items written must match the `etype` used to read the items, and for each data access operation, the current `etype` must also match the type declaration of the data access buffer.

Advice to users. In most cases, use of `MPI_BYTE` as a wild card will defeat the file interoperability features of MPI. File interoperability can only perform automatic conversion between heterogeneous data representations when the exact datatypes accessed are explicitly specified. (*End of advice to users.*)

13.6.6 Miscellaneous Clarifications

Once an I/O routine completes, it is safe to free any opaque objects passed as arguments to that routine. For example, the `comm` and `info` used in an `MPI_FILE_OPEN`, or the `etype` and `filetype` used in an `MPI_FILE_SET_VIEW`, can be freed without affecting access to the file. Note that for nonblocking routines and split collective operations, the operation must be completed before it is safe to reuse data buffers passed as arguments.

As in communication, datatypes must be committed before they can be used in file manipulation or data access operations. For example, the `etype` and `filetype` must be committed before calling `MPI_FILE_SET_VIEW`, and the `datatype` must be committed before calling `MPI_FILE_READ` or `MPI_FILE_WRITE`.

13.6.7 MPI_Offset Type

`MPI_Offset` is an integer type of size sufficient to represent the size (in bytes) of the largest file supported by MPI. Displacements and offsets are always specified as values of type `MPI_Offset`.

In Fortran, the corresponding integer is an integer **with kind parameter** `MPI_OFFSET_KIND`, **which is defined in the `mpi_f08` module, the `mpi` module and the `mpif.h` include file.**

In Fortran 77 environments that do not support `KIND` parameters, `MPI_Offset` arguments should be declared as an `INTEGER` of suitable size. The language interoperability implications for `MPI_Offset` are similar to those for addresses (see Section 16.3, page 600).

13.6.8 Logical vs. Physical File Layout

MPI specifies how the data should be laid out in a virtual file structure (the view), not how that file structure is to be stored on one or more disks. Specification of the physical file structure was avoided because it is expected that the mapping of files to disks will be system specific, and any specific control over file layout would therefore restrict program portability. However, there are still cases where some information may be necessary to

Chapter 14

Profiling Interface

14.1 Requirements

To meet the requirements for the MPI profiling interface, an implementation of the MPI functions *must*

1. provide a mechanism through which all of the MPI defined functions, except those allowed as macros (See Section 2.6.5), may be accessed with a name shift. This requires, in C and Fortran, an alternate entry point name, with the prefix PMPI_ for each MPI function in each provided language binding and language support method. The profiling interface in C++ is described in Section 16.1.10. For routines implemented as macros, it is still required that the PMPI_ version be supplied and work as expected, but it is not possible to replace at link time the MPI_ version with a user-defined version.

For Fortran, the different support methods cause several linker names. Therefore, several profiling routines (with these linker names) are needed for each Fortran MPI routine, as described in Section 16.2.5 on page 560.

2. ensure that those MPI functions that are not replaced may still be linked into an executable image without causing name clashes.
3. document the implementation of different language bindings of the MPI interface if they are layered on top of each other, so that the profiler developer knows whether she must implement the profile interface for each binding, or can economize by implementing it only for the lowest level routines.
4. where the implementation of different language bindings is done through a layered approach (e.g., the Fortran binding is a set of “wrapper” functions that call the C implementation), ensure that these wrapper functions are separable from the rest of the library.

This separability is necessary to allow a separate profiling library to be correctly implemented, since (at least with Unix linker semantics) the profiling library must contain these wrapper functions if it is to perform as expected. This requirement allows the person who builds the profiling library to extract these functions from the original MPI library and add them into the profiling library without bringing along any other unnecessary code.

Linker Oddities

The Unix linker traditionally operates in one **pass**: the effect of this is that functions from libraries are only included in the image if they are needed at the time the library is scanned. When combined with weak symbols, or multiple definitions of the same function, this can cause odd (and unexpected) effects.

Consider, for instance, an implementation of MPI in which the Fortran binding is achieved by using wrapper functions on top of the C implementation. The author of the profile library then assumes that it is reasonable only to provide profile functions for the C binding, since Fortran will eventually call these, and the cost of the wrappers is assumed to be small. However, if the wrapper functions are not in the profiling library, then none of the profiled entry points will be undefined when the profiling library is called. Therefore none of the profiling code will be included in the image. When the standard MPI library is scanned, the Fortran wrappers will be resolved, and will also pull in the base versions of the MPI functions. The overall effect is that the code will link successfully, but will not be profiled.

To overcome this we must ensure that the Fortran wrapper functions are included in the profiling version of the library. We ensure that this is possible by requiring that these be separable from the rest of the base MPI library. This allows them to be **ared** out of the base library and into the profiling one.

Fortran Support Methods

The different Fortran support methods and possible options for the support of subarrays (depending on whether the compiler can support **TYPE(*)**, **DIMENSION(..)** choice buffers) imply different linker names for the same Fortran MPI routine. The rules and implications for the profiling interface are described in Section 16.2.5 on page 560.

14.5 Multiple Levels of Interception

The scheme given here does not directly support the nesting of profiling functions, since it provides only a single alternative name for each MPI function. Consideration was given to an implementation that would allow multiple levels of call interception, however we were unable to construct an implementation of this that did not have the following disadvantages

- assuming a particular implementation language,
- imposing a run time cost even when no profiling was taking place.

Since one of the objectives of MPI is to permit efficient, low latency implementations, and it is not the business of a standard to require a particular implementation language, we decided to accept the scheme outlined above.

Note, however, that it is possible to use the scheme above to implement a multi-level system, since the function called by the user may call many different profiling functions before calling the underlying MPI function. This capability has been demonstrated in the P^N MPI tool infrastructure [46].

Chapter 15

Deprecated Functions

15.1 Deprecated since MPI-2.0

The following function is deprecated and is superseded by `MPI_TYPE_CREATE_HVECTOR` in MPI-2.0. The language independent definition and the C binding of the deprecated function is the same as of the new function, except of the function name. Only the Fortran language binding is different.

`MPI_TYPE_HVECTOR(count, blocklength, stride, oldtype, newtype)`

IN	count	number of blocks (non-negative integer)
IN	blocklength	number of elements in each block (non-negative integer)
IN	stride	number of bytes between start of each block (integer)
IN	oldtype	old datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPI_Type_hvector(int count, int blocklength, MPI_Aint stride,  
                    MPI_Datatype oldtype, MPI_Datatype *newtype)
```

For this routine, an interface within the `mpi_f08` module was never defined.

```
MPI_TYPE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)  
INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR
```

The following function is deprecated and is superseded by `MPI_TYPE_CREATE_HINDEXED` in MPI-2.0. The language independent definition and the C binding of the deprecated function is the same as of the new function, except of the function name. Only the Fortran language binding is different.

```

1 MPI_TYPE_HINDEXED( count, array_of_blocklengths, array_of_displacements, oldtype, new-
2     type)
3
4     IN          count          number of blocks – also number of entries in
5                               array_of_displacements and array_of_blocklengths (non-
6                               negative integer)
7
8     IN          array_of_blocklengths  number of elements in each block (array of non-negative
9                               integers)
10
11    IN          array_of_displacements  byte displacement of each block (array of integer)
12
13    IN          oldtype              old datatype (handle)
14
15    OUT         newtype              new datatype (handle)

```

```

16 int MPI_Type_hindexed(int count, int *array_of_blocklengths,
17     MPI_Aint *array_of_displacements, MPI_Datatype oldtype,
18     MPI_Datatype *newtype)

```

For this routine, an interface within the `mpi_f08` module was never defined.

```

19 MPI_TYPE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
20     OLDTYPE, NEWTYPE, IERROR)
21     INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
22     OLDTYPE, NEWTYPE, IERROR

```

The following function is deprecated and is superseded by `MPI_TYPE_CREATE_STRUCT` in MPI-2.0. The language independent definition and the C binding of the deprecated function is the same as of the new function, except of the function name. Only the Fortran language binding is different.

```

29 MPI_TYPE_STRUCT(count, array_of_blocklengths, array_of_displacements, array_of_types,
30     newtype)
31
32    IN          count          number of blocks (integer) (non-negative integer) –
33                               also number of entries in arrays array_of_types,
34                               array_of_displacements and array_of_blocklengths
35
36    IN          array_of_blocklength  number of elements in each block (array of non-negative
37                               integer)
38
39    IN          array_of_displacements  byte displacement of each block (array of integer)
40
41    IN          array_of_types        type of elements in each block (array of handles to
42                               datatype objects)
43
44    OUT         newtype              new datatype (handle)

```

```

45 int MPI_Type_struct(int count, int *array_of_blocklengths,
46     MPI_Aint *array_of_displacements,
47     MPI_Datatype *array_of_types, MPI_Datatype *newtype)

```

For this routine, an interface within the `mpi_f08` module was never defined.

ticket229.1.

For this routine, an interface within the `mpi_f08` module was never defined.

```
MPI_TYPE_LB( DATATYPE, DISPLACEMENT, IERROR)
    INTEGER DATATYPE, DISPLACEMENT, IERROR
```

```
MPI_TYPE_UB( datatype, displacement)
```

IN	datatype	datatype (handle)
OUT	displacement	displacement of upper bound from origin, in bytes (integer)

```
int MPI_Type_ub(MPI_Datatype datatype, MPI_Aint* displacement)
```

For this routine, an interface within the `mpi_f08` module was never defined.

```
MPI_TYPE_UB( DATATYPE, DISPLACEMENT, IERROR)
    INTEGER DATATYPE, DISPLACEMENT, IERROR
```

The following function is deprecated and is superseded by `MPI_COMM_CREATE_KEYVAL` in MPI-2.0. The language independent definition of the deprecated function is the same as that of the new function, except for the function name and a different behavior in the C/Fortran language interoperability, see Section 16.3.7 on page 610. The language bindings are modified.

```
MPI_KEYVAL_CREATE(copy_fn, delete_fn, keyval, extra_state)
```

IN	copy_fn	Copy callback function for keyval
IN	delete_fn	Delete callback function for keyval
OUT	keyval	key value for future access (integer)
IN	extra_state	Extra state for callback functions

```
int MPI_Keyval_create(MPI_Copy_function *copy_fn, MPI_Delete_function
    *delete_fn, int *keyval, void* extra_state)
```

For this routine, an interface within the `mpi_f08` module was never defined.

```
MPI_KEYVAL_CREATE(COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR)
    EXTERNAL COPY_FN, DELETE_FN
    INTEGER KEYVAL, EXTRA_STATE, IERROR
```

The `copy_fn` function is invoked when a communicator is duplicated by `MPI_COMM_DUP`. `copy_fn` should be of type `MPI_Copy_function`, which is defined as follows:

```
typedef int MPI_Copy_function(MPI_Comm oldcomm, int keyval,
    void *extra_state, void *attribute_val_in,
    void *attribute_val_out, int *flag)
```

A Fortran declaration for such a function is as follows:

For this routine, an interface within the `mpi_f08` module was never defined.

```
SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
    ATTRIBUTE_VAL_OUT, FLAG, IERR)
    INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
    ATTRIBUTE_VAL_OUT, IERR
    LOGICAL FLAG
```

`copy_fn` may be specified as `MPI_NULL_COPY_FN` or `MPI_DUP_FN` from either C or FORTRAN; `MPI_NULL_COPY_FN` is a function that does nothing other than returning `flag = 0` and `MPI_SUCCESS`. `MPI_DUP_FN` is a simple-minded copy function that sets `flag = 1`, returns the value of `attribute_val_in` in `attribute_val_out`, and returns `MPI_SUCCESS`. Note that `MPI_NULL_COPY_FN` and `MPI_DUP_FN` are also deprecated.

Analogous to `copy_fn` is a callback deletion function, defined as follows. The `delete_fn` function is invoked when a communicator is deleted by `MPI_COMM_FREE` or when a call is made explicitly to `MPI_ATTR_DELETE`. `delete_fn` should be of type `MPI_Delete_function`, which is defined as follows:

```
typedef int MPI_Delete_function(MPI_Comm comm, int keyval,
    void *attribute_val, void *extra_state);
```

A Fortran declaration for such a function is as follows:

For this routine, an interface within the `mpi_f08` module was never defined.

```
SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR)
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR
```

`delete_fn` may be specified as `MPI_NULL_DELETE_FN` from either C or FORTRAN; `MPI_NULL_DELETE_FN` is a function that does nothing, other than returning `MPI_SUCCESS`. Note that `MPI_NULL_DELETE_FN` is also deprecated.

The following function is deprecated and is superseded by `MPI_COMM_FREE_KEYVAL` in MPI-2.0. The language independent definition of the deprecated function is the same as of the new function, except of the function name. The language bindings are modified.

```
MPI_KEYVAL_FREE(keyval)
```

INOUT	keyval	Frees the integer key value (integer)
-------	--------	---------------------------------------

```
int MPI_Keyval_free(int *keyval)
```

For this routine, an interface within the `mpi_f08` module was never defined.

```
MPI_KEYVAL_FREE(KEYVAL, IERROR)
    INTEGER KEYVAL, IERROR
```

The following function is deprecated and is superseded by `MPI_COMM_SET_ATTR` in MPI-2.0. The language independent definition of the deprecated function is the same as of the new function, except of the function name. The language bindings are modified.

INTEGER COMM, KEYVAL, IERROR

The following function is deprecated and is superseded by MPI_COMM_CREATE_ERRHANDLER in MPI-2.0. The language independent definition of the deprecated function is the same as of the new function, except of the function name. The language bindings are modified.

MPI_ERRHANDLER_CREATE(*handler_fn*, errhandler)

IN [ticket252-W.]*handler_fn* user defined error handling procedure
OUT errhandler MPI error handler (handle)

int MPI_Errhandler_create(MPI_Handler_function **handler_fn*,
MPI_Errhandler *errhandler)

For this routine, an interface within the *mpi_f08* module was never defined.

MPI_ERRHANDLER_CREATE(*HANDLER_FN*, ERRHANDLER, IERROR)
EXTERNAL *HANDLER_FN*
INTEGER ERRHANDLER, IERROR

Register the user routine *handler_fn* for use as an MPI exception handler. Returns in errhandler a handle to the registered exception handler.

In the C language, the user routine should be a C function of type MPI_Handler_function, which is defined as:

typedef void (MPI_Handler_function)(MPI_Comm *, int *, ...);

The first argument is the communicator in use, the second is the error code to be returned.

In the Fortran language, the user routine should be of the form:

SUBROUTINE HANDLER_FUNCTION(COMM, ERROR_CODE)
INTEGER COMM, ERROR_CODE

The following function is deprecated and is superseded by MPI_COMM_SET_ERRHANDLER in MPI-2.0. The language independent definition of the deprecated function is the same as of the new function, except of the function name. The language bindings are modified.

MPI_ERRHANDLER_SET(comm, errhandler)

INOUT comm communicator to set the error handler for (handle)
IN errhandler new MPI error handler for communicator (handle)

int MPI_Errhandler_set(MPI_Comm comm, MPI_Errhandler errhandler)

For this routine, an interface within the *mpi_f08* module was never defined.

MPI_ERRHANDLER_SET(COMM, ERRHANDLER, IERROR)

15.3 Deprecated since MPI-3.0

1
2 ticket245-Q.
3 ticket229.1.
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48

MPI: A Message-Passing Interface Standard

Version **3.0**

(Draft, with MPI 3 Nonblocking Collectives

and new Fortran 2008 Interface)

Unofficial, for comment only

Message Passing Interface Forum

December 16, 2011

ticket0.

ticket229-A.

13.6.4	Collective File Operations	510
13.6.5	Type Matching	511
13.6.6	Miscellaneous Clarifications	511
13.6.7	MPI_Offset Type	511
13.6.8	Logical vs. Physical File Layout	511
13.6.9	File Size	512
13.6.10	Examples	512
	Asynchronous I/O	515
13.7	I/O Error Handling	516
13.8	I/O Error Classes	517
13.9	Examples	517
13.9.1	Double Buffering with Split Collective I/O	517
13.9.2	Subarray Filetype Constructor	520
14	Profiling Interface	523
14.1	Requirements	523
14.2	Discussion	524
14.3	Logic of the Design	524
14.3.1	Miscellaneous Control of Profiling	524
14.4	Examples	525
14.4.1	Profiler Implementation	525
14.4.2	MPI Library Implementation	526
	Systems with Weak Symbols	526
	Systems Without Weak Symbols	526
14.4.3	Complications	527
	Multiple Counting	527
	Linker Oddities	528
	Fortran Support Methods	528
14.5	Multiple Levels of Interception	528
15	Deprecated Functions	529
15.1	Deprecated since MPI-2.0	529
15.2	Deprecated since MPI-2.2	536
15.3	Deprecated since MPI-3.0	537
16	Language Bindings	539
16.1	C++	539
16.1.1	Overview	539
16.1.2	Design	539
16.1.3	C++ Classes for MPI	540
16.1.4	Class Member Functions for MPI	540
16.1.5	Semantics	541
16.1.6	C++ Datatypes	543
16.1.7	Communicators	546
16.1.8	Exceptions	548
16.1.9	Mixed-Language Operability	549
16.1.10	Profiling	549
16.2	Fortran Support	552

16.2.1	Overview	552	
16.2.2	Fortran Support Through the <code>mpi_f08</code> Module	553	ticket230-B.
16.2.3	Fortran Support Through the <code>mpi</code> Module	556	ticket230-B.
16.2.4	Fortran Support Through the <code>mpif.h</code> Include File	558	ticket230-B.
16.2.5	Interface Specifications, Linker Names and the Profiling Interface	560	ticket230-B.
16.2.6	MPI for Different Fortran Standard Versions	564	ticket230-B.
16.2.7	Requirements on Fortran Compilers	568	ticket247-S.
16.2.8	Additional Support for Fortran Register-Memory-Synchronization	569	ticket247-S.
16.2.9	Additional Support for Fortran Numeric Intrinsic Types	570	ticket230-B.
	Parameterized Datatypes with Specified Precision and Exponent Range	571	ticket238-J.
	Support for Size-specific MPI Datatypes	575	
	Communication With Size-specific Types	577	
16.2.10	Problems With Fortran Bindings for MPI	579	
16.2.11	Problems Due to Strong Typing	580	
16.2.12	Problems Due to Data Copying and Sequence Association with Subscript Triplets	581	ticket236-H.
16.2.13	Problems Due to Data Copying and Sequence Association with Vector Subscripts	584	ticket236-H.
16.2.14	Special Constants	584	
16.2.15	Fortran Derived Types	584	
16.2.16	Optimization Problems, an Overview	586	
16.2.17	Problems with Code Movement and Register Optimization	587	ticket238-J.
	Nonblocking operations	587	
	One-sided communication	588	
	MPI_BOTTOM and combining independent variables in datatypes	588	
	Solutions	589	ticket238-J.
	The Fortran ASYNCHRONOUS attribute	591	ticket238-J.
	Calling MPI_F_SYNC_REG	592	ticket238-J.
	A user defined routine instead of MPI_F_SYNC_REG	593	
	Module variables and COMMON blocks	594	
	The (poorly performing) Fortran VOLATILE attribute	594	
	The Fortran TARGET attribute	594	ticket238-J.
16.2.18	Temporary Data Movement and Temporary Memory Modification	594	ticket238-J.
16.2.19	Permanent Data Movement	597	ticket238-J.
16.2.20	Comparison with C	597	
16.3	Language Interoperability	600	
16.3.1	Introduction	600	
16.3.2	Assumptions	600	
16.3.3	Initialization	600	
16.3.4	Transfer of Handles	601	
16.3.5	Status	604	
16.3.6	MPI Opaque Objects	607	
	Datatypes	607	
	Callback Functions	609	
	Error Handlers	609	
	Reduce Operations	609	
	Addresses	609	
16.3.7	Attributes	610	

Example 16.10 `mpi_profile.cc`, to be compiled into `libmpi.a`.

```
int MPI::Comm::Get_size() const
{
    // Do profiling stuff
    int ret = pmpi_comm.Get_size();
    // More profiling stuff
    return ret;
}
```

(End of advice to implementors.)

16.2 Fortran Support

16.2.1 Overview

The Fortran **MPI** language bindings have been designed to be compatible with the Fortran 90 standard with additional features from Fortran 2003 and Fortran 2008 [34] + TR 29113 [36].

Rationale. Fortran 90 contains numerous features designed to make it a more “modern” language than Fortran 77. It seems natural that **MPI** should be able to take advantage of these new features with a set of bindings tailored to Fortran 90. In Fortran 2008 + TR 29113, the major new language features used are the **ASYNCHRONOUS** attribute to protect nonblocking **MPI** operations, and assumed-type and assumed-rank dummy arguments for choice buffer arguments. Further requirements for compiler support are listed in Section 16.2.7 on page 568. *(End of rationale.)*

MPI defines three methods of Fortran support:

1. **USE mpi_f08:** This method is described in Section 16.2.2 and requires compile-time argument checking with unique **MPI** handle types and provides techniques to fully solve the optimization problems with nonblocking calls. This is the only Fortran support method that is consistent with the Fortran standard (Fortran 2008 + TR 29113 and later). This method is highly recommended for all **MPI** applications.
2. **USE mpi:** This method is described in Section 16.2.3 and requires compile-time argument checking. Handles are defined as **INTEGER**. This Fortran support method is inconsistent with the Fortran standard, and its use is therefore not recommended. It exists only for backwards compatibility.
3. **INCLUDE 'mpif.h':** This method is described in Section 16.2.4. The use of the include file `mpif.h` is strongly discouraged starting with **MPI-3.0**, because this method neither guarantees compile-time argument checking nor provides sufficient techniques to solve the optimization problems with nonblocking calls, and is therefore inconsistent with the Fortran standard. It exists only for backwards compatibility with legacy **MPI** applications.

Compliant **MPI-3** implementations providing a Fortran interface must provide one or both of the following:

- The `USE mpi_f08` Fortran support method.
- The `USE mpi` and `INCLUDE 'mpif.h'` Fortran support methods.

Section 16.2.6 on page 564 describes restrictions if the compiler does not support all the needed features.

Application subroutines and functions may use either one of the modules or the `mpif.h` include file. An implementation may require the use of one of the modules to prevent type mismatch errors.

Advice to users. Users are advised to utilize one of the MPI modules even if `mpif.h` enforces type checking on a particular system. Using a module provides several potential advantages over using an include file; the `mpi_f08` module offers the most robust and complete Fortran support. (*End of advice to users.*)

In a single application, it must be possible to link together routines which `USE mpi_f08`, `USE mpi`, and `INCLUDE mpif.h`.

The `INTEGER` compile-time constant `MPI_SUBARRAYS_SUPPORTED` is set to `.TRUE.` if all buffer choice arguments are defined in explicit interfaces with assumed-type and assumed-rank [36]; otherwise it is set to `.FALSE.`. The `INTEGER` compile-time constant `MPI_ASYNC_PROTECTS_NONBLOCKING` is set to `.TRUE.` if the `ASYNCHRONOUS` attribute was added to the choice buffer arguments of all nonblocking interfaces and the underlying Fortran compiler supports the `ASYNCHRONOUS` attribute for MPI communication (as part of TR 29113), otherwise it is set to `.FALSE.`. These constants exist with each Fortran support method, but not in the C/C++ header files. The values may be different for each Fortran support method. All other constants and the integer values of handles must be the same for each Fortran support method.

Section 16.2.2 through 16.2.4 define the Fortran support methods. The Fortran interfaces of each MPI routine are shorthands. Section 16.2.5 defines the corresponding full interface specification together with the used linker names and implications for the profiling interface. Section 16.2.6 the implementation of the MPI routines for different versions of the Fortran standard. Section 16.2.7 summarizes major requirements for valid MPI-3.0 implementations with Fortran support. Section 16.2.8 and Section 16.2.9 describe additional functionality that is part of the Fortran support. `MPI_F_SYNC_REG` is needed for one of the methods to prevent register optimization problems. A set of functions provides additional support for Fortran intrinsic numeric types, including parameterized types: `MPI_SIZEOF`, `MPI_TYPE_MATCH_SIZE`, `MPI_TYPE_CREATE_F90_INTEGER`, `MPI_TYPE_CREATE_F90_REAL` and `MPI_TYPE_CREATE_F90_COMPLEX`. In the context of MPI, parameterized types are Fortran intrinsic types which are specified using `KIND` type parameters. Sections 16.2.10 through 16.2.19 give an overview and details on known problems when using Fortran together with MPI; Section 16.2.20 compares the Fortran problems with those in C.

16.2.2 Fortran Support Through the `mpi_f08` Module

An MPI implementation providing a Fortran interface must provide a module named `mpi_f08` that can be used in a Fortran program. Section 16.2.6 on page 564 describes restrictions if the compiler does not support all the needed features. Within all MPI function specifications, the first of the set of two Fortran routine interface specifications is provided by this module. This module must:

- Define all named MPI constants.
- Declare MPI functions that return a value.
- Provide explicit interfaces according to the Fortran routine interface specifications. This module therefore guarantees compile-time argument checking for all arguments which are not `TYPE(*)`, with the following exception:

Only one Fortran interface is defined for functions that are deprecated as of MPI-3.0. This interface must be provided as an explicit interface according to the rules defined for the `mpi` module, see Section 16.2.3 on page 556.

Advice to users. It is strongly recommended that developers substitute calls to deprecated routines when upgrading from `mpif.h` or the `mpi` module to the `mpi_f08` module. (*End of advice to users.*)

- Define all MPI handles with uniquely named handle types (instead of `INTEGER` handles, as in the `mpi` module). This is reflected in the first Fortran binding in each MPI function definition throughout this document (except for the deprecated routines).
- Use the `ASYNCHRONOUS` attribute to protect the buffers of nonblocking operations, and set the `INTEGER` compile-time constant `MPI_ASYNC_PROTECTS_NONBLOCKING` to `.TRUE.` if the underlying Fortran compiler supports the `ASYNCHRONOUS` attribute for MPI communication (as part of TR 29113). See Section 16.2.6 on page 564 for older compiler versions.
- Set the `INTEGER` compile-time constant `MPI_SUBARRAYS_SUPPORTED` to `.TRUE.` and declare choice buffers using the Fortran 2008 TR 29113 feature assumed-type and assumed-rank, i.e., `TYPE(*)`, `DIMENSION(..)` in all nonblocking, split collective and persistent communication routines, if the underlying Fortran compiler supports it. With this, non-contiguous sub-arrays can be used as buffers in nonblocking routines.

Rationale. In all blocking routines, i.e., if the choice-buffer is not declared as `ASYNCHRONOUS`, the TR 29113 feature is not needed for the support of non-contiguous buffers because the compiler can pass the buffer by in-and-out-copy through a contiguous scratch array. (*End of rationale.*)

- Set the `MPI_SUBARRAYS_SUPPORTED` compile-time constant to `.FALSE.` and declare choice buffers with a compiler-dependent mechanism that overrides type checking if the underlying Fortran compiler does not support the Fortran 2008 TR 29113 assumed-type and assumed-rank notation. In this case, the use of non-contiguous sub-arrays as buffers in nonblocking calls may be invalid. See Section 16.2.6 on page 564 for details.

- Declare each argument with an `INTENT` of `IN`, `OUT`, or `INOUT` as defined in this standard.

Rationale. For these definitions in the `mpi_f08` bindings, in most cases, `INTENT(IN)` is used if the C interface uses call-by-value. For all buffer arguments and for `OUT` and `INOUT` dummy arguments that allow one of the non-ordinary Fortran constants (see `MPI_BOTTOM`, etc. in Section 2.5.4 on page 15) as input, an `INTENT` is not specified. (*End of rationale.*)

ticket242-N.

Advice to users. If a dummy argument is declared with `INTENT(OUT)`, then the Fortran standard stipulates that the actual argument becomes undefined upon invocation of the MPI routine, i.e., it may be overwritten by some other values, e.g. zeros; according to [34], 12.5.2.4 Ordinary dummy variables, Paragraph 17: “If a dummy argument has `INTENT(OUT)`, the actual argument becomes undefined at the time the association is established, except [...]”. For example, if the dummy argument is an assumed-size array and the actual argument is a strided array, the call may be implemented with copy-in and copy-out of the argument. In the case of `INTENT(OUT)` the copy-in may be suppressed by the optimization and the routine starts execution using an array of undefined values. If the routine stores fewer elements into the dummy argument than is provided in the actual argument, then the remaining locations are overwritten with these undefined values. See also both advices to implementors in Section 16.2.3 on page 556. (*End of advice to users.*)

- Declare all `ierror` output arguments as `OPTIONAL`, except for user-defined callback functions (e.g., `COMM_COPY_ATTR_FUNCTION`) and predefined callbacks (e.g., `MPI_NULL_COPY_FN`).

Rationale. For user-defined callback functions (e.g., `COMM_COPY_ATTR_FUNCTION`) and their predefined callbacks (e.g., `MPI_NULL_COPY_FN`), the `ierror` argument is not optional. The MPI library must always call these routines with an actual `ierror` argument. Therefore, these user-defined functions need not check whether the MPI library calls these routines with or without an actual `ierror` output argument. (*End of rationale.*)

The MPI Fortran bindings in the `mpi_f08` module are designed based on the Fortran 2008 standard [34] together with the Technical Report “TR 29113 Further Interoperability with C” [36] of the ISO/IEC JTC1/SC22/WG5 (Fortran) working group.

Rationale. The features in TR 29113 on further interoperability with C were decided on by ISO/IEC JTC1/SC22/WG5 and designed by PL22.3 (formerly J3) to support a higher level of integration between Fortran-specific features and C than was provided in the Fortran 2008 standard; part of this design is based on requirements from the MPI Forum to support MPI-3.0. According to [35] page iv, last paragraph, “it is the intention of ISO/IEC JTC1/SC22/WG5 that the semantics and syntax specified by this technical report be included in the next revision of the Fortran International Standard without change unless experience in the implementation and use of this feature identifies errors that need to be corrected, or changes are needed to achieve proper integration, in which case every reasonable effort will be made to minimize the impact of such changes on existing implementations”.

The TR 29113 contains the following language features that are needed for the MPI bindings in the `mpi_f08` module: assumed-type and assumed-rank. It is important that any possible actual argument can be used for such dummy arguments, e.g., scalars, arrays, assumed-shape arrays, assumed-size arrays, allocatable arrays, and with any element type, e.g., `REAL`, `CHARACTER*5`, `CHARACTER*(*)`, sequence derived

types, or `BIND(C)` derived types. Especially for backward compatibility reasons, it is important that any possible actual argument in an implicit interface implementation of a choice buffer dummy argument (e.g., with `mpif.h` without argument-checking) can be used in an implementation with assumed-type and assumed-rank argument in an explicit interface (e.g., with the `mpi_f08` module).

The `INTERFACE` construct in combination with `BIND(C)` allows the implementation of the Fortran `mpi_f08` interface with a single set of portable wrapper routines written in C, which supports all desired features in the `mpi_f08` interface. TR 29113 also has a provision for `OPTIONAL` arguments in `BIND(C)` interfaces.

A further feature useful for MPI is the extension of the semantics of the `ASYNCHRONOUS` attribute: In F2003 and F2008, this attribute could be used only to protect buffers of Fortran asynchronous I/O. With TR29113, this attribute now also covers asynchronous communication occurring within library routines written in C.

The MPI Forum hereby wishes to acknowledge this important effort by the Fortran PL22.3 and WG5 committee. (*End of rationale.*)

16.2.3 Fortran Support Through the `mpi` Module

An MPI implementation **providing a Fortran interface** must provide a module named `mpi` that can be used in a Fortran program. Within all MPI function specifications, the second of the set of two Fortran routine interface specifications is provided by this module. This module must:

- Define all named MPI constants
- Declare MPI functions that return a value.
- Provide explicit interfaces according to the Fortran routine interface specifications. This module therefore guarantees compile-time argument checking and allows positional and keyword-based argument lists.
- Define all MPI handles as type `INTEGER`.
- Define all named handle types and the derived type `MPI_Status` that are used in the `mpi_f08` module.

Rationale. They are needed only when the application converts old-style `INTEGER` handles into new-style handles with a named type. (*End of rationale.*)

- A high quality MPI implementation may enhance the interface by using the `ASYNCHRONOUS` attribute in the same way as in the `mpi_f08` module if it is supported by the underlying compiler.
- Set the `INTEGER` compile-time constant `MPI_ASYNC_PROTECTS_NONBLOCKING` to `.TRUE.` if the `ASYNCHRONOUS` attribute is used in all nonblocking interfaces **and** the underlying Fortran compiler supports the `ASYNCHRONOUS` attribute for MPI communication (as part of TR 29113), otherwise to `.FALSE.`

ticket247-S.

Advice to users. For an MPI implementation that fully supports nonblocking calls with the `ASYNCHRONOUS` attribute for choice buffers, an existing MPI-2.2 application may fail to compile even if it compiled and executed with expected results with an MPI-2.2 implementation. One reason may be that the application uses ‘contiguous’ but not ‘simply contiguous’ `ASYNCHRONOUS` arrays as actual arguments for choice buffers of nonblocking routines, e.g., by using subscript triplets with stride one or specifying `(1:n)` for a whole dimension instead of using `(:)`. This should be fixed to fulfill the Fortran constraints for `ASYNCHRONOUS` dummy arguments. This is not considered a violation of backward compatibility because existing applications can not use the `ASYNCHRONOUS` attribute to protect nonblocking calls. Another reason may be that the application does not conform either to MPI-2.2, or to MPI-3.0, or to the Fortran standard, typically because the program forces the compiler to perform copyin/out for a choice buffer argument in a nonblocking MPI call. This is also not a violation of backward compatibility because the application itself is non-conforming. See Section 16.2.12 on page 581 for more details. (*End of advice to users.*)

- A high quality MPI implementation may enhance the interface by using `TYPE(*)`, `DIMENSION(..)` choice buffer dummy arguments instead of using non-standardized extensions such as `!$PRAGMA IGNORE_TKR` or a set of overloaded functions as described by M. Hennecke in [26], if the compiler supports this TR 29113 language feature. See Section 16.2.6 on page 564 for further details.
- Set the `INTEGER` compile-time constant `MPI_SUBARRAYS_SUPPORTED` to `.TRUE.` if all choice buffer arguments in all nonblocking, split collective and persistent communication routines are declared with `TYPE(*)`, `DIMENSION(..)`, otherwise set it to `.FALSE.` With `MPI_SUBARRAYS_SUPPORTED==.TRUE.`, non-contiguous sub-arrays can be used as buffers in nonblocking routines.
- Set the `MPI_SUBARRAYS_SUPPORTED` compile-time constant to `.FALSE.` and declare choice buffers with a compiler-dependent mechanism that overrides type checking if the underlying Fortran compiler does not support the TR 29113 assumed-type and assumed-rank features. In this case, the use of non-contiguous sub-arrays in nonblocking calls may be disallowed. See Section 16.2.6 on page 564 for details.

An MPI implementation may provide *other features* in the `mpi` module that enhance the usability of MPI while maintaining adherence to the standard. For example, it may provide `INTENT` information in these interface blocks.

Advice to implementors. The appropriate `INTENT` may be different from what is given in the MPI *language-neutral bindings*. Implementations must choose `INTENT` so that the function adheres to the MPI standard, e.g., by defining the `INTENT` as provided in the `mpi_f08 bindings`. (*End of advice to implementors.*)

Rationale. The intent given by the MPI generic interface is not precisely defined and does not in all cases correspond to the correct Fortran `INTENT`. For instance, receiving into a buffer specified by a datatype with absolute addresses may require associating `MPI_BOTTOM` with a dummy `OUT` argument. Moreover, “constants” such

as `MPI_BOTTOM` and `MPI_STATUS_IGNORE` are not constants as defined by Fortran, but “special addresses” used in a nonstandard way. Finally, the MPI-1 generic intent **was** changed in several places in MPI-2. For instance, `MPI_IN_PLACE` changes the **intent** of an `OUT` argument to be `INOUT`. (*End of rationale.*)

Advice to implementors. The Fortran 2008 standard illustrates in its Note 5.17 that “*INTENT(OUT)* means that the value of the argument after invoking the procedure is entirely the result of executing that procedure. If an argument should retain its value rather than being redefined, *INTENT(INOUT)* should be used rather than *INTENT(OUT)*, even if there is no explicit reference to the value of the dummy argument. Furthermore, *INTENT(INOUT)* is not equivalent to omitting the *INTENT* attribute, because *INTENT(INOUT)* always requires that the associated actual argument is *definable*”. Applications that include `mpif.h` may not expect that `INTENT(OUT)` is used. In particular, output array arguments are expected to keep their content as long as the MPI routine does not modify them. To keep this behavior, it is recommended that implementations not use `INTENT(OUT)` in the `mpi` module and the `mpif.h` include file, even though `INTENT(OUT)` is specified in an interface description of the `mpi_f08` module. (*End of advice to implementors.*)

16.2.4 Fortran Support Through the `mpif.h` Include File

The use of the `mpif.h` include file is strongly discouraged and may be deprecated in a future version of MPI.

An MPI implementation providing a Fortran interface must provide an include file named `mpif.h` that can be used in a Fortran program. Within all MPI function specifications, the second of the set of two Fortran routine interface specifications is supported by this include file. This include file must:

- Define all named MPI constants.
- Declare MPI functions that return a value.
- Define all handles as `INTEGER`.
- Be valid and equivalent for both fixed and free source form.

For each MPI routine, an implementation can choose to use an implicit or explicit interface for the second Fortran binding (in deprecated routines, the first one may be omitted).

- Set the `INTEGER` compile-time constants `MPI_SUBARRAYS_SUPPORTED` and `MPI_ASYNC_PROTECTS_NONBLOCKING` according to the same rules as for the `mpi` module. In the case of implicit interfaces for choice buffer or nonblocking routines, the constants must be set to `.FALSE.`

Advice to users. Instead of using `mpif.h`, the use of the `mpi_f08` or `mpi` module is strongly encouraged for the following reasons:

- Most `mpif.h` implementations do not include compile-time argument checking.

- Therefore, too many bugs in MPI applications remain undetected at compile-time, such as:
 - Missing `ierror` as last argument in most Fortran bindings.
 - Declaration of a `status` as an `INTEGER` variable instead of an `INTEGER` array with size `MPI_STATUS_SIZE`.
 - Wrong argument positions; e.g., interchanging the `count` and `datatype` arguments.
 - Passing wrong MPI handles; e.g., passing a datatype instead of a communicator.
- The migration from `mpif.h` to the `mpi` module should be relatively straightforward (i.e., substituting `include 'mpif.h'` after an `implicit` statement by `use mpi` before such `implicit` statement) as long as the application syntax is correct.
- Migrating portable and correctly written applications to the `mpi` module is not expected to be difficult. No compile or runtime problems should occur because an `mpif.h` include file was always allowed to provide explicit Fortran interfaces.

(End of advice to users.)

Rationale. With MPI-3.0, the `mpif.h` include file was not deprecated in order to retain strong backward compatibility. Internally, `mpif.h` and the `mpi` module may be implemented so that the same (or similar) library implementation of the MPI routines can be used. *(End of rationale.)*

Advice to implementors. To make `mpif.h` compatible with both fixed- and free-source forms, to allow automatic inclusion by preprocessors, and to allow extended fixed-form line length, it is recommended that the requirement of usability in free and fixed source form applications be met by constructing `mpif.h` without any continuation lines. This should be possible because `mpif.h` may contain only declarations, and because common block declarations can be split among several lines. The argument names may need to be shortened to keep the `SUBROUTINE` statement within the allowed 72-6=66 characters, e.g.,

```
INTERFACE
SUBROUTINE PMPI_DIST_GRAPH_CREATE_ADJACENT(a,b,c,d,e,f,g,h,i,j,k)
... ! dummy argument declarations
```

This line has 65 characters and is the longest in MPI-3.0.

As long as the MPI standard contains routines with choice buffers and a name length and argument count that implies that a `BIND(C)` implementation would need to shorten their linker names in `mpif.h`, the `mpif.h` cannot set `MPI_SUBARRAYS_SUPPORTED` and `MPI_ASYNC_PROTECTS_NONBLOCKING` equals `.TRUE.`, because such shortening is invalid. For example, `MPI_FILE_WRITE_AT_ALL_BEGIN` with 6 arguments, may be defined:

```
INTERFACE MPI_FILE_WRITE_AT_ALL_BEGIN
SUBROUTINE MPI_X(a,b,c,d,e,f)BIND(C,NAME='MPI_File_write_at_all_begin_f')
... ! dummy argument declarations
```

This would need a line length of 73 characters, i.e., the C routine name would need

to be shortened by 7 characters to stay within the available 66 characters. Note that the name `MPI_X` has no meaning for the compilation, and that this problem occurs only with routines with choice buffers implemented with the assumed-type and assumed-rank facility of TR 29113. To support Fortran 77 as well as Fortran 90 and later, it may be necessary to eliminate all comments from `mpif.h`. (*End of advice to implementors.*)

ticket229.4.

16.2.5 Interface Specifications, Linker Names and the Profiling Interface

The Fortran interface specifications of each MPI routine specifies the routine name that must be called by the application program, and the names and types of the dummy arguments together with additional attributes. The rules for the linker names and its implications for the profiling interface are specified within this section. The linker name of a Fortran routine is defined as the name that a C routine would have if both routines would have the same name visible for the linker. A typical linker name of the Fortran routine `FOOfoo` is `foofoo__`. In the case of `BIND(C,NAME='...')`, the linker name is directly defined through the given string.

The following rules for linker names apply:

- With the Fortran `mpi_f08` module, if `MPI_SUBARRAYS_SUPPORTED` equals `.TRUE.`:
The Fortran binding must use `BIND(C)` interfaces with an interface name identical to the language independent name, e.g., `MPI_SEND`. The linker name is a combination of the C name and an `_f08` suffix, e.g., `MPI_Send_f08`. Prototype example:

```
INTERFACE
  SUBROUTINE MPI_Send(...) BIND(C,NAME='MPI_Send_f08')
```

- With the Fortran `mpi_f08` module, if `MPI_SUBARRAYS_SUPPORTED` equals `.FALSE.` (i.e., with a preliminary implementation of this module without TR 29113):
The linker name of each routine is defined through the linker name mapping of the Fortran compiler for the name defined when subarrays are supported. For example, `MPI_Send_f08` may be mapped to `mpi_send_f08__`. Example:

```
INTERFACE MPI_Send
  SUBROUTINE MPI_Send_f08(...)
```

- With the Fortran `mpi` module or `mpif.h` include file, if `MPI_SUBARRAYS_SUPPORTED` equals `.FALSE.`:
The linker name of each routine is defined through the linker-name mapping of the Fortran compiler. For example, `MPI_SEND` may be mapped to `mpi_send__`. Example:

```
INTERFACE
  SUBROUTINE MPI_SEND(...)
```

- With the Fortran `mpi` module or `mpif.h` include file, if `MPI_SUBARRAYS_SUPPORTED` equals `.TRUE.`:
The Fortran binding must use `BIND(C)` interfaces with an interface name identical to the language independent name, e.g., `MPI_SEND`. The linker name is a combination of the C name and an `_f` suffix, e.g., `MPI_Send_f`. Prototype example:

```
INTERFACE
  SUBROUTINE MPI_SEND(...) BIND(C,NAME='MPI_Send_f')
```

If the support of subarrays is different for the `mpi` module and the `mpif.h` include file, then both linker-name methods can be used in the same application. If the application also uses the `mpi_f08` module and was compiled with this module partially before and after the subarrays were supported, then all four interfaces are used within the same application.

Rationale. After a compiler provides the facilities from TR29113, i.e., `TYPE(*)`, `DIMENSION(..)`, it is possible to change the bindings within a Fortran support method to support subarrays and without recompiling the complete application. Of course, only recompiled routines can benefit from the added facilities. There is no binary compatibility conflict because each interface uses its own linker names and all interfaces use the same constants and type definitions. (*End of rationale.*)

A user-written or middleware profiling routine that is written according to the same binding rules will have the same linker name, and therefore, can interpose itself as the `MPI` library routine. The profiling routine can internally call the matching `PMPI` routine with any of its existing bindings, except for routines that have callback routine dummy arguments. In this case, the profiling software must use the same Fortran support method as used in the calling application program, because the C, `mpi_f08` and `mpi` callback prototypes are different.

Advice to users. This advice is mainly for tool writers. Even if an `MPI` library supports subarrays in all three Fortran support methods, a portable profiling layer should also provide the two interfaces for `MPI_SUBARRAYS_SUPPORTED==.FALSE.` to support older binary user routines that were compiled before TR29113 level support was achieved.

If a user application calls `MPI_SEND`, then the chosen Fortran support method together with the `MPI` implement decision about `MPI_SUBARRAYS_SUPPORTED` imply, to which linker name the compiler will translate this call, i.e., whether the application calls `mpi_send__`, or `MPI_Send_f`, or `mpi_send_f08__`, or `MPI_Send_f08`. If the profiling layer wants to be independent of the decision of the user program and `MPI` implementation, then it should provide all four routines. For example:

