Advice to implementors. The intended semantics of opaque objects is that opaque objects are separate from one another; each call to allocate such an object copies all the information required for the object. Implementations may avoid excessive copying by substituting referencing for copying. For example, a derived datatype may contain references to its components, rather then copies of its components; a call to MPI_COMM_GROUP may return a reference to the group associated with the communicator, rather than a copy of this group. In such cases, the implementation must maintain reference counts, and allocate and deallocate objects in such a way that the visible effect is as if the objects were copied. (*End of advice to implementors.*)

2.5.2 Array Arguments

An MPI call may need an argument that is an array of opaque objects, or an array of handles. The array-of-handles is a regular array with entries that are handles to objects of the same type in consecutive locations in the array. Whenever such an array is used, an additional len argument is required to indicate the number of valid entries (unless this number can be derived otherwise). The valid entries are at the beginning of the array; len indicates how many of them there are, and need not be the size of the entire array. The same approach is followed for other array arguments. In some cases NULL handles are considered valid entries. When a NULL argument is desired for an array of statuses, one uses MPI_STATUSES_IGNORE.

2.5.3 State

MPI procedures use at various places arguments with *state* types. The values of such a data type are all identified by names, and no operation is defined on them. For example, the MPI_TYPE_CREATE_SUBARRAY routine has a state argument order with values MPI_ORDER_C and MPI_ORDER_FORTRAN.

2.5.4 Named Constants

MPI procedures sometimes assign a special meaning to a special value of a basic type argument; e.g., tag is an integer-valued argument of point-to-point communication operations, with a special wild-card value, MPI_ANY_TAG. Such arguments will have a range of regular values, which is a proper subrange of the range of values of the corresponding basic type; special values (such as MPI_ANY_TAG) will be outside the regular range. The range of regular values, such as tag, can be queried using environmental inquiry functions (Chapter 7 of the MPI-1 document). The range of other values, such as source, depends on values given by other MPI routines (in the case of source it is the communicator size).

MPI also provides predefined named constant handles, such as MPI_COMM_WORLD.

All named constants, with the exceptions noted below for Fortran, can be used in initialization expressions or assignments, but not necessarily in array declarations or as labels in C/C++ `switch` or Fortran `select/case` statements. This implies named constants to be link-time but not necessarily compile-time constants. The named constants listed below are required to be compile-time constants in both C/C++ and Fortran. These constants do not change values during execution. Opaque objects accessed by constant handles are defined and do not change value between MPI initialization (MPI_INIT) and MPI completion (MPI_FINALIZE). The handles themselves are constants and can be also used in initialization expressions or assignments.

The constants that are required to be compile-time constants (and can thus be used for array length declarations and labels in C/C++ `switch` and Fortran `case`/`select` statements) are:

MPI_MAX_PROCESSOR_NAME
MPI_MAX_ERROR_STRING
MPI_MAX_DATAREP_STRING
MPI_MAX_INFO_KEY
MPI_MAX_INFO_VAL
MPI_MAX_OBJECT_NAME
MPI_MAX_PORT_NAME
MPI_STATUS_SIZE (Fortran only)
MPI_ADDRESS_KIND (Fortran only)
MPI_INTEGER_KIND (Fortran only)
MPI_OFFSET_KIND (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, the document uses <type> to represent a choice variable; for C and C++, we use void *.

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.

ticket265.

2.5.8 Counts

Derived datatypes can be created representing more elements than can be encoded in a C int or Fortran `INTEGER`. MPI_GET_COUNT, MPI_GET_ELEMENTS , and associated functions cannot properly express these quantities. To overcome this limitation, these quantities are declared to be `INTEGER (KIND=MPI_COUNT_KIND)` in Fortran. In C one uses MPI_Count. These types must have the same width and encode values in the same manner such that count values in one language may be passed directly to another language without conversion. The size of the MPI_Count type is determined by the MPI implementation with the restriction that it must be minimally capable of encoding a C int and Fortran `INTEGER` and any value that may be stored in an MPI_Aint type.

