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. ticket141.

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. ticket141.

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 (Note that ANSI C has been replaced by ISO C.) The C++ language bindings have been deprecated.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. ticket150.

MPI bindings are for Fortran 90, though they are designed to be usable in Fortran 77 environments.

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 INTEGERs 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.

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

ticket150.

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 "#include" directive can be used to include the necessary C++ definitions in the mpi.h file. (*End of advice to implementors.*)

C++ functions that create objects or return information usually place the object or information in the return value. Since the language neutral prototypes of MPI functions include the C++ return value as an OUT parameter, semantic descriptions of MPI functions refer to the C++ return value by that parameter name. The remaining C++ functions return void.

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

ticket150.

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

ticket150.

The blocking semantics of this call are described in Section 3.4.

15.2 Deprecated since MPI-2.1

The entire set of C++ language bindings have been deprecated.

> *Rationale.* The C++ bindings add minimal functionality over the C bindings while incurring a significant amount of maintenance to the MPI specification. Since the C++ bindings are effectively a one-to-one mapping of the C bindings, it should be relatively easy to convert existing C++ MPI applications to use the MPI C bindings. Additionally, there are third party packages available that provide C++ class library functionality (i.e., C++-specific functionality layered on top of the MPI C bindings) that are likely more expressive and/or natural to C++ programmers and are not suitable for standardization in this specification. (*End of rationale.*)

Chapter 16

Language Bindings

16.1 C++

16.1.1 Overview

ticket150.

The C++ language bindings have been deprecated.

There are some issues specific to C++ that must be considered in the design of an interface that go beyond the simple description of language bindings. In particular, in C++, we must be concerned with the design of objects and their interfaces, rather than just the design of a language-specific functional interface to MPI. Fortunately, the design of MPI was based on the notion of objects, so a natural set of classes is already part of MPI.

MPI-2 includes C++ bindings as part of its function specifications. In some cases, MPI-2 provides new names for the C bindings of MPI-1 functions. In this case, the C++ binding matches the new C name — there is no binding for the deprecated name.

16.1.2 Design

The C++ language interface for MPI is designed according to the following criteria:

1. The C++ language interface consists of a small set of classes with a lightweight functional interface to MPI. The classes are based upon the fundamental MPI object types (e.g., communicator, group, etc.).

2. The MPI C++ language bindings provide a semantically correct interface to MPI.

3. To the greatest extent possible, the C++ bindings for MPI functions are member functions of MPI classes.

   Rationale. Providing a lightweight set of MPI objects that correspond to the basic MPI types is the best fit to MPI's implicit object-based design; methods can be supplied for these objects to realize MPI functionality. The existing C bindings can be used in C++ programs, but much of the expressive power of the C++ language is forfeited. On the other hand, while a comprehensive class library would make user programming more elegant, such a library it is not suitable as a language binding for MPI since a binding must provide a direct and unambiguous mapping to the specified functionality of MPI. (*End of rationale.*)

.

A.4 C++ Bindings (deprecated)

A.4.1 Point-to-Point Communication C++ Bindings

`namespace MPI {`

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

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

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

{`void Request::Cancel() const` *(binding deprecated, see Section 15.2)* }

{`int Detach_buffer(void*& buffer)` *(binding deprecated, see Section 15.2)* }

{`void Request::Free()` *(binding deprecated, see Section 15.2)* }

{`int Status::Get_count(const Datatype& datatype) const` *(binding deprecated, see Section 15.2)* }

{`int Status::Get_error() const` *(binding deprecated, see Section 15.2)* }

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

{`bool Request::Get_status() const` *(binding deprecated, see Section 15.2)* }

{`bool Request::Get_status(Status& status) const` *(binding deprecated, see Section 15.2)* }

{`int Status::Get_tag() const` *(binding deprecated, see Section 15.2)* }

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

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

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

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

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

{`bool Status::Is_cancelled() const` *(binding deprecated, see Section 15.2)* }

Annex B

Change-Log

This annex summarizes changes from the previous version of the MPI standard to the
ticket99. version presented by this document. [Only changes (i.e., clarifications and new features)
ticket99. are presented that may cause implementation effort in the MPI libraries.] 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
ticket100. are not shown.

B.1 Changes from Version 2.1 to Version 2.2

ticket65.

1. Section 2.5.4 on page 14.
   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.
   ticket150.
2. Section 2.6 on page 16, Section 2.6.4 on page 19, and Section 16.1 on page 469.
   The C++ language bindings have been deprecated and may be removed in a future version of the MPI specification.
   ticket63.
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.
   ticket18.
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.
   ticket45+98.
5. Section 3.4 on page 40, Section 3.7.2 on page 51, Section 3.9 on page 71, and Section 5.1 on page 133.
   The read access restriction on the send buffer for blocking, non blocking and collective API has been lifted. It is permitted to access for read the send buffer while the operation is in progress.
   ticket50.