```
SUBROUTINE MPI_SEND(...) BIND(C,NAME='MPI_Send_f')
  USE mpi
  CALL PMPI_SEND(...)
END SUBROUTINE
```

The `MPI` library must provide the `PMPI_SEND` routine according to the same rules as for providing the `MPI_SEND` routine. (*End of advice to users.*)

Advice to implementors. If an implementation provides in a first step two sets of routines, one for the `mpi` module and `mpif.h`, and the other for the `mpi_f08` module, and both sets without TR 29113, i.e., `MPI_SUBARRAYS_SUPPORTED` equals `.FALSE.`, and the implementor wants to add a TR 29113 based set of routines, then it is not necessary to add two full sets of routines. For full quality, it is enough to implement in each set only those routines that have a choice buffer argument. (*End of advice to implementors.*)

In the case that a Fortran binding consists of multiple routines through function overloading, the base names of overloaded routines are appended by a suffix notifying the difference in the argument list. For example, `MPI_ALLOC_MEM` (in the `mpi` module and `mpif.h`)

has an `INTEGER(KIND=...) baseptr` argument without a suffix. This routine is overloaded by a routine with `TYPE(C_PTR) baseptr` and the suffix `_CPTR`. The implied linker name base is `MPI_ALLOC_MEM_CPTR`. It is mapped to the linker names `MPI_Alloc_mem_cptra_f`, and, e.g., `mpi_alloc_mem_cptra_`. Note that these routines are always called via the interface name `MPI_ALLOC_MEM` by the application within all Fortran support methods.

For routines without `ASYNCHRONOUS` choice buffers and that are not predefined callback routines, the implementor can freely choose to implement the routines according to the rules for `MPI_SUBARRAYS_SUPPORTED` equals `.TRUE.` or `.FALSE.`, provided that the following rule about routine grouping is fulfilled. The implementation of routines with `ASYNCHRONOUS` choice buffers depends on the rules for the provided Fortran support method and language level of the underlying compiler. Predefined callback routines for the `mpi_f08` module must be implemented with `BIND(C)` interfaces, and for the `mpi` module and `mpif.h` without `BIND(C)`.

Similar MPI routines are grouped together for linker symbol scheme classification. If the peer routine of a group is available within an MPI library with one of its possible linker names then all of the routines in this group must provided according to the same linker name scheme. If the peer routine is not available through a linker name scheme then all other routines in the group must not be available through this scheme.

Peer routines and their groups:

<code>MPI_ALLOC_MEM</code>	<code>MPI_ALLOC_MEM</code> and <code>MPI_WIN_ALLOCATE</code> .
<code>MPI_FREE_MEM</code>	Only this routine is in this group.
<code>MPI_GET_ADDRESS</code>	<code>MPI_GET_ADDRESS</code> and <code>MPI_ADDRESS</code> .
<code>MPI_SEND</code>	All routines with choice buffer arguments that are not declared as <code>ASYNCHRONOUS</code> within the <code>mpi_f08</code> module.
<code>MPI_ISEND</code>	All routines with choice buffer arguments that are declared as <code>ASYNCHRONOUS</code> within the <code>mpi_f08</code> module.
<code>MPI_OP_CREATE</code>	Only this routine is in this group.
<code>MPI_REGISTER_DATAREP</code>	Only this routine is in this group.
<code>MPI_COMM_KEYVAL_CREATE</code>	All other routines with callback function arguments.
<code>MPI_COMM_DUP_FN</code>	All predefined callback routines.
<code>MPI_COMM_RANK</code>	All other MPI routines.

Additionally, four C preprocessor macros are available in `mpi.h` for each routine group. The name of the macros are the peer routine name written as in the list above and appended with one of the following suffixes and meanings:

`_mpi_f08_BIND_C` The macro is set to 1 if the `BIND(C)` linker name with the linker suffix `_f08` is available for all routines within this group (e.g., `MPI_Send_f08`), otherwise it is set to 0.

`_mpi_f08_BIND_F` The macro is set to 1 if the Fortran linker name with the linker suffix `_f08` is available for all routines within this group (e.g., `mpi_send_f08__`), otherwise it is set to 0.

`_mpi_BIND_C ...` The macro is set to 1 if the `BIND(C)` linker name with the linker suffix `_f` is available for all routines within this group (e.g., `MPI_Send_f`), otherwise it is set to 0.

`_mpi_BIND_F ...` The macro is set to 1 if the Fortran linker name without a linker suffix is available for all routines within this group (e.g., `mpi_send__`), otherwise it is set to 0.

For example

```
...
#define MPI_SEND_mpi_f08_BIND_C 0
#define MPI_SEND_mpi_f08_BIND_F 1
#define MPI_SEND_mpi_BIND_C 0
#define MPI_SEND_mpi_BIND_F 1

#define MPI_ISEND_mpi_f08_BIND_C 1
#define MPI_ISEND_mpi_f08_BIND_F 1
#define MPI_ISEND_mpi_BIND_C 1
#define MPI_ISEND_mpi_BIND_F 1
...
#define MPI_COMM_DUP_FN_mpi_f08_BIND_C 1
#define MPI_COMM_DUP_FN_mpi_f08_BIND_F 0
#define MPI_COMM_DUP_FN_mpi_BIND_C 0
#define MPI_COMM_DUP_FN_mpi_BIND_F 1
...
```

shows, that

- the routines in the `MPI_SEND` group are only available through their Fortran linker names (e.g., `mpi_send_f08__`, `mpi_send__`, `mpi_recv_f08__`, `mpi_recv__`, ...),
- the routines in the `MPI_ISEND` group are available with all four interfaces: the `MPI` library, the `mpi_f08` and `mpi` modules (that provide the TR 29113 quality), and this `MPI` library supports application routines that are compiled with an older `MPI` library version with `_BIND_C` set to 0 and `_BIND_F` set to 1.

For the predefined callbacks, there is no choice, because the interfaces must fit to the callback function prototypes which are `BIND(C)` based for `mpi_f08` and without `BIND(C)` for the `mpi` module and `mpif.h`.

Advice to implementors. If all following conditions are fulfilled (which is the case for most compilers):

- the handles in the `mpi_f08` module occupy one Fortran numerical storage unit (same as an `INTEGER` handle),

- the internal argument passing used to pass an actual `ierror` argument to a non optional `ierror` dummy argument is binary compatible to passing an actual `ierror` argument to an `ierror` dummy argument that is declared as `OPTIONAL`,
- the internal argument passing for `ASYNCHRONOUS` and non-`ASYNCHRONOUS` arguments is the same,
- the internal routine call mechanism is the same for the Fortran and the C compiler,
- the compiler does not provide TR 29113,

then for most groups, the implementor may use the same internal routine implementations for all Fortran support methods but with several different linker names. For TR 29113 quality, new routines are needed only for the routine group of `MPI_ISEND`. Typical settings for `_mpi_f08_BIND_C` / `_mpi_f08_BIND_F` / `_mpi_BIND_C` / `_mpi_BIND_F` may be:

	Without TR 29113	Upgrade to TR 29113	Upgrade for strided data optimization	New impl. with TR 29113
<code>MPI_ALLOC_MEM_...</code>	0/1/0/1	0/1/0/1	0/1/0/1	1/0/1/0
<code>MPI_FREE_MEM_...</code>	0/1/0/1	0/1/0/1	0/1/0/1	1/0/1/0
<code>MPI_GET_ADDRESS_...</code>	0/1/0/1	0/1/0/1	0/1/0/1	1/0/1/0
<code>MPI_SEND_...</code>	0/1/0/1	0/1/0/1	1/1/1/1	1/0/1/0
<code>MPI_ISEND_...</code>	0/1/0/1	1/1/1/1	1/1/1/1	1/0/1/0
<code>MPI_OP_CREATE_...</code>	0/1/0/1	0/1/0/1	0/1/0/1	1/0/1/0
<code>MPI_REGISTER_DATAREP_...</code>	0/1/0/1	0/1/0/1	0/1/0/1	1/0/1/0
<code>MPI_COMM_KEYVAL_CREATE_...</code>	0/1/0/1	0/1/0/1	0/1/0/1	1/0/1/0
<code>MPI_COMM_DUP_FN_...</code>	1/0/0/1	1/0/0/1	1/0/0/1	1/0/0/1
<code>MPI_COMM_RANK_...</code>	0/1/0/1	0/1/0/1	0/1/0/1	1/0/1/0

(End of advice to implementors.)

16.2.6 MPI for Different Fortran Standard Versions

This section describes which Fortran interface functionality can be provided for different versions of the Fortran standard.

- For Fortran 77 with some extensions:
 - MPI identifiers are limited to thirty or more, not six, significant characters.
 - MPI identifiers may contain underscores after the first character.
 - An MPI subroutine with a choice argument may be called with different argument types.
 - Although not required by the MPI standard, the `INCLUDE` statement should be available for including `mpif.h` into the user application source code.

Only MPI-1.1, MPI-1.2, and MPI-1.3 can be implemented. The use of absolute addresses from `MPI_ADDRESS` and `MPI_BOTTOM` may cause problems if an address does not fit into the memory space provided by an `INTEGER`. (In MPI-2.0 this problem is solved with `MPI_GET_ADDRESS`, but not for Fortran 77.)

- For Fortran 90:

The major additional features that are needed from Fortran 90 are:

- The `MODULE` and `INTERFACE` concept.
- The `KIND=` and `SELECTED_..._KIND` concept.
- Fortran derived `TYPE`s and the `SEQUENCE` attribute.
- The `OPTIONAL` attribute for dummy arguments.
- Cray pointers, which are a non-standard compiler extension, are needed for the use of `MPI_ALLOC_MEM`.

With these features, MPI-1.1 - MPI-2.2 can be implemented without restrictions. MPI-3.0 can be implemented with some restrictions. The Fortran support methods are abbreviated with `S1` = the `mpi_f08` module, `S2` = the `mpi` module, and `S3` = the `mpif.f` include file. If not stated otherwise, restrictions exist for each method which prevent implementing the complete semantics of MPI-3.0.

- `MPI_SUBARRAYS_SUPPORTED` equals `.FALSE.`, i.e., subscript triplets and non-contiguous subarrays cannot be used as buffers in nonblocking routines, RMA, or split-collective I/O.
- `S1`, `S2`, and `S3` can be implemented, but for `S1`, only a preliminary implementation is possible.
- In this preliminary interface of `S1`, the following changes are necessary:
 - * The routines are not `BIND(C)`.
 - * `TYPE(*)`, `DIMENSION(..)` is substituted by non-standardized extensions like `!$PRAGMA IGNORE_TKR`.
 - * The `ASYNCHRONOUS` attribute is omitted.
 - * `PROCEDURE(...)` callback declarations are substituted by `EXTERNAL`.
- The linker names are specified in Section 16.2.5 on page 560.
- Due to the rules specified in Section 16.2.5 on page 560, choice buffer declarations should be implemented only with non-standardized extensions like `!$PRAGMA IGNORE_TKR` (as long as F2008+TR29113 is not available).
 In `S2` and `S3`: Without such extensions, routines with choice buffers should be provided with an implicit interface, instead of overloading with a different MPI function for each possible buffer type (as mentioned in Section 16.2.11 on page 580). Such overloading would also imply restrictions for passing Fortran derived types as choice buffer, see also Section 16.2.15 on page 584.
 Only in `S1`: The implicit interfaces for routines with choice buffer arguments imply that the `ierror` argument cannot be defined as `OPTIONAL`. For this reason, it is recommended not to provide the `mpi_f08` module if such an extension is not available.
- The `ASYNCHRONOUS` attribute can **not** be used in applications to protect buffers in nonblocking MPI calls (`S1-S3`).
- The `TYPE(C_PTR)` binding of the `MPI_ALLOC_MEM` and `MPI_WIN_ALLOCATE` routines is not available.

- In S1 and S2, the definition of the handle types (e.g., `TYPE(MPI_Comm)` and the status type `TYPE(MPI_Status)` must be modified: The `SEQUENCE` attribute must be used instead of `BIND(C)` (which is not available in Fortran 90/95). This restriction implies that the application must be fully recompiled if one switches to an MPI library for Fortran 2003 and later because the internal memory size of the handles may have changed. For this reason, an implementor may choose not to provide the `mpi_f08` module for Fortran 90 compilers. In this case, the `mpi_f08` handle types and all routines, constants and types related to `TYPE(MPI_Status)` (see Section 16.3.5 on page 604) are also not available in the `mpi` module and `mpif.h`.

- For Fortran 95:

The quality of the MPI interface and the restrictions are the same as with Fortran 90.

- For Fortran 2003:

The major features that are needed from Fortran 2003 are:

- Interoperability with C, i.e.,
 - * `BIND(C, NAME='...')` interfaces.
 - * `BIND(C)` derived types.
 - * The `ISO_C_BINDING` intrinsic type `C_PTR` and routine `C_F_POINTER`.
- The ability to define an `ABSTRACT INTERFACE` and to use it for `PROCEDURE` dummy arguments.
- The `ASYNCHRONOUS` attribute is available to protect Fortran asynchronous I/O. This feature is not yet used by MPI, but it is the basis for the enhancement for MPI communication in the TR 29113.

With these features (but still without the features of TR29113), MPI-1.1 - MPI-2.2 can be implemented without restrictions, but with one enhancement:

- The user application can use `TYPE(C_PTR)` together with `MPI_ALLOC_MEM` as long as `MPI_ALLOC_MEM` is defined with an implicit interface because a `C_PTR` and an `INTEGER(KIND=MPI_ADDRESS_KIND)` argument must both map to a void * argument.

MPI-3.0 can be implemented with the following restrictions:

- `MPI_SUBARRAYS_SUPPORTED` equals `.FALSE..`
- For S1, only a preliminary implementation is possible. The following changes are necessary:
 - * The routines are not `BIND(C)`.
 - * `TYPE(*)`, `DIMENSION(..)` is substituted by non-standardized extensions like `!$PRAGMA IGNORE_TKR`.
- The linker names are specified in Section 16.2.5 on page 560.
- With S1, the `ASYNCHRONOUS` is required as specified in the second Fortran interfaces. With S2 and S3 the implementation can also add this attribute if explicit interfaces are used.

- The **ASYNCHRONOUS** Fortran attribute can be used in applications to **try to** protect buffers in nonblocking MPI calls, but the protection can work only if the compiler is able to protect asynchronous Fortran I/O and makes no difference between such asynchronous Fortran I/O and MPI communication.
- The **TYPE(C_PTR)** binding of the **MPI_ALLOC_MEM** and **MPI_WIN_ALLOCATE** routines can be used only for Fortran types that are C compatible.
- The same restriction as for Fortran 90 applies if non-standardized extensions like **!\$PRAGMA IGNORE_TKR** are not available.

- For Fortran 2008 + TR 29113 and later and
For Fortran 2003 + TR 29113:

The major feature that are needed from TR29113 are:

- **TYPE(*)**, **DIMENSION(..)** is available.
- The **ASYNCHRONOUS** attribute is extended to protect also nonblocking MPI communication.
- **OPTIONAL** dummy arguments are allowed in combination with **BIND(C)** interfaces.
- **CHARACTER(LEN=*)** dummy arguments are allowed in combination with **BIND(C)** interfaces.
- The array dummy argument of the **ISO_C_BINDING** intrinsic **C_F_POINTER** is not restricted to Fortran types for which a corresponding type in C exists.

Using these features, MPI-3.0 can be implemented without any restrictions.

- With **S1**, **MPI_SUBARRAYS_SUPPORTED** equals **.TRUE..** The **ASYNCHRONOUS** attribute can be used to protect buffers in nonblocking MPI calls. The **TYPE(C_PTR)** binding of the **MPI_ALLOC_MEM** and **MPI_WIN_ALLOCATE** routines can be used for any Fortran type.
- With **S2** and **S3**, the value of **MPI_SUBARRAYS_SUPPORTED** is implementation dependent. A high quality implementation will also provide **MPI_SUBARRAYS_SUPPORTED==.TRUE.** and will use the **ASYNCHRONOUS** attribute in the same way as in **S1**.
- If non-standardized extensions like **!\$PRAGMA IGNORE_TKR** are not available then **S2** must be implemented with **TYPE(*)**, **DIMENSION(..)**.

Advice to implementors. If **MPI_SUBARRAYS_SUPPORTED==.FALSE.**, the choice argument may be implemented with an explicit interface using compiler directives, for example:

```

INTERFACE
  SUBROUTINE MPI...(buf, ...)
    !DEC$ ATTRIBUTES NO_ARG_CHECK :: buf
    !$PRAGMA IGNORE_TKR buf
    !DIR$ IGNORE_TKR buf
    !IBM* IGNORE_TKR buf
    REAL, DIMENSION(*) :: buf
    ... ! declarations of the other arguments
  END SUBROUTINE
END INTERFACE

```

ticket234-F.

(End of advice to implementors.)

16.2.7 Requirements on Fortran Compilers

MPI-3.0 (and later) compliant Fortran bindings are not only a property of the MPI library itself, but rather a property of an MPI library together with the Fortran compiler suite for which it is compiled.

Advice to users. Users must take appropriate steps to ensure that proper options are specified to compilers. MPI libraries must document these options. Some MPI libraries are shipped together with special compilation scripts (e.g., `mpif90`, `mpicc`) that set these options automatically. (End of advice to users.)

An MPI library together with the Fortran compiler suite is only compliant with MPI-3.0 (and later), as referred by `MPI_GET_VERSION`, if all the solutions described in Sections 16.2.11 through 16.2.19 work correctly. Based on this rule, major requirements for all three Fortran support methods (i.e., the `mpi_f08` and `mpi` modules, and `mpif.h`) are:

- The language features assumed-type and assumed-rank from Fortran 2008 TR 29113 [36] are available. This is required only for `mpi_f08`. As long as this requirement is not supported by the compiler, it is valid to build an MPI library that implements the `mpi_f08` module with `MPI_SUBARRAYS_SUPPORTED` set to `.FALSE..`
- ‘Simply contiguous’ arrays and scalars must be passed to choice buffer dummy arguments of nonblocking routines with call by reference. This is needed only if one of the support methods does not use the `ASYNCHRONOUS` attribute. See Section 16.2.12 on page 581 for more details.
- `SEQUENCE` and `BIND(C)` derived types are valid as actual arguments passed to choice buffer dummy arguments, and, in the case of `MPI_SUBARRAYS_SUPPORTED==.FALSE.`, they are passed with call by reference, and passed by descriptor in the case of `.TRUE..`
- All actual arguments that are allowed for a dummy argument in an implicitly defined and separately compiled Fortran routine with the given compiler (e.g., `CHARACTER(LEN=*)` strings and array of strings) must also be valid for choice buffer dummy arguments with all Fortran support methods.
- The array dummy argument of the `ISO_C_BINDING` intrinsic module procedure `C_F_POINTER` is not restricted to Fortran types for which a corresponding type in C exists.
- The Fortran compiler shall not provide `TYPE(*)` unless the `ASYNCHRONOUS` attribute protects MPI communication as described in TR 29113. Specifically, the TR 29113 must be implemented as a whole.

The following rules are required at least as long as the compiler does not provide the extension of the `ASYNCHRONOUS` attribute as part of TR 29113 and there is still one Fortran support method with `MPI_ASYNC_PROTECTS_NONBLOCKING==.FALSE..` It is helpful when these rules are observed, especially for backward compatibility of existing applications that use the `mpi` module or the `mpif.h` include file. The rules are as follows:

- Separately compiled empty Fortran routines with implicit interfaces and separately compiled empty C routines with BIND(C) Fortran interfaces (e.g., MPI_F_SYNC_REG on page 592 and Section 16.2.8 on page 569, and DD on page 593) solve the problems described in Section 16.2.17 on page 587.
- The problems with temporary data movement (described in detail in Section 16.2.18 on page 594) are solved as long as the application uses different sets of variables for the nonblocking communication (or nonblocking or split collective IO) and the computation when overlapping communication and computation.
- Problems caused by automatic and permanent data movement (e.g., within a garbage collection, see Section 16.2.19 on page 597) are resolved **without** any further requirements on the application program, neither on the usage of the buffers, nor on the declaration of application routines that are involved in calling MPI operations.

All of these rules are valid independently of whether the MPI routine interfaces in the `mpi_f08` and `mpi` modules are internally defined with an `INTERFACE` or `CONTAINS` construct, and with or without `BIND(C)`, and also when `mpif.h` uses explicit interfaces.

Advice to implementors. Some of these rules are already part of the Fortran 2003 standard if the MPI interfaces are defined without BIND(C). Additional compiler support may be necessary if BIND(C) is used. Some of these additional requirements are defined in the Fortran 2008 TR 29113 [36]. Some of these requirements for MPI-3.0 are beyond the scope of TR 29113. (*End of advice to implementors.*)

Further requirements apply when the MPI library internally uses BIND(C) routine interfaces (i.e. for a full implementation of `mpi_f08`):

- Non-buffer arguments are `INTEGER`, `INTEGER(KIND=...)`, `CHARACTER(LEN=*)`, `LOGICAL`, and `BIND(C)` derived types, (handles and status in `mpi_f08`) variables and arrays; function results are `DOUBLE PRECISION`. All these types must be valid as dummy arguments in the `BIND(C)` MPI routine interfaces. When compiling an MPI application, the compiler should not issue warnings indicating that these types may not be interoperable with an existing type in C. Some of these types are already valid in `BIND(C)` interfaces since Fortran 2003, some may be valid based on TR 29113 (e.g., `CHARACTER*(*)`).
- `OPTIONAL` dummy arguments are also valid within `BIND(C)` interfaces. This requirement is fulfilled if TR 29113 is fully supported by the compiler.

16.2.8 Additional Support for Fortran Register-Memory-Synchronization

As described in Section 16.2.17 on page 587, a dummy call may be necessary to tell the compiler that registers are to be flushed for a given buffer or that accesses to a buffer may not be moved across a given point in the execution sequence. Only a Fortran binding exists for this call.

MPI_F_SYNC_REG(buf)

INOUT buf

initial address of buffer (choice)

```

1  MPI_F_sync_reg(buf) BIND(C)
2      TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
3
4  MPI_F_SYNC_REG(buf)
5      <type> buf(*)

```

This routine is a no-operation. It must be compiled in the MPI library in such a manner that a Fortran compiler cannot detect in the module that the routine has an empty body. It is used only to force the compiler to flush a cached register value of a variable or buffer back to memory (when necessary), or to invalidate the register value.

Rationale. This function is not available in other languages because it would not be useful. This routine has no `ierror` return argument because there is no operation that can fail. (*End of rationale.*)

Advice to implementors. This routine can be bound to a C routine to minimize the risk that the Fortran compiler can learn that this routine is empty (and that the call to this routine can be removed as part of an optimization). However, it is explicitly allowed to implement this routine within the `mpi_f08` module according to the definition for the `mpi` module or `mpif.h` to circumvent the overhead of building the internal dope vector to handle the assumed-type, assumed-rank argument. (*End of advice to implementors.*)

Rationale. This routine is not defined with `TYPE(*)`, `DIMENSION(*)`, i.e., assumed size instead of assumed rank, because this would restrict the usability to ‘simply contiguous’ arrays and would require overloading with another interface for scalar arguments. (*End of rationale.*)

Advice to users. If only a part of an array (e.g., defined by a subscript triplet) is used in a nonblocking routine, it is recommended to pass the whole array to `MPI_F_SYNC_REG` anyway to minimize the overhead of this no-operation call. Note that this routine need not to be called if `MPI_ASYNC_PROTECTS_NONBLOCKING` is `.TRUE.` and the application fully uses the facilities of `ASYNCHRONOUS` arrays. (*End of advice to users.*)

16.2.9 Additional Support for Fortran Numeric Intrinsic Types

MPI provides a small number of named datatypes that correspond to named intrinsic types supported by C and Fortran. These include `MPI_INTEGER`, `MPI_REAL`, `MPI_INT`, `MPI_DOUBLE`, etc., as well as the optional types `MPI_REAL4`, `MPI_REAL8`, etc. There is a one-to-one correspondence between language declarations and MPI types.

Fortran (starting with Fortran 90) provides so-called `KIND`-parameterized types. These types are declared using an intrinsic type (one of `INTEGER`, `REAL`, `COMPLEX`, `LOGICAL` and `CHARACTER`) with an optional integer `KIND` parameter that selects from among one or more variants. The specific meaning of different `KIND` values themselves are implementation dependent and not specified by the language. Fortran provides the `KIND` selection functions `selected_real_kind` for `REAL` and `COMPLEX` types, and `selected_int_kind` for `INTEGER` types that allow users to declare variables with a minimum precision or number of digits. These functions provide a portable way to declare `KIND`-parameterized `REAL`, `COMPLEX` and

INTEGER variables in Fortran. This scheme is backward compatible with Fortran 77. REAL and INTEGER Fortran variables have a default KIND if none is specified. Fortran DOUBLE PRECISION variables are of intrinsic type REAL with a non-default KIND. The following two declarations are equivalent:

```
double precision x
real(KIND(0.0d0)) x
```

MPI provides two orthogonal methods to communicate using numeric intrinsic types. The first method (see the following section) can be used when variables have been declared in a portable way — using default KIND or using KIND parameters obtained with the `selected_int_kind` or `selected_real_kind` functions. With this method, MPI automatically selects the correct data size (e.g., 4 or 8 bytes) and provides representation conversion in heterogeneous environments. The second method (see Support for size-specific MPI Datatypes on page 575) gives the user complete control over communication by exposing machine representations.

Parameterized Datatypes with Specified Precision and Exponent Range

MPI provides named datatypes corresponding to standard Fortran 77 numeric types — `MPI_INTEGER`, `MPI_COMPLEX`, `MPI_REAL`, `MPI_DOUBLE_PRECISION` and `MPI_DOUBLE_COMPLEX`. MPI automatically selects the correct data size and provides representation conversion in heterogeneous environments. The mechanism described in this section extends this model to support portable parameterized numeric types.

The model for supporting portable parameterized types is as follows. Real variables are declared (perhaps indirectly) using `selected_real_kind(p, r)` to determine the KIND parameter, where `p` is decimal digits of precision and `r` is an exponent range. Implicitly MPI maintains a two-dimensional array of predefined MPI datatypes `D(p, r)`. `D(p, r)` is defined for each value of `(p, r)` supported by the compiler, including pairs for which one value is unspecified. Attempting to access an element of the array with an index `(p, r)` not supported by the compiler is erroneous. MPI implicitly maintains a similar array of `COMPLEX` datatypes. For integers, there is a similar implicit array related to `selected_int_kind` and indexed by the requested number of digits `r`. Note that the predefined datatypes contained in these implicit arrays are not the same as the named MPI datatypes `MPI_REAL`, etc., but a new set.

Advice to implementors. The above description is for explanatory purposes only. It is not expected that implementations will have such internal arrays. (*End of advice to implementors.*)

Advice to users. `selected_real_kind()` maps a large number of `(p,r)` pairs to a much smaller number of KIND parameters supported by the compiler. KIND parameters are not specified by the language and are not portable. From the language point of view intrinsic types of the same base type and KIND parameter are of the same type. In order to allow interoperability in a heterogeneous environment, MPI is more stringent. The corresponding MPI datatypes match if and only if they have the same `(p,r)` value

(REAL and COMPLEX) or `r` value (INTEGER). Thus MPI has many more datatypes than there are fundamental language types. (*End of advice to users.*)

MPI_TYPE_CREATE_F90_REAL(`p`, `r`, `newtype`)

IN	<code>p</code>	precision, in decimal digits (integer)
IN	<code>r</code>	decimal exponent range (integer)
OUT	<code>newtype</code>	the requested MPI datatype (handle)

int MPI_Type_create_f90_real(int `p`, int `r`, MPI_Datatype *`newtype`)

```
MPI_Type_create_f90_real(p, r, newtype, ierror) BIND(C)
  INTEGER, INTENT(IN) :: p, r
  TYPE(MPI_Datatype), INTENT(OUT) :: newtype
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_TYPE_CREATE_F90_REAL(P, R, NEWTYPE, IERROR)
  INTEGER P, R, NEWTYPE, IERROR
```

```
{static MPI::Datatype MPI::Datatype::Create_f90_real(int p, int r) (binding
  deprecated, see Section 15.2) }
```

This function returns a predefined MPI datatype that matches a REAL variable of KIND `selected_real_kind(p, r)`. In the model described above it returns a handle for the element `D(p, r)`. Either `p` or `r` may be omitted from calls to `selected_real_kind(p, r)` (but not both). Analogously, either `p` or `r` may be set to `MPI_UNDEFINED`. In communication, an MPI datatype `A` returned by `MPI_TYPE_CREATE_F90_REAL` matches a datatype `B` if and only if `B` was returned by `MPI_TYPE_CREATE_F90_REAL` called with the same values for `p` and `r` or `B` is a duplicate of such a datatype. Restrictions on using the returned datatype with the “external32” data representation are given on page 574.

It is erroneous to supply values for `p` and `r` not supported by the compiler.

MPI_TYPE_CREATE_F90_COMPLEX(`p`, `r`, `newtype`)

IN	<code>p</code>	precision, in decimal digits (integer)
IN	<code>r</code>	decimal exponent range (integer)
OUT	<code>newtype</code>	the requested MPI datatype (handle)

int MPI_Type_create_f90_complex(int `p`, int `r`, MPI_Datatype *`newtype`)

```
MPI_Type_create_f90_complex(p, r, newtype, ierror) BIND(C)
  INTEGER, INTENT(IN) :: p, r
  TYPE(MPI_Datatype), INTENT(OUT) :: newtype
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_TYPE_CREATE_F90_COMPLEX(P, R, NEWTYPE, IERROR)
  INTEGER P, R, NEWTYPE, IERROR
```

```
{static MPI::Datatype MPI::Datatype::Create_f90_complex(int p,  
int r) (binding deprecated, see Section 15.2) }
```

This function returns a predefined MPI datatype that matches a COMPLEX variable of KIND `selected_real_kind(p, r)`. Either `p` or `r` may be omitted from calls to `selected_real_kind(p, r)` (but not both). Analogously, either `p` or `r` may be set to `MPI_UNDEFINED`. Matching rules for datatypes created by this function are analogous to the matching rules for datatypes created by `MPI_TYPE_CREATE_F90_REAL`. Restrictions on using the returned datatype with the “external32” data representation are given on page 574.

It is erroneous to supply values for `p` and `r` not supported by the compiler.

```
MPI_TYPE_CREATE_F90_INTEGER(r, newtype)
```

IN	r	decimal exponent range, i.e., number of decimal digits (integer)
----	---	---

OUT	newtype	the requested MPI datatype (handle)
-----	---------	-------------------------------------

```
int MPI_Type_create_f90_integer(int r, MPI_Datatype *newtype)
```

```
MPI_Type_create_f90_integer(r, newtype, ierror) BIND(C)
```

```
INTEGER, INTENT(IN) :: r  
TYPE(MPI_Datatype), INTENT(OUT) :: newtype  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_TYPE_CREATE_F90_INTEGER(R, NEWTYPE, IERROR)
```

```
INTEGER R, NEWTYPE, IERROR
```

```
{static MPI::Datatype MPI::Datatype::Create_f90_integer(int r) (binding  
deprecated, see Section 15.2) }
```

This function returns a predefined MPI datatype that matches a `INTEGER` variable of KIND `selected_int_kind(r)`. Matching rules for datatypes created by this function are analogous to the matching rules for datatypes created by `MPI_TYPE_CREATE_F90_REAL`. Restrictions on using the returned datatype with the “external32” data representation are given on page 574.

It is erroneous to supply a value for `r` that is not supported by the compiler.

Example:

```
integer      longtype, quadtype  
integer, parameter :: long = selected_int_kind(15)  
integer(long) ii(10)  
real(selected_real_kind(30)) x(10)  
call MPI_TYPE_CREATE_F90_INTEGER(15, longtype, ierror)  
call MPI_TYPE_CREATE_F90_REAL(30, MPI_UNDEFINED, quadtype, ierror)  
...  
  
call MPI_SEND(ii, 10, longtype, ...)  
call MPI_SEND(x, 10, quadtype, ...)
```

Advice to users. The datatypes returned by the above functions are predefined datatypes. They cannot be freed; they do not need to be committed; they can be used with predefined reduction operations. There are two situations in which they behave differently syntactically, but not semantically, from the MPI named predefined datatypes.

1. `MPI_TYPE_GET_ENVELOPE` returns special combiners that allow a program to retrieve the values of `p` and `r`.
2. Because the datatypes are not named, they cannot be used as compile-time initializers or otherwise accessed before a call to one of the `MPI_TYPE_CREATE_F90_XXXX` routines.

If a variable was declared specifying a non-default `KIND` value that was not obtained with `selected_real_kind()` or `selected_int_kind()`, the only way to obtain a matching MPI datatype is to use the size-based mechanism described in the next section.

(End of advice to users.)

Advice to implementors. An application may often repeat a call to `MPI_TYPE_CREATE_F90_XXXX` with the same combination of `(XXXX,p,r)`. The application is not allowed to free the returned predefined, unnamed datatype handles. To prevent the creation of a potentially huge amount of handles, a high quality MPI implementation should return the same datatype handle for the same `(REAL/COMPLEX/INTEGER,p,r)` combination. Checking for the combination `(p,r)` in the preceding call to `MPI_TYPE_CREATE_F90_XXXX` and using a hash table to find formerly generated handles should limit the overhead of finding a previously generated datatype with same combination of `(XXXX,p,r)`. *(End of advice to implementors.)*

Rationale. The `MPI_TYPE_CREATE_F90_REAL/COMPLEX/INTEGER` interface needs as input the original range and precision values to be able to define useful and compiler-independent external (Section 13.5.2 on page 500) or user-defined (Section 13.5.3 on page 501) data representations, and in order to be able to perform automatic and efficient data conversions in a heterogeneous environment. *(End of rationale.)*

We now specify how the datatypes described in this section behave when used with the “external32” external data representation described in Section 13.5.2 on page 500.

The external32 representation specifies data formats for integer and floating point values. Integer values are represented in two’s complement big-endian format. Floating point values are represented by one of three IEEE formats. These are the IEEE “Single,” “Double” and “Double Extended” formats, requiring 4, 8 and 16 bytes of storage, respectively. For the IEEE “Double Extended” formats, MPI specifies a Format Width of 16 bytes, with 15 exponent bits, bias = +10383, 112 fraction bits, and an encoding analogous to the “Double” format.

The external32 representations of the datatypes returned by `MPI_TYPE_CREATE_F90_REAL/COMPLEX/INTEGER` are given by the following rules.

For `MPI_TYPE_CREATE_F90_REAL`:

```

if      (p > 33) or (r > 4931) then  external32 representation
                                     is undefined
else if (p > 15) or (r > 307) then  external32_size = 16
else if (p > 6) or (r > 37) then   external32_size = 8
else                                external32_size = 4

```

For MPI_TYPE_CREATE_F90_COMPLEX: twice the size as for MPI_TYPE_CREATE_F90_REAL.

For MPI_TYPE_CREATE_F90_INTEGER:

```

if      (r > 38) then  external32 representation is undefined
else if (r > 18) then  external32_size = 16
else if (r > 9) then   external32_size = 8
else if (r > 4) then   external32_size = 4
else if (r > 2) then   external32_size = 2
else                  external32_size = 1

```

If the external32 representation of a datatype is undefined, the result of using the datatype directly or indirectly (i.e., as part of another datatype or through a duplicated datatype) in operations that require the external32 representation is undefined. These operations include MPI_PACK_EXTERNAL, MPI_UNPACK_EXTERNAL and many MPI_FILE functions, when the “external32” data representation is used. The ranges for which the external32 representation is undefined are reserved for future standardization.

Support for Size-specific MPI Datatypes

MPI provides named datatypes corresponding to optional Fortran 77 numeric types that contain explicit byte lengths — MPI_REAL4, MPI_INTEGER8, etc. This section describes a mechanism that generalizes this model to support all Fortran numeric intrinsic types.

We assume that for each **typeclass** (integer, real, complex) and each word size there is a unique machine representation. For every pair (**typeclass**, **n**) supported by a compiler, MPI must provide a named size-specific datatype. The name of this datatype is of the form MPI_<TYPE>n in C and Fortran and of the form MPI::<TYPE>n in C++ where <TYPE> is one of REAL, INTEGER and COMPLEX, and **n** is the length in bytes of the machine representation. This datatype locally matches all variables of type (**typeclass**, **n**). The list of names for such types includes:

```

MPI_REAL4
MPI_REAL8
MPI_REAL16
MPI_COMPLEX8
MPI_COMPLEX16
MPI_COMPLEX32
MPI_INTEGER1
MPI_INTEGER2
MPI_INTEGER4
MPI_INTEGER8
MPI_INTEGER16

```

One datatype is required for each representation supported by the compiler. To be backward compatible with the interpretation of these types in MPI-1, we assume that the nonstandard

declarations `REAL*n`, `INTEGER*n`, always create a variable whose representation is of size `n`. These datatypes may also be used for variables declared with `KIND=INT8/16/32/64` or `KIND=REAL32/64/128`, which are defined in the `ISO_FORTRAN_ENV` intrinsic module. Note that the MPI datatypes and the `REAL*n`, `INTEGER*n` declarations count bytes whereas the Fortran `KIND` values count bits. All these datatypes are predefined.

The following functions allow a user to obtain a size-specific MPI datatype for any intrinsic Fortran type.

`MPI_SIZEOF(x, size)`

IN	x	a Fortran variable of numeric intrinsic type (choice)
OUT	size	size of machine representation of that type (integer)

`MPI_Sizeof(x, size, ierror) BIND(C)`

```

TYPE(*), DIMENSION(...) :: x
INTEGER, INTENT(OUT) :: size
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

`MPI_SIZEOF(X, SIZE, IERROR)`

```

<type> X
INTEGER SIZE, IERROR

```

This function returns the size in bytes of the machine representation of the given variable. It is a generic Fortran routine and has a Fortran binding only.

Advice to users. This function is similar to the C and C++ `sizeof` operator but behaves slightly differently. If given an array argument, it returns the size of the base element, not the size of the whole array. (*End of advice to users.*)

Rationale. This function is not available in other languages because it would not be useful. (*End of rationale.*)

`MPI_TYPE_MATCH_SIZE(typeclass, size, datatype)`

IN	typeclass	generic type specifier (integer)
IN	size	size, in bytes, of representation (integer)
OUT	[ticket252-W.]datatype	datatype with correct type, size (handle)

`int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *datatype)`

```

MPI_Type_match_size(typeclass, size, datatype, ierror) BIND(C)
INTEGER, INTENT(IN) :: typeclass, size
TYPE(MPI_Datatype), INTENT(OUT) :: datatype
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

`MPI_TYPE_MATCH_SIZE(TYPECLASS, SIZE, DATATYPE, IERROR)`

```

INTEGER TYPECLASS, SIZE, DATATYPE, IERROR

```

```
{static MPI::Datatype MPI::Datatype::Match_size(int typeclass,
        int size) (binding deprecated, see Section 15.2) }
```

`typeclass` is one of `MPI_TYPECLASS_REAL`, `MPI_TYPECLASS_INTEGER` and `MPI_TYPECLASS_COMPLEX`, corresponding to the desired **typeclass**. The function returns an MPI datatype matching a local variable of type (**typeclass**, **size**).

This function returns a reference (handle) to one of the predefined named datatypes, not a duplicate. This type cannot be freed. `MPI_TYPE_MATCH_SIZE` can be used to obtain a size-specific type that matches a Fortran numeric intrinsic type by first calling `MPI_SIZEOF` in order to compute the variable size, and then calling `MPI_TYPE_MATCH_SIZE` to find a suitable datatype. In C and C++, one can use the C function `sizeof()`, instead of `MPI_SIZEOF`. In addition, for variables of default kind the variable's size can be computed by a call to `MPI_TYPE_GET_EXTENT`, if the **typeclass** is known. It is erroneous to specify a size not supported by the compiler.

Rationale. This is a convenience function. Without it, it can be tedious to find the correct named type. See note to implementors below. (*End of rationale.*)

Advice to implementors. This function could be implemented as a series of tests.

```
int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *rtype)
{
    switch(typeclass) {
        case MPI_TYPECLASS_REAL: switch(size) {
            case 4: *rtype = MPI_REAL4; return MPI_SUCCESS;
            case 8: *rtype = MPI_REAL8; return MPI_SUCCESS;
            default: error(...);
        }
        case MPI_TYPECLASS_INTEGER: switch(size) {
            case 4: *rtype = MPI_INTEGER4; return MPI_SUCCESS;
            case 8: *rtype = MPI_INTEGER8; return MPI_SUCCESS;
            default: error(...);
        }
        ... etc. ...
    }
}
```

(*End of advice to implementors.*)

Communication With Size-specific Types

The usual type matching rules apply to size-specific datatypes: a value sent with datatype `MPI_<TYPE>n` can be received with this same datatype on another process. Most modern computers use 2's complement for integers and IEEE format for floating point. Thus, communication using these size-specific datatypes will not entail loss of precision or truncation errors.

Advice to users. Care is required when communicating in a heterogeneous environment. Consider the following code:

```

1  real(selected_real_kind(5)) x(100)
2  call MPI_SIZEOF(x, size, ierror)
3  call MPI_TYPE_MATCH_SIZE(MPI_TYPECLASS_REAL, size, xtype, ierror)
4  if (myrank .eq. 0) then
5      ... initialize x ...
6      call MPI_SEND(x, xtype, 100, 1, ...)
7  else if (myrank .eq. 1) then
8      call MPI_RECV(x, xtype, 100, 0, ...)
9  endif

```

This may not work in a heterogeneous environment if the value of `size` is not the same on process 1 and process 0. There should be no problem in a homogeneous environment. To communicate in a heterogeneous environment, there are at least four options. The first is to declare variables of default type and use the MPI datatypes for these types, e.g., declare a variable of type `REAL` and use `MPI_REAL`. The second is to use `selected_real_kind` or `selected_int_kind` and with the functions of the previous section. The third is to declare a variable that is known to be the same size on all architectures (e.g., `selected_real_kind(12)` on almost all compilers will result in an 8-byte representation). The fourth is to carefully check representation size before communication. This may require explicit conversion to a variable of size that can be communicated and handshaking between sender and receiver to agree on a size.

Note finally that using the “external32” representation for I/O requires explicit attention to the representation sizes. Consider the following code:

```

26  real(selected_real_kind(5)) x(100)
27  call MPI_SIZEOF(x, size, ierror)
28  call MPI_TYPE_MATCH_SIZE(MPI_TYPECLASS_REAL, size, xtype, ierror)
29
30  if (myrank .eq. 0) then
31      call MPI_FILE_OPEN(MPI_COMM_SELF, 'foo',          &
32                        MPI_MODE_CREATE+MPI_MODE_WRONLY, &
33                        MPI_INFO_NULL, fh, ierror)
34      call MPI_FILE_SET_VIEW(fh, 0, xtype, xtype, 'external32', &
35                            MPI_INFO_NULL, ierror)
36      call MPI_FILE_WRITE(fh, x, 100, xtype, status, ierror)
37      call MPI_FILE_CLOSE(fh, ierror)
38  endif
39
40  call MPI_BARRIER(MPI_COMM_WORLD, ierror)
41
42  if (myrank .eq. 1) then
43      call MPI_FILE_OPEN(MPI_COMM_SELF, 'foo', MPI_MODE_RDONLY, &
44                        MPI_INFO_NULL, fh, ierror)
45      call MPI_FILE_SET_VIEW(fh, 0, xtype, xtype, 'external32', &
46                            MPI_INFO_NULL, ierror)
47      call MPI_FILE_WRITE(fh, x, 100, xtype, status, ierror)
48      call MPI_FILE_CLOSE(fh, ierror)

```



```
endif
```

If processes 0 and 1 are on different machines, this code may not work as expected if the size is different on the two machines. (*End of advice to users.*)

16.2.10 Problems With Fortran Bindings for MPI

This section discusses a number of problems that may arise when using MPI in a Fortran program. It is intended as advice to users, and clarifies how MPI interacts with Fortran. It does not add to the standard, but is intended to clarify the standard.

As noted in the original MPI specification, the interface violates the Fortran standard in several ways. While these **may** cause few problems for Fortran 77 programs, they become more significant for Fortran 90 programs, so that users must exercise care when using new Fortran 90 features. **With Fortran 2008 and the new semantics defined in TR 29113, most violations are resolved, and this is hinted at in an addendum to each item.** The violations were originally adopted and have been retained because they are important for the usability of MPI. The rest of this section describes the potential problems in detail.

The following MPI features are inconsistent with Fortran 90 **and Fortran 77.**

1. An MPI subroutine with a choice argument may be called with different argument types. **When using the `mpi_f08` module together with a compiler that supports Fortran 2008 + TR 29113, this problem is resolved.**
2. An MPI subroutine with an assumed-size dummy argument may be passed an actual scalar argument. **This is only solved for choice buffers through the use of `DIMENSION(...)`.**
3. **Nonblocking and split-collective** MPI routines assume that actual arguments are passed by address or descriptor and that arguments and the associated data are not copied on entrance to or exit from the subroutine. **This problem is solved with the use of the `ASYNCHRONOUS` attribute.**
4. An MPI implementation may read or modify user data (e.g., communication buffers used by nonblocking communications) concurrently with a user program that is executing outside of MPI calls. **This problem is resolved by relying on the extended semantics of the `ASYNCHRONOUS` attribute as specified in TR 29113.**
5. Several named “constants,” such as `MPI_BOTTOM`, `MPI_IN_PLACE`, `MPI_STATUS_IGNORE`, `MPI_STATUSES_IGNORE`, `MPI_ERRCODES_IGNORE`, `MPI_UNWEIGHTED`, `MPI_ARGV_NULL`, and `MPI_ARGVS_NULL` are not ordinary Fortran constants and require a special implementation. See Section 2.5.4 on page 15 for more information.
6. The memory allocation routine `MPI_ALLOC_MEM` can’t be usefully used in Fortran 77/90/95 without a language extension (for example, Cray pointers) that allows the allocated memory to be associated with a Fortran variable. **Therefore, address sized integers were used in MPI-2.0 - MPI-2.2. In Fortran 2003, `TYPE(C_PTR)` entities were added, which allow a standard-conforming implementation of the semantics of `MPI_ALLOC_MEM`. In MPI-3.0 and later, `MPI_ALLOC_MEM` has an additional,**

overloaded interface to support this language feature. The use of Cray pointers is deprecated. The `mpi_f08` module only supports `TYPE(C_PTR)` pointers.

Additionally, MPI is inconsistent with Fortran 77 in a number of ways, as noted below.

- MPI identifiers exceed 6 characters.
- MPI identifiers may contain underscores after the first character.
- MPI requires an include file, `mpif.h`. On systems that do not support include files, the implementation should specify the values of named constants.
- Many routines in MPI have KIND-parameterized integers (e.g., `MPI_ADDRESS_KIND` and `MPI_OFFSET_KIND`) that hold address information. On systems that do not support Fortran 90-style parameterized types, `INTEGER*8` or `INTEGER` should be used instead.

MPI-1 contained several routines that take address-sized information as input or return address-sized information as output. In C such arguments were of type `MPI_Aint` and in Fortran of type `INTEGER`. On machines where integers are smaller than addresses, these routines can lose information. In MPI-2 the use of these functions has been deprecated and they have been replaced by routines taking `INTEGER` arguments of `KIND=MPI_ADDRESS_KIND`. A number of new MPI-2 functions also take `INTEGER` arguments of non-default `KIND`. See Section 2.6 on page 17 and Section 4.1.1 on page 87 for more information.

Sections 16.2.11 through 16.2.19 describe several problems in detail which concern the interaction of MPI and Fortran as well as their solutions. Some of these solutions require special capabilities from the compilers. Major requirements are summarized in Section 16.2.7 on page 568.

16.2.11 Problems Due to Strong Typing

All MPI functions with choice arguments associate actual arguments of different Fortran datatypes with the same dummy argument. This is not allowed by Fortran 77, and in Fortran 90, it is technically only allowed if the function is overloaded with a different function for each type (see also Section 16.2.6 on page 564). In C, the use of `void*` formal arguments avoids these problems. Similar to C, with Fortran 2008 + TR 29113 (and later) together with the `mpi_f08` module, the problem is avoided by declaring choice arguments with `TYPE(*)`, `DIMENSION(...)`, i.e., as assumed-type and assumed-rank dummy arguments.

Using `INCLUDE mpif.h`, the following code fragment is technically invalid and may generate a compile-time error.