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, 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 `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_FN
MPI_Delete_function	MPI_Comm_delete_attr_function
DELETE_FUNCTION	COMM_DELETE_ATTR_FN
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.

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. 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. Binary-valued variables are of type `LOGICAL`.

Array arguments are indexed from one.

The MPI Fortran binding is inconsistent with the Fortran 90 standard in several respects. These inconsistencies, such as register optimization problems, have implications for user codes that are discussed in detail in Section 16.2.2. They are also inconsistent with Fortran 77.

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 "`#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`.

In some circumstances, MPI permits users to indicate that they do not want a return value. For example, the user may indicate that the status is not filled in. Unlike C and Fortran where this is achieved through a special input value, in C++ this is done by having two bindings where one has the optional argument and one does not.

C++ functions do not return error codes. If the default error handler has been set to MPI::ERRORS_THROW_EXCEPTIONS, the C++ exception mechanism is used to signal an error by throwing an MPI::Exception object.

It should be noted that the default error handler (i.e., MPI::ERRORS_ARE_FATAL) on a given type has not changed. User error handlers are also permitted. MPI::ERRORS_RETURN simply returns control to the calling function; there is no provision for the user to retrieve the error code.

User callback functions that return integer error codes should not throw exceptions; the returned error will be handled by the MPI implementation by invoking the appropriate error handler.

> *Advice to users.* C++ programmers that want to handle MPI errors on their own should use the MPI::ERRORS_THROW_EXCEPTIONS error handler, rather than MPI::ERRORS_RETURN, that is used for that purpose in C. Care should be taken using exceptions in mixed language situations. (*End of advice to users.*)

Opaque object handles must be objects in themselves, and have the assignment and equality operators overridden to perform semantically like their C and Fortran counterparts.

Array arguments are indexed from zero.

Logical flags are of type `bool`.

Choice arguments are pointers of type `void *`.

Address arguments are of MPI-defined integer type MPI::Aint, defined to be an integer of the size needed to hold any valid address on the target architecture. Analogously, MPI::Offset is an integer to hold file offsets.

Most MPI functions are methods of MPI C++ classes. MPI class names are generated from the language neutral MPI types by dropping the `MPI_` prefix and scoping the type within the `MPI` namespace. For example, MPI_DATATYPE becomes `MPI::Datatype`.

The names of MPI functions generally follow the naming rules given. In some circumstances, the MPI function is related to a function defined already for MPI-1 with a name that does not follow the naming conventions. In this circumstance, the language neutral name is in analogy to the MPI name even though this gives an MPI-2 name that violates the naming conventions. The C and Fortran names are the same as the language neutral name in this case. However, the C++ names do reflect the naming rules and can differ from the C and Fortran names. Thus, the analogous name in C++ to the MPI name may be different than the language neutral name. This results in the C++ name differing from the language neutral name. An example of this is the language neutral name of MPI_FINALIZED and a C++ name of MPI::Is_finalized.

In C++, function `typedef`s are made publicly within appropriate classes. However, these declarations then become somewhat cumbersome, as with the following:
{`typedef MPI::Grequest::Query_function();` *(binding deprecated, see Section 15.2)*}

would look like the following:

```
namespace MPI {
  class Request {
    // ...
  };

  class Grequest : public MPI::Request {
    // ...
    typedef Query_function(void* extra_state, MPI::Status& status);
  };
};
```

Rather than including this scaffolding when declaring C++ `typedef`s, we use an abbreviated form. In particular, we explicitly indicate the class and namespace scope for the `typedef` of the function. Thus, the example above is shown in the text as follows:

```
typedef int MPI::Grequest::Query_function(void* extra_state,
                                          MPI::Status& status)
```

The C++ bindings presented in Annex A.4 and throughout this document were generated by applying a simple set of name generation rules to the MPI function specifications. While these guidelines may be sufficient in most cases, they may not be suitable for all situations. In cases of ambiguity or where a specific semantic statement is desired, these guidelines may be superseded as the situation dictates.