```
integer i(5)
real    x(5)
...
call mpi_send(x, 5, MPI_REAL, ...)
call mpi_send(i, 5, MPI_INTEGER, ...)
```

In practice, it is rare for compilers to do more than issue a warning. Using the `mpi_f08` or `mpi` module, the problem is usually resolved through the assumed-type and assumed-rank declarations of the dummy arguments, or with a compiler-dependent mechanism that overrides type checking for choice arguments.

ticket235-G.

It is also technically *invalid* in Fortran to pass a scalar actual argument to an array dummy argument that is not a choice buffer argument. Thus, when using the `mpi_f08` or `mpi` module, the following code fragment usually generates an error since the `dims` and `periods` arguments to `MPI_CART_CREATE` are declared as assumed size arrays `INTEGER :: DIMS(*)` and `LOGICAL :: PERIODS(*)`.

```
USE mpi_f08      ! or  USE mpi
INTEGER size
CALL MPI_Cart_create( comm_old,1,size,.TRUE.,.TRUE.,comm_cart,ierror )
```

Although this is a non-conforming MPI call, compiler warnings are not expected (but may occur) when using `INCLUDE 'mpif.h'` and this include file does not use Fortran explicit interfaces.

16.2.12 Problems Due to Data Copying and Sequence Association with Subscript Triplets

Arrays with subscript **triplets** describe Fortran subarrays with or without strides, e.g.,

```
REAL a(100,100,100)
CALL MPI_Send( a(11:17, 12:99:3, 1:100), 7*30*100, MPI_REAL, ...)
```

The handling of subscript triplets depends on the value of the constant `MPI_SUBARRAYS_SUPPORTED`:

- If `MPI_SUBARRAYS_SUPPORTED` equals `.TRUE.:`

Choice buffer arguments are declared as `TYPE(*)`, `DIMENSION(...)`. For example, consider the following code fragment:

```
REAL s(100), r(100)
CALL MPI_Isend(s(1:100:5), 3, MPI_REAL, ..., rq, ierror)
CALL MPI_Wait(rq, status, ierror)
CALL MPI_Irecv(r(1:100:5), 3, MPI_REAL, ..., rq, ierror)
CALL MPI_Wait(rq, status, ierror)
```

In this case, the individual elements `s(1)`, `s(6)`, and `s(11)` are sent between the start of `MPI_ISEND` and the end of `MPI_WAIT` even though the compiled code will not copy `s(1:100:5)` to a real contiguous temporary scratch buffer. Instead, the compiled code will pass a descriptor to `MPI_ISEND` that allows MPI to operate directly on `s(1)`, `s(6)`, `s(11)`, ..., `s(96)`. The called `MPI_ISEND` routine will take only the first three of these elements due to the type signature “3, `MPI_REAL`”.

All nonblocking MPI functions (e.g., `MPI_ISEND`, `MPI_PUT`, `MPI_FILE_WRITE_ALL_BEGIN`) behave as if *the user-specified elements of choice buffers are copied to a contiguous scratch buffer in the MPI runtime environment*. All datatype descriptions (in the example above, “3, `MPI_REAL`”) read and store data from and to this virtual contiguous scratch buffer. Displacements in MPI derived datatypes are relative to the beginning of this virtual contiguous scratch buffer. Upon completion of a nonblocking receive operation (e.g., when `MPI_WAIT` on a corresponding `MPI_Request` returns), it is as if the received data has been copied from

the virtual contiguous scratch buffer back to the non-contiguous application buffer. In the example above, `r(1)`, `r(6)`, and `r(11)` are guaranteed to be defined with the received data when `MPI_WAIT` returns.

Advice to implementors. The Fortran descriptor for `TYPE(*)`, `DIMENSION(...)` arguments contains enough information that, if desired, the MPI library can make a real contiguous copy of non-contiguous user buffers when the nonblocking operation is started, and released this buffer not before the nonblocking communication has completed (e.g., in an MPI wait routine). Efficient implementations may avoid such additional memory-to-memory data copying. (*End of advice to implementors.*)

Rationale. If `MPI_SUBARRAYS_SUPPORTED` equals `.TRUE.`, non-contiguous buffers are handled inside of the MPI library instead of by the compiler through argument association conventions. Therefore, the scope of MPI library scratch buffers can be from the beginning of a nonblocking operation until the completion of the operation although beginning and completion are implemented in different routines. (*End of rationale.*)

- If `MPI_SUBARRAYS_SUPPORTED` equals `.FALSE.`:

Implicit in MPI is the idea of a contiguous chunk of memory accessible through a linear address space. MPI copies data to and from this memory. An MPI program specifies the location of data by providing memory addresses and offsets. In the C language, sequence association rules plus pointers provide all the necessary low-level structure.

In Fortran, `array` data is not necessarily stored contiguously. For example, the array section `A(1:N:2)` involves only the elements of `A` with indices 1, 3, 5, The same is true for a pointer array whose target is such a section. Most compilers ensure that an array that is a dummy argument is held in contiguous memory if it is declared with an explicit shape (e.g., `B(N)`) or is of assumed size (e.g., `B(*)`). If necessary, they do this by making a copy of the array into contiguous memory.¹

Because MPI dummy buffer arguments are assumed-size arrays if `MPI_SUBARRAYS_SUPPORTED` equals `.FALSE.`, this leads to a serious problem for a nonblocking call: the compiler copies the temporary array back on return but MPI continues to copy data to the memory that held it. For example, consider the following code fragment:

```
real a(100)
call MPI_Irecv(a(1:100:2), MPI_REAL, 50, ...)
```

Since the first dummy argument to `MPI_Irecv` is an assumed-size array (`<type> buf(*)`), the array section `a(1:100:2)` is copied to a temporary before being passed to `MPI_Irecv`, so that it is contiguous in memory. `MPI_Irecv` returns immediately, and data is copied from the temporary back into the array `a`. Sometime later, MPI may write to the address of the deallocated temporary. Copying is also a problem

¹Technically, the Fortran standard is worded to allow non-contiguous storage of any array data, unless the dummy argument has the `CONTIGUOUS` attribute.

for MPI_ISEND since the temporary array may be deallocated before the data has all been sent from it.

Most Fortran 90 compilers do not make a copy if the actual argument is the whole of an explicit-shape or assumed-size array or is a ‘**simply contiguous**’ section such as `A(1:N)` of such an array. (‘**Simply contiguous**’ is defined in the next paragraph.) Also, many compilers treat allocatable arrays the same as they treat explicit-shape arrays in this regard (though we know of one that does not). However, the same is not true for assumed-shape and pointer arrays; since they may be discontinuous, copying is often done. It is this copying that causes problems for MPI as described in the previous paragraph.

According to the Fortran 2008 Standard, Section 6.5.4, a ‘**simply contiguous**’ array section is

```
name ( [:,]... [<subscript>]:[<subscript>] [,<subscript>]... )
```

That is, there are zero or more dimensions that are selected in full, then one dimension selected without a stride, then zero or more dimensions that are selected with a simple subscript. The compiler can detect from analyzing the source code that the array is **contiguous**. Examples are

```
A(1:N), A(:,N), A(:,1:N,1), A(1:6,N), A(:, :, 1:N)
```

Because of Fortran’s column-major ordering, where the first index varies fastest, a ‘**simply contiguous**’ section of a contiguous array will also be contiguous.

The same problem can occur with a scalar argument. A compiler **may** make a copy of scalar dummy arguments within a called procedure **when passed as an actual argument to a choice buffer routine**. That this can cause a problem is illustrated by the example

```
[ticket236-H.]real :: a
call user1(a,rq)
call MPI_WAIT(rq,status,ierr)
write (*,*) a

subroutine user1(buf,request)
call MPI_IRecv(buf,...,request,...)
end
```

If `a` is copied, MPI_IRecv will alter the copy when it completes the communication and will not alter `a` itself.

Note that copying will almost certainly occur for an argument that is a non-trivial expression (one with at least one operator or function call), a section that does not select a contiguous part of its parent (e.g., `A(1:n:2)`), a pointer whose target is such a section, or an assumed-shape array that is (directly or indirectly) associated with such a section.

If a compiler option **exists** that inhibits copying of arguments, in either the calling or called procedure, this **must** be employed.

If a compiler makes copies in the calling procedure of arguments that are explicit-shape or assumed-size arrays, ‘**simply contiguous**’ array sections of such arrays, or scalars, and if **no compiler option exists to inhibit such copying**, then the compiler cannot be used for applications that use `MPI_GET_ADDRESS`, or any nonblocking MPI routine. If a compiler copies scalar arguments in the called procedure and there is no compiler option to inhibit this, then this compiler cannot be used for applications that use memory references across subroutine calls as in the example above.

ticket229.4.
ticket236-H.

16.2.13 Problems Due to Data Copying and Sequence Association with Vector Subscripts

Fortran arrays with **vector** subscripts describe subarrays containing a possibly irregular set of elements

```
REAL a(100)
CALL MPI_Send( A((/7,9,23,81,82/)), 5, MPI_REAL, ...)
```

Arrays with a vector subscript must not be used as actual choice buffer arguments in any nonblocking or split collective MPI operations. They may, however, be used in blocking MPI operations.

16.2.14 Special Constants

MPI requires a number of special “constants” that cannot be implemented as normal Fortran constants, e.g., `MPI_BOTTOM`. The complete list can be found in Section 2.5.4 on page 15. In C, these are implemented as constant pointers, usually as `NULL` and are used where the function prototype calls for a pointer to a variable, not the variable itself.

In Fortran, **using** special values for the constants (e.g., by defining them through **parameter** statements) is not possible because an implementation cannot distinguish these values from legal data. Typically these constants are implemented as predefined static variables (e.g., a variable in an MPI-declared `COMMON` block), relying on the fact that the target compiler passes data by address. Inside the subroutine, **the address of the actual choice buffer argument can be compared with the address of such a predefined static variable**.

These special constants also cause an exception with the usage of Fortran `INTENT`: with `USE mpi_f08`, the attributes `INTENT(IN)`, `INTENT(OUT)`, and `INTENT(INOUT)` are used in the Fortran interface. In most cases, `INTENT(IN)` is used if the C interface uses call-by-value. For all buffer arguments and for dummy arguments that may be modified and allow one of these special constants as input, an `INTENT` is not specified.

16.2.15 Fortran Derived Types

MPI supports passing Fortran **entities of `BIND(C)` and `SEQUENCE` derived types to choice dummy arguments**, **provided no type component has the `ALLOCATABLE` or `POINTER` attribute**.

The following code fragment shows **some possible ways to send scalars or arrays of interoperable** derived type in Fortran. The example assumes that all data is passed by address.

```
type[ticket237-I.], BIND(C) :: mytype
integer [ticket229.2.]:: i
```

```

real [ticket229.2.]:: x                                     1
double precision [ticket229.2.]:: d                         2
[ticket229.2.]logical :: l                                 3
end type mytype                                           4
                                                            5

type(mytype) [ticket250-V.]:: foo[ticket237-I.], fooarr(5) 6
integer [ticket250-V.]:: blocklen(4), type(4)             7
integer(KIND=MPI_ADDRESS_KIND) [ticket250-V.]:: disp(4), base[ticket237-I.], lb, extent 8, lb, extent
                                                            9

call MPI_GET_ADDRESS(foo%i, disp(1), ierr)                10
call MPI_GET_ADDRESS(foo%x, disp(2), ierr)                11
call MPI_GET_ADDRESS(foo%d, disp(3), ierr)                12
[ticket229.2.]call MPI_GET_ADDRESS(foo%l, disp(4), ierr)   13
                                                            14

base = disp(1)                                           15
disp(1) = disp(1) - base                                 16
disp(2) = disp(2) - base                                 17
disp(3) = disp(3) - base                                 18
[ticket229.2.]disp(4) = disp(4) - base                    19
                                                            20

blocklen(1) = 1                                           21
blocklen(2) = 1                                           22
blocklen(3) = 1                                           23
[ticket229.2.]blocklen(4) = 1                             24
                                                            25

type(1) = MPI_INTEGER                                    26
type(2) = MPI_REAL                                        27
type(3) = MPI_DOUBLE_PRECISION                           28
[ticket229.2.]type(4) = MPI_LOGICAL                       29
                                                            30

call MPI_TYPE_CREATE_STRUCT(4, blocklen, disp, type, newtype, ierr) 31
call MPI_TYPE_COMMIT(newtype, ierr)                      32
                                                            33

[ticket237-I.]                                           34
[ticket237-I.]                                           35
    call MPI_SEND(foo%i, 1, newtype, ...)                36
[ticket237-I.]! or                                       37
[ticket237-I.]    call MPI_SEND(foo, 1, newtype, ...)     38
[ticket237-I.]    ! expects that base == address(foo%i) == address(foo) 39
                                                            40

[ticket237-I.]    call MPI_GET_ADDRESS(fooarr(1), disp(1), ierr) 41
[ticket237-I.]    call MPI_GET_ADDRESS(fooarr(2), disp(2), ierr) 42
[ticket237-I.]    extent = disp(2) - disp(1)              43
[ticket237-I.]    lb = 0                                   44
[ticket237-I.]    call MPI_TYPE_CREATE_RESIZED(newtype, lb, extent, newarrtype, ierr) 45
[ticket237-I.]    call MPI_TYPE_COMMIT(newarrtype, ierr)    46
[ticket237-I.]                                           47
[ticket237-I.]    call MPI_SEND(fooarr, 5, newarrtype, ...) 48

```


Using the derived type variable `foo` instead of its first basic type element `foo%i` may be impossible if the MPI library implements choice buffer arguments through overloading instead of using `TYPE(*)`, `DIMENSION(..)`, or through a non-standardized extensions such as `!$PRAGMA IGNORE_TKR`; see Section 16.2.6 on page 564.

To use a derived type in an array requires a correct extent of the datatype handle to take care of the alignment rules applied by the compiler. These alignment rules may imply that there are gaps between the elements of a derived type, and also between the array elements. The extent of an iteroperable derived type (i.e., defined with `BIND(C)`) and a `SEQUENCE` derived type with the same content may be different because C and Fortran may apply different alignment rules. As recommended in the advice to users in Section 4.1 on page 85, one should add an additional fifth structure element with one numerical storage unit at the end of this structure to force in most cases that the array of structures is contiguous. Even with such an additional element, one should keep this resizing due to the special alignment rules that can be used by the compiler for structures, as also mentioned in this advice.

Using the extended semantics defined in TR 29113, it is also possible to use entities or derived types without either the `BIND(C)` or the `SEQUENCE` attribute as choice buffer arguments; some additional constraints must be observed e.g., no `ALLOCATABLE` or `POINTER` type components may exist. In this case, the `base` address in the example must be changed to become the address of `foo` instead of `foo%i`, because the Fortran compiler may rearrange type components or add padding as it may fit for such types. Sending the structure `foo` should then also be performed by providing it (and not `foo%i`) as actual argument for `MPI_Send`.

16.2.16 Optimization Problems, an Overview

MPI provides operations that may be hidden from the user code and run concurrently with it, accessing the same memory as user code. Examples include the data transfer for an `MPI_IRECV`. The optimizer of a compiler will assume that it can recognize periods when a copy of a variable can be kept in a register without reloading from or storing to memory. When the user code is working with a register copy of some variable while the hidden operation reads or writes the memory copy, problems occur. These problems are independent of the Fortran support method; i.e., they occur with the `mpi_f08` module, the `mpi` module, and the `mpif.h` include file.

This section shows four problematic usage areas (the abbreviations in parentheses are used in the table below):

- Use of nonblocking routines or persistent requests (*Nonbl.*).
- Use of one-sided routines (*1-sided*).
- Use of MPI parallel file I/O split collective operations (*Split*).
- Use of `MPI_BOTTOM` together with absolute displacements in MPI datatypes, or relative displacements between two variables in such datatypes (*Bottom*).

The following compiler optimization strategies (valid for serial code) may cause problems in MPI applications:

- Code movement and register optimization problems; see Section 16.2.17 on page 587.

- Temporary data movement and temporary memory modifications; see Section 16.2.18 on page 594.
- Permanent data movement (e.g., through garbage collection); see Section 16.2.19 on page 597.

Table 16.4 shows in which usage areas the optimization problems may only occur.

Optimization may cause a problem in following usage areas			
	Nonbl.	1-sided	Split	Bottom
Code movement and register optimization	yes	yes	no	yes
Temporary data movement	yes	yes	yes	no
Permanent data movement	yes	yes	yes	yes

Table 16.4: Occurrence of Fortran optimization problems in several usage areas

The solutions in the following sections are based on compromises:

- to minimize the burden for the application programmer, e.g., as shown in Sections “Solutions” to “VOLATILE” on pages 589-591,
- to minimize the drawbacks on compiler based optimization, and
- to minimize the requirements defined in Section 16.2.7 on page 568.

16.2.17 Problems with Code Movement and Register Optimization

Nonblocking operations

If a variable is local to a Fortran subroutine (i.e., not in a module or a COMMON block), the compiler will assume that it cannot be modified by a called subroutine unless it is an actual argument of the call. In the most common linkage convention, the subroutine is expected to save and restore certain registers. Thus, the optimizer will assume that a register which held a valid copy of such a variable before the call will still hold a valid copy on return.

Example 16.11 Fortran 90 register optimization – extreme.

Source	compiled as	or compiled as
[ticket238-J.] <code>REAL :: buf, b1</code>	<code>REAL :: buf, b1</code>	<code>REAL :: buf, b1</code>
<code>call MPI_Irecv(buf,..req)</code>	<code>call MPI_Irecv(buf,..req)</code>	<code>call MPI_Irecv(buf,..req)</code>
	<code>register = buf</code>	<code>b1 = buf</code>
<code>call MPI_WAIT(req,..)</code>	<code>call MPI_WAIT(req,..)</code>	<code>call MPI_WAIT(req,..)</code>
<code>b1 = buf</code>	<code>b1 = register</code>	

Example 16.11 shows extreme, but allowed, possibilities. MPI_WAIT on a concurrent thread modifies buf between the invocation of MPI_Irecv and the finish of MPI_WAIT. But the compiler cannot see any possibility that buf can be changed after MPI_Irecv has returned, and may schedule the load of buf earlier than typed in the source. The compiler

[ticket238-J.]

Example 16.12 Similar example with MPI_ISEND

Source	compiled as	with a possible MPI-internal execution sequence
REAL :: buf, copy	REAL :: buf, copy	REAL :: buf, copy
buf = val	buf = val	buf = val
call MPI_ISEND(buf,..req)	call MPI_ISEND(buf,..req)	addr = &buf
copy = buf	copy= buf	copy= buf
	buf = val_overwrite	buf = val_overwrite
call MPI_WAIT(req,..)	call MPI_WAIT(req,..)	send(*addr) ! within MPI_WAIT
buf = val_overwrite		

has no reason to avoid using a register to hold buf across the call to MPI_WAIT. It also may reorder the instructions as illustrated in the rightmost column.

Due to valid compiler code movement optimizations in Example 16.12, the content of buf may already be overwritten by the compiler when the content of buf is sent. The code movement is permitted because the compiler cannot detect a possible access to buf in MPI_WAIT (or in a second thread between the start of MPI_ISEND and the end of MPI_WAIT).

Such register optimization is based on moving code; here, the access to buf was moved from after MPI_WAIT to before MPI_WAIT. Note that code movement may also occur across subroutine boundaries when subroutines or functions are inlined.

This register optimization / code movement problem for nonblocking operations does not occur with MPI parallel file I/O split collective operations, because in the ..._BEGIN and ..._END calls, the same buffer has to be provided as an actual argument. The register optimization / code movement problem for MPI_BOTTOM and derived MPI datatypes may occur in each blocking and nonblocking communication or parallel file I/O operation.

One-sided communication

An example with instruction reordering due to register optimization can be found in Section 11.7.3 on page 433.

MPI_BOTTOM and combining independent variables in datatypes

This section is only relevant if the MPI program uses a buffer argument to an MPI_SEND, MPI_RECV etc., which hides the actual variables involved. MPI_BOTTOM with an MPI_Datatype containing absolute addresses is one example. Creating a datatype which uses one variable as an anchor and brings along others by using MPI_GET_ADDRESS to determine their offsets from the anchor is another. The anchor variable would be the only one referenced in the call. Also attention must be paid if MPI operations are used that run in parallel with the user's application.

Example 16.13 shows what Fortran compilers are allowed to do.

In Example 16.13, the compiler does not invalidate the register because it cannot see that MPI_RECV changes the value of buf. The access to buf is hidden by the use of MPI_GET_ADDRESS and MPI_BOTTOM.

In Example 16.14, several successive assignments to the same variable buf can be combined in a way such that only the last assignment is executed. "Successive" means that

Example 16.13 Fortran 90 register optimization.

This source ...

```
call MPI_GET_ADDRESS(buf,bufaddr,
                     ierror)
call MPI_TYPE_CREATE_STRUCT(1,1,
                             bufaddr,
                             MPI_REAL,type,ierror)
call MPI_TYPE_COMMIT(type,ierror)
val_old = buf

call MPI_RECV(MPI_BOTTOM,1,type,...)
val_new = buf
```

[ticket238-J.]

can be compiled as:

```
call MPI_GET_ADDRESS(buf,...)

call MPI_TYPE_CREATE_STRUCT(...)

call MPI_TYPE_COMMIT(...)
register = buf
val_old = register

call MPI_RECV(MPI_BOTTOM,...)
val_new = register
```

Example 16.14 Similar example with MPI_SEND

This source ...

```
! buf contains val_old
buf = val_new
call MPI_SEND(MPI_BOTTOM,1,type,...)
! with buf as a displacement in type

buf = val_overwrite
```

can be compiled as:

```
! buf contains val_old

call MPI_SEND(...)
! i.e. val_old is sent
!
! buf=val_new is moved to here
! and detected as dead code
! and therefore removed
!

buf = val_overwrite
```

no interfering load access to this variable occurs between the assignments. The compiler cannot detect that the call to MPI_SEND statement is interfering because the load access to buf is hidden by the usage of MPI_BOTTOM.

Solutions

The following sections show in detail how the problems with code movement and register optimization can be solved in a portable way. Application writers can partially or fully avoid these compiler optimization problems by using one or more of the special Fortran declarations with the send and receive buffers used in nonblocking operations, or in operations in which MPI_BOTTOM is used, or datatype handles that combine several variables are used:

- Use of the Fortran ASYNCHRONOUS attribute.
- Use of the helper routine MPI_F_SYNC_REG, or an equivalent user-written dummy routine.

- Declare the buffer as a Fortran module variable or within a Fortran common block.
- Use of the Fortran `VOLATILE` attribute.

[ticket238-J.]

Example 16.15 Protecting nonblocking communication with the `ASYNCHRONOUS` attribute.

```

USE mpi_f08
REAL, ASYNCHRONOUS :: b(0:101) ! elements 0 and 101 are halo cells
REAL :: bnew(0:101)           ! elements 1 and 100 are newly computed
TYPE(MPI_Request) :: req(4)
INTEGER :: left, right, i
CALL MPI_Cart_shift(...,left,right,...)
CALL MPI_Irecv(b( 0), ..., left, ..., req(1), ...)
CALL MPI_Irecv(b(101), ..., right, ..., req(2), ...)
CALL MPI_Isend(b( 1), ..., left, ..., req(3), ...)
CALL MPI_Isend(b(100), ..., right, ..., req(4), ...)

#ifdef WITHOUT_OVERLAPPING_COMMUNICATION_AND_COMPUTATION
! Case (a)
CALL MPI_Waitall(4,req,...)
DO i=1,100 ! compute all new local data
  bnew(i) = function(b(i-1), b(i), b(i+1))
END DO
#endif

#ifdef WITH_OVERLAPPING_COMMUNICATION_AND_COMPUTATION
! Case (b)
DO i=2,99 ! compute only elements for which halo data is not needed
  bnew(i) = function(b(i-1), b(i), b(i+1))
END DO
CALL MPI_Waitall(4,req,...)
i=1 ! compute leftmost element
  bnew(i) = function(b(i-1), b(i), b(i+1))
i=100 ! compute rightmost element
  bnew(i) = function(b(i-1), b(i), b(i+1))
#endif

```

Each of these methods solves the problems of code movement and register optimization, but may involve different degrees of performance impact, and may not be usable in every application context. These methods may not be guaranteed by the Fortran standard, but they must be guaranteed by a MPI-3.0 compliant (and later) MPI library and their compiler according to the requirements listed in Section 16.2.7 on page 568. The methods may have different impact on performance. `MPI_F_SYNC_REG` may have low impact, module data and the `ASYNCHRONOUS` attribute low through medium, and the `VOLATILE` attribute may have the most negative impact on performance. Note that there is one attribute that cannot be

used for this purpose: the Fortran **TARGET** attribute does not solve code movement problems in MPI applications.

The Fortran **ASYNCHRONOUS** attribute

Declaring an actual buffer argument with the **ASYNCHRONOUS** Fortran attribute in a scoping unit (or **BLOCK**) tells the compiler that any statement in the scoping unit may be executed while the buffer is affected by a pending asynchronous Fortran input/output operation (since Fortran 2003) or by an asynchronous communication (TR 29113 extension). Without the extensions specified in TR 29113, a Fortran compiler may totally ignore this attribute if the Fortran compiler implements asynchronous Fortran input/output operations with blocking I/O. The **ASYNCHRONOUS** attribute protects the buffer accesses from optimizations through code movements across routine calls, and the buffer itself from temporary and permanent data movements. If the choice buffer dummy argument of a nonblocking MPI routine is declared with **ASYNCHRONOUS** (which is mandatory for the `mpi_f08` module, with allowable exceptions listed in Section 16.2.6 on page 564), then the compiler has to guarantee call by reference and should report a compile-time error if call by reference is impossible, e.g., if vector subscripts are used. The `MPI_ASYNC_PROTECTS_NONBLOCKING` is set to `.TRUE.` if both the protection of the actual buffer argument through **ASYNCHRONOUS** according to the TR 29113 extension and the declaration of the dummy argument with **ASYNCHRONOUS** in the Fortran support method is guaranteed for all nonblocking routines, otherwise it is set to `.FALSE.`

The **ASYNCHRONOUS** attribute has some restrictions. The TR 29113 defines (in the PDTR N1869):

“Asynchronous communication for a Fortran variable occurs through the action of procedures defined by means other than Fortran. It is initiated by execution of an asynchronous communication initiation procedure and completed by execution of an asynchronous communication completion procedure. Between the execution of the initiation and completion procedures, any variable of which any part is associated with any part of the asynchronous communication variable is a pending communication affector. Whether a procedure is an asynchronous communication initiation or completion procedure is processor dependent. Asynchronous communication is either input communication or output communication. For input communication, a pending communication affector shall not be referenced, become defined, become undefined, become associated with a dummy argument that has the **VALUE** attribute, or have its pointer association status changed. For output communication, a pending communication affector shall not be redefined, become undefined, or have its pointer association status changed.”

In Example 16.15 Case (a) on page 590, the read accesses to `b` within `function(b(i-1), b(i), b(i+1))` cannot be moved by compiler optimizations to before the wait call because `b` was declared as **ASYNCHRONOUS**. Note that only the elements 0, 1, 100, and 101 of `b` are involved in asynchronous communication but by definition, the total variable `b` is the pending communication affector and is usable for input and output asynchronous communication between the `MPI_I...` routines and `MPI_Waitall`. Case (a) works fine because the read accesses to `b` occur after the communication completed.

In Case (b), the read accesses to `b(1:100)` in the loop `i=2,99` are read accesses to a pending communication affector while input communication (i.e., the two `MPI_Irecv` calls) is pending. This is a contradiction to the rule that *for input communication, a pending communication affector shall not be referenced*. The problem can be solved by using separate variables for the halos and the inner array, or by splitting a common array into disjunct subarrays which are passed through different dummy arguments into a subroutine, as shown in Example 16.19 on page 598.

If one does not overlap communication and computation on the same variable, then all optimization problems can be solved through the `ASYNCHRONOUS` attribute.

The problems with `MPI_BOTTOM`, as shown in Example 16.13 and Example 16.14, can also be solved by declaring the buffer `buf` with the `ASYNCHRONOUS` attribute.

In some MPI routines, a buffer dummy argument is defined as `ASYNCHRONOUS` to guarantee passing by reference, provided that the actual argument is also defined as `ASYNCHRONOUS`.

Calling `MPI_F_SYNC_REG`

The compiler may be prevented from moving a reference to a buffer across a call to an MPI subroutine by surrounding the call by calls to an external subroutine with the buffer as an actual argument. The MPI library provides the `MPI_F_SYNC_REG` routine for this purpose; see Section 16.2.8 on page 569.

- The problems illustrated by the Examples 16.11 and 16.12 can be solved by calling `MPI_F_SYNC_REG(buf)` once immediately after `MPI_WAIT`.

Example 16.11

can be solved with

```
call MPI_Irecv(buf, ..req)
```

```
call MPI_WAIT(req, ..)
```

```
call MPI_F_SYNC_REG(buf)
```

```
b1 = buf
```

Example 16.12

can be solved with

```
buf = val
```

```
call MPI_Isend(buf, ..req)
```

```
copy = buf
```

```
call MPI_WAIT(req, ..)
```

```
call MPI_F_SYNC_REG(buf)
```

```
buf = val_overwrite
```

The call to `MPI_F_SYNC_REG(buf)` prevents moving the last line before the `MPI_WAIT` call. Further calls to `MPI_F_SYNC_REG(buf)` are not needed because it is still correct if the additional read access `copy=buf` is moved below `MPI_WAIT` and before `buf=val_overwrite`.

- The problems illustrated by the Examples 16.13 and 16.14 can be solved with two additional `MPI_F_SYNC_REG(buf)` statements; one directly before `MPI_RECV/` `MPI_SEND`, and one directly after this communication operation.

Example 16.13

can be solved with

```
call MPI_F_SYNC_REG(buf)
```

```
call MPI_RECV(MPI_BOTTOM, ...)
```

```
call MPI_F_SYNC_REG(buf)
```

Example 16.14

can be solved with

```
call MPI_F_SYNC_REG(buf)
```

```
call MPI_SEND(MPI_BOTTOM, ...)
```

```
call MPI_F_SYNC_REG(buf)
```

The first call to `MPI_F_SYNC_REG(buf)` is needed to finish all load and store references to `buf` prior to `MPI_RECV/MPI_SEND`; the second call is needed to assure that the subsequent access to `buf` are not moved before `MPI_RECV/SEND`.

- In the example in Section 11.7.3 on page 433, two asynchronous accesses must be protected: in Process 1, the access to `bbbb` must be protected similar to Example 16.11, i.e., a call to `MPI_F_SYNC_REG(bbbb)` is needed after the second `MPI_WIN_FENCE` to guarantee that further accesses to `bbbb` are not moved ahead of the call to `MPI_WIN_FENCE`. In Process 2, both calls to `MPI_WIN_FENCE` together act as a communication call with `MPI_BOTTOM` as the buffer. That is, before the first fence and after the second fence, a call to `MPI_F_SYNC_REG(buff)` is needed to guarantee that accesses to `buff` are not moved after or ahead of the calls to `MPI_WIN_FENCE`. Using `MPI_GET` instead of `MPI_PUT`, the same calls to `MPI_F_SYNC_REG` are necessary.

Source of Process 1

```
bbbb = 777

call MPI_WIN_FENCE
call MPI_PUT(bbbb
into buff of process 2)
```

```
call MPI_WIN_FENCE
call MPI_F_SYNC_REG(bbbb)
```

Source of Process 2

```
buff = 999
call MPI_F_SYNC_REG(buff)
call MPI_WIN_FENCE

call MPI_WIN_FENCE
call MPI_F_SYNC_REG(buff)
ccc = buff
```

- The temporary memory modification problem, i.e., Example 16.16 on page 595, can **not** be solved with this method.

A user defined routine instead of `MPI_F_SYNC_REG`

Instead of `MPI_F_SYNC_REG`, one can also use a user defined external subroutine, which is separately compiled:

```
subroutine DD(buf)
  integer buf
end
```

Note that if the intent is declared in an explicit interface for the external subroutine, it must be `OUT` or `INOUT`. The subroutine itself may have an empty body, but the compiler does not know this and has to assume that the buffer may be altered. For example, a call to `MPI_RECV` with `MPI_BOTTOM` as buffer might be replaced by

```
call DD(buf)
call MPI_RECV(MPI_BOTTOM,...)
call DD(buf)
```

Such a user-defined routine was introduced in MPI-2.0 and is still included here to document such usage in existing application programs although new applications should prefer

MPI_F_SYNC_REG or one of the other possibilities. In an existing application, calls to such a user-written routine should be substituted by a call to MPI_F_SYNC_REG because the user-written routine may not be implemented according to the rules specified in Section 16.2.7 on page 568.

Module variables and COMMON blocks

An alternative to the already mentioned methods is to put the buffer or variable into a module or a common block and access it through a USE or COMMON statement in each scope where it is referenced, defined or appears as an actual argument in a call to an MPI routine. The compiler will then have to assume that the MPI procedure may alter the buffer or variable, provided that the compiler cannot infer that the MPI procedure does not reference the module or common block.

- This method solves problems of instruction reordering, code movement, and register optimization related to nonblocking and one-sided communication, or related to the usage of MPI_BOTTOM and derived datatype handles.
- Unfortunately, this method does **not** solve problems caused by asynchronous accesses between the start and end of a nonblocking or one-sided communication. Specifically, problems caused by temporary memory modifications are not solved.

The (poorly performing) Fortran VOLATILE attribute

The VOLATILE attribute gives the buffer or variable the properties needed, but it may inhibit optimization of any code containing references or definitions of the buffer or variable.

The Fortran TARGET attribute

The TARGET attribute does not solve the code movement problem because it is not specified for the choice buffer dummy arguments of nonblocking routines. If the compiler detects that the application program specifies the TARGET attribute for an actual buffer argument used in the call to a nonblocking routine, the compiler may ignore this attribute if no pointer reference to this buffer exists.

Rationale. The Fortran standardization body decided to extend the ASYNCHRONOUS attribute within the TR 29113 to protect buffers in nonblocking calls from all kinds of optimization, instead of extending the TARGET attribute. (*End of rationale.*)

16.2.18 Temporary Data Movement and Temporary Memory Modification

The compiler is allowed to temporarily modify data in memory. Normally, this problem may occur only when overlapping communication and computation, as in Example 16.15, Case (b) on page 590. Example 16.16 on page 595 shows a possibility that could be problematic.

In the compiler-generated, possible optimization in Example 16.17, buf(100,100) from Example 16.16 is equivalenced with the 1-dimensional array buf_1dim(10000). The nonblocking receive may asynchronously receive the data in the boundary buf(1,1:100) while the fused loop is temporarily using this part of the buffer.

[ticket238-J.]

Example 16.16 Overlapping Communication and Computation.

```

USE mpi_f08
REAL :: buf(100,100)
CALL MPI_Irecv(buf(1,1:100),...req,...)
DO j=1,100
  DO i=2,100
    buf(i,j)=...
  END DO
END DO
CALL MPI_Wait(req,...)

```

[ticket238-J.]

Example 16.17 The compiler may substitute the nested loops through loop fusion.

```

REAL :: buf(100,100), buf_1dim(10000)
EQUIVALENCE (buf(1,1), buf_1dim(1))
CALL MPI_Irecv(buf(1,1:100),...req,...)
tmp(1:100) = buf(1,1:100)
DO j=1,10000
  buf_1dim(h)=...
END DO
buf(1,1:100) = tmp(1:100)
CALL MPI_Wait(req,...)

```

When the tmp data is written back to buf, the previous data of buf(1,1:100) is restored and the received data is lost. The principle behind this optimization is that the receive buffer data buf(1,1:100) was temporarily moved to tmp.

Example 16.18 shows a second possible optimization. The whole array is temporarily moved to local_buf. When storing local_buf back to the original location buf, then this includes also an overwriting of the receive buffer part buf(1,1:100), i.e., this storing back may overwrite the asynchronously received data.

Note, that this problem may also occur:

- With the local buffer at the origin process, between an RMA communication call and the ensuing synchronization call; see Chapter 11 on page 395.
- With the window buffer at the target process between two ensuing RMA synchronization calls.
- With the local buffer in MPI parallel file I/O split collective operations with between the ..._BEGIN and ..._END calls; see Section 13.4.5 on page 488.

As already mentioned in subsection *The Fortran ASYNCHRONOUS attribute* on page 591 in Section 16.2.17 on page 587, the ASYNCHRONOUS attribute can prevent compiler optimization with temporary data movement, but only if the receive buffer and the numerical read

[ticket238-J.]

Example 16.18 Another optimization is based on the usage of a separate memory storage area, e.g., in a GPU.

```

REAL :: buf(100,100), local_buf(100,100)
CALL MPI_Irecv(buf(1,1:100),...req,...)
local_buf = buf
DO j=1,100
  DO i=2,100
    local_buf(i,j)=....
  END DO
END DO
buf = local_buf ! may overwrite asynchronously received
                ! data in buf(1,1:100)
CALL MPI_Wait(req,...)

```

accesses are separated into different variables, as shown in Example 16.19 on page 598 and in Example 16.20 on page 599.

Note also that the methods

- calling `MPI_F_SYNC_REG` (or such a user-defined routine),
- using module variables and `COMMON` blocks, and
- the `TARGET` attribute

cannot be used to prevent such temporary data movement. These methods influence compiler optimization when library routines are called. They cannot prevent the optimizations of the numerical code shown in Example 16.16 and 16.17.

Note also that compiler optimization with temporary data movement should **not** be prevented by declaring `buf` as `VOLATILE` because the `VOLATILE` implies that all accesses to any storage unit (word) of `buf` must be directly done in the main memory exactly in the sequence defined by the application program. The `VOLATILE` attribute prevents all register and cache optimizations. Therefore, `VOLATILE` may cause a huge performance degradation.

Instead of solving the problem, it is needed to **prevent** the problem. When overlapping communication and computation, the nonblocking communication (or nonblocking or split collective IO) and the computation should be executed **on different sets of variables**. In this case, the temporary memory modifications are done only on the variables used in the computation and cannot have any side effect on the data used in the nonblocking MPI operations.

Rationale. This is a strong restriction for application programs. To weaken this restriction, a new or modified asynchronous feature in the Fortran language would be necessary: an asynchronous attribute that can be used on parts of an array and together with asynchronous operations outside the scope of Fortran. If such a feature is available in a later version of the Fortran standard, then this restriction also may be weakened in a later version of the MPI standard. (*End of rationale.*)

In Example 16.19 on page 598 (which is a solution for the problem shown in Example 16.15 on page 590) and in Example 16.20 on page 599 (which is a solution for the problem shown in Example 16.18 on page 596), the array is split into inner and halo part and both disjunct parts are passed to a subroutine `separated_sections`. This routine overlaps the receiving of the halo data and the calculations on the inner part of the array. In a second step, the whole array is used to do the calculation on the elements where inner+halo is needed. Note that the halo and the inner area are strided arrays. Those can be used in non-blocking communication only with a TR 29113 based MPI library.

16.2.19 Permanent Data Movement

A Fortran compiler may implement permanent data movement during the execution of a Fortran program. This would require that pointers to such data are appropriately updated. Automatic garbage collection implementation is one use case. Such permanent data movement is in conflict with MPI in several areas:

- MPI datatype handles with absolute addresses in combination with `MPI_BOTTOM`.
- Nonblocking MPI operations (communication, one-sided, I/O) if the internally used pointers to the buffers are not updated by the Fortran runtime, or if within an MPI process, the data movement is executed in parallel with the MPI operation.

This problem can be also solved by using the `ASYNCHRONOUS` attribute for such buffers. This MPI standard requires that the problems with permanent data movement do not occur by imposing suitable restrictions on the MPI library together with the compiler used; see Section 16.2.7 on page 568.

16.2.20 Comparison with C

In C, subroutines which modify variables that are not in the argument list will not cause register optimization problems. This is because taking pointers to storage objects by using the `&` operator and later referencing the objects by way of the pointer is an integral part of the language. A C compiler understands the implications, so that the problem should not occur, in general. However, some compilers do offer optional aggressive optimization levels which may not be safe. Problems due to temporary memory modifications can also occur in C. As above, the best advice is to avoid the problem: use different variables for buffers in nonblocking MPI operations and computation that is executed while the nonblocking operations are pending.

[ticket238-J.]

Example 16.19 Using separated variables for overlapping communication and computation to allow the protection of nonblocking communication with the `ASYNCHRONOUS` attribute.

```

USE mpi_f08
REAL :: b(0:101)      ! elements 0 and 101 are halo cells
REAL :: bnew(0:101)   ! elements 1 and 100 are newly computed
INTEGER :: i
CALL separated_sections(b(0), b(1:100), b(101), bnew(0:101))
i=1 ! compute leftmost element
  bnew(i) = function(b(i-1), b(i), b(i+1))
i=100 ! compute rightmost element
  bnew(i) = function(b(i-1), b(i), b(i+1))
END

SUBROUTINE separated_sections(b_lefthalo, b_inner, b_righthalo, bnew)
USE mpi_f08
REAL, ASYNCHRONOUS :: b_lefthalo(0:0), b_inner(1:100), b_righthalo(101:101)
REAL :: bnew(0:101) ! elements 1 and 100 are newly computed
TYPE(MPI_Request) :: req(4)
INTEGER :: left, right, i
CALL MPI_Cart_shift(...,left,right,...)
CALL MPI_Irecv(b_lefthalo ( 0), ..., left, ..., req(1), ...)
CALL MPI_Irecv(b_righthalo(101), ..., right, ..., req(2), ...)
! b_lefthalo and b_righthalo is written asynchronously.
! There is no other concurrent access to b_lefthalo and b_righthalo.
CALL MPI_Isend(b_inner( 1), ..., left, ..., req(3), ...)
CALL MPI_Isend(b_inner(100), ..., right, ..., req(4), ...)