1. All functions, types, and constants are declared within the scope of a `namespace` called `MPI`.

2. Arrays of MPI handles are always left in the argument list (whether they are IN or OUT arguments).

3. If the argument list of an MPI function contains a scalar IN handle, and it makes sense to define the function as a method of the object corresponding to that handle, the function is made a member function of the corresponding MPI class. The member functions are named according to the corresponding MPI function name, but without the "MPI_" prefix and without the object name prefix (if applicable). In addition:

   (a) The scalar IN handle is dropped from the argument list, and `this` corresponds to the dropped argument.

   (b) The function is declared `const`.

4. MPI functions are made into class functions (static) when they belong on a class but do not have a unique scalar IN or INOUT parameter of that class.

5. If the argument list contains a single OUT argument that is not of type MPI_STATUS (or an array), that argument is dropped from the list and the function returns that value.

   Example 2.1 The C++ binding for MPI_COMM_SIZE is
   int MPI::Comm::Get_size(void) const.

6. If there are multiple OUT arguments in the argument list, one is chosen as the return value and is removed from the list.

7. If the argument list does not contain any OUT arguments, the function returns `void`.

   Example 2.2 The C++ binding for MPI_REQUEST_FREE is
   void MPI::Request::Free(void)

8. MPI functions to which the above rules do not apply are not members of any class, but are defined in the `MPI` namespace.

   Example 2.3 The C++ binding for MPI_BUFFER_ATTACH is
   void MPI::Attach_buffer(void* buffer, int size).

9. All class names, defined types, and function names have only their first letter capitalized. Defined constants are in all capital letters.

10. Any IN pointer, reference, or array argument must be declared `const`.

11. Handles are passed by reference.

12. Array arguments are denoted with square brackets (`[]`), not pointers, as this is more semantically precise.

2.6.5 Functions and Macros

An implementation is allowed to implement MPI_WTIME, MPI_WTICK, PMPI_WTIME, PMPI_WTICK, and the handle-conversion functions (MPI_Group_f2c, etc.) in Section 16.3.4, and no others, as macros in C.

Advice to implementors. Implementors should document which routines are implemented as macros. (*End of advice to implementors.*)

Advice to users. If these routines are implemented as macros, they will not work with the MPI profiling interface. (*End of advice to users.*)

2.7 Processes

An MPI program consists of autonomous processes, executing their own code, in an MIMD style. The codes executed by each process need not be identical. The processes communicate via calls to MPI communication primitives. Typically, each process executes in its own address space, although shared-memory implementations of MPI are possible.

This document specifies the behavior of a parallel program assuming that only MPI calls are used. The interaction of an MPI program with other possible means of communication, I/O, and process management is not specified. Unless otherwise stated in the specification of the standard, MPI places no requirements on the result of its interaction with external mechanisms that provide similar or equivalent functionality. This includes, but is not limited to, interactions with external mechanisms for process control, shared and remote memory access, file system access and control, interprocess communication, process signaling, and terminal I/O. High quality implementations should strive to make the results of such interactions intuitive to users, and attempt to document restrictions where deemed necessary.

Advice to implementors. Implementations that support such additional mechanisms for functionality supported within MPI are expected to document how these interact with MPI. (*End of advice to implementors.*)

The interaction of MPI and threads is defined in Section 12.4.

2.8 Error Handling

MPI provides the user with reliable message transmission. A message sent is always received correctly, and the user does not need to check for transmission errors, time-outs, or other error conditions. In other words, MPI does not provide mechanisms for dealing with failures in the communication system. If the MPI implementation is built on an unreliable underlying mechanism, then it is the job of the implementor of the MPI subsystem to insulate the user from this unreliability, or to reflect unrecoverable errors as failures. Whenever possible, such failures will be reflected as errors in the relevant communication call. Similarly, MPI itself provides no mechanisms for handling processor failures.