DO i=2,99 ! compute only elements for which halo data is not needed
  bnew(i) = function(b_inner(i-1), b_inner(i), b_inner(i+1))
  ! b_inner is read and send at the same time.
  ! This is allowed based on the rules for ASYNCHRONOUS.
END DO
CALL MPI_Waitall(4,req,...)
END SUBROUTINE

```

[ticket238-J.]

Example 16.20 Protecting GPU optimizations with the ASYNCHRONOUS attribute.

```
USE mpi_f08
REAL :: buf(100,100)
CALL separated_sections(buf(1:1,1:100), buf(2:100,1:100))
END

SUBROUTINE separated_sections(buf_halo, buf_inner)
REAL, ASYNCHRONOUS :: buf_halo(1:1,1:100)
REAL :: buf_inner(2:100,1:100)
REAL :: local_buf(2:100,100)

CALL MPI_Irecv(buf_halo(1,1:100),...req,...)
local_buf = buf_inner
DO j=1,100
  DO i=2,100
    local_buf(i,j)=....
  END DO
END DO
buf_inner = local_buf ! buf_halo is not touched!!!

CALL MPI_Wait(req,...)
```

all executing processes. Use of the Fortran version of `MPI_INIT` to initialize MPI may result in a loss of this ability. (*End of advice to users.*)

The function `MPI_INITIALIZED` returns the same answer in all languages.

The function `MPI_FINALIZE` finalizes the MPI environments for all languages.

The function `MPI_FINALIZED` returns the same answer in all languages.

The function `MPI_ABORT` kills processes, irrespective of the language used by the caller or by the processes killed.

The MPI environment is initialized in the same manner for all languages by `MPI_INIT`. E.g., `MPI_COMM_WORLD` carries the same information regardless of language: same processes, same environmental attributes, same error handlers.

Information can be added to info objects in one language and retrieved in another.

Advice to users. The use of several languages in one MPI program may require the use of special options at compile and/or link time. (*End of advice to users.*)

Advice to implementors. Implementations may selectively link language specific MPI libraries only to codes that need them, so as not to increase the size of binaries for codes that use only one language. The MPI initialization code need perform initialization for a language only if that language library is loaded. (*End of advice to implementors.*)

16.3.4 Transfer of Handles

Handles are passed between Fortran and C or C++ by using an explicit C wrapper to convert Fortran handles to C handles. There is no direct access to C or C++ handles in Fortran. Handles are passed between C and C++ using overloaded C++ operators called from C++ code. There is no direct access to C++ objects from C.

The type definition `MPI_Fint` is provided in C/C++ for an integer of the size that matches a Fortran `INTEGER`; usually, `MPI_Fint` will be equivalent to `int`. **With the Fortran `mpi` module or the `mpif.h` include file, a Fortran handle is a Fortran `INTEGER` value that can be used in the following conversion functions. With the Fortran `mpi_f08` module, a Fortran handle is a `BIND(C)` derived type that contains an `INTEGER` field named `MPI_VAL`. This `INTEGER` value can be used in the following conversion functions.**

The following functions are provided in C to convert from a Fortran communicator handle (which is an integer) to a C communicator handle, and vice versa. See also Section 2.6.5 on page 22.

```
MPI_Comm MPI_Comm_f2c(MPI_Fint comm)
```

If `comm` is a valid Fortran handle to a communicator, then `MPI_Comm_f2c` returns a valid C handle to that same communicator; if `comm = MPI_COMM_NULL` (Fortran value), then `MPI_Comm_f2c` returns a null C handle; if `comm` is an invalid Fortran handle, then `MPI_Comm_f2c` returns an invalid C handle.

```
MPI_Fint MPI_Comm_c2f(MPI_Comm comm)
```

The function `MPI_Comm_c2f` translates a C communicator handle into a Fortran handle to the same communicator; it maps a null handle into a null handle and an invalid handle into an invalid handle.

Similar functions are provided for the other types of opaque objects.

```

1  MPI_Datatype MPI_Type_f2c(MPI_Fint datatype)
2
3  MPI_Fint MPI_Type_c2f(MPI_Datatype datatype)
4
5  MPI_Group MPI_Group_f2c(MPI_Fint group)
6  MPI_Fint MPI_Group_c2f(MPI_Group group)
7
8  MPI_Request MPI_Request_f2c(MPI_Fint request)
9  MPI_Fint MPI_Request_c2f(MPI_Request request)
10
11 MPI_File MPI_File_f2c(MPI_Fint file)
12 MPI_Fint MPI_File_c2f(MPI_File file)
13
14 MPI_Win MPI_Win_f2c(MPI_Fint win)
15 MPI_Fint MPI_Win_c2f(MPI_Win win)
16
17 MPI_Op MPI_Op_f2c(MPI_Fint op)
18 MPI_Fint MPI_Op_c2f(MPI_Op op)
19
20 MPI_Info MPI_Info_f2c(MPI_Fint info)
21 MPI_Fint MPI_Info_c2f(MPI_Info info)
22
23 MPI_Errhandler MPI_Errhandler_f2c(MPI_Fint errhandler)
24 MPI_Fint MPI_Errhandler_c2f(MPI_Errhandler errhandler)
25

```

Example 16.21 The example below illustrates how the Fortran MPI function `MPI_TYPE_COMMIT` can be implemented by wrapping the C MPI function `MPI_Type_commit` with a C wrapper to do handle conversions. In this example a Fortran-C interface is assumed where a Fortran function is all upper case when referred to from C and arguments are passed by addresses.

```

32  ! FORTRAN PROCEDURE
33  SUBROUTINE MPI_TYPE_COMMIT( DATATYPE, IERR)
34  INTEGER [ticket250-V.]:: DATATYPE, IERR
35  CALL MPI_X_TYPE_COMMIT(DATATYPE, IERR)
36  RETURN
37  END
38
39  /* C wrapper */
40
41  void MPI_X_TYPE_COMMIT( MPI_Fint *f_handle, MPI_Fint *ierr)
42  {
43      MPI_Datatype datatype;
44
45      datatype = MPI_Type_f2c( *f_handle);
46      *ierr = (MPI_Fint)MPI_Type_commit( &datatype);
47      *f_handle = MPI_Type_c2f(datatype);
48      return;

```

The following function allows conversion from C++ objects to C MPI handles. In this case, the casting operator is overloaded to provide the functionality.

```
MPI::<CLASS>::operator MPI_<CLASS>() const
```

Example 16.23 A C library routine is called from a C++ program. The C library routine is prototyped to take an MPI_Comm as an argument.

```
// C function prototype
extern "C" {
    void c_lib_call(MPI_Comm c_comm);
}

void cpp_function()
{
    // Create a C++ communicator, and initialize it with a dup of
    // MPI::COMM_WORLD
    MPI::Intracomm cpp_comm(MPI::COMM_WORLD.Dup());
    c_lib_call(cpp_comm);
}
```

Rationale. Providing conversion from C to C++ via constructors and from C++ to C via casting allows the compiler to make automatic conversions. Calling C from C++ becomes trivial, as does the provision of a C or Fortran interface to a C++ library. (*End of rationale.*)

Advice to users. Note that the casting and promotion operators return new handles by value. Using these new handles as INOUT parameters will affect the internal MPI object, but will *not* affect the original handle from which it was cast. (*End of advice to users.*)

It is important to note that all C++ objects with corresponding C handles can be used interchangeably by an application. For example, an application can cache an attribute on MPI_COMM_WORLD and later retrieve it from MPI::COMM_WORLD.

16.3.5 Status

The following two procedures are provided in C to convert from a Fortran (**with the mpi module or mpif.h**) status (which is an array of integers) to a C status (which is a structure), and vice versa. The conversion occurs on all the information in status, including that which is hidden. That is, no status information is lost in the conversion.

```
int MPI_Status_f2c(MPI_Fint *f_status, MPI_Status *c_status)
```

If f_status is a valid Fortran status, but not the Fortran value of MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE, then MPI_Status_f2c returns in c_status a valid C status with the same content. If f_status is the Fortran value of MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE, or if f_status is not a valid Fortran status, then the call is erroneous.

The C status has the same source, tag and error code values as the Fortran status, and returns the same answers when queried for count, elements, and cancellation. The

conversion function may be called with a Fortran status argument that has an undefined error field, in which case the value of the error field in the C status argument is undefined.

Two global variables of type `MPI_Fint*`, `MPI_F_STATUS_IGNORE` and `MPI_F_STATUSES_IGNORE` are declared in `mpi.h`. They can be used to test, in C, whether `f_status` is the Fortran value of `MPI_STATUS_IGNORE` or `MPI_STATUSES_IGNORE` defined in the `mpi` module or `mpif.h`. These are global variables, not C constant expressions and cannot be used in places where C requires constant expressions. Their value is defined only between the calls to `MPI_INIT` and `MPI_FINALIZE` and should not be changed by user code.

To do the conversion in the other direction, we have the following:

```
int MPI_Status_c2f(MPI_Status *c_status, MPI_Fint *f_status)
```

This call converts a C status into a Fortran status, and has a behavior similar to `MPI_Status_f2c`. That is, the value of `c_status` must not be either `MPI_STATUS_IGNORE` or `MPI_STATUSES_IGNORE`.

Advice to users. There exists no separate conversion function for arrays of statuses, since one can simply loop through the array, converting each status with the routines in Fig. 16.1 on page 606. (*End of advice to users.*)

Rationale. The handling of `MPI_STATUS_IGNORE` is required in order to layer libraries with only a C wrapper: if the Fortran call has passed `MPI_STATUS_IGNORE`, then the C wrapper must handle this correctly. Note that this constant need not have the same value in Fortran and C. If `MPI_Status_f2c` were to handle `MPI_STATUS_IGNORE`, then the type of its result would have to be `MPI_Status**`, which was considered an inferior solution. (*End of rationale.*)

Using the `mpi_f08` Fortran module, a status is declared as `TYPE(MPI_Status)`. The C type `MPI_F08_status` can be used to pass a Fortran `TYPE(MPI_Status)` argument into a C routine. Figure 16.1 illustrates all status conversion routines. Some are only available in C, some in both C and Fortran.

```
int MPI_Status_f082c(MPI_F08_status *f08_status, MPI_Status *c_status)
```

This C routine converts a Fortran `mpi_f08 TYPE(MPI_Status)` into a C `MPI_Status`.

```
int MPI_Status_c2f08(MPI_Status *c_status, MPI_F08_status *f08_status)
```

This C routine converts a C `MPI_Status` into a Fortran `mpi_f08 TYPE(MPI_Status)`. Two global variables of type `MPI_F08_status*`, `MPI_F08_STATUS_IGNORE` and `MPI_F08_STATUSES_IGNORE` are declared in `mpi.h`. They can be used to test, in C, whether `f_status` is the Fortran value of `MPI_STATUS_IGNORE` or `MPI_STATUSES_IGNORE` defined in the `mpi_f08` module. These are global variables, not C constant expressions and cannot be used in places where C requires constant expressions. Their value is defined only between the calls to `MPI_INIT` and `MPI_FINALIZE` and should not be changed by user code.

Conversion between the two Fortran versions of a status can be done with:

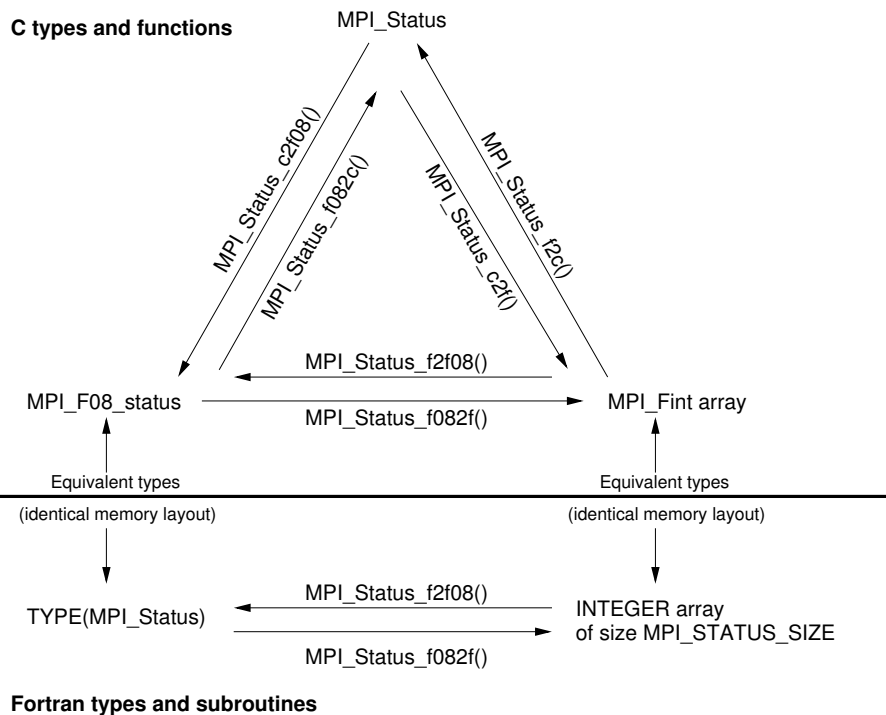


Figure 16.1: Status conversion routines

MPI_STATUS_F2F08(f_status, f08_status)

IN **f_status** status object declared as array
OUT **f08_status** status object declared as named type

```
int MPI_Status_f2f08(MPI_Fint *f_status, MPI_F08_status *f08_status)
```

```
MPI_Status_f2f08(f_status, f08_status, ierror) BIND(C)
```

```
  INTEGER, INTENT(IN) :: f_status(MPI_STATUS_SIZE)
```

```
  TYPE(MPI_Status), INTENT(OUT) :: f08_status
```

```
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_STATUS_F2F08(F_STATUS, F08_STATUS, IERROR)
```

```
  INTEGER :: F_STATUS(MPI_STATUS_SIZE)
```

```
  TYPE(MPI_Status) :: F08_STATUS
```

```
  INTEGER IERROR
```

This routine converts a Fortran `INTEGER, DIMENSION(MPI_STATUS_SIZE)` status array into a Fortran `mpi_f08 TYPE(MPI_Status)`.

```
MPI_STATUS_F082F(f08_status, f_status)
```

```
IN      f08_status      status object declared as named type
OUT     f_status        status object declared as array
```

```
int MPI_Status_f082f(MPI_F08_status *f08_status, MPI_Fint *f_status)
```

```
MPI_Status_f082f(f08_status, f_status, ierror) BIND(C)
  TYPE(MPI_Status), INTENT(IN) :: f08_status
  INTEGER, INTENT(OUT) :: f_status(MPI_STATUS_SIZE)
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_STATUS_F082F(F08_STATUS, F_STATUS, IERROR)
  TYPE(MPI_Status) :: F08_STATUS
  INTEGER :: F_STATUS(MPI_STATUS_SIZE)
  INTEGER IERROR
```

This routine converts a Fortran `mpi_f08` `TYPE(MPI_Status)` into a Fortran `INTEGER`, `DIMENSION(MPI_STATUS_SIZE)` status array.

16.3.6 MPI Opaque Objects

Unless said otherwise, opaque objects are “the same” in all languages: they carry the same information, and have the same meaning in both languages. The mechanism described in the previous section can be used to pass references to MPI objects from language to language. An object created in one language can be accessed, modified or freed in another language.

We examine below in more detail, issues that arise for each type of MPI object.

Datatypes

Datatypes encode the same information in all languages. E.g., a datatype accessor like `MPI_TYPE_GET_EXTENT` will return the same information in all languages. If a datatype defined in one language is used for a communication call in another language, then the message sent will be identical to the message that would be sent from the first language: the same communication buffer is accessed, and the same representation conversion is performed, if needed. All predefined datatypes can be used in datatype constructors in any language. If a datatype is committed, it can be used for communication in any language.

The function `MPI_GET_ADDRESS` returns the same value in all languages. Note that we do not require that the constant `MPI_BOTTOM` have the same value in all languages (see [16.3.9](#), page 614).

Example 16.24

```
! FORTRAN CODE
REAL [ticket250-V.]:: R(5)
INTEGER [ticket250-V.]:: TYPE, IERR, AOBLN(1), AOTYPE(1)
INTEGER (KIND=MPI_ADDRESS_KIND) [ticket250-V.]:: AODISP(1)

! create an absolute datatype for array R
```

i.e., the regular address - c, as the MPI address returned by `MPI_GET_ADDRESS` and stored in absolute datatypes. (*End of advice to implementors.*)

Callback Functions

MPI calls may associate callback functions with MPI objects: error handlers are associated with communicators and files, attribute copy and delete functions are associated with attribute keys, reduce operations are associated with operation objects, etc. In a multilanguage environment, a function passed in an MPI call in one language may be invoked by an MPI call in another language. MPI implementations must make sure that such invocation will use the calling convention of the language the function is bound to.

Advice to implementors. Callback functions need to have a language tag. This tag is set when the callback function is passed in by the library function (which is presumably different for each language **and language support method**), and is used to generate the right calling sequence when the callback function is invoked. (*End of advice to implementors.*)

Advice to users. If a subroutine written in one language or Fortran support method wants to pass a callback routine including the predefined Fortran functions (e.g., `MPI_COMM_NULL_COPY_FN`) to another application routine written in another language or Fortran support method, then it must be guaranteed that both routines use the callback interface definition that is defined for the argument when passing the callback to an MPI routine (e.g., `MPI_COMM_CREATE_KEYVAL`); see also the advice to users on page 276. (*End of advice to users.*)

Error Handlers

Advice to implementors. Error handlers, have, in C and C++, a “`stdargs`” argument list. It might be useful to provide to the handler information on the language environment where the error occurred. (*End of advice to implementors.*)

Reduce Operations

Advice to users. Reduce operations receive as one of their arguments the datatype of the operands. Thus, one can define “polymorphic” reduce operations that work for C, C++, and Fortran datatypes. (*End of advice to users.*)

Addresses

Some of the datatype accessors and constructors have arguments of type `MPI_Aint` (in C) or `MPI::Aint` in C++, to hold addresses. The corresponding arguments, in Fortran, have type `INTEGER`. This causes Fortran and C/C++ to be incompatible, in an environment where addresses have 64 bits, but Fortran `INTEGER`s have 32 bits.

This is a problem, irrespective of interlanguage issues. Suppose that a Fortran process has an address space of ≥ 4 GB. What should be the value returned in Fortran by `MPI_ADDRESS`, for a variable with an address above 2^{32} ? The design described here addresses this issue, while maintaining compatibility with current Fortran codes.

Advice to users. This definition means that it is safe in C/C++ to allocate a buffer to receive a string using a declaration like

```
char name [MPI_MAX_OBJECT_NAME];
```

(End of advice to users.)

Also constant “addresses,” i.e., special values for reference arguments that are not handles, such as MPI_BOTTOM or MPI_STATUS_IGNORE may have different values in different languages.

Rationale. The current MPI standard specifies that MPI_BOTTOM can be used in initialization expressions in C, but not in Fortran. Since Fortran does not normally support call by value, then MPI_BOTTOM must be in Fortran the name of a predefined static variable, e.g., a variable in an MPI declared COMMON block. On the other hand, in C, it is natural to take MPI_BOTTOM = 0 (Caveat: Defining MPI_BOTTOM = 0 implies that NULL pointer cannot be distinguished from MPI_BOTTOM; it may be that MPI_BOTTOM = 1 is better ...) Requiring that the Fortran and C values be the same will complicate the initialization process. *(End of rationale.)*

16.3.10 Interlanguage Communication

The type matching rules for communication in MPI are not changed: the datatype specification for each item sent should match, in type signature, the datatype specification used to receive this item (unless one of the types is MPI_PACKED). Also, the type of a message item should match the type declaration for the corresponding communication buffer location, unless the type is MPI_BYTE or MPI_PACKED. Interlanguage communication is allowed if it complies with these rules.

Example 16.28 In the example below, a Fortran array is sent from Fortran and received in C.

```
! FORTRAN CODE
USE mpi_f08
REAL [ticket250-V.]:: R(5)
INTEGER [ticket250-V.]:: IERR, MYRANK, AOBLN(1), AOTYPE(1)
[ticket250-V.]TYPE(MPI_Type) :: TYPE
INTEGER (KIND=MPI_ADDRESS_KIND) [ticket250-V.]:: AODISP(1)

! create an absolute datatype for array R
AOBLN(1) = 5
CALL MPI_GET_ADDRESS( R, AODISP(1), IERR)
AOTYPE(1) = MPI_REAL
CALL MPI_TYPE_CREATE_STRUCT(1, AOBLN,AODISP,AOTYPE, TYPE, IERR)
CALL MPI_TYPE_COMMIT(TYPE, IERR)

CALL MPI_COMM_RANK( MPI_COMM_WORLD, MYRANK, IERR)
IF (MYRANK.EQ.0) THEN
    CALL MPI_SEND( MPI_BOTTOM, 1, TYPE, 1, 0, MPI_COMM_WORLD, IERR)
```

ticket250-V.

```
1  ELSE
2      CALL C_ROUTINE(TYPE[ticket250-V.]%MPI_VAL)
3  END IF
4
5
6  /* C code */
7
8  void C_ROUTINE(MPI_Fint *fhandle)
9  {
10     MPI_Datatype type;
11     MPI_Status status;
12
13     type = MPI_Type_f2c(*fhandle);
14
15     MPI_Recv( MPI_BOTTOM, 1, type, 0, 0, MPI_COMM_WORLD, &status);
16 }
17
```

MPI implementors may weaken these type matching rules, and allow messages to be sent with Fortran types and received with C types, and vice versa, when those types match. I.e., if the Fortran type `INTEGER` is identical to the C type `int`, then an MPI implementation may allow data to be sent with datatype `MPI_INTEGER` and be received with datatype `MPI_INT`. However, such code is not portable.

Assorted Constants

C type: <code>const int</code> (or unnamed <code>enum</code>)	C++ type:
Fortran type: <code>INTEGER</code>	<code>const int</code> (or unnamed <code>enum</code>)
<code>MPI_PROC_NULL</code>	<code>MPI::PROC_NULL</code>
<code>MPI_ANY_SOURCE</code>	<code>MPI::ANY_SOURCE</code>
<code>MPI_ANY_TAG</code>	<code>MPI::ANY_TAG</code>
<code>MPI_UNDEFINED</code>	<code>MPI::UNDEFINED</code>
<code>MPI_BSEND_OVERHEAD</code>	<code>MPI::BSEND_OVERHEAD</code>
<code>MPI_KEYVAL_INVALID</code>	<code>MPI::KEYVAL_INVALID</code>
<code>MPI_LOCK_EXCLUSIVE</code>	<code>MPI::LOCK_EXCLUSIVE</code>
<code>MPI_LOCK_SHARED</code>	<code>MPI::LOCK_SHARED</code>
<code>MPI_ROOT</code>	<code>MPI::ROOT</code>

[ticket247-S.]Fortran Support Method Specific Constants

Fortran type: <code>LOGICAL</code>
[ticket234-F.] <code>MPI_SUBARRAYS_SUPPORTED</code> (Fortran only)
[ticket238-J.] <code>[ticket229.1.]MPI_ASYNC_PROTECTS_NONBLOCKING</code> (Fortran only)

Status size and reserved index values (Fortran only)

Fortran type: <code>INTEGER</code>
<code>MPI_STATUS_SIZE</code> Not defined for C++
<code>MPI_SOURCE</code> Not defined for C++
<code>MPI_TAG</code> Not defined for C++
<code>MPI_ERROR</code> Not defined for C++

Variable Address Size (Fortran only)

Fortran type: <code>INTEGER</code>
<code>MPI_ADDRESS_KIND</code> Not defined for C++
<code>MPI_INTEGER_KIND</code> Not defined for C++
<code>MPI_OFFSET_KIND</code> Not defined for C++

Error-handling specifiers

C type: <code>MPI_Errhandler</code>	C++ type: <code>MPI::Errhandler</code>
Fortran type: <code>INTEGER</code>	
[ticket231-C.] <code>or TYPE(MPI_Errhandler)</code>	
<code>MPI_ERRORS_ARE_FATAL</code>	<code>MPI::ERRORS_ARE_FATAL</code>
<code>MPI_ERRORS_RETURN</code>	<code>MPI::ERRORS_RETURN</code>
	<code>MPI::ERRORS_THROW_EXCEPTIONS</code>

Named Predefined Datatypes		C/C++ types	1
C type: MPI_Datatype	C++ type: MPI::Datatype		2
Fortran type: INTEGER			3
[ticket231-C.]or TYPE(MPI_Datatype)			4
MPI_CHAR	MPI::CHAR	char	5
		(treated as printable	6
		character)	7
MPI_SHORT	MPI::SHORT	signed short int	8
MPI_INT	MPI::INT	signed int	9
MPI_LONG	MPI::LONG	signed long	10
MPI_LONG_LONG_INT	MPI::LONG_LONG_INT	signed long long	11
MPI_LONG_LONG	MPI::LONG_LONG	long long (synonym)	12
MPI_SIGNED_CHAR	MPI::SIGNED_CHAR	signed char	13
		(treated as integral value)	14
MPI_UNSIGNED_CHAR	MPI::UNSIGNED_CHAR	unsigned char	15
		(treated as integral value)	16
MPI_UNSIGNED_SHORT	MPI::UNSIGNED_SHORT	unsigned short	17
MPI_UNSIGNED	MPI::UNSIGNED	unsigned int	18
MPI_UNSIGNED_LONG	MPI::UNSIGNED_LONG	unsigned long	19
MPI_UNSIGNED_LONG_LONG	MPI::UNSIGNED_LONG_LONG	unsigned long long	20
MPI_FLOAT	MPI::FLOAT	float	21
MPI_DOUBLE	MPI::DOUBLE	double	22
MPI_LONG_DOUBLE	MPI::LONG_DOUBLE	long double	23
MPI_WCHAR	MPI::WCHAR	wchar_t	24
		(defined in <stddef.h>)	25
		(treated as printable	26
		character)	27
MPI_C_BOOL	(use C datatype handle)	_Bool	28
MPI_INT8_T	(use C datatype handle)	int8_t	29
MPI_INT16_T	(use C datatype handle)	int16_t	30
MPI_INT32_T	(use C datatype handle)	int32_t	31
MPI_INT64_T	(use C datatype handle)	int64_t	32
MPI_UINT8_T	(use C datatype handle)	uint8_t	33
MPI_UINT16_T	(use C datatype handle)	uint16_t	34
MPI_UINT32_T	(use C datatype handle)	uint32_t	35
MPI_UINT64_T	(use C datatype handle)	uint64_t	36
MPI_AINT	(use C datatype handle)	MPI_Aint	37
MPI_OFFSET	(use C datatype handle)	MPI_Offset	38
MPI_C_COMPLEX	(use C datatype handle)	float _Complex	39
MPI_C_FLOAT_COMPLEX	(use C datatype handle)	float _Complex	40
MPI_C_DOUBLE_COMPLEX	(use C datatype handle)	double _Complex	41
MPI_C_LONG_DOUBLE_COMPLEX	(use C datatype handle)	long double _Complex	42
MPI_BYTE	MPI::BYTE	(any C/C++ type)	43
MPI_PACKED	MPI::PACKED	(any C/C++ type)	44

Named Predefined Datatypes		Fortran types
C type: MPI_Datatype	C++ type: MPI::Datatype	
Fortran type: INTEGER		
[ticket231-C.]or TYPE(MPI_Datatype)		
MPI_INTEGER	MPI::INTEGER	INTEGER
MPI_REAL	MPI::REAL	REAL
MPI_DOUBLE_PRECISION	MPI::DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	MPI::F_COMPLEX	COMPLEX
MPI_LOGICAL	MPI::LOGICAL	LOGICAL
MPI_CHARACTER	MPI::CHARACTER	CHARACTER(1)
MPI_AINT	(use C datatype handle)	INTEGER (KIND=MPI_ADDRESS_KIND)
MPI_OFFSET	(use C datatype handle)	INTEGER (KIND=MPI_OFFSET_KIND)
MPI_BYTE	MPI::BYTE	(any Fortran type)
MPI_PACKED	MPI::PACKED	(any Fortran type)

C++-Only Named Predefined Datatypes	C++ types
C++ type: MPI::Datatype	
MPI::BOOL	bool
MPI::COMPLEX	Complex<float>
MPI::DOUBLE_COMPLEX	Complex<double>
MPI::LONG_DOUBLE_COMPLEX	Complex<long double>

Optional datatypes (Fortran)		Fortran types
C type: MPI_Datatype	C++ type: MPI::Datatype	
Fortran type: INTEGER		
[ticket231-C.]or TYPE(MPI_Datatype)		
MPI_DOUBLE_COMPLEX	MPI::F_DOUBLE_COMPLEX	DOUBLE COMPLEX
MPI_INTEGER1	MPI::INTEGER1	INTEGER*1
MPI_INTEGER2	MPI::INTEGER2	INTEGER*[ticket231-C.] ²
MPI_INTEGER4	MPI::INTEGER4	INTEGER*4
MPI_INTEGER8	MPI::INTEGER8	INTEGER*8
MPI_INTEGER16		INTEGER*16
MPI_REAL2	MPI::REAL2	REAL*2
MPI_REAL4	MPI::REAL4	REAL*4
MPI_REAL8	MPI::REAL8	REAL*8
MPI_REAL16		REAL*16
MPI_COMPLEX4		COMPLEX*4
MPI_COMPLEX8		COMPLEX*8
MPI_COMPLEX16		COMPLEX*16
MPI_COMPLEX32		COMPLEX*32

Datatypes for reduction functions (C and C++)

C type: MPI_Datatype	C++ type: MPI::Datatype
Fortran type: INTEGER	
[ticket231-C.]or TYPE(MPI_Datatype)	
MPI_FLOAT_INT	MPI::FLOAT_INT
MPI_DOUBLE_INT	MPI::DOUBLE_INT
MPI_LONG_INT	MPI::LONG_INT
MPI_2INT	MPI::TWOINT
MPI_SHORT_INT	MPI::SHORT_INT
MPI_LONG_DOUBLE_INT	MPI::LONG_DOUBLE_INT

Datatypes for reduction functions (Fortran)

C type: MPI_Datatype	C++ type: MPI::Datatype
Fortran type: INTEGER	
[ticket231-C.]or TYPE(MPI_Datatype)	
MPI_2REAL	MPI::TWOREAL
MPI_2DOUBLE_PRECISION	MPI::TWODOUBLE_PRECISION
MPI_2INTEGER	MPI::TWOINTEGER

Special datatypes for constructing derived datatypes

C type: MPI_Datatype	C++ type: MPI::Datatype
Fortran type: INTEGER	
[ticket231-C.]or TYPE(MPI_Datatype)	
MPI_UB	MPI::UB
MPI_LB	MPI::LB

Reserved communicators

C type: MPI_Comm	C++ type: MPI::Intracomm
Fortran type: INTEGER	
[ticket231-C.]or TYPE(MPI_Comm)	
MPI_COMM_WORLD	MPI::COMM_WORLD
MPI_COMM_SELF	MPI::COMM_SELF

Results of communicator and group comparisons

C type: const int (or unnamed enum)	C++ type: const int
Fortran type: INTEGER	(or unnamed enum)
MPI_IDENT	MPI::IDENT
MPI_CONGRUENT	MPI::CONGRUENT
MPI_SIMILAR	MPI::SIMILAR
MPI_UNEQUAL	MPI::UNEQUAL

Environmental inquiry keys

C type: <code>const int</code> (or unnamed <code>enum</code>)	C++ type: <code>const int</code> (or unnamed <code>enum</code>)
Fortran type: <code>INTEGER</code>	
<code>MPI_TAG_UB</code>	<code>MPI::TAG_UB</code>
<code>MPI_IO</code>	<code>MPI::IO</code>
<code>MPI_HOST</code>	<code>MPI::HOST</code>
<code>MPI_WTIME_IS_GLOBAL</code>	<code>MPI::WTIME_IS_GLOBAL</code>

Collective Operations

C type: <code>MPI_Op</code>	C++ type: <code>const MPI::Op</code>
Fortran type: <code>INTEGER</code> [ticket231-C.] or <code>TYPE(MPI_Op)</code>	
<code>MPI_MAX</code>	<code>MPI::MAX</code>
<code>MPI_MIN</code>	<code>MPI::MIN</code>
<code>MPI_SUM</code>	<code>MPI::SUM</code>
<code>MPI_PROD</code>	<code>MPI::PROD</code>
<code>MPI_MAXLOC</code>	<code>MPI::MAXLOC</code>
<code>MPI_MINLOC</code>	<code>MPI::MINLOC</code>
<code>MPI_BAND</code>	<code>MPI::BAND</code>
<code>MPI_BOR</code>	<code>MPI::BOR</code>
<code>MPI_BXOR</code>	<code>MPI::BXOR</code>
<code>MPI_LAND</code>	<code>MPI::LAND</code>
<code>MPI_LOR</code>	<code>MPI::LOR</code>
<code>MPI_LXOR</code>	<code>MPI::LXOR</code>
<code>MPI_REPLACE</code>	<code>MPI::REPLACE</code>

Null Handles	
C/Fortran name	C++ name
C type / Fortran type	C++ type
MPI_GROUP_NULL	MPI::GROUP_NULL
MPI_Group / INTEGER	const MPI::Group
[ticket231-C.] or TYPE(MPI_Group)	
MPI_COMM_NULL	MPI::COMM_NULL
MPI_Comm / INTEGER	¹⁾
[ticket231-C.] or TYPE(MPI_Comm)	
MPI_DATATYPE_NULL	MPI::DATATYPE_NULL
MPI_Datatype / INTEGER	const MPI::Datatype
[ticket231-C.] or TYPE(MPI_Datatype)	
MPI_REQUEST_NULL	MPI::REQUEST_NULL
MPI_Request / INTEGER	const MPI::Request
[ticket231-C.] or TYPE(MPI_Request)	
MPI_OP_NULL	MPI::OP_NULL
MPI_Op / INTEGER	const MPI::Op
[ticket231-C.] or TYPE(MPI_Op)	
MPI_ERRHANDLER_NULL	MPI::ERRHANDLER_NULL
MPI_Errhandler / INTEGER	const MPI::Errhandler
[ticket231-C.] or TYPE(MPI_Errhandler)	
MPI_FILE_NULL	MPI::FILE_NULL
MPI_File / INTEGER	
[ticket231-C.] or TYPE(MPI_File)	
MPI_INFO_NULL	MPI::INFO_NULL
MPI_Info / INTEGER	const MPI::Info
[ticket231-C.] or TYPE(MPI_Info)	
MPI_WIN_NULL	MPI::WIN_NULL
MPI_Win / INTEGER	
[ticket231-C.] or TYPE(MPI_Win)	
¹⁾ C++ type: See Section 16.1.7 on page 546 regarding class hierarchy and the specific type of MPI::COMM_NULL	

Empty group	
C type: MPI_Group	C++ type: const MPI::Group
Fortran type: INTEGER	
[ticket231-C.] or TYPE(MPI_Group)	
MPI_GROUP_EMPTY	MPI::GROUP_EMPTY

Topologies	
C type: const int (or unnamed enum)	C++ type: const int
Fortran type: INTEGER	(or unnamed enum)
MPI_GRAPH	MPI::GRAPH
MPI_CART	MPI::CART
MPI_DIST_GRAPH	MPI::DIST_GRAPH

Predefined functions	
C/Fortran name	C++ name
C type / Fortran type [ticket230-B.]with mpi module [ticket230-B.]/ Fortran type with mpi_f08 module	C++ type
MPI_COMM_NULL_COPY_FN	MPI_COMM_NULL_COPY_FN
MPI_Comm_copy_attr_function	same as in C ¹⁾
/ COMM_COPY_ATTR_[ticket250-V.] FUNCTION	
/ [ticket230-B.] PROCEDURE(MPI_Comm_copy_attr_function) ²⁾	
MPI_COMM_DUP_FN	MPI_COMM_DUP_FN
MPI_Comm_copy_attr_function	same as in C ¹⁾
/ COMM_COPY_ATTR_[ticket250-V.] FUNCTION	
/ [ticket230-B.] PROCEDURE(MPI_Comm_copy_attr_function) ²⁾	
MPI_COMM_NULL_DELETE_FN	MPI_COMM_NULL_DELETE_FN
MPI_Comm_delete_attr_function	same as in C ¹⁾
/ COMM_DELETE_ATTR_[ticket250-V.] FUNCTION	
/ [ticket230-B.] PROCEDURE(MPI_Comm_delete_attr_function) ²⁾	
MPI_WIN_NULL_COPY_FN	MPI_WIN_NULL_COPY_FN
MPI_Win_copy_attr_function	same as in C ¹⁾
/ WIN_COPY_ATTR_[ticket250-V.] FUNCTION	
/ [ticket230-B.] PROCEDURE(MPI_Win_copy_attr_function) ²⁾	
MPI_WIN_DUP_FN	MPI_WIN_DUP_FN
MPI_Win_copy_attr_function	same as in C ¹⁾
/ WIN_COPY_ATTR_[ticket250-V.] FUNCTION	
/ [ticket230-B.] PROCEDURE(MPI_Win_copy_attr_function) ²⁾	
MPI_WIN_NULL_DELETE_FN	MPI_WIN_NULL_DELETE_FN
MPI_Win_delete_attr_function	same as in C ¹⁾
/ WIN_DELETE_ATTR_[ticket250-V.] FUNCTION	
/ [ticket230-B.] PROCEDURE(MPI_Win_delete_attr_function) ²⁾	
MPI_TYPE_NULL_COPY_FN	MPI_TYPE_NULL_COPY_FN
MPI_Type_copy_attr_function	same as in C ¹⁾
/ TYPE_COPY_ATTR_[ticket250-V.] FUNCTION	
/ [ticket230-B.] PROCEDURE(MPI_Type_copy_attr_function) ²⁾	
MPI_TYPE_DUP_FN	MPI_TYPE_DUP_FN
MPI_Type_copy_attr_function	same as in C ¹⁾
/ TYPE_COPY_ATTR_[ticket250-V.] FUNCTION	
/ [ticket230-B.] PROCEDURE(MPI_Type_copy_attr_function) ²⁾	
MPI_TYPE_NULL_DELETE_FN	MPI_TYPE_NULL_DELETE_FN
MPI_Type_delete_attr_function	same as in C ¹⁾
/ TYPE_DELETE_ATTR_[ticket250-V.] FUNCTION	
/ [ticket230-B.] PROCEDURE(MPI_Type_delete_attr_function) ²⁾	
¹ See the advice to implementors [ticket230-B.](on page 275) and advice to users (on page 275) on [ticket230-B.]the predefined C functions MPI_COMM_NULL_COPY_FN, ... in Section 6.7.2 on page 272	
[ticket230-B.] ² See the advice to implementors (on page 275) and advice to users (on page 276) [ticket230-B.] on the predefined Fortran functions MPI_COMM_NULL_COPY_FN, ... in [ticket230-B.] Section 6.7.2 on page 272	

File Operation Constants, Part 2

C type: <code>const int</code> (or unnamed <code>enum</code>)	C++ type:
Fortran type: <code>INTEGER</code>	<code>const int</code> (or unnamed <code>enum</code>)
<code>MPI_DISTRIBUTE_BLOCK</code>	<code>MPI::DISTRIBUTE_BLOCK</code>
<code>MPI_DISTRIBUTE_CYCLIC</code>	<code>MPI::DISTRIBUTE_CYCLIC</code>
<code>MPI_DISTRIBUTE_DFLT_DARG</code>	<code>MPI::DISTRIBUTE_DFLT_DARG</code>
<code>MPI_DISTRIBUTE_NONE</code>	<code>MPI::DISTRIBUTE_NONE</code>
<code>MPI_ORDER_C</code>	<code>MPI::ORDER_C</code>
<code>MPI_ORDER_FORTRAN</code>	<code>MPI::ORDER_FORTRAN</code>
<code>MPI_SEEK_CUR</code>	<code>MPI::SEEK_CUR</code>
<code>MPI_SEEK_END</code>	<code>MPI::SEEK_END</code>
<code>MPI_SEEK_SET</code>	<code>MPI::SEEK_SET</code>

F90 Datatype Matching Constants

C type: <code>const int</code> (or unnamed <code>enum</code>)	C++ type:
Fortran type: <code>INTEGER</code>	<code>const int</code> (or unnamed <code>enum</code>)
<code>MPI_TYPECLASS_COMPLEX</code>	<code>MPI::TYPECLASS_COMPLEX</code>
<code>MPI_TYPECLASS_INTEGER</code>	<code>MPI::TYPECLASS_INTEGER</code>
<code>MPI_TYPECLASS_REAL</code>	<code>MPI::TYPECLASS_REAL</code>

Constants Specifying Empty or Ignored Input

C/Fortran name	C++ name
C type / Fortran type	C++ type
<code>MPI_ARGVS_NULL</code> <code>char***</code> / 2-dim. array of <code>CHARACTER*(*)</code>	<code>MPI::ARGVS_NULL</code> <code>const char ***</code>
<code>MPI_ARGV_NULL</code> <code>char**</code> / array of <code>CHARACTER*(*)</code>	<code>MPI::ARGV_NULL</code> <code>const char **</code>
<code>MPI_ERRCODES_IGNORE</code> <code>int*</code> / <code>INTEGER</code> array	Not defined for C++
<code>MPI_STATUSES_IGNORE</code> <code>MPI_Status*</code> / <code>INTEGER</code> , <code>DIMENSION(MPI_STATUS_SIZE,*)</code> [ticket231-C.] <code>or TYPE(MPI_Status), DIMENSION(*)</code>	Not defined for C++
<code>MPI_STATUS_IGNORE</code> <code>MPI_Status*</code> / <code>INTEGER</code> , <code>DIMENSION(MPI_STATUS_SIZE)</code> [ticket231-C.] <code>or TYPE(MPI_Status)</code>	Not defined for C++
<code>MPI_UNWEIGHTED</code> <code>int*</code> / <code>INTEGER</code> array	Not defined for C++

C Constants Specifying Ignored Input (no C++ or Fortran)

C type: <code>MPI_Fint*</code>	[ticket243-O.] equivalent to Fortran
<code>MPI_F_STATUSES_IGNORE</code>	[ticket243-O.] <code>MPI_STATUSES_IGNORE</code> in <code>mpi / mpif.h</code>
<code>MPI_F_STATUS_IGNORE</code>	[ticket243-O.] <code>MPI_STATUS_IGNORE</code> in <code>mpi / mpif.h</code>
[ticket243-O.] C type: <code>MPI_F08_status*</code>	[ticket243-O.] equivalent to Fortran
[ticket243-O.] <code>MPI_F08_STATUSES_IGNORE</code>	[ticket243-O.] <code>MPI_STATUSES_IGNORE</code> in <code>mpi_f08</code>
[ticket243-O.] <code>MPI_F08_STATUS_IGNORE</code>	[ticket243-O.] <code>MPI_STATUS_IGNORE</code> in <code>mpi_f08</code>

C and C++ preprocessor Constants and Fortran Parameters

C/C++ type: `const int` (or unnamed `enum`)

Fortran type: `INTEGER`

`MPI_SUBVERSION`

`MPI_VERSION`

A.1.2 Types

The following are defined C type definitions, included in the file `mpi.h`.

`/* C opaque types */`

`MPI_Aint`

`MPI_Fint`

`MPI_Offset`

`MPI_Status`

`MPI_F08_status`

`/* C handles to assorted structures */`

`MPI_Comm`

`MPI_Datatype`

`MPI_Errhandler`

`MPI_File`

`MPI_Group`

`MPI_Info`

`MPI_Op`

`MPI_Request`

`MPI_Win`

`// C++ opaque types (all within the MPI namespace)`

`MPI::Aint`

`MPI::Offset`

`MPI::Status`

`// C++ handles to assorted structures (classes,`

`// all within the MPI namespace)`

`MPI::Comm`

`MPI::Intracomm`

`MPI::Graphcomm`

`MPI::Distgraphcomm`

`MPI::Cartcomm`

`MPI::Intercomm`

`MPI::Datatype`

`MPI::Errhandler`

`MPI::Exception`

`MPI::File`

`MPI::Group`

`MPI::Info`

`MPI::Op`

`MPI::Request`

MPI::Prequest
 MPI::Grequest
 MPI::Win

The following are defined Fortran type definitions, included in the `mpi_f08` and `mpi` module.

```
! Fortran opaque types in the mpi_f08 and mpi module
TYPE(MPI_Status)
```

```
! Fortran handles in the mpi_f08 and mpi module
TYPE(MPI_Comm)
TYPE(MPI_Datatype)
TYPE(MPI_Errhandler)
TYPE(MPI_File)
TYPE(MPI_Group)
TYPE(MPI_Info)
TYPE(MPI_Op)
TYPE(MPI_Request)
TYPE(MPI_Win)
```

A.1.3 Prototype Definitions

C Bindings

The following are defined C typedefs for user-defined functions, also included in the file `mpi.h`.

```
/* prototypes for user-defined functions */
typedef void MPI_User_function(void *invec, void *inoutvec, int *len,
                               MPI_Datatype *datatype);

typedef int MPI_Comm_copy_attr_function(MPI_Comm oldcomm,
                                         int comm_keyval, void *extra_state, void *attribute_val_in,
                                         void *attribute_val_out, int*flag);
typedef int MPI_Comm_delete_attr_function(MPI_Comm comm,
                                         int comm_keyval, void *attribute_val, void *extra_state);

typedef int MPI_Win_copy_attr_function(MPI_Win oldwin, int win_keyval,
                                         void *extra_state, void *attribute_val_in,
                                         void *attribute_val_out, int *flag);
typedef int MPI_Win_delete_attr_function(MPI_Win win, int win_keyval,
                                         void *attribute_val, void *extra_state);

typedef int MPI_Type_copy_attr_function(MPI_Datatype oldtype,
                                         int type_keyval, void *extra_state,
                                         void *attribute_val_in, void *attribute_val_out, int *flag);
typedef int MPI_Type_delete_attr_function(MPI_Datatype [ticket252-W.]datatype,
                                         int type_keyval, void *attribute_val, void *extra_state);
```



```

1
2 typedef void MPI_Comm_errhandler_function(MPI_Comm *, int *, ...);
3 typedef void MPI_Win_errhandler_function(MPI_Win *, int *, ...);
4 typedef void MPI_File_errhandler_function(MPI_File *, int *, ...);
5
6 typedef int MPI_Grequest_query_function(void *extra_state,
7     MPI_Status *status);
8 typedef int MPI_Grequest_free_function(void *extra_state);
9 typedef int MPI_Grequest_cancel_function(void *extra_state, int complete);
10
11 typedef int MPI_Datarep_extent_function(MPI_Datatype datatype,
12     MPI_Aint *file_extent, void *extra_state);
13 typedef int MPI_Datarep_conversion_function(void *userbuf,
14     MPI_Datatype datatype, int count, void *filebuf,
15     MPI_Offset position, void *extra_state);

```

Fortran 2008 Bindings with the mpi_f08 Module

With the Fortran mpi_f08 module, the callback prototypes are:

The user-function argument to MPI_Op_create should be declared according to:

```

ABSTRACT INTERFACE
SUBROUTINE MPI_User_function(invec, inoutvec, len, datatype) BIND(C)
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
    TYPE(C_PTR), VALUE :: invec, inoutvec
    INTEGER :: len
    TYPE(MPI_Datatype) :: datatype

```

The copy and delete function arguments to MPI_Comm_create_keyval should be declared according to:

```

ABSTRACT INTERFACE
SUBROUTINE MPI_Comm_copy_attr_function(oldcomm, comm_keyval, extra_state,
    attribute_val_in, attribute_val_out, flag, ierror) BIND(C)
    TYPE(MPI_Comm) :: oldcomm
    INTEGER :: comm_keyval, ierror
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
    attribute_val_out
    LOGICAL :: flag

```

```

ABSTRACT INTERFACE
SUBROUTINE MPI_Comm_delete_attr_function(comm, comm_keyval,
    attribute_val, extra_state, ierror) BIND(C)
    TYPE(MPI_Comm) :: comm
    INTEGER :: comm_keyval, ierror
    INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state

```

The copy and delete function arguments to MPI_Win_create_keyval should be declared according to:

```

ABSTRACT INTERFACE

```

```

SUBROUTINE MPI_Win_copy_attr_function(oldwin, win_keyval, extra_state,
attribute_val_in, attribute_val_out, flag, ierror) BIND(C)
  TYPE(MPI_Win) :: oldwin
  INTEGER :: win_keyval, ierror
  INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
attribute_val_out
  LOGICAL :: flag

```

ABSTRACT INTERFACE

```

SUBROUTINE MPI_Win_delete_attr_function(win, win_keyval, attribute_val,
extra_state, ierror) BIND(C)
  TYPE(MPI_Win) :: win
  INTEGER :: win_keyval, ierror
  INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state

```

The copy and delete function arguments to MPI_Type_create_keyval should be declared according to:

ABSTRACT INTERFACE

```

SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_state,
attribute_val_in, attribute_val_out, flag, ierror) BIND(C)
  TYPE(MPI_Datatype) :: oldtype
  INTEGER :: type_keyval, ierror
  INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
attribute_val_out
  LOGICAL :: flag

```

ABSTRACT INTERFACE

```

SUBROUTINE MPI_Type_delete_attr_function(datatype, type_keyval,
attribute_val, extra_state, ierror) BIND(C)
  TYPE(MPI_Datatype) :: datatype
  INTEGER :: type_keyval, ierror
  INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state

```

The handler-function argument to MPI_Comm_create_errhandler should be declared like this:

ABSTRACT INTERFACE

```

SUBROUTINE MPI_Comm_errhandler_function(comm, error_code) BIND(C)
  TYPE(MPI_Comm) :: comm
  INTEGER :: error_code

```

The handler-function argument to MPI_Win_create_errhandler should be declared like this:

ABSTRACT INTERFACE

```

SUBROUTINE MPI_Win_errhandler_function(win, error_code) BIND(C)
  TYPE(MPI_Win) :: win
  INTEGER :: error_code

```

The handler-function argument to MPI_File_create_errhandler should be declared like this:

ABSTRACT INTERFACE

```

1      SUBROUTINE MPI_File_errhandler_function(file, error_code) BIND(C)
2          TYPE(MPI_File) :: file
3          INTEGER :: error_code

```

The query, free, and cancel function arguments to MPI_Grequest_start should be declared according to:

```

7  ABSTRACT INTERFACE
8      SUBROUTINE MPI_Grequest_query_function(extra_state, status, ierror)
9          BIND(C)
10         TYPE(MPI_Status) :: status
11         INTEGER :: ierror
12         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state

```

```

13 ABSTRACT INTERFACE
14     SUBROUTINE MPI_Grequest_free_function(extra_state, ierror) BIND(C)
15         INTEGER :: ierror
16         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state

```

```

18 ABSTRACT INTERFACE
19     SUBROUTINE MPI_Grequest_cancel_function(extra_state, complete, ierror)
20         BIND(C)
21         INTEGER :: ierror
22         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
23         LOGICAL :: complete

```

The extend and conversion function arguments to MPI_Register_datarep should be declared according to:

```

26 ABSTRACT INTERFACE
27     SUBROUTINE MPI_Datarep_extent_function(datatype, extent, extra_state,
28         ierror) BIND(C)
29         TYPE(MPI_Datatype) :: datatype
30         INTEGER(KIND=MPI_ADDRESS_KIND) :: extent, extra_state
31         INTEGER :: ierror

```

```

33 ABSTRACT INTERFACE
34     SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count,
35         filebuf, position, extra_state, ierror) BIND(C)
36         USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
37         TYPE(C_PTR), VALUE :: userbuf, filebuf
38         TYPE(MPI_Datatype) :: datatype
39         INTEGER :: count, ierror
40         INTEGER(KIND=MPI_OFFSET_KIND) :: position
41         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state

```

Fortran Bindings with mpif.h or the mpi Module

With the Fortran `mpi` module or `mpif.h`, here are examples of how each of the user-defined subroutines should be declared.

The user-function argument to MPI_OP_CREATE should be declared like this:

```

47 SUBROUTINE USER_FUNCTION(INVEC, INOUTVEC, LEN, [ticket252-W.]DATATYPE)
48     <type> INVEC(LEN), INOUTVEC(LEN)

```

```
INTEGER LEN, [ticket252-W.]DATATYPE
```

The copy and delete function arguments to MPI_COMM_CREATE_KEYVAL should be declared like these:

```
SUBROUTINE COMM_COPY_ATTR_[ticket250-V.]FUNCTION(OLDCOMM, COMM_KEYVAL, EXTRA_STATE,  
        ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)  
        INTEGER OLDCOMM, COMM_KEYVAL, IERROR  
        INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,  
        ATTRIBUTE_VAL_OUT  
        LOGICAL FLAG
```

```
SUBROUTINE COMM_DELETE_ATTR_[ticket250-V.]FUNCTION(COMM, COMM_KEYVAL, ATTRIBUTE_VAL,  
        EXTRA_STATE, IERROR)  
        INTEGER COMM, COMM_KEYVAL, IERROR  
        INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
```

The copy and delete function arguments to MPI_WIN_CREATE_KEYVAL should be declared like these:

```
SUBROUTINE WIN_COPY_ATTR_[ticket250-V.]FUNCTION(OLDWIN, WIN_KEYVAL, EXTRA_STATE,  
        ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)  
        INTEGER OLDWIN, WIN_KEYVAL, IERROR  
        INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,  
        ATTRIBUTE_VAL_OUT  
        LOGICAL FLAG
```

```
SUBROUTINE WIN_DELETE_ATTR_[ticket250-V.]FUNCTION(WIN, WIN_KEYVAL, ATTRIBUTE_VAL,  
        EXTRA_STATE, IERROR)  
        INTEGER WIN, WIN_KEYVAL, IERROR  
        INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
```

The copy and delete function arguments to MPI_TYPE_CREATE_KEYVAL should be declared like these:

```
SUBROUTINE TYPE_COPY_ATTR_[ticket250-V.]FUNCTION(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE,  
        ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)  
        INTEGER OLDTYPE, TYPE_KEYVAL, IERROR  
        INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE,  
        ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT  
        LOGICAL FLAG
```

```
SUBROUTINE TYPE_DELETE_ATTR_[ticket250-V.]FUNCTION([ticket252-W.]DATATYPE, TYPE_KEYVAL, ATT  
        EXTRA_STATE, IERROR)  
        INTEGER [ticket252-W.]DATATYPE, TYPE_KEYVAL, IERROR  
        INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
```

The handler-function argument to MPI_COMM_CREATE_ERRHANDLER should be declared like this:

```
1  SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE)
```

```
2      INTEGER COMM, ERROR_CODE
```

3
4 The handler-function argument to MPI_WIN_CREATE_ERRHANDLER should be de-
5 clared like this:

```
6  
7  SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)
```

```
8      INTEGER WIN, ERROR_CODE
```

9
10 The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de-
11 clared like this:

```
12  
13 SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE)
```

```
14     INTEGER FILE, ERROR_CODE
```

15
16 The query, free, and cancel function arguments to MPI_GREQUEST_START should be
17 declared like these:

```
18 SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)
```

```
19     INTEGER STATUS(MPI_STATUS_SIZE), IERROR
```

```
20     INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

```
21  
22 SUBROUTINE GREQUEST_FREE_FUNCTION(EXTRA_STATE, IERROR)
```

```
23     INTEGER IERROR
```

```
24     INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

```
25  
26 SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR)
```

```
27     INTEGER IERROR
```

```
28     INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

```
29     LOGICAL COMPLETE
```

30
31 The extend and conversion function arguments to MPI_REGISTER_DATAREP should
32 be declared like these:

```
33  
34 SUBROUTINE DATAREP_EXTENT_FUNCTION(DATATYPE, EXTENT, EXTRA_STATE, IERROR)
```

```
35     INTEGER DATATYPE, IERROR
```

```
36     INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, EXTRA_STATE
```

```
37  
38 SUBROUTINE DATAREP_CONVERSION_FUNCTION(USERBUF, DATATYPE, COUNT, FILEBUF,  
39     POSITION, EXTRA_STATE, IERROR)
```

```
40     <TYPE> USERBUF(*), FILEBUF(*)
```

```
41     INTEGER COUNT, DATATYPE, IERROR
```

```
42     INTEGER(KIND=MPI_OFFSET_KIND) POSITION
```

```
43     INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

44 ticket230-B.

C++ Bindings (deprecated)

The following are defined C++ typedefs, also included in the file `mpi.h`.

```
namespace MPI {
    typedef void User_function(const void* invec, void *inoutvec,
                              int len, const Datatype& datatype);

    typedef int Comm::Copy_attr_function(const Comm& oldcomm,
                                         int comm_keyval, void* extra_state, void* attribute_val_in,
                                         void* attribute_val_out, bool& flag);
    typedef int Comm::Delete_attr_function(Comm& comm, int
                                         comm_keyval, void* attribute_val, void* extra_state);

    typedef int Win::Copy_attr_function(const Win& oldwin,
                                         int win_keyval, void* extra_state, void* attribute_val_in,
                                         void* attribute_val_out, bool& flag);
    typedef int Win::Delete_attr_function(Win& win, int
                                         win_keyval, void* attribute_val, void* extra_state);

    typedef int Datatype::Copy_attr_function(const Datatype& oldtype,
                                              int type_keyval, void* extra_state,
                                              const void* attribute_val_in, void* attribute_val_out,
                                              bool& flag);
    typedef int Datatype::Delete_attr_function(Datatype& [ticket3.].0){252-W}{data}type,
                                              int type_keyval, void* attribute_val, void* extra_state);

    typedef void Comm::Errhandler_function(Comm &, int *, ...);
    typedef void Win::Errhandler_function(Win &, int *, ...);
    typedef void File::Errhandler_function(File &, int *, ...);

    typedef int Grequest::Query_function(void* extra_state, Status& status);
    typedef int Grequest::Free_function(void* extra_state);
    typedef int Grequest::Cancel_function(void* extra_state, bool complete);

    typedef void Datarep_extent_function(const Datatype& datatype,
                                         Aint& file_extent, void* extra_state);
    typedef void Datarep_conversion_function(void* userbuf,
                                              Datatype& datatype, int count, void* filebuf,
                                              Offset position, void* extra_state);
}
```

A.1.4 Deprecated Prototype Definitions

The following are defined C typedefs for deprecated user-defined functions, also included in the file `mpi.h`.

```
/* prototypes for user-defined functions */
typedef int MPI_Copy_function(MPI_Comm oldcomm, int keyval,
                             void *extra_state, void *attribute_val_in,
```

```

1  int MPI_Pack(void* inbuf, int incount, MPI_Datatype datatype, void *outbuf,
2      int outsize, int *position, MPI_Comm comm)
3
4  int MPI_Pack_external(char *datarep, void *inbuf, int incount,
5      MPI_Datatype datatype, void *outbuf, MPI_Aint outsize,
6      MPI_Aint *position)
7
8  int MPI_Pack_external_size(char *datarep, int incount,
9      MPI_Datatype datatype, MPI_Aint *size)
10
11 int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm,
12     int *size)
13
14 int MPI_Type_commit(MPI_Datatype *datatype)
15
16 int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
17     MPI_Datatype *newtype)
18
19 int MPI_Type_create_darray(int size, int rank, int ndims,
20     int array_of_gsizes[], int array_of_distribs[], int
21     array_of_dargs[], int array_of_psize[], int order,
22     MPI_Datatype oldtype, MPI_Datatype *newtype)
23
24 int MPI_Type_create_hindexed(int count, int array_of_blocklengths[],
25     MPI_Aint array_of_displacements[], MPI_Datatype oldtype,
26     MPI_Datatype *newtype)
27
28 int MPI_Type_create_hvector(int count, int blocklength, MPI_Aint stride,
29     MPI_Datatype oldtype, MPI_Datatype *newtype)
30
31 int MPI_Type_create_indexed_block(int count, int blocklength,
32     int array_of_displacements[], MPI_Datatype oldtype,
33     MPI_Datatype *newtype)
34
35 int MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb, MPI_Aint
36     extent, MPI_Datatype *newtype)
37
38 int MPI_Type_create_struct(int count, int array_of_blocklengths[],
39     MPI_Aint array_of_displacements[],
40     MPI_Datatype array_of_types[], MPI_Datatype *newtype)
41
42 int MPI_Type_create_subarray(int ndims, int array_of_sizes[],
43     int array_of_subsizes[], int array_of_starts[], int order,
44     MPI_Datatype oldtype, MPI_Datatype *newtype)
45
46 int MPI_Type_dup(MPI_Datatype oldtype, MPI_Datatype *newtype)
47
48 int MPI_Type_free(MPI_Datatype *datatype)
49
50 int MPI_Type_get_contents(MPI_Datatype datatype, int max_integers,
51     int max_addresses, int max_datatypes, int array_of_integers[],
52     MPI_Aint array_of_addresses[],
53     MPI_Datatype array_of_datatypes[])
54
55 int MPI_Type_get_envelope(MPI_Datatype datatype, int *num_integers,

```

```

int MPI_Ireduce(void* sendbuf, void* recvbuf, int count,
                MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,
                MPI_Request* request)
int MPI_Ireduce_scatter(void* sendbuf, void* recvbuf, int recvcounts[],
                        MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
                        MPI_Request* request)
int MPI_Ireduce_scatter_block(void* sendbuf, void* recvbuf, int recvcount,
                              MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
                              MPI_Request* request)
int MPI_Iscan(void* sendbuf, void* recvbuf, int count,
              MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
              MPI_Request* request)
int MPI_Iscatter(void* sendbuf, int sendcount, MPI_Datatype sendtype,
                void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
                MPI_Comm comm, MPI_Request* request)
int MPI_Iscatterv(void* sendbuf, int sendcounts[], int displs[],
                  MPI_Datatype sendtype, void* recvbuf, int recvcount,
                  MPI_Datatype recvtype, int root, MPI_Comm comm,
                  MPI_Request* request)
int MPI_Op_commutative(MPI_Op op, int *commute)
int MPI_Op_create(MPI_User_function* user_fn, int commute, MPI_Op* op)
int MPI_Reduce(void* sendbuf, void* recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
int MPI_Reduce_local(void* inbuf, void* inoutbuf, int count,
                     MPI_Datatype datatype, MPI_Op op)
int MPI_Reduce_scatter(void* sendbuf, void* recvbuf, int recvcounts[],
                       MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
int MPI_Reduce_scatter_block(void* sendbuf, void* recvbuf, int recvcount,
                              MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
int MPI_Scan(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype,
                void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
                MPI_Comm comm)
int MPI_Scatterv(void* sendbuf, int sendcounts[], int displs[],
                 MPI_Datatype sendtype, void* recvbuf, int recvcount,
                 MPI_Datatype recvtype, int root, MPI_Comm comm)
int MPI_op_free(MPI_Op *op)

```



```

int MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
int MPI_Group_intersection(MPI_Group group1, MPI_Group group2,
    MPI_Group *newgroup)
int MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3],
    MPI_Group *newgroup)
int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3],
    MPI_Group *newgroup)
int MPI_Group_rank(MPI_Group group, int *rank)
int MPI_Group_size(MPI_Group group, int *size)
int MPI_Group_translate_ranks (MPI_Group group1, int n, int *ranks1,
    MPI_Group group2, int *ranks2)
int MPI_Group_union(MPI_Group group1, MPI_Group group2,
    MPI_Group *newgroup)
int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader,
    MPI_Comm peer_comm, int remote_leader, int tag,
    MPI_Comm *newintercomm)
int MPI_Intercomm_merge(MPI_Comm intercomm, int high,
    MPI_Comm *newintracomm)
int MPI_TYPE_DUP_FN(MPI_Datatype oldtype, int type_keyval,
    void *extra_state, void *attribute_val_in,
    void *attribute_val_out, int *flag)
int MPI_TYPE_NULL_COPY_FN(MPI_Datatype oldtype, int type_keyval,
    void *extra_state, void *attribute_val_in,
    void *attribute_val_out, int *flag)
int MPI_TYPE_NULL_DELETE_FN(MPI_Datatype datatype, int type_keyval, void
    *attribute_val, void *extra_state)
int MPI_Type_create_keyval(MPI_Type_copy_attr_function *type_copy_attr_fn,
    MPI_Type_delete_attr_function *type_delete_attr_fn,
    int *type_keyval, void *extra_state)
int MPI_Type_delete_attr(MPI_Datatype datatype, int type_keyval)
int MPI_Type_free_keyval(int *type_keyval)
int MPI_Type_get_attr(MPI_Datatype datatype, int type_keyval, void
    *attribute_val, int *flag)
int MPI_Type_get_name(MPI_Datatype datatype, char *type_name, int
    *resultlen)
int MPI_Type_set_attr(MPI_Datatype datatype, int type_keyval,
    void *attribute_val)

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48

ticket252-W.

ticket252-W.

ticket252-W.

ticket252-W.

ticket252-W.

```

ticket252-W. 1  int MPI_Type_set_name(MPI_Datatype datatype, char *type_name)
2
3  int MPI_WIN_DUP_FN(MPI_Win oldwin, int win_keyval, void *extra_state,
4                    void *attribute_val_in, void *attribute_val_out, int *flag)
5
6  int MPI_WIN_NULL_COPY_FN(MPI_Win oldwin, int win_keyval, void *extra_state,
7                          void *attribute_val_in, void *attribute_val_out, int *flag)
8
9  int MPI_WIN_NULL_DELETE_FN(MPI_Win win, int win_keyval, void
10                           *attribute_val, void *extra_state)
11
12 int MPI_Win_create_keyval(MPI_Win_copy_attr_function *win_copy_attr_fn,
13                          MPI_Win_delete_attr_function *win_delete_attr_fn,
14                          int *win_keyval, void *extra_state)
15
16 int MPI_Win_delete_attr(MPI_Win win, int win_keyval)
17
18 int MPI_Win_free_keyval(int *win_keyval)
19
20 int MPI_Win_get_attr(MPI_Win win, int win_keyval, void *attribute_val,
21                    int *flag)
22
23 int MPI_Win_get_name(MPI_Win win, char *win_name, int *resultlen)
24
25 int MPI_Win_set_attr(MPI_Win win, int win_keyval, void *attribute_val)
26
27 int MPI_Win_set_name(MPI_Win win, char *win_name)

```

A.2.5 Process Topologies C Bindings

```

26 int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)
27
28 int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods,
29                   int reorder, MPI_Comm *comm_cart)
30
31 int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims, int *periods,
32                int *coords)
33
34 int MPI_Cart_map(MPI_Comm comm, int ndims, int *dims, int *periods,
35                int *newrank)
36
37 int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
38
39 int MPI_Cart_shift(MPI_Comm comm, int direction, int disp,
40                  int *rank_source, int *rank_dest)
41
42 int MPI_Cart_sub(MPI_Comm comm, int *remain_dims, MPI_Comm *newcomm)
43
44 int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
45
46 int MPI_Dims_create(int nnodes, int ndims, int *dims)
47
48 int MPI_Dist_graph_create(MPI_Comm comm_old, int n, int sources[],
49                          int degrees[], int destinations[], int weights[],
50                          MPI_Info info, int reorder, MPI_Comm *comm_dist_graph)

```

```

int MPI_Dist_graph_create_adjacent(MPI_Comm comm_old, int indegree,
    int sources[], int sourceweights[], int outdegree,
    int destinations[], int destweights[], MPI_Info info,
    int reorder, MPI_Comm *comm_dist_graph)

int MPI_Dist_graph_neighbors(MPI_Comm comm, int maxindegree, int sources[],
    int sourceweights[], int maxoutdegree, int destinations[],
    int destweights[])

int MPI_Dist_graph_neighbors_count(MPI_Comm comm, int *indegree,
    int *outdegree, int *weighted)

int MPI_Graph_create(MPI_Comm comm_old, int nnodes, int *index, int *edges,
    int reorder, MPI_Comm *comm_graph)

int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int *index,
    int *edges)

int MPI_Graph_map(MPI_Comm comm, int nnodes, int *index, int *edges,
    int *newrank)

int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors,
    int *neighbors)

int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)

int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)

int MPI_Topo_test(MPI_Comm comm, int *status)

```

A.2.6 MPI Environmental Management C Bindings

```

double MPI_Wtick(void)

double MPI_Wtime(void)

int MPI_Abort(MPI_Comm comm, int errorcode)

int MPI_Add_error_class(int *errorclass)

int MPI_Add_error_code(int errorclass, int *errorcode)

int MPI_Add_error_string(int errorcode, char *string)

int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr)

int MPI_Comm_call_errhandler(MPI_Comm comm, int errorcode)

int MPI_Comm_create_errhandler(MPI_Comm_errhandler_function
    *comm_errhandler_fn, MPI_Errhandler *errhandler)

int MPI_Comm_get_errhandler(MPI_Comm comm, MPI_Errhandler *errhandler)

int MPI_Comm_set_errhandler(MPI_Comm comm, MPI_Errhandler errhandler)

int MPI_Errhandler_free(MPI_Errhandler *errhandler)

```

ticket252-W.

```

1  int MPI_Error_class(int errorcode, int *errorclass)
2
3  int MPI_Error_string(int errorcode, char *string, int *resultlen)
4
5  int MPI_File_call_errhandler(MPI_File fh, int errorcode)
6
7  int MPI_File_create_errhandler(MPI_File_errhandler_function
   *file_errhandler_fn, MPI_Errhandler *errhandler)
8
9  int MPI_File_get_errhandler(MPI_File file, MPI_Errhandler *errhandler)
10
11 int MPI_File_set_errhandler(MPI_File file, MPI_Errhandler errhandler)
12
13 int MPI_Finalize(void)
14
15 int MPI_Finalized(int *flag)
16
17 int MPI_Free_mem(void *base)
18
19 int MPI_Get_processor_name(char *name, int *resultlen)
20
21 int MPI_Get_version(int *version, int *subversion)
22
23 int MPI_Init(int *argc, char ***argv)
24
25 int MPI_Initialized(int *flag)
26
27 int MPI_Win_call_errhandler(MPI_Win win, int errorcode)
28
29 int MPI_Win_create_errhandler(MPI_Win_errhandler_function
   *win_errhandler_fn, MPI_Errhandler *errhandler)
30
31 int MPI_Win_get_errhandler(MPI_Win win, MPI_Errhandler *errhandler)
32
33 int MPI_Win_set_errhandler(MPI_Win win, MPI_Errhandler errhandler)

```

A.2.7 The Info Object C Bindings

```

34 int MPI_Info_create(MPI_Info *info)
35
36 int MPI_Info_delete(MPI_Info info, char *key)
37
38 int MPI_Info_dup(MPI_Info info, MPI_Info *newinfo)
39
40 int MPI_Info_free(MPI_Info *info)
41
42 int MPI_Info_get(MPI_Info info, char *key, int valuelen, char *value,
   int *flag)
43
44 int MPI_Info_get_nkeys(MPI_Info info, int *nkeys)
45
46 int MPI_Info_get_nthkey(MPI_Info info, int n, char *key)
47
48 int MPI_Info_get_valuelen(MPI_Info info, char *key, int *valuelen,
   int *flag)
49
50 int MPI_Info_set(MPI_Info info, char *key, char *value)

```

```
void *extra_state)
```

A.2.12 Language Bindings C Bindings

```
int MPI_Status_f082f(MPI_F08_status *f08_status, MPI_Fint *f_status)
int MPI_Status_f2f08(MPI_Fint *f_status, MPI_F08_status *f08_status)
int MPI_Type_create_f90_complex(int p, int r, MPI_Datatype *newtype)
int MPI_Type_create_f90_integer(int r, MPI_Datatype *newtype)
int MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype)
int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *datatype)
MPI_Fint MPI_Comm_c2f(MPI_Comm comm)
MPI_Comm MPI_Comm_f2c(MPI_Fint comm)
MPI_Fint MPI_Errhandler_c2f(MPI_Errhandler errhandler)
MPI_Errhandler MPI_Errhandler_f2c(MPI_Fint errhandler)
MPI_Fint MPI_File_c2f(MPI_File file)
MPI_File MPI_File_f2c(MPI_Fint file)
MPI_Fint MPI_Group_c2f(MPI_Group group)
MPI_Group MPI_Group_f2c(MPI_Fint group)
MPI_Fint MPI_Info_c2f(MPI_Info info)
MPI_Info MPI_Info_f2c(MPI_Fint info)
MPI_Fint MPI_Op_c2f(MPI_Op op)
MPI_Op MPI_Op_f2c(MPI_Fint op)
MPI_Fint MPI_Request_c2f(MPI_Request request)
MPI_Request MPI_Request_f2c(MPI_Fint request)
int MPI_Status_c2f(MPI_Status *c_status, MPI_Fint *f_status)
int MPI_Status_c2f08(MPI_Status *c_status, MPI_F08_status *f08_status)
int MPI_Status_f082c(MPI_F08_status *f08_status, MPI_Status *c_status)
int MPI_Status_f2c(MPI_Fint *f_status, MPI_Status *c_status)
MPI_Fint MPI_Type_c2f(MPI_Datatype datatype)
MPI_Datatype MPI_Type_f2c(MPI_Fint datatype)
MPI_Fint MPI_Win_c2f(MPI_Win win)
MPI_Win MPI_Win_f2c(MPI_Fint win)
```

A.2.13 Profiling Interface C Bindings

```
int MPI_Pcontrol(const int level, ...)
```

A.2.14 Deprecated C Bindings

```
int MPI_Address(void* location, MPI_Aint *address)
```

```
int MPI_Attr_delete(MPI_Comm comm, int keyval)
```

```
int MPI_Attr_get(MPI_Comm comm, int keyval, void *attribute_val, int *flag)
```

```
int MPI_Attr_put(MPI_Comm comm, int keyval, void* attribute_val)
```

```
int MPI_DUP_FN(MPI_Comm oldcomm, int keyval, void *extra_state,
               void *attribute_val_in, void *attribute_val_out, int *flag)
```

```
int MPI_Errhandler_create(MPI_Handler_function *handler_fn,
                          MPI_Errhandler *errhandler)
```

```
int MPI_Errhandler_get(MPI_Comm comm, MPI_Errhandler *errhandler)
```

```
int MPI_Errhandler_set(MPI_Comm comm, MPI_Errhandler errhandler)
```

```
int MPI_Keyval_create(MPI_Copy_function *copy_fn, MPI_Delete_function
                     *delete_fn, int *keyval, void* extra_state)
```

```
int MPI_Keyval_free(int *keyval)
```

```
int MPI_NULL_COPY_FN(MPI_Comm oldcomm, int keyval, void *extra_state,
                     void *attribute_val_in, void *attribute_val_out, int *flag)
```

```
int MPI_NULL_DELETE_FN(MPI_Comm comm, int keyval, void *attribute_val,
                       void *extra_state)
```

```
int MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent)
```

```
int MPI_Type_hindexed(int count, int *array_of_blocklengths,
                      MPI_Aint *array_of_displacements, MPI_Datatype oldtype,
                      MPI_Datatype *newtype)
```

```
int MPI_Type_hvector(int count, int blocklength, MPI_Aint stride,
                    MPI_Datatype oldtype, MPI_Datatype *newtype)
```

```
int MPI_Type_lb(MPI_Datatype datatype, MPI_Aint* displacement)
```

```
int MPI_Type_struct(int count, int *array_of_blocklengths,
                   MPI_Aint *array_of_displacements,
                   MPI_Datatype *array_of_types, MPI_Datatype *newtype)
```

```
int MPI_Type_ub(MPI_Datatype datatype, MPI_Aint* displacement)
```

A.4 Fortran Bindings with mpif.h or the mpi Module

A.4.1 Point-to-Point Communication Fortran Bindings

```

MPI_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR

MPI_BSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI_BUFFER_ATTACH(BUFFER, SIZE, IERROR)
    <type> BUFFER(*)
    INTEGER SIZE, IERROR

MPI_BUFFER_DETACH(BUFFER_ADDR, SIZE, IERROR)
    <type> BUFFER_ADDR(*)
    INTEGER SIZE, IERROR

MPI_CANCEL(REQUEST, IERROR)
    INTEGER REQUEST, IERROR

MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR

MPI_IBSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR)
    LOGICAL FLAG
    INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR

MPI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)
    INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)

```

```

1      <type> BUF(*)
2      INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),
3      IERROR
4
5      MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
6      <type> BUF(*)
7      INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
8
9      MPI_REQUEST_FREE(REQUEST, IERROR)
10     INTEGER REQUEST, IERROR
11
12     MPI_REQUEST_GET_STATUS( REQUEST, FLAG, STATUS, IERROR)
13     INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
14     LOGICAL FLAG
15
16     MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
17     <type> BUF(*)
18     INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
19
20     MPI_RSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
21     <type> BUF(*)
22     INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
23
24     MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
25     <type> BUF(*)
26     INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
27
28     MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF,
29     RECVCOUNT, RECVMODE, SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR)
30     <type> SENDBUF(*), RECVBUF(*)
31     INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVMODE,
32     SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
33
34     MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG,
35     COMM, STATUS, IERROR)
36     <type> BUF(*)
37     INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM,
38     STATUS(MPI_STATUS_SIZE), IERROR
39
40     MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
41     <type> BUF(*)
42     INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
43
44     MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
45     <type> BUF(*)
46     INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
47
48     MPI_SSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
49     <type> BUF(*)
50     INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
51
52     MPI_START(REQUEST, IERROR)
53     INTEGER REQUEST, IERROR

```



```

    INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)
1
2
MPI_TYPE_CREATE_SUBARRAY(NDIMS, ARRAY_OF_SIZES, ARRAY_OF_SUBSIZES,
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
    INTEGER NDIMS, ARRAY_OF_SIZES(*), ARRAY_OF_SUBSIZES(*),
    ARRAY_OF_STARTS(*), ORDER, OLDTYPE, NEWTYPE, IERROR)

MPI_TYPE_DUP(OLDTYPE, NEWTYPE, IERROR)
    INTEGER OLDTYPE, NEWTYPE, IERROR

MPI_TYPE_FREE(DATATYPE, IERROR)
    INTEGER DATATYPE, IERROR

MPI_TYPE_GET_CONTENTS(DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,
    ARRAY_OF_INTEGERS, ARRAY_OF_ADDRESSES, ARRAY_OF_DATATYPES,
    IERROR)
    INTEGER DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,
    ARRAY_OF_INTEGERS(*), ARRAY_OF_DATATYPES(*), IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_ADDRESSES(*)

MPI_TYPE_GET_ENVELOPE(DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES,
    COMBINER, IERROR)
    INTEGER DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES, COMBINER,
    IERROR

MPI_TYPE_GET_EXTENT(DATATYPE, LB, EXTENT, IERROR)
    INTEGER DATATYPE, IERROR
    INTEGER(KIND = MPI_ADDRESS_KIND) LB, EXTENT

MPI_TYPE_GET_TRUE_EXTENT(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)
    INTEGER DATATYPE, IERROR
    INTEGER(KIND = MPI_ADDRESS_KIND) TRUE_LB, TRUE_EXTENT

MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
    OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
    OLDTYPE, NEWTYPE, IERROR

MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR)
    INTEGER DATATYPE, SIZE, IERROR

MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR

MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, COMM,
    IERROR)
    <type> INBUF(*), OUTBUF(*)
    INTEGER INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM, IERROR

MPI_UNPACK_EXTERNAL(DATAREP, INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT,
    DATATYPE, IERROR)
    INTEGER OUTCOUNT, DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) INSIZE, POSITION

```

```

1      <type> SENDBUF(*), RECVBUF(*)
2      INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,
3      IERROR
4
5      MPI_IGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
6                  RECVTYPE, ROOT, COMM, REQUEST, IERROR)
7      <type> SENDBUF(*), RECVBUF(*)
8      INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
9      COMM, REQUEST, IERROR
10
11     MPI_IREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, REQUEST,
12                IERROR)
13     <type> SENDBUF(*), RECVBUF(*)
14     INTEGER COUNT, DATATYPE, OP, ROOT, COMM, REQUEST, IERROR
15
16     MPI_IREDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
17                        REQUEST, IERROR)
18     <type> SENDBUF(*), RECVBUF(*)
19     INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, REQUEST, IERROR
20
21     MPI_IREDUCE_SCATTER_BLOCK(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,
22                              REQUEST, IERROR)
23     <type> SENDBUF(*), RECVBUF(*)
24     INTEGER RECVCOUNT, DATATYPE, OP, COMM, REQUEST, IERROR
25
26     MPI_ISCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)
27     <type> SENDBUF(*), RECVBUF(*)
28     INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR
29
30     MPI_ISCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
31                 ROOT, COMM, REQUEST, IERROR)
32     <type> SENDBUF(*), RECVBUF(*)
33     INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,
34     IERROR
35
36     MPI_ISCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,
37                  RECVTYPE, ROOT, COMM, REQUEST, IERROR)
38     <type> SENDBUF(*), RECVBUF(*)
39     INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
40     COMM, REQUEST, IERROR
41
42     MPI_OP_COMMUTATIVE(OP, COMMUTE, IERROR)
43     LOGICAL COMMUTE
44     INTEGER OP, IERROR
45
46     MPI_OP_CREATE( USER_FN, COMMUTE, OP, IERROR)
47     EXTERNAL USER_FN
48     LOGICAL COMMUTE
49     INTEGER OP, IERROR
50
51     MPI_OP_FREE(OP, IERROR)
52     INTEGER OP, IERROR

```

ticket252-W. 42 MPI_OP_CREATE(**USER_FN**, COMMUTE, OP, IERROR)

ticket252-W. 43 EXTERNAL **USER_FN**

```

MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR

MPI_REDUCE_LOCAL(INBUF, INOUTBUF, COUNT, DATATYPE, OP, IERROR)
    <type> INBUF(*), INOUTBUF(*)
    INTEGER COUNT, DATATYPE, OP, IERROR

MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
    IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR

MPI_REDUCE_SCATTER_BLOCK(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,
    IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER RECVCOUNT, DATATYPE, OP, COMM, IERROR

MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, IERROR

MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
    ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR

MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,
    RECVTYPE, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
    COMM, IERROR

```

A.4.4 Groups, Contexts, Communicators, and Caching Fortran Bindings

```

MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERROR)
    INTEGER COMM1, COMM2, RESULT, IERROR

MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)
    INTEGER COMM, GROUP, NEWCOMM, IERROR

MPI_COMM_CREATE_KEYVAL(COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN, COMM_KEYVAL,
    EXTRA_STATE, IERROR)
    EXTERNAL COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN
    INTEGER COMM_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE

MPI_COMM_DELETE_ATTR(COMM, COMM_KEYVAL, IERROR)
    INTEGER COMM, COMM_KEYVAL, IERROR

MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
    INTEGER COMM, NEWCOMM, IERROR

```

MPI_COMM_SIZE(COMM, SIZE, IERROR)	1	
INTEGER COMM, SIZE, IERROR	2	
	3	
MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR)	4	
INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR	5	
	6	
MPI_COMM_TEST_INTER(COMM, FLAG, IERROR)	7	
INTEGER COMM, IERROR	8	
LOGICAL FLAG	9	
	10	
MPI_GROUP_COMPARE(GROUP1, GROUP2, RESULT, IERROR)	11	
INTEGER GROUP1, GROUP2, RESULT, IERROR	12	
	13	
MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR)	14	
INTEGER GROUP1, GROUP2, NEWGROUP, IERROR	15	
	16	
MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, IERROR)	17	
INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR	18	
	19	
MPI_GROUP_FREE(GROUP, IERROR)	20	
INTEGER GROUP, IERROR	21	
	22	
MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR)	23	
INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR	24	
	25	
MPI_GROUP_INTERSECTION(GROUP1, GROUP2, NEWGROUP, IERROR)	26	
INTEGER GROUP1, GROUP2, NEWGROUP, IERROR	27	
	28	
MPI_GROUP_RANGE_EXCL(GROUP, N, RANGES, NEWGROUP, IERROR)	29	
INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR	30	
	31	
MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERROR)	32	
INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR	33	
	34	
MPI_GROUP_RANK(GROUP, RANK, IERROR)	35	
INTEGER GROUP, RANK, IERROR	36	
	37	
MPI_GROUP_SIZE(GROUP, SIZE, IERROR)	38	
INTEGER GROUP, SIZE, IERROR	39	
	40	
MPI_GROUP_TRANSLATE_RANKS(GROUP1, N, RANKS1, GROUP2, RANKS2, IERROR)	41	
INTEGER GROUP1, N, RANKS1(*), GROUP2, RANKS2(*), IERROR	42	
	43	
MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP, IERROR)	44	
INTEGER GROUP1, GROUP2, NEWGROUP, IERROR	45	
	46	
MPI_INTERCOMM_CREATE(LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER,	47	
TAG, NEWINTERCOMM, IERROR)	48	
INTEGER LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG,		
NEWINTERCOMM, IERROR		
MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, NEWINTRACOMM, IERROR)		ticket250-V.
INTEGER INTERCOMM, NEWINTRACOMM, IERROR		ticket250-V.
LOGICAL HIGH		

```

1  MPI_TYPE_CREATE_KEYVAL(TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN, TYPE_KEYVAL,
2      EXTRA_STATE, IERROR)
3      EXTERNAL TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN
4      INTEGER TYPE_KEYVAL, IERROR
5      INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
6
ticket252-W. 7  MPI_TYPE_DELETE_ATTR(DATATYPE, TYPE_KEYVAL, IERROR)
ticket252-W. 8      INTEGER DATATYPE, TYPE_KEYVAL, IERROR
9
10 MPI_TYPE_DUP_FN(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
11     ATTRIBUTE_VAL_OUT, FLAG, IERROR)
12     INTEGER OLDTYPE, TYPE_KEYVAL, IERROR
13     INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
14     ATTRIBUTE_VAL_OUT
15     LOGICAL FLAG
16
17 MPI_TYPE_FREE_KEYVAL(TYPE_KEYVAL, IERROR)
18     INTEGER TYPE_KEYVAL, IERROR
19
ticket252-W. 18 MPI_TYPE_GET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
ticket252-W. 19     INTEGER DATATYPE, TYPE_KEYVAL, IERROR
20     INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
21     LOGICAL FLAG
22
ticket252-W. 23 MPI_TYPE_GET_NAME(DATATYPE, TYPE_NAME, RESULTLEN, IERROR)
ticket252-W. 24     INTEGER DATATYPE, RESULTLEN, IERROR
25     CHARACTER*(*) TYPE_NAME
26
27 MPI_TYPE_NULL_COPY_FN(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
28     ATTRIBUTE_VAL_OUT, FLAG, IERROR)
29     INTEGER OLDTYPE, TYPE_KEYVAL, IERROR
30     INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
31     ATTRIBUTE_VAL_OUT
32     LOGICAL FLAG
33
ticket252-W. 33 MPI_TYPE_NULL_DELETE_FN(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE,
34     IERROR)
35
ticket252-W. 35     INTEGER DATATYPE, TYPE_KEYVAL, IERROR
36     INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
37
38 MPI_TYPE_SET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR)
39     INTEGER DATATYPE, TYPE_KEYVAL, IERROR
40     INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
41
ticket252-W. 41 MPI_TYPE_SET_NAME(DATATYPE, TYPE_NAME, IERROR)
ticket252-W. 42     INTEGER DATATYPE, IERROR
43     CHARACTER*(*) TYPE_NAME
44
45 MPI_WIN_CREATE_KEYVAL(WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN, WIN_KEYVAL,
46     EXTRA_STATE, IERROR)
47     EXTERNAL WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN
48     INTEGER WIN_KEYVAL, IERROR

```

```

    INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*), IERROR
MPI_GRAPH_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR)
    INTEGER COMM, NNODES, INDEX(*), EDGES(*), NEWRANK, IERROR
MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR)
    INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR
MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS, IERROR)
    INTEGER COMM, RANK, NNEIGHBORS, IERROR
MPI_TOPO_TEST(COMM, STATUS, IERROR)
    INTEGER COMM, STATUS, IERROR

```

A.4.6 MPI Environmental Management Fortran Bindings

```

DOUBLE PRECISION MPI_WTICK()
DOUBLE PRECISION MPI_WTIME()
MPI_ABORT(COMM, ERRORCODE, IERROR)
    INTEGER COMM, ERRORCODE, IERROR
MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR)
    INTEGER ERRORCLASS, IERROR
MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR)
    INTEGER ERRORCLASS, ERRORCODE, IERROR
MPI_ADD_ERROR_STRING(ERRORCODE, STRING, IERROR)
    INTEGER ERRORCODE, IERROR
    CHARACTER*(*) STRING
MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
    INTEGER INFO, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
MPI_COMM_CALL_ERRHANDLER(COMM, ERRORCODE, IERROR)
    INTEGER COMM, ERRORCODE, IERROR
MPI_COMM_CREATE_ERRHANDLER(COMM_ERRHANDLER_FN, ERRHANDLER, IERROR)
    EXTERNAL COMM_ERRHANDLER_FN
    INTEGER ERRHANDLER, IERROR
MPI_COMM_GET_ERRHANDLER(COMM, ERRHANDLER, IERROR)
    INTEGER COMM, ERRHANDLER, IERROR
MPI_COMM_SET_ERRHANDLER(COMM, ERRHANDLER, IERROR)
    INTEGER COMM, ERRHANDLER, IERROR
MPI_ERRHANDLER_FREE(ERRHANDLER, IERROR)
    INTEGER ERRHANDLER, IERROR
MPI_ERROR_CLASS(ERRORCODE, ERRORCLASS, IERROR)
    INTEGER ERRORCODE, ERRORCLASS, IERROR

```

```

1  MPI_ERROR_STRING(ERRORCODE, STRING, RESULTLEN, IERROR)
2      INTEGER ERRORCODE, RESULTLEN, IERROR
3      CHARACTER*(*) STRING
4
5  MPI_FILE_CALL_ERRHANDLER(FH, ERRORCODE, IERROR)
6      INTEGER FH, ERRORCODE, IERROR
7
8  MPI_FILE_CREATE_ERRHANDLER(FILE_ERRHANDLER_FN, ERRHANDLER, IERROR)
9      EXTERNAL FILE_ERRHANDLER_FN
10     INTEGER ERRHANDLER, IERROR
11
12 MPI_FILE_GET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
13     INTEGER FILE, ERRHANDLER, IERROR
14
15 MPI_FILE_SET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
16     INTEGER FILE, ERRHANDLER, IERROR
17
18 MPI_FINALIZE(IERROR)
19     INTEGER IERROR
20
21 MPI_FINALIZED(FLAG, IERROR)
22     LOGICAL FLAG
23     INTEGER IERROR
24
25 MPI_FREE_MEM(BASE, IERROR)
26     <type> BASE(*)
27     INTEGER IERROR
28
29 MPI_GET_PROCESSOR_NAME( NAME, RESULTLEN, IERROR)
30     CHARACTER*(*) NAME
31     INTEGER RESULTLEN, IERROR
32
33 MPI_GET_VERSION(VERSION, SUBVERSION, IERROR)
34     INTEGER VERSION, SUBVERSION, IERROR
35
36 MPI_INIT(IERROR)
37     INTEGER IERROR
38
39 MPI_INITIALIZED(FLAG, IERROR)
40     LOGICAL FLAG
41     INTEGER IERROR
42
43 MPI_WIN_CALL_ERRHANDLER(WIN, ERRORCODE, IERROR)
44     INTEGER WIN, ERRORCODE, IERROR
45
46 MPI_WIN_CREATE_ERRHANDLER(WIN_ERRHANDLER_FN, ERRHANDLER, IERROR)
47     EXTERNAL WIN_ERRHANDLER_FN
48     INTEGER ERRHANDLER, IERROR
49
50 MPI_WIN_GET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
51     INTEGER WIN, ERRHANDLER, IERROR
52
53 MPI_WIN_SET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
54     INTEGER WIN, ERRHANDLER, IERROR
55

```

```

1 MPI_FILE_WRITE_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
2     <type> BUF(*)
3     INTEGER FH, COUNT, DATATYPE, IERROR
4
5 MPI_FILE_WRITE_ORDERED_END(FH, BUF, STATUS, IERROR)
6     <type> BUF(*)
7     INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
8
9 MPI_FILE_WRITE_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
10    <type> BUF(*)
11    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
12
13 MPI_REGISTER_DATAREP(DATAREP, READ_CONVERSION_FN, WRITE_CONVERSION_FN,
14                     DTYPE_FILE_EXTENT_FN, EXTRA_STATE, IERROR)
15    CHARACTER*(*) DATAREP
16    EXTERNAL READ_CONVERSION_FN, WRITE_CONVERSION_FN, DTYPE_FILE_EXTENT_FN
17    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
18    INTEGER IERROR

```

A.4.12 Language Bindings Fortran Bindings

```

19 MPI_F_SYNC_REG(buf)
20     <type> buf(*)
21
22 MPI_SIZEOF(X, SIZE, IERROR)
23     <type> X
24     INTEGER SIZE, IERROR
25
26 MPI_STATUS_F082F(F08_STATUS, F_STATUS, IERROR)
27     TYPE(MPI_Status) :: F08_STATUS
28     INTEGER :: F_STATUS(MPI_STATUS_SIZE)
29     INTEGER IERROR
30
31 MPI_STATUS_F2F08(F_STATUS, F08_STATUS, IERROR)
32     INTEGER :: F_STATUS(MPI_STATUS_SIZE)
33     TYPE(MPI_Status) :: F08_STATUS
34     INTEGER IERROR
35
36 MPI_TYPE_CREATE_F90_COMPLEX(P, R, NEWTYPE, IERROR)
37     INTEGER P, R, NEWTYPE, IERROR
38
39 MPI_TYPE_CREATE_F90_INTEGER(R, NEWTYPE, IERROR)
40     INTEGER R, NEWTYPE, IERROR
41
42 MPI_TYPE_CREATE_F90_REAL(P, R, NEWTYPE, IERROR)
43     INTEGER P, R, NEWTYPE, IERROR
44
45 MPI_TYPE_MATCH_SIZE(TYPECLASS, SIZE, DATATYPE, IERROR)
46     INTEGER TYPECLASS, SIZE, DATATYPE, IERROR

```

ticket252-W.
ticket252-W.