Of course, MPI programs may still be erroneous. A **program error** can occur when an MPI call is made with an incorrect argument (non-existing destination in a send operation, buffer too small in a receive operation, etc.). This type of error would occur in any implementation. In addition, a **resource error** may occur when a program exceeds the amount of available system resources (number of pending messages, system buffers, etc.). The occurrence of this type of error depends on the amount of available resources in the system and the resource allocation mechanism used; this may differ from system to system. A high-quality implementation will provide generous limits on the important resources so as to alleviate the portability problem this represents.

In C and Fortran, almost all MPI calls return a code that indicates successful completion of the operation. Whenever possible, MPI calls return an error code if an error occurred during the call. By default, an error detected during the execution of the MPI library causes the parallel computation to abort, except for file operations. However, MPI provides mechanisms for users to change this default and to handle recoverable errors. The user may specify that no error is fatal, and handle error codes returned by MPI calls by himself or herself. Also, the user may provide his or her own error-handling routines, which will be invoked whenever an MPI call returns abnormally. The MPI error handling facilities are described in Section 8.3. The return values of C++ functions are not error codes. If the default error handler has been set to MPI::ERRORS_THROW_EXCEPTIONS, the C++ exception mechanism is used to signal an error by throwing an MPI::Exception object. See also Section 16.1.8 on page 506.

Several factors limit the ability of MPI calls to return with meaningful error codes when an error occurs. MPI may not be able to detect some errors; other errors may be too expensive to detect in normal execution mode; finally some errors may be "catastrophic" and may prevent MPI from returning control to the caller in a consistent state.

Another subtle issue arises because of the nature of asynchronous communications: MPI calls may initiate operations that continue asynchronously after the call returned. Thus, the operation may return with a code indicating successful completion, yet later cause an error exception to be raised. If there is a subsequent call that relates to the same operation (e.g., a call that verifies that an asynchronous operation has completed) then the error argument associated with this call will be used to indicate the nature of the error. In a few cases, the error may occur after all calls that relate to the operation have completed, so that no error value can be used to indicate the nature of the error (e.g., an error on the receiver in a send with the ready mode). Such an error must be treated as fatal, since information cannot be returned for the user to recover from it.

This document does not specify the state of a computation after an erroneous MPI call has occurred. The desired behavior is that a relevant error code be returned, and the effect of the error be localized to the greatest possible extent. E.g., it is highly desirable that an erroneous receive call will not cause any part of the receiver's memory to be overwritten, beyond the area specified for receiving the message.

Implementations may go beyond this document in supporting in a meaningful manner MPI calls that are defined here to be erroneous. For example, MPI specifies strict type matching rules between matching send and receive operations: it is erroneous to send a floating point variable and receive an integer. Implementations may go beyond these type matching rules, and provide automatic type conversion in such situations. It will be helpful to generate warnings for such non-conforming behavior.

MPI defines a way for users to create new error codes as defined in Section 8.5.

2.9 Implementation Issues

There are a number of areas where an MPI implementation may interact with the operating environment and system. While MPI does not mandate that any services (such as signal handling) be provided, it does strongly suggest the behavior to be provided if those services are available. This is an important point in achieving portability across platforms that provide the same set of services.

2.9.1 Independence of Basic Runtime Routines

MPI programs require that library routines that are part of the basic language environment (such as `write` in Fortran and `printf` and `malloc` in ISO C) and are executed after MPI_INIT and before MPI_FINALIZE operate independently and that their *completion* is independent of the action of other processes in an MPI program.

Note that this in no way prevents the creation of library routines that provide parallel services whose operation is collective. However, the following program is expected to complete in an ISO C environment regardless of the size of MPI_COMM_WORLD (assuming that `printf` is available at the executing nodes).

```
int rank;
MPI_Init((void *)0, (void *)0);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) printf("Starting program\n");
MPI_Finalize();
```

The corresponding Fortran and C++ programs are also expected to complete.

An example of what is *not* required is any particular ordering of the action of these routines when called by several tasks. For example, MPI makes neither requirements nor recommendations for the output from the following program (again assuming that I/O is available at the executing nodes).

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
printf("Output from task rank %d\n", rank);
```

In addition, calls that fail because of resource exhaustion or other error are not considered a violation of the requirements here (however, they are required to complete, just not to complete successfully).

2.9.2 Interaction with Signals

MPI does not specify the interaction of processes with signals and does not require that MPI be signal safe. The implementation may reserve some signals for its own use. It is required that the implementation document which signals it uses, and it is strongly recommended that it not use `SIGALRM`, `SIGFPE`, or `SIGIO`. Implementations may also prohibit the use of MPI calls from within signal handlers.

In multithreaded environments, users can avoid conflicts between signals and the MPI library by catching signals only on threads that do not execute MPI calls. High quality single-threaded implementations will be signal safe: an MPI call suspended by a signal will resume and complete normally after the signal is handled.

2.10 Examples

The examples in this document are for illustration purposes only. They are not intended to specify the standard. Furthermore, the examples have not been carefully checked or verified.

Chapter 3

Point-to-Point Communication

3.1 Introduction

Sending and receiving of messages by processes is the basic MPI communication mechanism. The basic point-to-point communication operations are **send** and **receive**. Their use is illustrated in the example below.

```
#include "mpi.h"
int main( int argc, char **argv )
{
  char message[20];
  int myrank;
  MPI_Status status;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
  if (myrank == 0)    /* code for process zero */
  {
      strcpy(message,"Hello, there");
      MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
  }
  else if (myrank == 1)  /* code for process one */
  {
      MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
      printf("received :%s:\n", message);
  }
  MPI_Finalize();
}
```

In this example, process zero (myrank = 0) sends a message to process one using the **send** operation MPI_SEND. The operation specifies a **send buffer** in the sender memory from which the message data is taken. In the example above, the send buffer consists of the storage containing the variable message in the memory of process zero. The location, size and type of the send buffer are specified by the first three parameters of the send operation. The message sent will contain the 13 characters of this variable. In addition, the send operation associates an **envelope** with the message. This envelope specifies the message destination and contains distinguishing information that can be used by the **receive**

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

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

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

3.2.2 Message Data

The send buffer specified by the MPI_SEND operation consists of count successive entries of the type indicated by datatype, starting with the entry at address buf. Note that we specify the message length in terms of number of *elements*, not number of *bytes*. The former is machine independent and closer to the application level.

The data part of the message consists of a sequence of count values, each of the type indicated by datatype. count may be zero, in which case the data part of the message is empty. The basic datatypes that can be specified for message data values correspond to the basic datatypes of the host language. Possible values of this argument for Fortran and the corresponding Fortran types are listed in Table 3.1.

MPI datatype	Fortran datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

Table 3.1: Predefined MPI datatypes corresponding to Fortran datatypes

Possible values for this argument for C and the corresponding C types are listed in Table 3.2.

The datatypes MPI_BYTE and MPI_PACKED do not correspond to a Fortran or C datatype. A value of type MPI_BYTE consists of a byte (8 binary digits). A byte is uninterpreted and is different from a character. Different machines may have different representations for characters, or may use more than one byte to represent characters. On the other hand, a byte has the same binary value on all machines. The use of the type MPI_PACKED is explained in Section 4.2.