A.4.13 Profiling Interface Fortran Bindings

```

MPI_PCONTROL(LEVEL)
    INTEGER LEVEL

```

A.4.14 Deprecated Fortran Bindings

```

MPI_ADDRESS(LOCATION, ADDRESS, IERROR)
    <type> LOCATION(*)
    INTEGER ADDRESS, IERROR

MPI_ATTR_DELETE(COMM, KEYVAL, IERROR)
    INTEGER COMM, KEYVAL, IERROR

MPI_ATTR_GET(COMM, KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR
    LOGICAL FLAG

MPI_ATTR_PUT(COMM, KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR

MPI_DUP_FN(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
           ATTRIBUTE_VAL_OUT, FLAG, IERR)
    INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
    ATTRIBUTE_VAL_OUT, IERR
    LOGICAL FLAG

MPI_ERRHANDLER_CREATE(HANDLER_FN, ERRHANDLER, IERROR)
    EXTERNAL HANDLER_FN
    INTEGER ERRHANDLER, IERROR

MPI_ERRHANDLER_GET(COMM, ERRHANDLER, IERROR)
    INTEGER COMM, ERRHANDLER, IERROR

MPI_ERRHANDLER_SET(COMM, ERRHANDLER, IERROR)
    INTEGER COMM, ERRHANDLER, IERROR

MPI_KEYVAL_CREATE(COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR)
    EXTERNAL COPY_FN, DELETE_FN
    INTEGER KEYVAL, EXTRA_STATE, IERROR

MPI_KEYVAL_FREE(KEYVAL, IERROR)
    INTEGER KEYVAL, IERROR

MPI_NULL_COPY_FN(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
                ATTRIBUTE_VAL_OUT, FLAG, IERR)
    INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
    ATTRIBUTE_VAL_OUT, IERR
    LOGICAL FLAG

MPI_NULL_DELETE_FN(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR)
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR

```

ticket252-W.
ticket252-W.

```

const int displs[], const Datatype& recvtype, int root)
const = 0(binding deprecated, see Section 15.2) }

{Request Comm::Iallgather(const void* sendbuf, int sendcount, const
Datatype& sendtype, void* recvbuf, int recvcount,
const Datatype& recvtype) const = 0(binding deprecated, see
Section 15.2) }

{Request Comm::Iallgatherv(const void* sendbuf, int sendcount, const
Datatype& sendtype, void* recvbuf, const int recvcounts[],
const int displs[], const Datatype& recvtype) const = 0(binding
deprecated, see Section 15.2) }

{Request Comm::Iallreduce(const void* sendbuf, void* recvbuf, int count,
const Datatype& datatype, const Op& op) const = 0(binding
deprecated, see Section 15.2) }

{Request Comm::Ialltoall(const void* sendbuf, int sendcount, const
Datatype& sendtype, void* recvbuf, int recvcount,
const Datatype& recvtype) const = 0(binding deprecated, see
Section 15.2) }

{Request Comm::Ialltoallv(const void* sendbuf, const int sendcounts[],
const int sdispls[], const Datatype& sendtype, void* recvbuf,
const int recvcounts[], const int rdispls[],
const Datatype& recvtype) const = 0(binding deprecated, see
Section 15.2) }

{Request Comm::Ialltoallw(const void* sendbuf, const int sendcounts[],
const int sdispls[], const Datatype sendtypes[], void*
recvbuf, const int recvcounts[], const int rdispls[], const
Datatype recvtypes[]) const = 0(binding deprecated, see
Section 15.2) }

{Request Comm::Ibarrier() const = 0(binding deprecated, see Section 15.2) }

{Request Comm::Ibcast(void* buffer, int count, const Datatype& datatype,
int root) const = 0(binding deprecated, see Section 15.2) }

{Request Intracomm::Iexscan(const void* sendbuf, void* recvbuf, int
count, const Datatype& datatype, const Op& op) const(binding
deprecated, see Section 15.2) }

{Request Comm::Igather(const void* sendbuf, int sendcount, const
Datatype& sendtype, void* recvbuf, int recvcount,
const Datatype& recvtype, int root) const = 0(binding deprecated,
see Section 15.2) }

{Request Comm::Igatherv(const void* sendbuf, int sendcount, const
Datatype& sendtype, void* recvbuf, const int recvcounts[],
const int displs[], const Datatype& recvtype, int root)
const = 0(binding deprecated, see Section 15.2) }

```

```

1      {static Errhandler Comm::Create_errhandler(Comm::Errhandler_function*
ticket252-W. 2          comm_errhandler_fn) (binding deprecated, see Section 15.2) }
3
4      {static Errhandler File::Create_errhandler(File::Errhandler_function*
ticket252-W. 5          file_errhandler_fn) (binding deprecated, see Section 15.2) }
6
7      {static Errhandler Win::Create_errhandler(Win::Errhandler_function*
ticket252-W. 8          win_errhandler_fn) (binding deprecated, see Section 15.2) }
9
10     {void Finalize() (binding deprecated, see Section 15.2) }
11
12     {void Errhandler::Free() (binding deprecated, see Section 15.2) }
13
14     {void Free_mem(void *base) (binding deprecated, see Section 15.2) }
15
16     {Errhandler Comm::Get_errhandler() const (binding deprecated, see Section 15.2) }
17
18     {Errhandler File::Get_errhandler() const (binding deprecated, see Section 15.2) }
19
20     {Errhandler Win::Get_errhandler() const (binding deprecated, see Section 15.2) }
21
22     {int Get_error_class(int errorcode) (binding deprecated, see Section 15.2) }
23
24     {void Get_error_string(int errorcode, char* name, int& resultlen) (binding
25         deprecated, see Section 15.2) }
26
27     {void Get_processor_name(char* name, int& resultlen) (binding deprecated, see
28         Section 15.2) }
29
30     {void Get_version(int& version, int& subversion) (binding deprecated, see
31         Section 15.2) }
32
33     {void Init() (binding deprecated, see Section 15.2) }
34
35     {void Init(int& argc, char**& argv) (binding deprecated, see Section 15.2) }
36
37     {bool Is_finalized() (binding deprecated, see Section 15.2) }
38
39     {bool Is_initialized() (binding deprecated, see Section 15.2) }
40
41     {void Comm::Set_errhandler(const Errhandler& errhandler) (binding deprecated,
42         see Section 15.2) }
43
44     {void File::Set_errhandler(const Errhandler& errhandler) (binding deprecated,
45         see Section 15.2) }
46
47     {void Win::Set_errhandler(const Errhandler& errhandler) (binding deprecated,
48         see Section 15.2) }
49
50     {double Wtick() (binding deprecated, see Section 15.2) }
51
52     {double Wtime() (binding deprecated, see Section 15.2) }
53
54 };

```

A.5.7 The Info Object C++ Bindings

```
namespace MPI {
```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48

Annex B

Change-Log

This annex summarizes changes from the previous version of the MPI standard to the version presented by this document. Only significant changes (i.e., clarifications and new features) that might either require implementation effort in the MPI libraries or change the understanding of MPI from a user's perspective are presented. Editorial modifications, formatting, typo corrections and minor clarifications are not shown.

B.1 Changes from Version 2.2 to Version 3.0

1. Section 2.3 on page 10, and Sections 16.2.1, 16.2.2, 16.2.7 on pages 552, 553, and 568. The new `mpi_08` Fortran module is introduced.
2. Section 2.5.1 on page 12, Section 16.2.3 on page 556, Section 16.2.2 on page 553, and Section 16.2.7 on page 568.
Handles to opaque objects are defined as named types within the `mpi_08` Fortran module. The handle types are also available through the `mpi` Fortran module.
3. Sections 2.5.4, 2.5.5 on pages 15, 16, Sections 16.2.1, 16.2.10, 16.2.11, 16.2.12, 16.2.13 on pages 552, 579, 580, 581, 584, and Sections 16.2.3, 16.2.2, 16.2.7 on pages 556, 553, 568.
Within the `mpi_08` Fortran module, choice buffers are defined as assumed-type and assumed-rank according to Fortran 2008 TR 29113 [36], and the compile-time constant `MPI_SUBARRAYS_SUPPORTED` is set to `.TRUE..` With this, Fortran subscript triplets can be used in nonblocking MPI operations; vector subscripts are not supported in nonblocking operations. If the compiler does not support this Fortran TR 29113 feature, the constant is set to `.FALSE..`
4. Section 2.6.2 on page 18, Section 16.2.2 on page 553, and Section 16.2.7 on page 568. The `ierror` dummy arguments are `OPTIONAL` within the `mpi_08` Fortran module.
5. Section 3.2.5 on page 34, Section 16.2.3 on page 556, Section 16.2.2 on page 553, Section 16.2.7 on page 568, and Section 16.3.5 on page 604.
Within the `mpi_08` Fortran module, the status is defined as `TYPE(MPI_Status)`. New conversion routines are added: `MPI_STATUS_F2F08`, `MPI_STATUS_F082F`, `MPI_Status_c2f08`, and `MPI_Status_f082c`. In `mpi.h`, the new type `MPI_F08_status`, and the external variables `MPI_F08_STATUS_IGNORE` and `MPI_F08_STATUSES_IGNORE` are added.

- 1
2
3
4
ticket229.2. 5
6
7
ticket237-I. 8
ticket229.2. 9
10
11
12
13
14
ticket252-W. 15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
ticket251-V. 30
31
32
ticket245-Q. 33
34
35
36
37
38
ticket237-I. 39
40
41
42
ticket238-J. 43
44
45
46
47
ticket229.1. 48
6. Section 3.2.6 on page 36, and Section 3.8 on page 71.
MPI_STATUS_IGNORE can be also used in MPI_IProbe, MPI_Probe, MPI_Improbe, and MPI_Mprobe.
 7. Section 3.6 on page 49.
In Fortran with the `mpi` module or `mpif.h`, the type of the `buffer_addr` argument of `MPI_BUFFER_DETACH` is wrongly defined and the argument is therefore unused.
 8. Section 4.1 on page 85, Section 4.1.6 on page 106, and Section 16.2.15 on page 584.
The Fortran alignments of basic datatypes are implementation dependent. It is recommended that they are computed according to `BIND(C)` derived types. If an array of structures (in C/C++) or derived types (in Fortran) should be communicated, it is recommended that the user creates a portable datatype handle and applies additionally `MPI_TYPE_CREATE_RESIZED` to this datatype handle.
 9. Sections 4.1.10, 5.9.5, 5.9.7, 6.7.4, 6.8, 8.3.1, 8.3.2, 8.3.3, 15.1, 16.2.9 on pages 112, 188, 194, 282, 288, 333, 335, 337, 529, and 570. In some routines, the dummy argument names were changed, because they were identical to the Fortran keywords `TYPE` and `FUNCTION`. The new dummy argument names must be used because the `mpi` and `mpi_08` modules guarantee keyword-based actual argument lists. The argument name `type` was changed into `oldtype` in `MPI_TYPE_DUP`, and into `datatype` in the Fortran `USER_FUNCTION` of `MPI_OP_CREATE`, and in `MPI_TYPE_SET_ATTR`, `MPI_TYPE_GET_ATTR`, `MPI_TYPE_DELETE_ATTR`, `MPI_TYPE_SET_NAME`, `MPI_TYPE_GET_NAME`, `MPI_TYPE_MATCH_SIZE`, in the callback prototype definition `MPI_Type_delete_attr_function`, and the predefined callback function `MPI_TYPE_NULL_DELETE_FN`; function was changed into `user_fn` in `MPI_OP_CREATE`, into `comm_errhandler_fn` in `MPI_COMM_CREATE_ERRHANDLER`, into `win_errhandler_fn` in `MPI_WIN_CREATE_ERRHANDLER`, into `file_errhandler_fn` in `MPI_FILE_CREATE_ERRHANDLER`, into `handler_fn` in `MPI_ERRHANDLER_CREATE`. For consistency reasons, `INOUBUF` was changed into `INOUTBUF` in `MPI_REDUCE_LOCAL`, and `intracomm` into `newintracomm` in `MPI_INTERCOMM_MERGE`.
 10. Section 8.2 on page 328.
In Fortran with the `mpi` and `mpi_f08` modules, `MPI_ALLOC_MEM` now also supports `TYPE(C_PTR)` C-pointer instead of only returning an address-sized integer that may be usable together with non-standard Cray-pointer. The Fortran interfaces with `INTEGER(KIND=MPI_ADDRESS_KIND) BASEPTR` in the `mpi` module and the `mpif.h` include file are deprecated since MPI-3.0.
 11. Section 16.2.15 on page 584, and Section 16.2.7 on page 568.
Fortran `SEQUENCE` and `BIND(C)` derived application types can be used as buffers in MPI operations.
 12. Section 16.2.16 on page 586 to Section 16.2.19 on page 597, Section 16.2.7 on page 568, and Section 16.2.8 on page 569.
The sections about Fortran optimization problems and their solution is partially rewritten and new methods are added, e.g., the use of the `ASYNCHRONOUS` attribute. The constant `MPI_ASYNC_PROTECTS_NONBLOCKING` tells whether the meaning of the

ticket38.
ticket274.
ticket229.2.
ticket229.4.

ASYNCHRONOUS attribute is extended to protect nonblocking operations. The Fortran routine `MPI_F_SYNC_REG` is added. To achieve a secure and portable programming interfaces, in Section 16.2.7, several requirements are defined for the combination of an MPI library and a Fortran compiler to be MPI-3.0 compliant.

13. Section 16.2.4 on page 558.
The use of the `mpif.h` Fortran include file is strongly discouraged.
14. Section 16.2.3 on page 556, and Section 16.2.7 on page 568.
The existing `mpi` Fortran module must implement compile-time argument checking.
15. Section 16.2.2 on page 553.
Within the `mpi_08` Fortran module, dummy arguments are declared with `INTENT=IN`, `OUT`, or `INOUT` as defined in the `mpi_08` interfaces.
16. Section 16.2.7 on page 568.
This new section summarizes requirements that an MPI library together with a Fortran compiler is compliant to the MPI standard.
17. Section A.1.1, Table “*Predefined functions*” on page 626, Section A.1.3 on page 631, and Section A.3.4 on page 673.
Within the new `mpi_f08` module, all callback prototype definitions are defined with explicit interfaces `PROCEDURE(MPI_...)` with `BIND(C)` attribute.
18. Section A.1.3 on page 631.
In some routines, the Fortran callback prototype names were changed from `..._FN` to `..._FUNCTION` to be consistent with the other language bindings.

B.2 Changes from Version 2.1 to Version 2.2

1. Section 2.5.4 on page 15.
It is now guaranteed that predefined named constant handles (as other constants) can be used in initialization expressions or assignments, i.e., also before the call to `MPI_INIT`.
2. Section 2.6 on page 17, Section 2.6.4 on page 19, and Section 16.1 on page 539.
The C++ language bindings have been deprecated and may be removed in a future version of the MPI specification.
3. Section 3.2.2 on page 29.
`MPI_CHAR` for printable characters is now defined for C type char (instead of signed char). This change should not have any impact on applications nor on MPI libraries (except some comment lines), because printable characters could and can be stored in any of the C types char, signed char, and unsigned char, and `MPI_CHAR` is not allowed for predefined reduction operations.
4. Section 3.2.2 on page 29.
`MPI_(U)INT{8,16,32,64}_T`, `MPI_AINT`, `MPI_OFFSET`, `MPI_C_BOOL`, `MPI_C_COMPLEX`, `MPI_C_FLOAT_COMPLEX`, `MPI_C_DOUBLE_COMPLEX`, and `MPI_C_LONG_DOUBLE_COMPLEX` are now valid predefined MPI datatypes.

- [26] Michael Hennecke. A Fortran 90 interface to MPI version 1.1. Technical Report Internal Report 63/96, Rechenzentrum, Universität Karlsruhe, D-76128 Karlsruhe, Germany, June 1996. Available via world wide web from http://www.uni-karlsruhe.de/~Michael.Hennecke/Publications/#MPI_F90.
16.2.3
- [27] T. Hoefer, P. Gottschling, A. Lumsdaine, and W. Rehm. Optimizing a Conjugate Gradient Solver with Non-Blocking Collective Operations. *Elsevier Journal of Parallel Computing (PARCO)*, 33(9):624–633, Sep. 2007. 5.12
- [28] T. Hoefer and A. Lumsdaine. Message Progression in Parallel Computing - To Thread or not to Thread? In *Proceedings of the 2008 IEEE International Conference on Cluster Computing*. IEEE Computer Society, Oct. 2008. 5.12
- [29] T. Hoefer, A. Lumsdaine, and W. Rehm. Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In *Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07*. IEEE Computer Society/ACM, Nov. 2007. 5.12
- [30] T. Hoefer, M. Schellmann, S. Gorlatch, and A. Lumsdaine. Communication Optimization for Medical Image Reconstruction Algorithms. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 15th European PVM/MPI Users' Group Meeting*, volume LNCS 5205, pages 75–83. Springer, Sep. 2008. 5.12
- [31] Institute of Electrical and Electronics Engineers, New York. *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*, 1985. 13.5.2
- [32] International Organization for Standardization, Geneva, ISO 8859-1:1987. *Information processing — 8-bit single-byte coded graphic character sets — Part 1: Latin alphabet No. 1*, 1987. 13.5.2
- [33] International Organization for Standardization, Geneva, ISO/IEC 9945-1:1996(E). *Information technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language]*, December 1996. 12.4, 13.2.1
- [34] International Organization for Standardization, Geneva, ISO/IEC 1539-1:2010. *Information technology – Programming languages – Fortran – Part 1: Base language*, November 2010. 16.2.1, 16.2.2
- [35] International Organization for Standardization, ISO/IEC/SC22/WG5 (Fortran), Geneva, TR 29113, Draft N1869. *TR on further interoperability with C*, July, 18 2011. <http://www.nag.co.uk/sc22wg5/> and <ftp://ftp.nag.co.uk/sc22wg5/N1851-N1900/N1869.pdf>. 16.2.2
- [36] International Organization for Standardization, ISO/IEC/SC22/WG5 (Fortran), Geneva, TR 29113. *TR on further interoperability with C*, 2012. <http://www.nag.co.uk/sc22wg5/>. 16.2.1, 16.2.1, 16.2.2, 16.2.7, 3
- [37] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1993. 4.1.4

MPI: A Message-Passing Interface Standard

Version **3.0**

(Draft, with MPI 3 Nonblocking Collectives

and new Fortran 2008 Interface)

Unofficial, for comment only

Message Passing Interface Forum

December 16, 2011

ticket0.

ticket229-A.

	16.3.8 Extra State	614
	16.3.9 Constants	614
	16.3.10 Interlanguage Communication	615
	A Language Bindings Summary	617
	A.1 Defined Values and Handles	617
	A.1.1 Defined Constants	617
	A.1.2 Types	630
ticket0.	A.1.3 Prototype Definitions	631
	C Bindings	631
	Fortran 2008 Bindings with the mpi_f08 Module	632
	Fortran Bindings with mpif.h or the mpi Module	634
	C++ Bindings (deprecated)	637
ticket0.	A.1.4 Deprecated Prototype Definitions	637
ticket0.	A.1.5 Info Keys	638
	A.1.6 Info Values	639
	A.2 C Bindings	640
	A.2.1 Point-to-Point Communication C Bindings	640
	A.2.2 Datatypes C Bindings	641
	A.2.3 Collective Communication C Bindings	643
	A.2.4 Groups, Contexts, Communicators, and Caching C Bindings	646
	A.2.5 Process Topologies C Bindings	648
	A.2.6 MPI Environmental Management C Bindings	649
	A.2.7 The Info Object C Bindings	650
	A.2.8 Process Creation and Management C Bindings	651
	A.2.9 One-Sided Communications C Bindings	651
	A.2.10 External Interfaces C Bindings	652
	A.2.11 I/O C Bindings	652
	A.2.12 Language Bindings C Bindings	655
	A.2.13 Profiling Interface C Bindings	656
	A.2.14 Deprecated C Bindings	656
	A.3 Fortran 2008 Bindings with the mpi_f08 Module	657
	A.3.1 Point-to-Point Communication Fortran 2008 Bindings	657
	A.3.2 Datatypes Fortran 2008 Bindings	662
	A.3.3 Collective Communication Fortran 2008 Bindings	666
	A.3.4 Groups, Contexts, Communicators, and Caching Fortran 2008 Bindings	673
	A.3.5 Process Topologies Fortran 2008 Bindings	679
	A.3.6 MPI Environmental Management Fortran 2008 Bindings	682
	A.3.7 The Info Object Fortran 2008 Bindings	685
	A.3.8 Process Creation and Management Fortran 2008 Bindings	686
	A.3.9 One-Sided Communications Fortran 2008 Bindings	687
	A.3.10 External Interfaces Fortran 2008 Bindings	689
	A.3.11 I/O Fortran 2008 Bindings	690
	A.3.12 Language Bindings Fortran 2008 Bindings	697
	A.3.13 Profiling Interface Fortran 2008 Bindings	698
	A.3.14 Deprecated Fortran 2008 Bindings	698
	A.4 Fortran Bindings with mpif.h or the mpi Module	699
	A.4.1 Point-to-Point Communication Fortran Bindings	699

A.3 Fortran 2008 Bindings with the mpi_f08 Module

A.3.1 Point-to-Point Communication Fortran 2008 Bindings

```

MPI_Bsend(buf, count, datatype, dest, tag, comm, ierror) BIND(C)
  TYPE(*), DIMENSION(..), INTENT(IN) :: buf
  INTEGER, INTENT(IN) :: count, dest, tag
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Bsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
  BIND(C)
  TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
  INTEGER, INTENT(IN) :: count, dest, tag
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Comm), INTENT(IN) :: comm
  TYPE(MPI_Request), INTENT(OUT) :: request
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Buffer_attach(buffer, size, ierror) BIND(C)
  TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buffer
  INTEGER, INTENT(IN) :: size
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Buffer_detach(buffer_addr, size, ierror) BIND(C)
  USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
  TYPE(C_PTR), INTENT(OUT) :: buffer_addr
  INTEGER, INTENT(OUT) :: size
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Cancel(request, ierror) BIND(C)
  TYPE(MPI_Request), INTENT(IN) :: request
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Get_count(status, datatype, count, ierror) BIND(C)
  TYPE(MPI_Status), INTENT(IN) :: status
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  INTEGER, INTENT(OUT) :: count
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Ibsend(buf, count, datatype, dest, tag, comm, request, ierror) BIND(C)
  TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
  INTEGER, INTENT(IN) :: count, dest, tag
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Comm), INTENT(IN) :: comm
  TYPE(MPI_Request), INTENT(OUT) :: request
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

1  MPI_Iprobe(source, tag, comm, flag, status, ierror) BIND(C)
2      INTEGER, INTENT(IN) :: source, tag
3      TYPE(MPI_Comm), INTENT(IN) :: comm
4      LOGICAL, INTENT(OUT) :: flag
5      TYPE(MPI_Status) :: status
6      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 7
8  MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror) BIND(C)
9      TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
10     INTEGER, INTENT(IN) :: count, source, tag
11     TYPE(MPI_Datatype), INTENT(IN) :: datatype
12     TYPE(MPI_Comm), INTENT(IN) :: comm
13     TYPE(MPI_Request), INTENT(OUT) :: request
14     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 14
15  MPI_Irsend(buf, count, datatype, dest, tag, comm, request, ierror) BIND(C)
16     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
17     INTEGER, INTENT(IN) :: count, dest, tag
18     TYPE(MPI_Datatype), INTENT(IN) :: datatype
19     TYPE(MPI_Comm), INTENT(IN) :: comm
20     TYPE(MPI_Request), INTENT(OUT) :: request
21     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 21
22  MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierror) BIND(C)
23     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
24     INTEGER, INTENT(IN) :: count, dest, tag
25     TYPE(MPI_Datatype), INTENT(IN) :: datatype
26     TYPE(MPI_Comm), INTENT(IN) :: comm
27     TYPE(MPI_Request), INTENT(OUT) :: request
28     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 28
29  MPI_Issend(buf, count, datatype, dest, tag, comm, request, ierror) BIND(C)
30     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
31     INTEGER, INTENT(IN) :: count, dest, tag
32     TYPE(MPI_Datatype), INTENT(IN) :: datatype
33     TYPE(MPI_Comm), INTENT(IN) :: comm
34     TYPE(MPI_Request), INTENT(OUT) :: request
35     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 35
36  MPI_Probe(source, tag, comm, status, ierror) BIND(C)
37     INTEGER, INTENT(IN) :: source, tag
38     TYPE(MPI_Comm), INTENT(IN) :: comm
39     TYPE(MPI_Status) :: status
40     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 40
41  MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror) BIND(C)
42     TYPE(*), DIMENSION(..) :: buf
43     INTEGER, INTENT(IN) :: count, source, tag
44     TYPE(MPI_Datatype), INTENT(IN) :: datatype
45     TYPE(MPI_Comm), INTENT(IN) :: comm
46
47
48

```

```

TYPE(MPI_Status) :: status
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror)
    BIND(C)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count, source, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Request_free(request, ierror) BIND(C)
    TYPE(MPI_Request), INTENT(INOUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Request_get_status(request, flag, status, ierror) BIND(C)
    TYPE(MPI_Request), INTENT(IN) :: request
    LOGICAL, INTENT(OUT) :: flag
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Rsend(buf, count, datatype, dest, tag, comm, ierror) BIND(C)
    TYPE(*), DIMENSION(..), INTENT(IN) :: buf
    INTEGER, INTENT(IN) :: count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Rsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
    BIND(C)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Send(buf, count, datatype, dest, tag, comm, ierror) BIND(C)
    TYPE(*), DIMENSION(..), INTENT(IN) :: buf
    INTEGER, INTENT(IN) :: count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Send_init(buf, count, datatype, dest, tag, comm, request, ierror)
    BIND(C)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm

```

1
2
3 ticket-248T.
4
5
6
7
8
9
10
11 ticket-248T.
12
13
14
15 ticket-248T.
16
17
18
19
20 ticket-248T.
21
22
23
24
25
26
27 ticket-248T.
28
29
30
31
32
33
34
35 ticket-248T.
36
37
38
39
40
41
42 ticket-248T.
43
44
45
46
47
48

```

1      TYPE(MPI_Request), INTENT(OUT) :: request
2      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 3
4      MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
5                  recvcount, recvtype, source, recvtag, comm, status, ierror)
6                  BIND(C)
7      TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
8      TYPE(*), DIMENSION(..) :: recvbuf
9      INTEGER, INTENT(IN) :: sendcount, dest, sendtag, recvcount, source,
10     recvtag
11     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
12     TYPE(MPI_Comm), INTENT(IN) :: comm
13     TYPE(MPI_Status) :: status
14     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 15
16     MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,
17                          comm, status, ierror) BIND(C)
18     TYPE(*), DIMENSION(..) :: buf
19     INTEGER, INTENT(IN) :: count, dest, sendtag, source, recvtag
20     TYPE(MPI_Datatype), INTENT(IN) :: datatype
21     TYPE(MPI_Comm), INTENT(IN) :: comm
22     TYPE(MPI_Status) :: status
23     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 24
25     MPI_Ssend(buf, count, datatype, dest, tag, comm, ierror) BIND(C)
26     TYPE(*), DIMENSION(..), INTENT(IN) :: buf
27     INTEGER, INTENT(IN) :: count, dest, tag
28     TYPE(MPI_Datatype), INTENT(IN) :: datatype
29     TYPE(MPI_Comm), INTENT(IN) :: comm
30     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 31
32     MPI_Ssend_init(buf, count, datatype, dest, tag, comm, request, ierror)
33     BIND(C)
34     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
35     INTEGER, INTENT(IN) :: count, dest, tag
36     TYPE(MPI_Datatype), INTENT(IN) :: datatype
37     TYPE(MPI_Comm), INTENT(IN) :: comm
38     TYPE(MPI_Request), INTENT(OUT) :: request
39     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 40
41     MPI_Start(request, ierror) BIND(C)
42     TYPE(MPI_Request), INTENT(INOUT) :: request
43     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 44
45     MPI_Startall(count, array_of_requests, ierror) BIND(C)
46     INTEGER, INTENT(IN) :: count
47     TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
48     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 49
50     MPI_Test(request, flag, status, ierror) BIND(C)
51     TYPE(MPI_Request), INTENT(INOUT) :: request

```

```

LOGICAL, INTENT(OUT) :: flag
TYPE(MPI_Status) :: status
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Test_cancelled(status, flag, ierror) BIND(C)
TYPE(MPI_Status), INTENT(IN) :: status
LOGICAL, INTENT(OUT) :: flag
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Testall(count, array_of_requests, flag, array_of_statuses, ierror)
    BIND(C)
INTEGER, INTENT(IN) :: count
TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
LOGICAL, INTENT(OUT) :: flag
TYPE(MPI_Status) :: array_of_statuses(*)
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Testany(count, array_of_requests, index, flag, status, ierror) BIND(C)
INTEGER, INTENT(IN) :: count
TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
INTEGER, INTENT(OUT) :: index
LOGICAL, INTENT(OUT) :: flag
TYPE(MPI_Status) :: status
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Testsome(incount, array_of_requests, outcount, array_of_indices,
    array_of_statuses, ierror) BIND(C)
INTEGER, INTENT(IN) :: incount
TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(incount)
INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)
TYPE(MPI_Status) :: array_of_statuses(*)
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Wait(request, status, ierror) BIND(C)
TYPE(MPI_Request), INTENT(INOUT) :: request
TYPE(MPI_Status) :: status
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Waitall(count, array_of_requests, array_of_statuses, ierror) BIND(C)
INTEGER, INTENT(IN) :: count
TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
TYPE(MPI_Status) :: array_of_statuses(*)
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Waitany(count, array_of_requests, index, status, ierror) BIND(C)
INTEGER, INTENT(IN) :: count
TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
INTEGER, INTENT(OUT) :: index
TYPE(MPI_Status) :: status
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

1
2
3
4 ticket-248T.
5
6
7
8 ticket-248T.
9
10
11
12
13
14
15
16 ticket-248T.
17
18
19
20
21
22
23 ticket-248T.
24
25
26
27
28
29
30
31 ticket-248T.
32
33
34
35 ticket-248T.
36
37
38
39
40
41 ticket-248T.
42
43
44
45
46
47 ticket-248T.
48

```

1  MPI_Waitsome(incount, array_of_requests, outcount, array_of_indices,
2              array_of_statuses, ierror) BIND(C)
3      INTEGER, INTENT(IN) :: incount
4      TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(incount)
5      INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)
6      TYPE(MPI_Status) :: array_of_statuses(*)
7      INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

A.3.2 Datatypes Fortran 2008 Bindings

```

11 MPI_Get_address(location, address, ierror) BIND(C)
12     TYPE(*), DIMENSION(..) :: location
13     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: address
14     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

16 MPI_Get_elements(status, datatype, count, ierror) BIND(C)
17     TYPE(MPI_Status), INTENT(IN) :: status
18     TYPE(MPI_Datatype), INTENT(IN) :: datatype
19     INTEGER, INTENT(OUT) :: count
20     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

21 MPI_Pack(inbuf, incount, datatype, outbuf, outsize, position, comm, ierror)
22     BIND(C)
23     TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf
24     TYPE(*), DIMENSION(..) :: outbuf
25     INTEGER, INTENT(IN) :: incount, outsize
26     TYPE(MPI_Datatype), INTENT(IN) :: datatype
27     INTEGER, INTENT(INOUT) :: position
28     TYPE(MPI_Comm), INTENT(IN) :: comm
29     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

31 MPI_Pack_external(datarep, inbuf, incount, datatype, outbuf, outsize,
32                  position, ierror) BIND(C)
33     CHARACTER(LEN=*), INTENT(IN) :: datarep
34     TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf
35     TYPE(*), DIMENSION(..) :: outbuf
36     INTEGER, INTENT(IN) :: incount
37     TYPE(MPI_Datatype), INTENT(IN) :: datatype
38     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: outsize
39     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position
40     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

41 MPI_Pack_external_size(datarep, incount, datatype, size, ierror) BIND(C)
42     TYPE(MPI_Datatype), INTENT(IN) :: datatype
43     INTEGER, INTENT(IN) :: incount
44     CHARACTER(LEN=*), INTENT(IN) :: datarep
45     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size
46     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

48 MPI_Pack_size(incount, datatype, comm, size, ierror) BIND(C)

```



```

INTEGER, INTENT(IN) :: incount
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, INTENT(OUT) :: size
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_commit(datatype, ierror) BIND(C)
TYPE(MPI_Datatype), INTENT(INOUT) :: datatype
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_contiguous(count, oldtype, newtype, ierror) BIND(C)
INTEGER, INTENT(IN) :: count
TYPE(MPI_Datatype), INTENT(IN) :: oldtype
TYPE(MPI_Datatype), INTENT(OUT) :: newtype
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_create_darray(size, rank, ndims, array_of_gsizes,
                      array_of_distribs, array_of_dargs, array_of_psize, order,
                      oldtype, newtype, ierror) BIND(C)
INTEGER, INTENT(IN) :: size, rank, ndims, array_of_gsizes(ndims),
array_of_distribs(ndims), array_of_dargs(ndims),
array_of_psize(ndims), order
TYPE(MPI_Datatype), INTENT(IN) :: oldtype
TYPE(MPI_Datatype), INTENT(OUT) :: newtype
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_create_hindexed(count, array_of_blocklengths,
                        array_of_displacements, oldtype, newtype, ierror) BIND(C)
INTEGER, INTENT(IN) :: count, array_of_blocklengths(count)
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
array_of_displacements(count)
TYPE(MPI_Datatype), INTENT(IN) :: oldtype
TYPE(MPI_Datatype), INTENT(OUT) :: newtype
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_create_hvector(count, blocklength, stride, oldtype, newtype,
                       ierror) BIND(C)
INTEGER, INTENT(IN) :: count, blocklength
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: stride
TYPE(MPI_Datatype), INTENT(IN) :: oldtype
TYPE(MPI_Datatype), INTENT(OUT) :: newtype
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_create_indexed_block(count, blocklength, array_of_displacements,
                             oldtype, newtype, ierror) BIND(C)
INTEGER, INTENT(IN) :: count, blocklength,
array_of_displacements(count)
TYPE(MPI_Datatype), INTENT(IN) :: oldtype
TYPE(MPI_Datatype), INTENT(OUT) :: newtype
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

ticket-248T.

ticket-248T.

ticket-248T.

ticket-248T.

ticket-248T.

ticket-248T.

ticket-248T.

```

1  MPI_Type_create_resized(oldtype, lb, extent, newtype, ierror) BIND(C)
2      INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: lb, extent
3      TYPE(MPI_Datatype), INTENT(IN) :: oldtype
4      TYPE(MPI_Datatype), INTENT(OUT) :: newtype
5      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 6
7  MPI_Type_create_struct(count, array_of_blocklengths,
8      array_of_displacements, array_of_types, newtype, ierror)
9      BIND(C)
10     INTEGER, INTENT(IN) :: count, array_of_blocklengths(count)
11     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
12     array_of_displacements(count)
13     TYPE(MPI_Datatype), INTENT(IN) :: array_of_types(count)
14     TYPE(MPI_Datatype), INTENT(OUT) :: newtype
15     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 16
17 MPI_Type_create_subarray(ndims, array_of_sizes, array_of_subsizes,
18     array_of_starts, order, oldtype, newtype, ierror) BIND(C)
19     INTEGER, INTENT(IN) :: ndims, array_of_sizes(ndims),
20     array_of_subsizes(ndims), array_of_starts(ndims), order
21     TYPE(MPI_Datatype), INTENT(IN) :: oldtype
22     TYPE(MPI_Datatype), INTENT(OUT) :: newtype
23     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 24
25 MPI_Type_dup(oldtype, newtype, ierror) BIND(C)
26     TYPE(MPI_Datatype), INTENT(IN) :: oldtype
27     TYPE(MPI_Datatype), INTENT(OUT) :: newtype
28     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 29
30 MPI_Type_free(datatype, ierror) BIND(C)
31     TYPE(MPI_Datatype), INTENT(INOUT) :: datatype
32     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 33
34 MPI_Type_get_contents(datatype, max_integers, max_addresses, max_datatypes,
35     array_of_integers, array_of_addresses, array_of_datatypes,
36     ierror) BIND(C)
37     TYPE(MPI_Datatype), INTENT(IN) :: datatype
38     INTEGER, INTENT(IN) :: max_integers, max_addresses, max_datatypes
39     INTEGER, INTENT(OUT) :: array_of_integers(max_integers)
40     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::
41     array_of_addresses(max_addresses)
42     TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes)
43     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 44
45 MPI_Type_get_envelope(datatype, num_integers, num_addresses, num_datatypes,
46     combiner, ierror) BIND(C)
47     TYPE(MPI_Datatype), INTENT(IN) :: datatype
48     INTEGER, INTENT(OUT) :: num_integers, num_addresses, num_datatypes,
49     combiner
50     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 51

```

```

MPI_Type_get_extent(datatype, lb, extent, ierror) BIND(C)
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: lb, extent
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_get_true_extent(datatype, true_lb, true_extent, ierror) BIND(C)
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: true_lb, true_extent
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_indexed(count, array_of_blocklengths, array_of_displacements,
  oldtype, newtype, ierror) BIND(C)
  INTEGER, INTENT(IN) :: count, array_of_blocklengths(count),
  array_of_displacements(count)
  TYPE(MPI_Datatype), INTENT(IN) :: oldtype
  TYPE(MPI_Datatype), INTENT(OUT) :: newtype
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_size(datatype, size, ierror) BIND(C)
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  INTEGER, INTENT(OUT) :: size
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_vector(count, blocklength, stride, oldtype, newtype, ierror)
  BIND(C)
  INTEGER, INTENT(IN) :: count, blocklength, stride
  TYPE(MPI_Datatype), INTENT(IN) :: oldtype
  TYPE(MPI_Datatype), INTENT(OUT) :: newtype
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Unpack(inbuf, insize, position, outbuf, outcount, datatype, comm,
  ierror) BIND(C)
  TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf
  TYPE(*), DIMENSION(..) :: outbuf
  INTEGER, INTENT(IN) :: insize, outcount
  INTEGER, INTENT(INOUT) :: position
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Unpack_external(datarep, inbuf, insize, position, outbuf, outcount,
  datatype, ierror) BIND(C)
  CHARACTER(LEN=*), INTENT(IN) :: datarep
  TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf
  TYPE(*), DIMENSION(..) :: outbuf
  INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: insize
  INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position
  INTEGER, INTENT(IN) :: outcount
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

A.3.3 Collective Communication Fortran 2008 Bindings

```

1  MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
2      comm, ierror) BIND(C)
3      TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
4      TYPE(*), DIMENSION(..) :: recvbuf
5      INTEGER, INTENT(IN) :: sendcount, recvcount
6      TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
7      TYPE(MPI_Comm), INTENT(IN) :: comm
8      INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

10 MPI_Allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
11     recvtype, comm, ierror) BIND(C)
12     TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
13     TYPE(*), DIMENSION(..) :: recvbuf
14     INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*)
15     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
16     TYPE(MPI_Comm), INTENT(IN) :: comm
17     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

19 MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror) BIND(C)
20     TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
21     TYPE(*), DIMENSION(..) :: recvbuf
22     INTEGER, INTENT(IN) :: count
23     TYPE(MPI_Datatype), INTENT(IN) :: datatype
24     TYPE(MPI_Op), INTENT(IN) :: op
25     TYPE(MPI_Comm), INTENT(IN) :: comm
26     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

28 MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
29     comm, ierror) BIND(C)
30     TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
31     TYPE(*), DIMENSION(..) :: recvbuf
32     INTEGER, INTENT(IN) :: sendcount, recvcount
33     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
34     TYPE(MPI_Comm), INTENT(IN) :: comm
35     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

36 MPI_Alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
37     rdispls, recvtype, comm, ierror) BIND(C)
38     TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
39     TYPE(*), DIMENSION(..) :: recvbuf
40     INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
41     rdispls(*)
42     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
43     TYPE(MPI_Comm), INTENT(IN) :: comm
44     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

46 MPI_Alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts,
47     rdispls, recvtypes, comm, ierror) BIND(C)

```

```

TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcoun
counts(*), rdispls(*)
TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*)
TYPE(MPI_Datatype), INTENT(IN) :: recvtypes(*)
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Barrier(comm, ierror) BIND(C)
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Bcast(buffer, count, datatype, root, comm, ierror) BIND(C)
TYPE(*), DIMENSION(..) :: buffer
INTEGER, INTENT(IN) :: count, root
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Exscan(sendbuf, recvbuf, count, datatype, op, comm, ierror) BIND(C)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: count
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Op), INTENT(IN) :: op
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcoun
t, recvttype, root, comm, ierror) BIND(C)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: sendcount, recvcoun
t, root
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvttype
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcoun
ts, displs, recvttype, root, comm, ierror) BIND(C)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: sendcount, recvcoun
ts(*), displs(*), root
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvttype
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Iallgather(sendbuf, sendcount, sendtype, recvbuf, recvcoun
t, recvttype, comm, request, ierror) BIND(C)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf

```

```

1      TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
2      INTEGER, INTENT(IN) :: sendcount, recvcount
3      TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
4      TYPE(MPI_Comm), INTENT(IN) :: comm
5      TYPE(MPI_Request), INTENT(OUT) :: request
6      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 7
8      MPI_Iallgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
9                  recvtype, comm, request, ierror) BIND(C)
10     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
11     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
12     INTEGER, INTENT(IN) :: sendcount
13     INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
14     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
15     TYPE(MPI_Comm), INTENT(IN) :: comm
16     TYPE(MPI_Request), INTENT(OUT) :: request
17     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 17
18     MPI_Iallreduce(sendbuf, recvbuf, count, datatype, op, comm, request,
19                  ierror) BIND(C)
20     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
21     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
22     INTEGER, INTENT(IN) :: count
23     TYPE(MPI_Datatype), INTENT(IN) :: datatype
24     TYPE(MPI_Op), INTENT(IN) :: op
25     TYPE(MPI_Comm), INTENT(IN) :: comm
26     TYPE(MPI_Request), INTENT(OUT) :: request
27     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 27
28
29     MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
30                  comm, request, ierror) BIND(C)
31     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
32     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
33     INTEGER, INTENT(IN) :: sendcount, recvcount
34     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
35     TYPE(MPI_Comm), INTENT(IN) :: comm
36     TYPE(MPI_Request), INTENT(OUT) :: request
37     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 37
38
39     MPI_Ialltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
40                  rdispls, recvtype, comm, request, ierror) BIND(C)
41     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
42     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
43     INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
44     recvcounts(*), rdispls(*)
45     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
46     TYPE(MPI_Comm), INTENT(IN) :: comm
47     TYPE(MPI_Request), INTENT(OUT) :: request
48     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 48

```

```

MPI_Ialltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
               recvcounts, rdispls, recvtypes, comm, request, ierror) BIND(C)
  TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
  TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
  INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
  recvcounts(*), rdispls(*)
  TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
  recvtypes(*)
  TYPE(MPI_Comm), INTENT(IN) :: comm
  TYPE(MPI_Request), INTENT(OUT) :: request
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

1
2
3
4
5
6
7
8
9
10
11
12 ticket-248T.

```

MPI_Ibarrier(comm, request, ierror) BIND(C)
  TYPE(MPI_Comm), INTENT(IN) :: comm
  TYPE(MPI_Request), INTENT(OUT) :: request
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

13
14
15
16 ticket-248T.

```

MPI_Ibcast(buffer, count, datatype, root, comm, request, ierror) BIND(C)
  TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buffer
  INTEGER, INTENT(IN) :: count, root
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Comm), INTENT(IN) :: comm
  TYPE(MPI_Request), INTENT(OUT) :: request
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

17
18
19
20
21
22
23
24 ticket-248T

```

MPI_Iexscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
  BIND(C)
  TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
  TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
  INTEGER, INTENT(IN) :: count
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Op), INTENT(IN) :: op
  TYPE(MPI_Comm), INTENT(IN) :: comm
  TYPE(MPI_Request), INTENT(OUT) :: request
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

25
26
27
28
29
30
31
32
33
34 ticket-248T.

```

MPI_Igather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
            root, comm, request, ierror) BIND(C)
  TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
  TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
  INTEGER, INTENT(IN) :: sendcount, recvcount, root
  TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
  TYPE(MPI_Comm), INTENT(IN) :: comm
  TYPE(MPI_Request), INTENT(OUT) :: request
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

35
36
37
38
39
40
41
42
43
44 ticket-248T.

```

MPI_Igatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
             recvtype, root, comm, request, ierror) BIND(C)
  TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
  TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf

```

45
46
47
48

```

1      INTEGER, INTENT(IN) :: sendcount, root
2      INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
3      TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
4      TYPE(MPI_Comm), INTENT(IN) :: comm
5      TYPE(MPI_Request), INTENT(OUT) :: request
6      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 7
8      MPI_Ireduce(sendbuf, recvbuf, count, datatype, op, root, comm, request,
9                ierror) BIND(C)
10     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
11     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
12     INTEGER, INTENT(IN) :: count, root
13     TYPE(MPI_Datatype), INTENT(IN) :: datatype
14     TYPE(MPI_Op), INTENT(IN) :: op
15     TYPE(MPI_Comm), INTENT(IN) :: comm
16     TYPE(MPI_Request), INTENT(OUT) :: request
17     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 17
18     MPI_Ireduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
19                      request, ierror) BIND(C)
20     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
21     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
22     INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
23     TYPE(MPI_Datatype), INTENT(IN) :: datatype
24     TYPE(MPI_Op), INTENT(IN) :: op
25     TYPE(MPI_Comm), INTENT(IN) :: comm
26     TYPE(MPI_Request), INTENT(OUT) :: request
27     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 27
28
29     MPI_Ireduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
30                             request, ierror) BIND(C)
31     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
32     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
33     INTEGER, INTENT(IN) :: recvcount
34     TYPE(MPI_Datatype), INTENT(IN) :: datatype
35     TYPE(MPI_Op), INTENT(IN) :: op
36     TYPE(MPI_Comm), INTENT(IN) :: comm
37     TYPE(MPI_Request), INTENT(OUT) :: request
38     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 37
39     MPI_Iscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
40             BIND(C)
41     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
42     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
43     INTEGER, INTENT(IN) :: count
44     TYPE(MPI_Datatype), INTENT(IN) :: datatype
45     TYPE(MPI_Op), INTENT(IN) :: op
46     TYPE(MPI_Comm), INTENT(IN) :: comm
47     TYPE(MPI_Request), INTENT(OUT) :: request
48

```



```

    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Isscatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
              root, comm, request, ierror) BIND(C)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Isscatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
               recvtype, root, comm, request, ierror) BIND(C)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), displs(*)
    INTEGER, INTENT(IN) :: recvcount, root
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Op_commutative(op, commute, ierror) BIND(C)
    TYPE(MPI_Op), INTENT(IN) :: op
    LOGICAL, INTENT(OUT) :: commute
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Op_create(user_fn, commute, op, ierror) BIND(C)
    PROCEDURE(MPI_User_function) :: user_fn
    LOGICAL, INTENT(IN) :: commute
    TYPE(MPI_Op), INTENT(OUT) :: op
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Op_free(op, ierror) BIND(C)
    TYPE(MPI_Op), INTENT(INOUT) :: op
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
    BIND(C)
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
    TYPE(*), DIMENSION(..) :: recvbuf
    INTEGER, INTENT(IN) :: count, root
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Reduce_local(inbuf, inoutbuf, count, datatype, op, ierror) BIND(C)
    TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf
    TYPE(*), DIMENSION(..) :: inoutbuf

```

1
2 ticket-248T.
3
4
5
6
7
8
9
10
11 ticket-248T.
12
13
14
15
16
17
18
19
20
21
22 ticket-248T.
23
24
25
26 ticket-248T.
27
28
29
30
31
32 ticket-248T.
33
34
35 ticket-248T.
36
37
38
39
40
41
42
43
44
45 ticket-248T.
46
47
48

```

1      INTEGER, INTENT(IN) :: count
2      TYPE(MPI_Datatype), INTENT(IN) :: datatype
3      TYPE(MPI_Op), INTENT(IN) :: op
4      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 5
6      MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
7          ierror) BIND(C)
8      TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
9      TYPE(*), DIMENSION(..) :: recvbuf
10     INTEGER, INTENT(IN) :: recvcounts(*)
11     TYPE(MPI_Datatype), INTENT(IN) :: datatype
12     TYPE(MPI_Op), INTENT(IN) :: op
13     TYPE(MPI_Comm), INTENT(IN) :: comm
14     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 15
16     MPI_Reduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
17         ierror) BIND(C)
18     TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
19     TYPE(*), DIMENSION(..) :: recvbuf
20     INTEGER, INTENT(IN) :: recvcount
21     TYPE(MPI_Datatype), INTENT(IN) :: datatype
22     TYPE(MPI_Op), INTENT(IN) :: op
23     TYPE(MPI_Comm), INTENT(IN) :: comm
24     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 25
26     MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm, ierror) BIND(C)
27     TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
28     TYPE(*), DIMENSION(..) :: recvbuf
29     INTEGER, INTENT(IN) :: count
30     TYPE(MPI_Datatype), INTENT(IN) :: datatype
31     TYPE(MPI_Op), INTENT(IN) :: op
32     TYPE(MPI_Comm), INTENT(IN) :: comm
33     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 34
35     MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
36         root, comm, ierror) BIND(C)
37     TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
38     TYPE(*), DIMENSION(..) :: recvbuf
39     INTEGER, INTENT(IN) :: sendcount, recvcount, root
40     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
41     TYPE(MPI_Comm), INTENT(IN) :: comm
42     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 43
44     MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
45         recvtype, root, comm, ierror) BIND(C)
46     TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
47     TYPE(*), DIMENSION(..) :: recvbuf
48     INTEGER, INTENT(IN) :: sendcounts(*), displs(*), recvcount, root
49     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
50     TYPE(MPI_Comm), INTENT(IN) :: comm

```

```
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

A.3.4 Groups, Contexts, Communicators, and Caching Fortran 2008 Bindings

```
MPI_COMM_DUP_FN(oldcomm, comm_keyval, extra_state, attribute_val_in,
    attribute_val_out, flag, ierror) BIND(C)
TYPE(MPI_Comm), INTENT(IN) :: oldcomm
INTEGER, INTENT(IN) :: comm_keyval
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state,
attribute_val_in
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val_out
LOGICAL, INTENT(OUT) :: flag
INTEGER, INTENT(OUT) :: ierror

MPI_COMM_NULL_COPY_FN(oldcomm, comm_keyval, extra_state, attribute_val_in,
    attribute_val_out, flag, ierror) BIND(C)
TYPE(MPI_Comm), INTENT(IN) :: oldcomm
INTEGER, INTENT(IN) :: comm_keyval
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state,
attribute_val_in
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val_out
LOGICAL, INTENT(OUT) :: flag
INTEGER, INTENT(OUT) :: ierror

MPI_COMM_NULL_DELETE_FN(comm, comm_keyval, attribute_val, extra_state,
    ierror) BIND(C)
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, INTENT(IN) :: comm_keyval
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val,
extra_state
INTEGER, INTENT(OUT) :: ierror

MPI_Comm_compare(comm1, comm2, result, ierror) BIND(C)
TYPE(MPI_Comm), INTENT(IN) :: comm1, comm2
INTEGER, INTENT(OUT) :: result
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_create(comm, group, newcomm, ierror) BIND(C)
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Group), INTENT(IN) :: group
TYPE(MPI_Comm), INTENT(OUT) :: newcomm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_create_keyval(comm_copy_attr_fn, comm_delete_attr_fn, comm_keyval,
    extra_state, ierror) BIND(C)
PROCEDURE(MPI_Comm_copy_attr_function) :: comm_copy_attr_fn
PROCEDURE(MPI_Comm_delete_attr_function) :: comm_delete_attr_fn
INTEGER, INTENT(OUT) :: comm_keyval
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```

1
ticket-248T. 2 MPI_Comm_delete_attr(comm, comm_keyval, ierror) BIND(C)
3 TYPE(MPI_Comm), INTENT(IN) :: comm
4 INTEGER, INTENT(IN) :: comm_keyval
5 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 6 MPI_Comm_dup(comm, newcomm, ierror) BIND(C)
7 TYPE(MPI_Comm), INTENT(IN) :: comm
8 TYPE(MPI_Comm), INTENT(OUT) :: newcomm
9 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 10 MPI_Comm_free(comm, ierror) BIND(C)
11 TYPE(MPI_Comm), INTENT(INOUT) :: comm
12 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 13 MPI_Comm_free_keyval(comm_keyval, ierror) BIND(C)
14 INTEGER, INTENT(INOUT) :: comm_keyval
15 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 16 MPI_Comm_get_attr(comm, comm_keyval, attribute_val, flag, ierror) BIND(C)
17 TYPE(MPI_Comm), INTENT(IN) :: comm
18 INTEGER, INTENT(IN) :: comm_keyval
19 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
20 LOGICAL, INTENT(OUT) :: flag
21 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 22 MPI_Comm_get_name(comm, comm_name, resultlen, ierror) BIND(C)
23 TYPE(MPI_Comm), INTENT(IN) :: comm
24 CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: comm_name
25 INTEGER, INTENT(OUT) :: resultlen
26 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 27 MPI_Comm_group(comm, group, ierror) BIND(C)
28 TYPE(MPI_Comm), INTENT(IN) :: comm
29 TYPE(MPI_Group), INTENT(OUT) :: group
30 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 31 MPI_Comm_rank(comm, rank, ierror) BIND(C)
32 TYPE(MPI_Comm), INTENT(IN) :: comm
33 INTEGER, INTENT(OUT) :: rank
34 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 35 MPI_Comm_remote_group(comm, group, ierror) BIND(C)
36 TYPE(MPI_Comm), INTENT(IN) :: comm
37 TYPE(MPI_Group), INTENT(OUT) :: group
38 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 39 MPI_Comm_remote_size(comm, size, ierror) BIND(C)
40 TYPE(MPI_Comm), INTENT(IN) :: comm
41 INTEGER, INTENT(OUT) :: size
42 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 43
44
45
46
47
48

```

```

MPI_Comm_set_attr(comm, comm_keyval, attribute_val, ierror) BIND(C)
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, INTENT(IN) :: comm_keyval
  INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_set_name(comm, comm_name, ierror) BIND(C)
  TYPE(MPI_Comm), INTENT(IN) :: comm
  CHARACTER(LEN=*), INTENT(IN) :: comm_name
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_size(comm, size, ierror) BIND(C)
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, INTENT(OUT) :: size
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_split(comm, color, key, newcomm, ierror) BIND(C)
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, INTENT(IN) :: color, key
  TYPE(MPI_Comm), INTENT(OUT) :: newcomm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_test_inter(comm, flag, ierror) BIND(C)
  TYPE(MPI_Comm), INTENT(IN) :: comm
  LOGICAL, INTENT(OUT) :: flag
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Group_compare(group1, group2, result, ierror) BIND(C)
  TYPE(MPI_Group), INTENT(IN) :: group1, group2
  INTEGER, INTENT(OUT) :: result
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Group_difference(group1, group2, newgroup, ierror) BIND(C)
  TYPE(MPI_Group), INTENT(IN) :: group1, group2
  TYPE(MPI_Group), INTENT(OUT) :: newgroup
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Group_excl(group, n, ranks, newgroup, ierror) BIND(C)
  TYPE(MPI_Group), INTENT(IN) :: group
  INTEGER, INTENT(IN) :: n, ranks(n)
  TYPE(MPI_Group), INTENT(OUT) :: newgroup
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Group_free(group, ierror) BIND(C)
  TYPE(MPI_Group), INTENT(INOUT) :: group
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Group_incl(group, n, ranks, newgroup, ierror) BIND(C)
  TYPE(MPI_Group), INTENT(IN) :: group
  INTEGER, INTENT(IN) :: n, ranks(n)
  TYPE(MPI_Group), INTENT(OUT) :: newgroup
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

ticket-248T.

ticket-248T.

ticket-248T.

ticket-248T.

ticket-248T.

ticket-248T.

ticket-248T.

ticket-248T.

ticket-248T.

```

1
ticket-248T. 2 MPI_Group_intersection(group1, group2, newgroup, ierror) BIND(C)
3 TYPE(MPI_Group), INTENT(IN) :: group1, group2
4 TYPE(MPI_Group), INTENT(OUT) :: newgroup
5 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 6
7 MPI_Group_range_excl(group, n, ranges, newgroup, ierror) BIND(C)
8 TYPE(MPI_Group), INTENT(IN) :: group
9 INTEGER, INTENT(IN) :: n, ranges(3,n)
10 TYPE(MPI_Group), INTENT(OUT) :: newgroup
11 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 12
13 MPI_Group_range_incl(group, n, ranges, newgroup, ierror) BIND(C)
14 TYPE(MPI_Group), INTENT(IN) :: group
15 INTEGER, INTENT(IN) :: n, ranges(3,n)
16 TYPE(MPI_Group), INTENT(OUT) :: newgroup
17 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 18
19 MPI_Group_rank(group, rank, ierror) BIND(C)
20 TYPE(MPI_Group), INTENT(IN) :: group
21 INTEGER, INTENT(OUT) :: rank
22 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 23
24 MPI_Group_size(group, size, ierror) BIND(C)
25 TYPE(MPI_Group), INTENT(IN) :: group
26 INTEGER, INTENT(OUT) :: size
27 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 28
29 MPI_Group_translate_ranks(group1, n, ranks1, group2, ranks2, ierror)
30 BIND(C)
31 TYPE(MPI_Group), INTENT(IN) :: group1, group2
32 INTEGER, INTENT(IN) :: n, ranks1(n)
33 INTEGER, INTENT(OUT) :: ranks2(n)
34 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 35
36 MPI_Group_union(group1, group2, newgroup, ierror) BIND(C)
37 TYPE(MPI_Group), INTENT(IN) :: group1, group2
38 TYPE(MPI_Group), INTENT(OUT) :: newgroup
39 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 40
41 MPI_Intercomm_create(local_comm, local_leader, peer_comm, remote_leader,
42 tag, newintercomm, ierror) BIND(C)
43 TYPE(MPI_Comm), INTENT(IN) :: local_comm, peer_comm
44 INTEGER, INTENT(IN) :: local_leader, remote_leader, tag
45 TYPE(MPI_Comm), INTENT(OUT) :: newintercomm
46 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 47
48 MPI_Intercomm_merge(intercomm, high, newintracomm, ierror) BIND(C)
49 TYPE(MPI_Comm), INTENT(IN) :: intercomm
50 LOGICAL, INTENT(IN) :: high
51 TYPE(MPI_Comm), INTENT(OUT) :: newintracomm
52

```

```

    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_TYPE_DUP_FN(oldtype, type_keyval, extra_state, attribute_val_in,
    attribute_val_out, flag, ierror) BIND(C)
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
    INTEGER, INTENT(IN) :: type_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state,
    attribute_val_in
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val_out
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, INTENT(OUT) :: ierror
MPI_TYPE_NULL_COPY_FN(oldtype, type_keyval, extra_state, attribute_val_in,
    attribute_val_out, flag, ierror) BIND(C)
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
    INTEGER, INTENT(IN) :: type_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state,
    attribute_val_in
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val_out
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, INTENT(OUT) :: ierror
MPI_TYPE_NULL_DELETE_FN(datatype, type_keyval, attribute_val, extra_state,
    ierror) BIND(C)
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, INTENT(IN) :: type_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val,
    extra_state
    INTEGER, INTENT(OUT) :: ierror
MPI_Type_create_keyval(type_copy_attr_fn, type_delete_attr_fn, type_keyval,
    extra_state, ierror) BIND(C)
    PROCEDURE(MPI_Type_copy_attr_function) :: type_copy_attr_fn
    PROCEDURE(MPI_Type_delete_attr_function) :: type_delete_attr_fn
    INTEGER, INTENT(OUT) :: type_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Type_delete_attr(datatype, type_keyval, ierror) BIND(C)
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, INTENT(IN) :: type_keyval
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Type_free_keyval(type_keyval, ierror) BIND(C)
    INTEGER, INTENT(INOUT) :: type_keyval
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Type_get_attr(datatype, type_keyval, attribute_val, flag, ierror)
    BIND(C)
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, INTENT(IN) :: type_keyval

```

1
2 ticket-248T.
3
4
5
6
7
8
9
10
11 ticket-248T.
12
13
14
15
16
17
18
19
20
21 ticket-248T.
22
23
24
25
26
27
28 ticket-248T.
29
30
31
32
33
34
35
36 ticket-248T.
37
38
39
40 ticket-248T.
41
42
43
44 ticket-248T.
45
46
47
48

```

1      INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
2      LOGICAL, INTENT(OUT) :: flag
3      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 4
5      MPI_Type_get_name(datatype, type_name, resultlen, ierror) BIND(C)
6      TYPE(MPI_Datatype), INTENT(IN) :: datatype
7      CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: type_name
8      INTEGER, INTENT(OUT) :: resultlen
9      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 10
11     MPI_Type_set_attr(datatype, type_keyval, attribute_val, ierror) BIND(C)
12     TYPE(MPI_Datatype), INTENT(IN) :: datatype
13     INTEGER, INTENT(IN) :: type_keyval
14     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
15     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 16
17     MPI_Type_set_name(datatype, type_name, ierror) BIND(C)
18     TYPE(MPI_Datatype), INTENT(IN) :: datatype
19     CHARACTER(LEN=*), INTENT(IN) :: type_name
20     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 21
22     MPI_WIN_DUP_FN(oldwin, win_keyval, extra_state, attribute_val_in,
23                   attribute_val_out, flag, ierror) BIND(C)
24     INTEGER, INTENT(IN) :: oldwin, win_keyval
25     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state,
26     attribute_val_in
27     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val_out
28     LOGICAL, INTENT(OUT) :: flag
29     INTEGER, INTENT(OUT) :: ierror
ticket-248T. 30
31     MPI_WIN_NULL_COPY_FN(oldwin, win_keyval, extra_state, attribute_val_in,
32                          attribute_val_out, flag, ierror) BIND(C)
33     INTEGER, INTENT(IN) :: oldwin, win_keyval
34     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state,
35     attribute_val_in
36     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val_out
37     LOGICAL, INTENT(OUT) :: flag
38     INTEGER, INTENT(OUT) :: ierror
ticket-248T. 39
40     MPI_WIN_NULL_DELETE_FN(win, win_keyval, attribute_val, extra_state, ierror)
41     BIND(C)
42     TYPE(MPI_Win), INTENT(IN) :: win
43     INTEGER, INTENT(IN) :: win_keyval
44     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val,
45     extra_state
46     INTEGER, INTENT(OUT) :: ierror
ticket-248T. 47
48     MPI_Win_create_keyval(win_copy_attr_fn, win_delete_attr_fn, win_keyval,
49                          extra_state, ierror) BIND(C)
50     PROCEDURE(MPI_Win_copy_attr_function) :: win_copy_attr_fn
51     PROCEDURE(MPI_Win_delete_attr_function) :: win_delete_attr_fn

```



```

    INTEGER, INTENT(OUT) :: win_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Win_delete_attr(win, win_keyval, ierror) BIND(C)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, INTENT(IN) :: win_keyval
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Win_free_keyval(win_keyval, ierror) BIND(C)
    INTEGER, INTENT(INOUT) :: win_keyval
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Win_get_attr(win, win_keyval, attribute_val, flag, ierror) BIND(C)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, INTENT(IN) :: win_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Win_get_name(win, win_name, resultlen, ierror) BIND(C)
    TYPE(MPI_Win), INTENT(IN) :: win
    CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: win_name
    INTEGER, INTENT(OUT) :: resultlen
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Win_set_attr(win, win_keyval, attribute_val, ierror) BIND(C)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, INTENT(IN) :: win_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Win_set_name(win, win_name, ierror) BIND(C)
    TYPE(MPI_Win), INTENT(IN) :: win
    CHARACTER(LEN=*), INTENT(IN) :: win_name
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

A.3.5 Process Topologies Fortran 2008 Bindings

```

MPI_Cart_coords(comm, rank, maxdims, coords, ierror) BIND(C)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: rank, maxdims
    INTEGER, INTENT(OUT) :: coords(maxdims)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Cart_create(comm_old, ndims, dims, periods, reorder, comm_cart, ierror)
    BIND(C)
    TYPE(MPI_Comm), INTENT(IN) :: comm_old
    INTEGER, INTENT(IN) :: ndims, dims(ndims)
    LOGICAL, INTENT(IN) :: periods(ndims), reorder
    TYPE(MPI_Comm), INTENT(OUT) :: comm_cart

```

```

1      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 2
3      MPI_Cart_get(comm, maxdims, dims, periods, coords, ierror) BIND(C)
4      TYPE(MPI_Comm), INTENT(IN) :: comm
5      INTEGER, INTENT(IN) :: maxdims
6      INTEGER, INTENT(OUT) :: dims(maxdims), coords(maxdims)
7      LOGICAL, INTENT(OUT) :: periods(maxdims)
8      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 9
10     MPI_Cart_map(comm, ndims, dims, periods, newrank, ierror) BIND(C)
11     TYPE(MPI_Comm), INTENT(IN) :: comm
12     INTEGER, INTENT(IN) :: ndims, dims(ndims)
13     LOGICAL, INTENT(IN) :: periods(ndims)
14     INTEGER, INTENT(OUT) :: newrank
15     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 16
17     MPI_Cart_rank(comm, coords, rank, ierror) BIND(C)
18     TYPE(MPI_Comm), INTENT(IN) :: comm
19     INTEGER, INTENT(IN) :: coords(*)
20     INTEGER, INTENT(OUT) :: rank
21     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 22
23     MPI_Cart_shift(comm, direction, disp, rank_source, rank_dest, ierror)
24     BIND(C)
25     TYPE(MPI_Comm), INTENT(IN) :: comm
26     INTEGER, INTENT(IN) :: direction, disp
27     INTEGER, INTENT(OUT) :: rank_source, rank_dest
28     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 29
30     MPI_Cart_sub(comm, remain_dims, newcomm, ierror) BIND(C)
31     TYPE(MPI_Comm), INTENT(IN) :: comm
32     LOGICAL, INTENT(IN) :: remain_dims(*)
33     TYPE(MPI_Comm), INTENT(OUT) :: newcomm
34     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 35
36     MPI_Cartdim_get(comm, ndims, ierror) BIND(C)
37     TYPE(MPI_Comm), INTENT(IN) :: comm
38     INTEGER, INTENT(OUT) :: ndims
39     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 40
41     MPI_Dims_create(nnodes, ndims, dims, ierror) BIND(C)
42     INTEGER, INTENT(IN) :: nnodes, ndims
43     INTEGER, INTENT(INOUT) :: dims(ndims)
44     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 45
46     MPI_Dist_graph_create(comm_old, n, sources, degrees, destinations, weights,
47     info, reorder, comm_dist_graph, ierror) BIND(C)
48     TYPE(MPI_Comm), INTENT(IN) :: comm_old
49     INTEGER, INTENT(IN) :: n, sources(n), degrees(n), destinations(*)
50     INTEGER, INTENT(IN) :: weights(*)
51     TYPE(MPI_Info), INTENT(IN) :: info

```

```

LOGICAL, INTENT(IN) :: reorder
TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Dist_graph_create_adjacent(comm_old, indegree, sources, sourceweights,
                               outdegree, destinations, destweights, info, reorder,
                               comm_dist_graph, ierror) BIND(C)
TYPE(MPI_Comm), INTENT(IN) :: comm_old
INTEGER, INTENT(IN) :: indegree, sources(indegree), outdegree,
destinations(outdegree)
INTEGER, INTENT(IN) :: sourceweights(*), destweights(*)
TYPE(MPI_Info), INTENT(IN) :: info
LOGICAL, INTENT(IN) :: reorder
TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Dist_graph_neighbors(comm, maxindegree, sources, sourceweights,
                        maxoutdegree, destinations, destweights, ierror) BIND(C)
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, INTENT(IN) :: maxindegree, maxoutdegree
INTEGER, INTENT(OUT) :: sources(maxindegree),
destinations(maxoutdegree)
INTEGER :: sourceweights(*), destweights(*)
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Dist_graph_neighbors_count(comm, indegree, outdegree, weighted, ierror)
BIND(C)
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, INTENT(OUT) :: indegree, outdegree
LOGICAL, INTENT(OUT) :: weighted
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Graph_create(comm_old, nnodes, index, edges, reorder, comm_graph,
                 ierror) BIND(C)
TYPE(MPI_Comm), INTENT(IN) :: comm_old
INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*)
LOGICAL, INTENT(IN) :: reorder
TYPE(MPI_Comm), INTENT(OUT) :: comm_graph
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Graph_get(comm, maxindex, maxedges, index, edges, ierror) BIND(C)
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, INTENT(IN) :: maxindex, maxedges
INTEGER, INTENT(OUT) :: index(maxindex), edges(maxedges)
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Graph_map(comm, nnodes, index, edges, newrank, ierror) BIND(C)
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*)
INTEGER, INTENT(OUT) :: newrank

```

1
2
3
4 ticket-248T.

15 ticket-248T.

24 ticket-248T.

30 ticket-248T.

38 ticket-248T.

43 ticket-248T.

```

1      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 2
3      MPI_Graph_neighbors(comm, rank, maxneighbors, neighbors, ierror) BIND(C)
4      TYPE(MPI_Comm), INTENT(IN) :: comm
5      INTEGER, INTENT(IN) :: rank, maxneighbors
6      INTEGER, INTENT(OUT) :: neighbors(maxneighbors)
7      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 8
9      MPI_Graph_neighbors_count(comm, rank, nneighbors, ierror) BIND(C)
10     TYPE(MPI_Comm), INTENT(IN) :: comm
11     INTEGER, INTENT(IN) :: rank
12     INTEGER, INTENT(OUT) :: nneighbors
13     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 14
15     MPI_Graphdims_get(comm, nnodes, nedges, ierror) BIND(C)
16     TYPE(MPI_Comm), INTENT(IN) :: comm
17     INTEGER, INTENT(OUT) :: nnodes, nedges
18     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 19
20     MPI_Topo_test(comm, status, ierror) BIND(C)
21     TYPE(MPI_Comm), INTENT(IN) :: comm
22     INTEGER, INTENT(OUT) :: status
23     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
25     A.3.6 MPI Environmental Management Fortran 2008 Bindings
ticket-248T. 26
27     DOUBLE PRECISION MPI_Wtick() BIND(C)
ticket-248T. 28
29     DOUBLE PRECISION MPI_Wtime() BIND(C)
30
31     MPI_Abort(comm, errorcode, ierror) BIND(C)
32     TYPE(MPI_Comm), INTENT(IN) :: comm
33     INTEGER, INTENT(IN) :: errorcode
34     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 35
36     MPI_Add_error_class(errorclass, ierror) BIND(C)
37     INTEGER, INTENT(OUT) :: errorclass
38     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 39
40     MPI_Add_error_code(errorclass, errorcode, ierror) BIND(C)
41     INTEGER, INTENT(IN) :: errorclass
42     INTEGER, INTENT(OUT) :: errorcode
43     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 44
45     MPI_Add_error_string(errorcode, string, ierror) BIND(C)
46     INTEGER, INTENT(IN) :: errorcode
47     CHARACTER(LEN=*), INTENT(IN) :: string
48     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 49
50     MPI_Alloc_mem(size, info, baseptr, ierror) BIND(C)
51     USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
52     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size

```

```

TYPE(MPI_Info), INTENT(IN) :: info
TYPE(C_PTR), INTENT(OUT) :: baseptr
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Comm_call_errhandler(comm, errorcode, ierror) BIND(C)
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, INTENT(IN) :: errorcode
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Comm_create_errhandler(comm_errhandler_fn, errhandler, ierror) BIND(C)
PROCEDURE(MPI_Comm_errhandler_function) :: comm_errhandler_fn
TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Comm_get_errhandler(comm, errhandler, ierror) BIND(C)
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Comm_set_errhandler(comm, errhandler, ierror) BIND(C)
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Errhandler_free(errhandler, ierror) BIND(C)
TYPE(MPI_Errhandler), INTENT(INOUT) :: errhandler
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Error_class(errorcode, errorclass, ierror) BIND(C)
INTEGER, INTENT(IN) :: errorcode
INTEGER, INTENT(OUT) :: errorclass
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Error_string(errorcode, string, resultlen, ierror) BIND(C)
INTEGER, INTENT(IN) :: errorcode
CHARACTER(LEN=MPI_MAX_ERROR_STRING), INTENT(OUT) :: string
INTEGER, INTENT(OUT) :: resultlen
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_File_call_errhandler(fh, errorcode, ierror) BIND(C)
TYPE(MPI_File), INTENT(IN) :: fh
INTEGER, INTENT(IN) :: errorcode
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_File_create_errhandler(file_errhandler_fn, errhandler, ierror) BIND(C)
PROCEDURE(MPI_File_errhandler_function) :: file_errhandler_fn
TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_File_get_errhandler(file, errhandler, ierror) BIND(C)
TYPE(MPI_File), INTENT(IN) :: file
TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler

```

```

1      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 2
3      MPI_File_set_errhandler(file, errhandler, ierror) BIND(C)
4      TYPE(MPI_File), INTENT(IN) :: file
5      TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
6      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 7
8      MPI_Finalize(ierr) BIND(C)
9      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 10
11     MPI_Finalized(flag, ierror) BIND(C)
12     LOGICAL, INTENT(OUT) :: flag
13     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 14
15     MPI_Free_mem(base, ierror) BIND(C)
16     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: base
17     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 18
19     MPI_Get_processor_name(name, resultlen, ierror) BIND(C)
20     CHARACTER(LEN=MPI_MAX_PROCESSOR_NAME), INTENT(OUT) :: name
21     INTEGER, INTENT(OUT) :: resultlen
22     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 23
24     MPI_Get_version(version, subversion, ierror) BIND(C)
25     INTEGER, INTENT(OUT) :: version, subversion
26     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 27
28     MPI_Init(ierr) BIND(C)
29     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 30
31     MPI_Initialized(flag, ierror) BIND(C)
32     LOGICAL, INTENT(OUT) :: flag
33     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 34
35     MPI_Win_call_errhandler(win, errorcode, ierror) BIND(C)
36     TYPE(MPI_Win), INTENT(IN) :: win
37     INTEGER, INTENT(IN) :: errorcode
38     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 39
40     MPI_Win_create_errhandler(win_errhandler_fn, errhandler, ierror) BIND(C)
41     PROCEDURE(MPI_Win_errhandler_function) :: win_errhandler_fn
42     TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
43     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 44
45     MPI_Win_get_errhandler(win, errhandler, ierror) BIND(C)
46     TYPE(MPI_Win), INTENT(IN) :: win
47     TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
48     INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

A.3.7 The Info Object Fortran 2008 Bindings

```

MPI_Info_create(info, ierror) BIND(C)
  TYPE(MPI_Info), INTENT(OUT) :: info
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Info_delete(info, key, ierror) BIND(C)
  TYPE(MPI_Info), INTENT(IN) :: info
  CHARACTER(LEN=*), INTENT(IN) :: key
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Info_dup(info, newinfo, ierror) BIND(C)
  TYPE(MPI_Info), INTENT(IN) :: info
  TYPE(MPI_Info), INTENT(OUT) :: newinfo
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Info_free(info, ierror) BIND(C)
  TYPE(MPI_Info), INTENT(INOUT) :: info
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Info_get(info, key, valuelen, value, flag, ierror) BIND(C)
  TYPE(MPI_Info), INTENT(IN) :: info
  CHARACTER(LEN=*), INTENT(IN) :: key
  INTEGER, INTENT(IN) :: valuelen
  CHARACTER(LEN=valuelen), INTENT(OUT) :: value
  LOGICAL, INTENT(OUT) :: flag
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Info_get_nkeys(info, nkeys, ierror) BIND(C)
  TYPE(MPI_Info), INTENT(IN) :: info
  INTEGER, INTENT(OUT) :: nkeys
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Info_get_nthkey(info, n, key, ierror) BIND(C)
  TYPE(MPI_Info), INTENT(IN) :: info
  INTEGER, INTENT(IN) :: n
  CHARACTER(LEN=*), INTENT(OUT) :: key
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Info_get_valuelen(info, key, valuelen, flag, ierror) BIND(C)
  TYPE(MPI_Info), INTENT(IN) :: info
  CHARACTER(LEN=*), INTENT(IN) :: key
  INTEGER, INTENT(OUT) :: valuelen
  LOGICAL, INTENT(OUT) :: flag
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Info_set(info, key, value, ierror) BIND(C)
  TYPE(MPI_Info), INTENT(IN) :: info
  CHARACTER(LEN=*), INTENT(IN) :: key, value
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

1
2 ticket-248T.
3
4
5 ticket-248T.
6
7
8
9
10 ticket-248T.
11
12
13
14 ticket-248T.
15
16
17
18 ticket-248T.
19
20
21
22
23
24
25 ticket-248T.
26
27
28
29
30 ticket-248T.
31
32
33
34
35 ticket-248T.
36
37
38
39
40
41
42 ticket-248T.
43
44
45
46
47
48

A.3.8 Process Creation and Management Fortran 2008 Bindings

```

1  MPI_Close_port(port_name, ierror) BIND(C)
2  CHARACTER(LEN=*), INTENT(IN) :: port_name
3  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
5  MPI_Comm_accept(port_name, info, root, comm, newcomm, ierror) BIND(C)
6  CHARACTER(LEN=*), INTENT(IN) :: port_name
7  TYPE(MPI_Info), INTENT(IN) :: info
8  INTEGER, INTENT(IN) :: root
9  TYPE(MPI_Comm), INTENT(IN) :: comm
10 TYPE(MPI_Comm), INTENT(OUT) :: newcomm
11 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
13 MPI_Comm_connect(port_name, info, root, comm, newcomm, ierror) BIND(C)
14 CHARACTER(LEN=*), INTENT(IN) :: port_name
15 TYPE(MPI_Info), INTENT(IN) :: info
16 INTEGER, INTENT(IN) :: root
17 TYPE(MPI_Comm), INTENT(IN) :: comm
18 TYPE(MPI_Comm), INTENT(OUT) :: newcomm
19 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21 MPI_Comm_disconnect(comm, ierror) BIND(C)
22 TYPE(MPI_Comm), INTENT(INOUT) :: comm
23 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
25 MPI_Comm_get_parent(parent, ierror) BIND(C)
26 TYPE(MPI_Comm), INTENT(OUT) :: parent
27 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
29 MPI_Comm_join(fd, intercomm, ierror) BIND(C)
30 INTEGER, INTENT(IN) :: fd
31 TYPE(MPI_Comm), INTENT(OUT) :: intercomm
32 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34 MPI_Comm_spawn(command, argv, maxprocs, info, root, comm, intercomm,
35               array_of_errcodes, ierror) BIND(C)
36 CHARACTER(LEN=*), INTENT(IN) :: command, argv(*)
37 INTEGER, INTENT(IN) :: maxprocs, root
38 TYPE(MPI_Info), INTENT(IN) :: info
39 TYPE(MPI_Comm), INTENT(IN) :: comm
40 TYPE(MPI_Comm), INTENT(OUT) :: intercomm
41 INTEGER :: array_of_errcodes(*)
42 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
44 MPI_Comm_spawn_multiple(count, array_of_commands, array_of_argv,
45                       array_of_maxprocs, array_of_info, root, comm, intercomm,
46                       array_of_errcodes, ierror) BIND(C)
47 INTEGER, INTENT(IN) :: count, array_of_maxprocs(*), root
48 CHARACTER(LEN=*), INTENT(IN) :: array_of_commands(*),
49   array_of_argv(count, *)

```



```

TYPE(MPI_Info), INTENT(IN) :: array_of_info(*)
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Comm), INTENT(OUT) :: intercomm
INTEGER :: array_of_errcodes(*)
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Lookup_name(service_name, info, port_name, ierror) BIND(C)
  CHARACTER(LEN=*), INTENT(IN) :: service_name
  TYPE(MPI_Info), INTENT(IN) :: info
  CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Open_port(info, port_name, ierror) BIND(C)
  TYPE(MPI_Info), INTENT(IN) :: info
  CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Publish_name(service_name, info, port_name, ierror) BIND(C)
  TYPE(MPI_Info), INTENT(IN) :: info
  CHARACTER(LEN=*), INTENT(IN) :: service_name, port_name
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Unpublish_name(service_name, info, port_name, ierror) BIND(C)
  CHARACTER(LEN=*), INTENT(IN) :: service_name, port_name
  TYPE(MPI_Info), INTENT(IN) :: info
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

A.3.9 One-Sided Communications Fortran 2008 Bindings

MPI_Accumulate(origin_addr, origin_count, origin_datatype, target_rank,
               target_disp, target_count, target_datatype, op, win, ierror)
  BIND(C)
  TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
  INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
  TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
  INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
  TYPE(MPI_Op), INTENT(IN) :: op
  TYPE(MPI_Win), INTENT(IN) :: win
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Get(origin_addr, origin_count, origin_datatype, target_rank,
        target_disp, target_count, target_datatype, win, ierror)
  BIND(C)
  TYPE(*), DIMENSION(..), ASYNCHRONOUS :: origin_addr
  INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
  TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
  INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
  TYPE(MPI_Win), INTENT(IN) :: win
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

1  MPI_Put(origin_addr, origin_count, origin_datatype, target_rank,
2          target_disp, target_count, target_datatype, win, ierror)
3          BIND(C)
4          TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
5          INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
6          TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
7          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
8          TYPE(MPI_Win), INTENT(IN) :: win
9          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 10
11  MPI_Win_complete(win, ierror) BIND(C)
12          TYPE(MPI_Win), INTENT(IN) :: win
13          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 14
15  MPI_Win_create(base, size, disp_unit, info, comm, win, ierror) BIND(C)
16          TYPE(*), DIMENSION(..), ASYNCHRONOUS :: base
17          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
18          INTEGER, INTENT(IN) :: disp_unit
19          TYPE(MPI_Info), INTENT(IN) :: info
20          TYPE(MPI_Comm), INTENT(IN) :: comm
21          TYPE(MPI_Win), INTENT(OUT) :: win
22          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 23
24  MPI_Win_fence(assert, win, ierror) BIND(C)
25          INTEGER, INTENT(IN) :: assert
26          TYPE(MPI_Win), INTENT(IN) :: win
27          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 28
29  MPI_Win_free(win, ierror) BIND(C)
30          TYPE(MPI_Win), INTENT(INOUT) :: win
31          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 32
33  MPI_Win_get_group(win, group, ierror) BIND(C)
34          TYPE(MPI_Win), INTENT(IN) :: win
35          TYPE(MPI_Group), INTENT(OUT) :: group
36          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 37
38  MPI_Win_lock(lock_type, rank, assert, win, ierror) BIND(C)
39          INTEGER, INTENT(IN) :: lock_type, rank, assert
40          TYPE(MPI_Win), INTENT(IN) :: win
41          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 42
43  MPI_Win_post(group, assert, win, ierror) BIND(C)
44          TYPE(MPI_Group), INTENT(IN) :: group
45          INTEGER, INTENT(IN) :: assert
46          TYPE(MPI_Win), INTENT(IN) :: win
47          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 48
49  MPI_Win_start(group, assert, win, ierror) BIND(C)
50          TYPE(MPI_Group), INTENT(IN) :: group
51          INTEGER, INTENT(IN) :: assert

```

```

TYPE(MPI_Win), INTENT(IN) :: win
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Win_test(win, flag, ierror) BIND(C)
TYPE(MPI_Win), INTENT(IN) :: win
LOGICAL, INTENT(OUT) :: flag
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Win_unlock(rank, win, ierror) BIND(C)
INTEGER, INTENT(IN) :: rank
TYPE(MPI_Win), INTENT(IN) :: win
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Win_wait(win, ierror) BIND(C)
TYPE(MPI_Win), INTENT(IN) :: win
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

A.3.10 External Interfaces Fortran 2008 Bindings

```

MPI_Grequest_complete(request, ierror) BIND(C)
TYPE(MPI_Request), INTENT(IN) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request,
ierror) BIND(C)
PROCEDURE(MPI_Grequest_query_function) :: query_fn
PROCEDURE(MPI_Grequest_free_function) :: free_fn
PROCEDURE(MPI_Grequest_cancel_function) :: cancel_fn
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Init_thread(required, provided, ierror) BIND(C)
INTEGER, INTENT(IN) :: required
INTEGER, INTENT(OUT) :: provided
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Is_thread_main(flag, ierror) BIND(C)
LOGICAL, INTENT(OUT) :: flag
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Query_thread(provided, ierror) BIND(C)
INTEGER, INTENT(OUT) :: provided
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Status_set_cancelled(status, flag, ierror) BIND(C)
TYPE(MPI_Status), INTENT(INOUT) :: status
LOGICAL, INTENT(OUT) :: flag
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Status_set_elements(status, datatype, count, ierror) BIND(C)
TYPE(MPI_Status), INTENT(INOUT) :: status

```

```

1      TYPE(MPI_Datatype), INTENT(IN) :: datatype
2      INTEGER, INTENT(IN) :: count
3      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
5

```

ticket-248T. 6 A.3.11 I/O Fortran 2008 Bindings

```

7      MPI_File_close(fh, ierror) BIND(C)
8          TYPE(MPI_File), INTENT(INOUT) :: fh
9          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11     ticket-248T. MPI_File_delete(filename, info, ierror) BIND(C)
12         CHARACTER(LEN=*), INTENT(IN) :: filename
13         TYPE(MPI_Info), INTENT(IN) :: info
14         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16     ticket-248T. MPI_File_get_amode(fh, amode, ierror) BIND(C)
17         TYPE(MPI_File), INTENT(IN) :: fh
18         INTEGER, INTENT(OUT) :: amode
19         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21     ticket-248T. MPI_File_get_atomicity(fh, flag, ierror) BIND(C)
22         TYPE(MPI_File), INTENT(IN) :: fh
23         LOGICAL, INTENT(OUT) :: flag
24         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
26     ticket-248T. MPI_File_get_byte_offset(fh, offset, disp, ierror) BIND(C)
27         TYPE(MPI_File), INTENT(IN) :: fh
28         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
29         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp
30         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
32     ticket-248T. MPI_File_get_group(fh, group, ierror) BIND(C)
33         TYPE(MPI_File), INTENT(IN) :: fh
34         TYPE(MPI_Group), INTENT(OUT) :: group
35         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37     ticket-248T. MPI_File_get_info(fh, info_used, ierror) BIND(C)
38         TYPE(MPI_File), INTENT(IN) :: fh
39         TYPE(MPI_Info), INTENT(OUT) :: info_used
40         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
42     ticket-248T. MPI_File_get_position(fh, offset, ierror) BIND(C)
43         TYPE(MPI_File), INTENT(IN) :: fh
44         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset
45         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47     ticket-248T. MPI_File_get_position_shared(fh, offset, ierror) BIND(C)
48         TYPE(MPI_File), INTENT(IN) :: fh
49         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset
50         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
51
52     MPI_File_get_size(fh, size, ierror) BIND(C)

```

```

TYPE(MPI_File), INTENT(IN) :: fh
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: size
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_File_get_type_extent(fh, datatype, extent, ierror) BIND(C)
TYPE(MPI_File), INTENT(IN) :: fh
TYPE(MPI_Datatype), INTENT(IN) :: datatype
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: extent
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_File_get_view(fh, disp, etype, filetype, datarep, ierror) BIND(C)
TYPE(MPI_File), INTENT(IN) :: fh
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp
TYPE(MPI_Datatype), INTENT(OUT) :: etype, filetype
CHARACTER(LEN=*), INTENT(OUT) :: datarep
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_File_iread(fh, buf, count, datatype, request, ierror) BIND(C)
TYPE(MPI_File), INTENT(IN) :: fh
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
INTEGER, INTENT(IN) :: count
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_File_iread_at(fh, offset, buf, count, datatype, request, ierror)
BIND(C)
TYPE(MPI_File), INTENT(IN) :: fh
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
INTEGER, INTENT(IN) :: count
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_File_iread_shared(fh, buf, count, datatype, request, ierror) BIND(C)
TYPE(MPI_File), INTENT(IN) :: fh
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
INTEGER, INTENT(IN) :: count
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_File_iwrite(fh, buf, count, datatype, request, ierror) BIND(C)
TYPE(MPI_File), INTENT(IN) :: fh
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
INTEGER, INTENT(IN) :: count
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

1  MPI_File_iwrite_at(fh, offset, buf, count, datatype, request, ierror)
2      BIND(C)
3      TYPE(MPI_File), INTENT(IN) :: fh
4      INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
5      TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
6      INTEGER, INTENT(IN) :: count
7      TYPE(MPI_Datatype), INTENT(IN) :: datatype
8      TYPE(MPI_Request), INTENT(OUT) :: request
9      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 10
11  MPI_File_iwrite_shared(fh, buf, count, datatype, request, ierror) BIND(C)
12      TYPE(MPI_File), INTENT(IN) :: fh
13      TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
14      INTEGER, INTENT(IN) :: count
15      TYPE(MPI_Datatype), INTENT(IN) :: datatype
16      TYPE(MPI_Request), INTENT(OUT) :: request
17      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 17
18  MPI_File_open(comm, filename, amode, info, fh, ierror) BIND(C)
19      TYPE(MPI_Comm), INTENT(IN) :: comm
20      CHARACTER(LEN=*), INTENT(IN) :: filename
21      INTEGER, INTENT(IN) :: amode
22      TYPE(MPI_Info), INTENT(IN) :: info
23      TYPE(MPI_File), INTENT(OUT) :: fh
24      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 24
25
26  MPI_File_preallocate(fh, size, ierror) BIND(C)
27      TYPE(MPI_File), INTENT(IN) :: fh
28      INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: size
29      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 29
30
31  MPI_File_read(fh, buf, count, datatype, status, ierror) BIND(C)
32      TYPE(MPI_File), INTENT(IN) :: fh
33      TYPE(*), DIMENSION(..) :: buf
34      INTEGER, INTENT(IN) :: count
35      TYPE(MPI_Datatype), INTENT(IN) :: datatype
36      TYPE(MPI_Status) :: status
37      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 37
38
39  MPI_File_read_all(fh, buf, count, datatype, status, ierror) BIND(C)
40      TYPE(MPI_File), INTENT(IN) :: fh
41      TYPE(*), DIMENSION(..) :: buf
42      INTEGER, INTENT(IN) :: count
43      TYPE(MPI_Datatype), INTENT(IN) :: datatype
44      TYPE(MPI_Status) :: status
45      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 44
46
47  MPI_File_read_all_begin(fh, buf, count, datatype, ierror) BIND(C)
48      TYPE(MPI_File), INTENT(IN) :: fh
49      TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf

```

```

    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_File_read_all_end(fh, buf, status, ierror) BIND(C)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_File_read_at(fh, offset, buf, count, datatype, status, ierror) BIND(C)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_File_read_at_all(fh, offset, buf, count, datatype, status, ierror)
    BIND(C)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_File_read_at_all_begin(fh, offset, buf, count, datatype, ierror)
    BIND(C)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_File_read_at_all_end(fh, buf, status, ierror) BIND(C)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_File_read_ordered(fh, buf, count, datatype, status, ierror) BIND(C)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status) :: status

```

```

1      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 2
3      MPI_File_read_ordered_begin(fh, buf, count, datatype, ierror) BIND(C)
4      TYPE(MPI_File), INTENT(IN) :: fh
5      TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
6      INTEGER, INTENT(IN) :: count
7      TYPE(MPI_Datatype), INTENT(IN) :: datatype
8      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 9
10     MPI_File_read_ordered_end(fh, buf, status, ierror) BIND(C)
11     TYPE(MPI_File), INTENT(IN) :: fh
12     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
13     TYPE(MPI_Status) :: status
14     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 15
16     MPI_File_read_shared(fh, buf, count, datatype, status, ierror) BIND(C)
17     TYPE(MPI_File), INTENT(IN) :: fh
18     TYPE(*), DIMENSION(..) :: buf
19     INTEGER, INTENT(IN) :: count
20     TYPE(MPI_Datatype), INTENT(IN) :: datatype
21     TYPE(MPI_Status) :: status
22     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 23
24     MPI_File_seek(fh, offset, whence, ierror) BIND(C)
25     TYPE(MPI_File), INTENT(IN) :: fh
26     INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
27     INTEGER, INTENT(IN) :: whence
28     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 29
30     MPI_File_seek_shared(fh, offset, whence, ierror) BIND(C)
31     TYPE(MPI_File), INTENT(IN) :: fh
32     INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
33     INTEGER, INTENT(IN) :: whence
34     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 35
36     MPI_File_set_atomicity(fh, flag, ierror) BIND(C)
37     TYPE(MPI_File), INTENT(IN) :: fh
38     LOGICAL, INTENT(IN) :: flag
39     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 40
41     MPI_File_set_info(fh, info, ierror) BIND(C)
42     TYPE(MPI_File), INTENT(IN) :: fh
43     TYPE(MPI_Info), INTENT(IN) :: info
44     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 45
46     MPI_File_set_size(fh, size, ierror) BIND(C)
47     TYPE(MPI_File), INTENT(IN) :: fh
48     INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: size
49     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 50
51     MPI_File_set_view(fh, disp, etype, filetype, datarep, info, ierror) BIND(C)
52     TYPE(MPI_File), INTENT(IN) :: fh

```



```

    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: disp
    TYPE(MPI_Datatype), INTENT(IN) :: etype, filetype
    CHARACTER(LEN=*), INTENT(IN) :: datarep
    TYPE(MPI_Info), INTENT(IN) :: info
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_sync(fh, ierror) BIND(C)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_write(fh, buf, count, datatype, status, ierror) BIND(C)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), INTENT(IN) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_write_all(fh, buf, count, datatype, status, ierror) BIND(C)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), INTENT(IN) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_write_all_begin(fh, buf, count, datatype, ierror) BIND(C)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_write_all_end(fh, buf, status, ierror) BIND(C)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_write_at(fh, offset, buf, count, datatype, status, ierror) BIND(C)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    TYPE(*), DIMENSION(..), INTENT(IN) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror)
    BIND(C)
    TYPE(MPI_File), INTENT(IN) :: fh

```

ticket-248T.

ticket-248T.

ticket-248T.

ticket-248T.

ticket-248T.

ticket-248T.

ticket-248T.

```

1      INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
2      TYPE(*), DIMENSION(..), INTENT(IN) :: buf
3      INTEGER, INTENT(IN) :: count
4      TYPE(MPI_Datatype), INTENT(IN) :: datatype
5      TYPE(MPI_Status) :: status
6      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 7
8      MPI_File_write_at_all_begin(fh, offset, buf, count, datatype, ierror)
9          BIND(C)
10         TYPE(MPI_File), INTENT(IN) :: fh
11         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
12         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
13         INTEGER, INTENT(IN) :: count
14         TYPE(MPI_Datatype), INTENT(IN) :: datatype
15         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 16
17         MPI_File_write_at_all_end(fh, buf, status, ierror) BIND(C)
18         TYPE(MPI_File), INTENT(IN) :: fh
19         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
20         TYPE(MPI_Status) :: status
21         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 22
23         MPI_File_write_ordered(fh, buf, count, datatype, status, ierror) BIND(C)
24         TYPE(MPI_File), INTENT(IN) :: fh
25         TYPE(*), DIMENSION(..), INTENT(IN) :: buf
26         INTEGER, INTENT(IN) :: count
27         TYPE(MPI_Datatype), INTENT(IN) :: datatype
28         TYPE(MPI_Status) :: status
29         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 30
31         MPI_File_write_ordered_begin(fh, buf, count, datatype, ierror) BIND(C)
32         TYPE(MPI_File), INTENT(IN) :: fh
33         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
34         INTEGER, INTENT(IN) :: count
35         TYPE(MPI_Datatype), INTENT(IN) :: datatype
36         TYPE(MPI_Status) :: status
37         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 38
39         MPI_File_write_ordered_end(fh, buf, status, ierror) BIND(C)
40         TYPE(MPI_File), INTENT(IN) :: fh
41         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
42         TYPE(MPI_Status) :: status
43         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
ticket-248T. 44
45         MPI_File_write_shared(fh, buf, count, datatype, status, ierror) BIND(C)
46         TYPE(MPI_File), INTENT(IN) :: fh
47         TYPE(*), DIMENSION(..), INTENT(IN) :: buf
48         INTEGER, INTENT(IN) :: count
49         TYPE(MPI_Datatype), INTENT(IN) :: datatype
50         TYPE(MPI_Status) :: status
51         INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

MPI_Register_datarep(datarep, read_conversion_fn, write_conversion_fn,
                     dtype_file_extent_fn, extra_state, ierror) BIND(C)
CHARACTER(LEN=*), INTENT(IN) :: datarep
PROCEDURE(MPI_Datarep_conversion_function) :: read_conversion_fn
PROCEDURE(MPI_Datarep_conversion_function) :: write_conversion_fn
PROCEDURE(MPI_Datarep_extent_function) :: dtype_file_extent_fn
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

A.3.12 Language Bindings Fortran 2008 Bindings

```

MPI_F_sync_reg(buf) BIND(C)
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf

MPI_Sizeof(x, size, ierror) BIND(C)
TYPE(*), DIMENSION(..) :: x
INTEGER, INTENT(OUT) :: size
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Status_f082f(f08_status, f_status, ierror) BIND(C)
TYPE(MPI_Status), INTENT(IN) :: f08_status
INTEGER, INTENT(OUT) :: f_status(MPI_STATUS_SIZE)
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Status_f2f08(f_status, f08_status, ierror) BIND(C)
INTEGER, INTENT(IN) :: f_status(MPI_STATUS_SIZE)
TYPE(MPI_Status), INTENT(OUT) :: f08_status
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_create_f90_complex(p, r, newtype, ierror) BIND(C)
INTEGER, INTENT(IN) :: p, r
TYPE(MPI_Datatype), INTENT(OUT) :: newtype
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_create_f90_integer(r, newtype, ierror) BIND(C)
INTEGER, INTENT(IN) :: r
TYPE(MPI_Datatype), INTENT(OUT) :: newtype
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_create_f90_real(p, r, newtype, ierror) BIND(C)
INTEGER, INTENT(IN) :: p, r
TYPE(MPI_Datatype), INTENT(OUT) :: newtype
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_match_size(typeclass, size, datatype, ierror) BIND(C)
INTEGER, INTENT(IN) :: typeclass, size
TYPE(MPI_Datatype), INTENT(OUT) :: datatype
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

ticket-248T.

ticket-248T.

ticket-248T.

ticket-248T.

ticket-248T.

ticket-248T.

ticket-248T.

ticket-248T.

A.3.13 Profiling Interface Fortran 2008 Bindings

```
MPI_Pcontrol(level) BIND(C)
    INTEGER, INTENT(IN) :: level
```

A.3.14 Deprecated Fortran 2008 Bindings

MPI: A Message-Passing Interface Standard

Version **3.0**

(Draft, with MPI 3 Nonblocking Collectives

and new Fortran 2008 Interface)

Unofficial, for comment only

Message Passing Interface Forum

December 16, 2011

ticket0.

ticket229-A.

	16.3.8 Extra State	614
	16.3.9 Constants	614
	16.3.10 Interlanguage Communication	615
	A Language Bindings Summary	617
	A.1 Defined Values and Handles	617
	A.1.1 Defined Constants	617
	A.1.2 Types	630
ticket0.	A.1.3 Prototype Definitions	631
	C Bindings	631
	Fortran 2008 Bindings with the mpi_f08 Module	632
	Fortran Bindings with mpif.h or the mpi Module	634
	C++ Bindings (deprecated)	637
ticket0.	A.1.4 Deprecated Prototype Definitions	637
ticket0.	A.1.5 Info Keys	638
	A.1.6 Info Values	639
	A.2 C Bindings	640
	A.2.1 Point-to-Point Communication C Bindings	640
	A.2.2 Datatypes C Bindings	641
	A.2.3 Collective Communication C Bindings	643
	A.2.4 Groups, Contexts, Communicators, and Caching C Bindings	646
	A.2.5 Process Topologies C Bindings	648
	A.2.6 MPI Environmental Management C Bindings	649
	A.2.7 The Info Object C Bindings	650
	A.2.8 Process Creation and Management C Bindings	651
	A.2.9 One-Sided Communications C Bindings	651
	A.2.10 External Interfaces C Bindings	652
	A.2.11 I/O C Bindings	652
	A.2.12 Language Bindings C Bindings	655
	A.2.13 Profiling Interface C Bindings	656
	A.2.14 Deprecated C Bindings	656
	A.3 Fortran 2008 Bindings with the mpi_f08 Module	657
	A.3.1 Point-to-Point Communication Fortran 2008 Bindings	657
	A.3.2 Datatypes Fortran 2008 Bindings	662
	A.3.3 Collective Communication Fortran 2008 Bindings	666
	A.3.4 Groups, Contexts, Communicators, and Caching Fortran 2008 Bindings	673
	A.3.5 Process Topologies Fortran 2008 Bindings	679
	A.3.6 MPI Environmental Management Fortran 2008 Bindings	682
	A.3.7 The Info Object Fortran 2008 Bindings	685
	A.3.8 Process Creation and Management Fortran 2008 Bindings	686
	A.3.9 One-Sided Communications Fortran 2008 Bindings	687
	A.3.10 External Interfaces Fortran 2008 Bindings	689
	A.3.11 I/O Fortran 2008 Bindings	690
	A.3.12 Language Bindings Fortran 2008 Bindings	697
	A.3.13 Profiling Interface Fortran 2008 Bindings	698
	A.3.14 Deprecated Fortran 2008 Bindings	698
	A.4 Fortran Bindings with mpif.h or the mpi Module	699
	A.4.1 Point-to-Point Communication Fortran Bindings	699

A.4.2	Datatypes Fortran Bindings	701
A.4.3	Collective Communication Fortran Bindings	704
A.4.4	Groups, Contexts, Communicators, and Caching Fortran Bindings	707
A.4.5	Process Topologies Fortran Bindings	711
A.4.6	MPI Environmental Management Fortran Bindings	713
A.4.7	The Info Object Fortran Bindings	715
A.4.8	Process Creation and Management Fortran Bindings	715
A.4.9	One-Sided Communications Fortran Bindings	716
A.4.10	External Interfaces Fortran Bindings	717
A.4.11	I/O Fortran Bindings	718
A.4.12	Language Bindings Fortran Bindings	722
A.4.13	Profiling Interface Fortran Bindings	723
A.4.14	Deprecated Fortran Bindings	723
A.5	C++ Bindings (deprecated)	725
A.5.1	Point-to-Point Communication C++ Bindings	725
A.5.2	Datatypes C++ Bindings	728
A.5.3	Collective Communication C++ Bindings	730
A.5.4	Groups, Contexts, Communicators, and Caching C++ Bindings	733
A.5.5	Process Topologies C++ Bindings	736
A.5.6	MPI Environmental Management C++ Bindings	737
A.5.7	The Info Object C++ Bindings	738
A.5.8	Process Creation and Management C++ Bindings	739
A.5.9	One-Sided Communications C++ Bindings	740
A.5.10	External Interfaces C++ Bindings	741
A.5.11	I/O C++ Bindings	741
A.5.12	Language Bindings C++ Bindings	745
A.5.13	Profiling Interface C++ Bindings	746
A.5.14	C++ Bindings on all MPI Classes	746
A.5.15	Construction / Destruction	746
A.5.16	Copy / Assignment	746
A.5.17	Comparison	746
A.5.18	Inter-language Operability	747
B	Change-Log	749
B.1	Changes from Version 2.2 to Version 3.0	749
B.2	Changes from Version 2.1 to Version 2.2	751
B.3	Changes from Version 2.0 to Version 2.1	754
B.4	Other New Fortran Interfaces for New MPI-3 Routines	758
B.4.1	Matched Probe & Matched Receive, ticket #38 and #274	758
B.4.2	Nonblocking Collective Communication, ticket #109	759
B.4.3	Neighborhood Collective Communication, ticket #258	764
B.4.4	MPI_Count, ticket #265	767
B.4.5	MPI_T_..., ticket #266	768
B.4.6	RMA, ticket #270	768
B.4.7	Fault Tolerance, ticket #276	774
B.4.8	Hybrid Programming, ticket #284	774
	Bibliography	775

33. Section [A.1.1](#) on page 617.

MPI_BOTTOM is defined as `void * const MPI::BOTTOM`.

B.4 Other New Fortran Interfaces for New MPI-3 Routines

This section is not part of the MPI standard. It is only part of this draft document. The Fortran bindings with the `mpi` module or the `mpif.h` header file should be identical to those provided in the appropriate tickets. The `mpi_f08` bindings are added here. They are provided as part of this draft. The provided interfaces or changes will be part of the appropriate chapter when the cited ticket are voted into this chapter of the MPI standard.

B.4.1 Matched Probe & Matched Receive, ticket #38 and #274

This section requires the new type `TYPE(MPI_Message)`.

`MPI_Improbe(source, tag, comm, flag, message, status, ierror) BIND(C)`

`INTEGER, INTENT(IN) :: source, tag`

`TYPE(MPI_Comm), INTENT(IN) :: comm`

`INTEGER, INTENT(OUT) :: flag`

`TYPE(MPI_Message), INTENT(OUT) :: message`

`TYPE(MPI_Status) :: status`

`INTEGER, OPTIONAL, INTENT(OUT) :: ierror`

`MPI_IMPROBE(SOURCE, TAG, COMM, FLAG, MESSAGE, STATUS, IERROR)`

`INTEGER SOURCE, TAG, COMM, FLAG, MESSAGE, STATUS(MPI_STATUS_SIZE),
IERROR`

`MPI_Mprobe(source, tag, comm, message, status, ierror) BIND(C)`

`INTEGER, INTENT(IN) :: source, tag`

`TYPE(MPI_Comm), INTENT(IN) :: comm`

`TYPE(MPI_Message), INTENT(OUT) :: message`

`TYPE(MPI_Status) :: status`

`INTEGER, OPTIONAL, INTENT(OUT) :: ierror`

`MPI_MPROBE(SOURCE, TAG, COMM, MESSAGE, STATUS, IERROR)`

`INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR`

`MPI_Mrecv(buf, count, datatype, message, status, ierror) BIND(C)`

`TYPE(*), DIMENSION(..) :: buf`

`INTEGER, INTENT(IN) :: count`

`TYPE(MPI_Datatype), INTENT(IN) :: datatype`

`TYPE(MPI_Message), INTENT(INOUT) :: message`

`TYPE(MPI_Status) :: status`

`INTEGER, OPTIONAL, INTENT(OUT) :: ierror`

`MPI_MRECV(BUF, COUNT, DATATYPE, MESSAGE, STATUS, IERROR)`

`<type> BUF(*)`

`INTEGER COUNT, DATATYPE, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR`

`MPI_Imrecv(buf, count, datatype, message, request, ierror) BIND(C)`

`TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf`


```

    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Message), INTENT(INOUT) :: message
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_IMRECV(BUF, COUNT, DATATYPE, MESSAGE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, MESSAGE, REQUEST, IERROR

```

B.4.2 Nonblocking Collective Communication, ticket #109

```

MPI_Iallgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
               comm, request, ierror) BIND(C)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

MPI_IALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
               COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR

```

```

MPI_Iallgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
                recvtype, comm, request, ierror) BIND(C)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

```

```

MPI_IALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
                RECVTYPE, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
    REQUEST, IERROR

```

```

MPI_Iallreduce(sendbuf, recvbuf, count, datatype, op, comm, request,
               ierror) BIND(C)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype

```

```

1      TYPE(MPI_Op), INTENT(IN) :: op
2      TYPE(MPI_Comm), INTENT(IN) :: comm
3      TYPE(MPI_Request), INTENT(OUT) :: request
4      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
6      MPI_IALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST,
7                    IERROR)
8      <type> SENDBUF(*), RECVBUF(*)
9      INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR
ticket-248T.
10     MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
11                  comm, request, ierror) BIND(C)
12     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
13     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
14     INTEGER, INTENT(IN) :: sendcount, recvcount
15     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
16     TYPE(MPI_Comm), INTENT(IN) :: comm
17     TYPE(MPI_Request), INTENT(OUT) :: request
18     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
20     MPI_IALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, REVCOUNT, RECVTYPE,
21                  COMM, REQUEST, IERROR)
22     <type> SENDBUF(*), RECVBUF(*)
23     INTEGER SENDCOUNT, SENDTYPE, REVCOUNT, RECVTYPE, COMM, REQUEST, IERROR
ticket229.1.
ticket-248T.
24     MPI_Ialltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
25                   rdispls, recvtype, comm, request, ierror) BIND(C)
26     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
27     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
28     INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
29     recvcounts(*), rdispls(*)
30     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
31     TYPE(MPI_Comm), INTENT(IN) :: comm
32     TYPE(MPI_Request), INTENT(OUT) :: request
33     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
35     MPI_IALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, REVCOUNTS,
36                   RDISPLS, RECVTYPE, COMM, REQUEST, IERROR)
37     <type> SENDBUF(*), RECVBUF(*)
38     INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, REVCOUNTS(*), RDISPLS(*),
39     RECVTYPE, COMM, REQUEST, IERROR
ticket229.1.
ticket-248T.
40     MPI_Ialltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
41                   recvcounts, rdispls, recvtypes, comm, request, ierror) BIND(C)
42     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
43     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
44     INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
45     recvcounts(*), rdispls(*)
46     TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
47     recvtypes(*)
48

```

```

TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_IALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
               RECVCOUNTS, RDISPLS, RECVTYPES, COMM, REQUEST, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
RDISPLS(*), RECVTYPES(*), COMM, REQUEST, IERROR

MPI_Ibarrier(comm, request, ierror) BIND(C)
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_IBARRIER(COMM, REQUEST, IERROR)
INTEGER COMM, REQUEST, IERROR

MPI_Ibcast(buffer, count, datatype, root, comm, request, ierror) BIND(C)
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buffer
INTEGER, INTENT(IN) :: count, root
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_IBCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR)
<type> BUFFER(*)
INTEGER COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR

MPI_Iexscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
BIND(C)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
INTEGER, INTENT(IN) :: count
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Op), INTENT(IN) :: op
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_IEXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR

MPI_Igather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvttype,
            root, comm, request, ierror) BIND(C)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
INTEGER, INTENT(IN) :: sendcount, recvcount, root
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvttype

```

```

1      TYPE(MPI_Comm), INTENT(IN) :: comm
2      TYPE(MPI_Request), INTENT(OUT) :: request
3      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
5      MPI_IGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
6                  ROOT, COMM, REQUEST, IERROR)
7      <type> SENDBUF(*), RECVBUF(*)
8      INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,
9      IERROR
ticket229.1.
ticket-248T. 10      MPI_Igatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
11                  recvtype, root, comm, request, ierror) BIND(C)
12      TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
13      TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
14      INTEGER, INTENT(IN) :: sendcount, root
15      INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
16      TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
17      TYPE(MPI_Comm), INTENT(IN) :: comm
18      TYPE(MPI_Request), INTENT(OUT) :: request
19      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21      MPI_IGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
22                  RECVTYPE, ROOT, COMM, REQUEST, IERROR)
23      <type> SENDBUF(*), RECVBUF(*)
24      INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
25      COMM, REQUEST, IERROR
ticket-248T. 26      MPI_Ireduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
27                  request, ierror) BIND(C)
28      TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
29      TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
30      INTEGER, INTENT(IN) :: recvcount
31      TYPE(MPI_Datatype), INTENT(IN) :: datatype
32      TYPE(MPI_Op), INTENT(IN) :: op
33      TYPE(MPI_Comm), INTENT(IN) :: comm
34      TYPE(MPI_Request), INTENT(OUT) :: request
35      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37      MPI_IREDUCE_SCATTER_BLOCK(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,
38                  REQUEST, IERROR)
39      <type> SENDBUF(*), RECVBUF(*)
40      INTEGER RECVCOUNT, DATATYPE, OP, COMM, REQUEST, IERROR
ticket229.1.
ticket-248T. 41      MPI_Ireduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
42                  request, ierror) BIND(C)
43      TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
44      TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
45      INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
46      TYPE(MPI_Datatype), INTENT(IN) :: datatype
47      TYPE(MPI_Op), INTENT(IN) :: op
48

```

```

TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_IREDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
    REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, REQUEST, IERROR

MPI_Ireduce(sendbuf, recvbuf, count, datatype, op, root, comm, request,
    ierror) BIND(C)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: count, root
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_IREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, REQUEST,
    IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, ROOT, COMM, REQUEST, IERROR

MPI_Iscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
    BIND(C)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_ISCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR

MPI_Iscatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
    root, comm, request, ierror) BIND(C)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_ISCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,

```

```

1          ROOT, COMM, REQUEST, IERROR)
2      <type> SENDBUF(*), RECVBUF(*)
3      INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,
4      IERROR
5
6      MPI_Iscatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
7          recvtype, root, comm, request, ierror) BIND(C)
8      TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
9      TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
10     INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), displs(*)
11     INTEGER, INTENT(IN) :: recvcount, root
12     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
13     TYPE(MPI_Comm), INTENT(IN) :: comm
14     TYPE(MPI_Request), INTENT(OUT) :: request
15     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17     MPI_ISCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,
18         RECVTYPE, ROOT, COMM, REQUEST, IERROR)
19     <type> SENDBUF(*), RECVBUF(*)
20     INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
21     COMM, REQUEST, IERROR
22

```

B.4.3 Neighborhood Collective Communication, ticket #258

```

23
24     MPI_Neighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
25         recvtype, comm, ierror) BIND(C)
26     TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
27     TYPE(*), DIMENSION(..) :: recvbuf
28     INTEGER, INTENT(IN) :: sendcount, recvcount
29     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
30     TYPE(MPI_Comm), INTENT(IN) :: comm
31     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
33     MPI_NEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
34         RECVTYPE, COMM, IERROR)
35     <type> SENDBUF(*), RECVBUF(*)
36     INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
37
38     MPI_Neighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
39         displs, recvtype, comm, ierror) BIND(C)
40     TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
41     TYPE(*), DIMENSION(..) :: recvbuf
42     INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*)
43     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
44     TYPE(MPI_Comm), INTENT(IN) :: comm
45     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47     MPI_NEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
48         DISPLS, RECVTYPE, COMM, IERROR)
49     <type> SENDBUF(*), RECVBUF(*)
50

```

```

INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
IERROR
MPI_Neighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
    recvtype, comm, ierror) BIND(C)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: sendcount, recvcount
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_NEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
    RECVTYPE, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR

MPI_Neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
    recvcounts, rdispls, recvtype, comm, ierror) BIND(C)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
rdispls(*)
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_NEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
    RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
RECVTYPE, COMM, IERROR

Decision at the 05/2011 meeting: INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS,
RDISPLS should be used in new bindings of ALLTOALLW. This is changed manually
in the following interface:
MPI_Neighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
    recvcounts, rdispls, recvtypes, comm, ierror) BIND(C)
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: sendcounts(*), recvcounts(*)
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_NEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
    RECVCOUNTS, RDISPLS, RECVTYPES, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,

```

ticket-248T.

ticket-248T.

ticket-248T.

```

1      IERROR
2      INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)
ticket-248T. 3
4      MPI_Ineighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
5          recvtype, comm, request, ierror) BIND(C)
6      TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
7      TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
8      INTEGER, INTENT(IN) :: sendcount, recvcount
9      TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
10     TYPE(MPI_Comm), INTENT(IN) :: comm
11     TYPE(MPI_Request), INTENT(OUT) :: request
12     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14     MPI_INEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
15         RECVTYPE, COMM, REQUEST, IERROR)
16     <type> SENDBUF(*), RECVBUF(*)
ticket229.2. 17
ticket-248T. 18     MPI_Ineighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcoun
19         displs, recvtype, comm, request, ierror) BIND(C)
20     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
21     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
22     INTEGER, INTENT(IN) :: sendcount
23     INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcoun
24     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
25     TYPE(MPI_Comm), INTENT(IN) :: comm
26     TYPE(MPI_Request), INTENT(OUT) :: request
27     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
29     MPI_INEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
30         DISPLS, RECVTYPE, COMM, REQUEST, IERROR)
31     <type> SENDBUF(*), RECVBUF(*)
32     INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
33     REQUEST, IERROR
ticket-248T. 34
35     MPI_Ineighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
36         recvtype, comm, request, ierror) BIND(C)
37     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
38     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
39     INTEGER, INTENT(IN) :: sendcount, recvcount
40     TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
41     TYPE(MPI_Comm), INTENT(IN) :: comm
42     TYPE(MPI_Request), INTENT(OUT) :: request
43     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45     MPI_INEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
46         RECVTYPE, COMM, REQUEST, IERROR)
47     <type> SENDBUF(*), RECVBUF(*)
ticket229.2. 48     INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR
ticket-248T. 49
50     MPI_Ineighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,

```



```

        recvcounts, rdispls, recvtype, comm, request, ierror) BIND(C)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
recvcounts(*), rdispls(*)
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_INEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
        RECVCOUNTS, RDISPLS, RECVTYPE, COMM, REQUEST, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
RECVTYPE, COMM, REQUEST, IERROR

```

Decision at the 05/2011 meeting: INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS, RDISPLS should be used in new bindings of ALLTOALLW. This is changed manually in the following interface:

```

MPI_Ineighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
        recvcounts, rdispls, recvtypes, comm, request, ierror) BIND(C)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcounts(*)
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS ::
sdispls(*), rdispls(*)
TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
recvtypes(*)
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_INEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
        RECVCOUNTS, RDISPLS, RECVTYPES, COMM, REQUEST, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,
REQUEST, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)

```

B.4.4 MPI_Count, ticket #265

Be aware of new constant MPI_COUNT_KIND in the following interfaces:

```

MPI_Type_size_x(datatype, size, ierror) BIND(C)
TYPE(MPI_Datatype), INTENT(IN) :: datatype
INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_TYPE_SIZE_X(DATATYPE, SIZE, IERROR)
INTEGER DATATYPE, IERROR

```

1 INTEGER (KIND=MPI_COUNT_KIND) SIZE
 ticket-248T. 2
 3 MPI_Type_get_extent_x(datatype, lb, extent, ierror) BIND(C)
 4 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 5 INTEGER(KIND = MPI_COUNT_KIND), INTENT(OUT) :: lb, extent
 6 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 7
 8 MPI_TYPE_GET_EXTENT_X(DATATYPE, LB, EXTENT, IERROR)
 9 INTEGER DATATYPE, IERROR
 10 INTEGER(KIND = MPI_COUNT_KIND) LB, EXTENT
 ticket-248T. 11
 12 MPI_Type_get_true_extent_x(datatype, true_lb, true_extent, ierror) BIND(C)
 13 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 14 INTEGER(KIND = MPI_COUNT_KIND), INTENT(OUT) :: true_lb, true_extent
 15 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 16
 17 MPI_TYPE_GET_TRUE_EXTENT_X(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)
 18 INTEGER DATATYPE, IERROR
 19 INTEGER(KIND = MPI_COUNT_KIND) TRUE_LB, TRUE_EXTENT
 ticket-248T. 20
 21 MPI_Get_elements_x(status, datatype, count, ierror) BIND(C)
 22 TYPE(MPI_Status), INTENT(IN) :: status
 23 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 24 INTEGER(KIND = MPI_COUNT_KIND), INTENT(OUT) :: count
 25 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 26
 27 MPI_GET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR)
 28 INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR
 29 INTEGER (KIND=MPI_COUNT_KIND) COUNT
 ticket-248T. 30
 31 MPI_Status_set_elements_x(status, datatype, count, ierror) BIND(C)
 32 TYPE(MPI_Status), INTENT(INOUT) :: status
 33 TYPE(MPI_Datatype), INTENT(IN) :: datatype
 34 INTEGER(KIND = MPI_COUNT_KIND), INTENT(IN) :: count
 35 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 36
 37 MPI_STATUS_SET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR)
 38 INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR
 39 INTEGER (KIND=MPI_COUNT_KIND) COUNT
 40
 41 B.4.5 MPI_T_..., ticket #266
 42 There isn't any Fortran interface for this ticket.
 43
 44 B.4.6 RMA, ticket #270
 ticket229.4. 45 **The following interface in ticket #270**
 46 MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR)
 47 INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
 48 INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
 ticket-248T. **should be substituted by**

```

MPI_Win_allocate(size, disp_unit, info, comm, baseptr, win, ierror) BIND(C)
  USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
  INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
  INTEGER, INTENT(IN) :: disp_unit
  TYPE(MPI_Info), INTENT(IN) :: info
  TYPE(MPI_Comm), INTENT(IN) :: comm
  TYPE(C_PTR), INTENT(OUT) :: baseptr
  TYPE(MPI_Win), INTENT(OUT) :: win
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR)
  INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
  INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR

```

If the Fortran compiler provides `TYPE(C_PTR)`, then the following interface must be provided in the `mpi` module and should be provided in `mpif.h` through overloading, i.e., with the same routine name as the routine with `INTEGER(KIND=MPI_ADDRESS_KIND) BASEPTR`, but with a different linker name:

```

INTERFACE MPI_WIN_ALLOCATE
  SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, &
    & BASEPTR, WIN, IERROR)
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
    INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
    TYPE(C_PTR) :: BASEPTR
  END SUBROUTINE
END INTERFACE

```

The linker name base of this overloaded function is `MPI_WIN_ALLOCATE_CPTR`. The implied linker names are described in Section 16.2.5 on page 560.

```

MPI_Win_create_dynamic(info, comm, win, ierror) BIND(C)
  TYPE(MPI_Info), INTENT(IN) :: info
  TYPE(MPI_Comm), INTENT(IN) :: comm
  TYPE(MPI_Win), INTENT(OUT) :: win
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_WIN_CREATE_DYNAMIC(INFO, COMM, WIN, IERROR)
  INTEGER INFO, COMM, WIN, IERROR

MPI_Win_attach(win, base, size, ierror) BIND(C)
  TYPE(MPI_Win), INTENT(IN) :: win
  TYPE(*), ASYNCHRONOUS :: base
  INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_WIN_ATTACH(WIN, BASE, SIZE, IERROR)
  INTEGER WIN, IERROR
  <type> base

```

```

1      INTEGER (KIND=MPI_ADDRESS_KIND) size
2
3      MPI_Win_detach(win, base, ierror) BIND(C)
4      TYPE(MPI_Win), INTENT(IN) :: win
5      TYPE(*), ASYNCHRONOUS :: base
6      INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
8      MPI_WIN_DETACH(WIN, BASE, IERROR)
9      INTEGER WIN, IERROR
10     <type> base
11
12     MPI_Get_accumulate(origin_addr, origin_count, origin_datatype, result_addr,
13         result_count, result_datatype, target_rank, target_disp,
14         target_count, target_datatype, op, win, ierror) BIND(C)
15     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
16     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: result_addr
17     INTEGER, INTENT(IN) :: origin_count, result_count, target_rank,
18     target_count
19     TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype,
20     result_datatype
21     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
22     TYPE(MPI_Op), INTENT(IN) :: op
23     TYPE(MPI_Win), INTENT(IN) :: win
24     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
26     MPI_GET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_ADDR,
27         RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK, TARGET_DISP,
28         TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
29     <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
30     INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
31     INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE,
32     TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR
33
34     MPI_Fetch_and_op(origin_addr, result_addr, datatype, target_rank,
35         target_disp, op, win, ierror) BIND(C)
36     TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
37     TYPE(*), DIMENSION(..), ASYNCHRONOUS :: result_addr
38     TYPE(MPI_Datatype), INTENT(IN) :: datatype
39     INTEGER, INTENT(IN) :: target_rank
40     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
41     TYPE(MPI_Op), INTENT(IN) :: op
42     TYPE(MPI_Win), INTENT(IN) :: win
43     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45     MPI_FETCH_AND_OP(ORIGIN_ADDR, RESULT_ADDR, DATATYPE, TARGET_RANK,
46         TARGET_DISP, OP, WIN, IERROR)
47     <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
48     INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
49     INTEGER DATATYPE, TARGET_RANK, OP, WIN, IERROR
50
51     MPI_Compare_and_swap(origin_addr, compare_addr, result_addr, datatype,

```

```

        target_rank, target_disp, win, ierror) BIND(C)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  origin_addr,
compare_addr
TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  result_addr
TYPE(MPI_Datatype), INTENT(IN) ::  datatype
INTEGER, INTENT(IN) ::  target_rank
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  target_disp
TYPE(MPI_Win), INTENT(IN) ::  win
INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_COMPARE_AND_SWAP(ORIGIN_ADDR, COMPARE_ADDR, RESULT_ADDR, DATATYPE,
        TARGET_RANK, TARGET_DISP, WIN, IERROR)
<type> ORIGIN_ADDR(*), COMPARE_ADDR(*), RESULT_ADDR(*)
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
INTEGER DATATYPE, TARGET_RANK, WIN, IERROR

MPI_Rput(origin_addr, origin_count, origin_datatype, target_rank,
        target_disp, target_count, target_datatype, win, request,
        ierror) BIND(C)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  origin_addr
INTEGER, INTENT(IN) ::  origin_count, target_rank, target_count
TYPE(MPI_Datatype), INTENT(IN) ::  origin_datatype, target_datatype
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  target_disp
TYPE(MPI_Win), INTENT(IN) ::  win
TYPE(MPI_Request), INTENT(OUT) ::  request
INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_RPUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
        TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, REQUEST,
        IERROR)
<type> ORIGIN_ADDR(*)
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
TARGET_DATATYPE, WIN, REQUEST, IERROR

MPI_Rget(origin_addr, origin_count, origin_datatype, target_rank,
        target_disp, target_count, target_datatype, win, request,
        ierror) BIND(C)
TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  origin_addr
INTEGER, INTENT(IN) ::  origin_count, target_rank, target_count
TYPE(MPI_Datatype), INTENT(IN) ::  origin_datatype, target_datatype
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  target_disp
TYPE(MPI_Win), INTENT(IN) ::  win
TYPE(MPI_Request), INTENT(OUT) ::  request
INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_RGET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
        TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, REQUEST,
        IERROR)
<type> ORIGIN_ADDR(*)

```

1 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
 2 INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
 3 TARGET_DATATYPE, WIN, REQUEST, IERROR
 ticket-248T. 4 MPI_Raccumulate(origin_addr, origin_count, origin_datatype, target_rank,
 5 target_disp, target_count, target_datatype, op, win, request,
 6 ierror) BIND(C)
 7 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
 8 INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
 9 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
 10 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
 11 TYPE(MPI_Op), INTENT(IN) :: op
 12 TYPE(MPI_Win), INTENT(IN) :: win
 13 TYPE(MPI_Request), INTENT(OUT) :: request
 14 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 15
 16 MPI_RACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
 17 TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,
 18 IERROR)
 19 <type> ORIGIN_ADDR(*)
 20 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
 21 INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
 22 TARGET_DATATYPE, OP, WIN, REQUEST, IERROR
 ticket-248T. 23 MPI_Rget_accumulate(origin_addr, origin_count, origin_datatype,
 24 result_addr, result_count, result_datatype, target_rank,
 25 target_disp, target_count, target_datatype, op, win, request,
 26 ierror) BIND(C)
 27 TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
 28 TYPE(*), DIMENSION(..), ASYNCHRONOUS :: result_addr
 29 INTEGER, INTENT(IN) :: origin_count, result_count, target_rank,
 30 target_count
 31 TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype,
 32 result_datatype
 33 INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
 34 TYPE(MPI_Op), INTENT(IN) :: op
 35 TYPE(MPI_Win), INTENT(IN) :: win
 36 TYPE(MPI_Request), INTENT(OUT) :: request
 37 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
 38
 39 MPI_RGET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE,
 40 RESULT_ADDR, RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK,
 41 TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,
 42 IERROR)
 43 <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
 44 INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
 45 INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE,
 46 TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST, IERROR
 ticket-248T. 47 MPI_Win_lock_all(assert, win, ierror) BIND(C)
 48

```

    INTEGER, INTENT(IN) :: assert
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_WIN_LOCK_ALL(ASSERT, WIN, IERROR)
    INTEGER ASSERT, WIN, IERROR

MPI_Win_unlock_all(win, ierror) BIND(C)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_WIN_UNLOCK_ALL(WIN, IERROR)
    INTEGER WIN, IERROR

MPI_Win_flush(rank, win, ierror) BIND(C)
    INTEGER, INTENT(IN) :: rank
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_WIN_FLUSH(RANK, WIN, IERROR)
    INTEGER RANK, WIN, IERROR

MPI_Win_flush_all(win, ierror) BIND(C)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_WIN_FLUSH_ALL(WIN, IERROR)
    INTEGER WIN, IERROR

MPI_Win_flush_local(rank, win, ierror) BIND(C)
    INTEGER, INTENT(IN) :: rank
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_WIN_FLUSH_LOCAL(RANK, WIN, IERROR)
    INTEGER RANK, WIN, IERROR

MPI_Win_flush_local_all(win, ierror) BIND(C)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_WIN_FLUSH_LOCAL_ALL(WIN, IERROR)
    INTEGER WIN, IERROR

MPI_Win_sync(win, ierror) BIND(C)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_WIN_SYNC(WIN, IERROR)
    INTEGER WIN, IERROR

```

1
2
3
4
5
6 ticket-248T.
7
8
9
10
11
12 ticket-248T.
13
14
15
16
17
18
19 ticket-248T.
20
21
22
23
24
25 ticket-248T.
26
27
28
29
30
31
32 ticket-248T.
33
34
35
36
37
38
39 ticket-248T.
40
41
42
43
44
45
46
47
48

B.4.7 Fault Tolerance, ticket #276

The Fortran interface is not yet available.

B.4.8 Hybrid Programming, ticket #284

```
MPI_Comm_split_type(comm, comm_type, key, info, newcomm, ierror) BIND(C)
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, INTENT(IN) :: comm_type, key
  TYPE(MPI_Info), INTENT(IN) :: info
  TYPE(MPI_Comm), INTENT(OUT) :: newcomm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_COMM_SPLIT_TYPE(COMM, COMM_TYPE, KEY, INFO, NEWCOMM, IERROR)
  INTEGER COMM, COMM_TYPE, KEY, INFO, NEWCOMM, IERROR
```