MPI requires support of these datatypes, which match the basic datatypes of Fortran and ISO C. Additional MPI datatypes should be provided if the host language has additional data types: MPI_DOUBLE_COMPLEX for double precision complex in Fortran declared to be of type DOUBLE COMPLEX; MPI_REAL2, MPI_REAL4 and MPI_REAL8 for Fortran reals, declared to be of type REAL*2, REAL*4 and REAL*8, respectively; MPI_INTEGER1 MPI_INTEGER2 and MPI_INTEGER4 for Fortran integers, declared to be of type INTEGER*1, INTEGER*2 and INTEGER*4, respectively; etc.

> *Rationale.* One goal of the design is to allow for MPI to be implemented as a library, with no need for additional preprocessing or compilation. Thus, one cannot assume that a communication call has information on the datatype of variables in the communication buffer; this information must be supplied by an explicit argument. The need for such datatype information will become clear in Section 3.3.2. (*End of rationale.*)

> *Rationale.* The datatypes MPI_C_BOOL, MPI_INT8_T, MPI_INT16_T, MPI_INT32_T, MPI_UINT8_T, MPI_UINT16_T, MPI_UINT32_T, MPI_C_COMPLEX,

MPI datatype	C datatype
MPI_CHAR	`char` (treated as printable character)
MPI_SHORT	`signed short int`
MPI_INT	`signed int`
MPI_LONG	`signed long int`
MPI_LONG_LONG_INT	`signed long long int`
MPI_LONG_LONG (as a synonym)	`signed long long int`
MPI_SIGNED_CHAR	`signed char` (treated as integral value)
MPI_UNSIGNED_CHAR	`unsigned char` (treated as integral value)
MPI_UNSIGNED_SHORT	`unsigned short int`
MPI_UNSIGNED	`unsigned int`
MPI_UNSIGNED_LONG	`unsigned long int`
MPI_UNSIGNED_LONG_LONG	`unsigned long long int`
MPI_FLOAT	`float`
MPI_DOUBLE	`double`
MPI_LONG_DOUBLE	`long double`
MPI_WCHAR	`wchar_t` (defined in `<stddef.h>`) (treated as printable character)
MPI_C_BOOL	`_Bool`
MPI_INT8_T	`int8_t`
MPI_INT16_T	`int16_t`
MPI_INT32_T	`int32_t`
MPI_INT64_T	`int64_t`
MPI_UINT8_T	`uint8_t`
MPI_UINT16_T	`uint16_t`
MPI_UINT32_T	`uint32_t`
MPI_UINT64_T	`uint64_t`
MPI_C_COMPLEX	`float _Complex`
MPI_C_FLOAT_COMPLEX (as a synonym)	`float _Complex`
MPI_C_DOUBLE_COMPLEX	`double _Complex`
MPI_C_LONG_DOUBLE_COMPLEX	`long double _Complex`
MPI_BYTE	
MPI_PACKED	

Table 3.2: Predefined MPI datatypes corresponding to C datatypes

MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX, and MPI_C_LONG_DOUBLE_COMPLEX have no corresponding C++ bindings. This was intentionally done to avoid potential collisions with the C preprocessor and namespaced C++ names. C++ applications can use the C bindings with no loss of functionality. (*End of rationale.*)

ticket265. The datatypes MPI_AINT [and], MPI_OFFSET , and MPI_COUNT correspond to the ticket265.

MPI datatype	C datatype	Fortran datatype
MPI_AINT	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND)
MPI_OFFSET	MPI_Offset	INTEGER (KIND=MPI_OFFSET_KIND)
[ticket265.] MPI_COUNT	[ticket265.]MPI_Count	[ticket265.]INTEGER (KIND=MPI_COUNT_KIND)

Table 3.3: Predefined MPI datatypes corresponding to both C and Fortran datatypes

MPI-defined C types MPI_Aint [and], MPI_Offset , and MPI_COUNT and their Fortran equivalents INTEGER (KIND=MPI_ADDRESS_KIND) [and], INTEGER (KIND= MPI_OFFSET_KIND) , and INTEGER (KIND=MPI_COUNT_KIND) . This is described in Table 3.3. See Section 16.3.10 for information on interlanguage communication with these types. ticket265. ticket265. ticket265. ticket265.

3.2.3 Message Envelope

In addition to the data part, messages carry information that can be used to distinguish messages and selectively receive them. This information consists of a fixed number of fields, which we collectively call the **message envelope**. These fields are

source
destination
tag
communicator

The message source is implicitly determined by the identity of the message sender. The other fields are specified by arguments in the send operation.

The message destination is specified by the dest argument.

The integer-valued message tag is specified by the tag argument. This integer can be used by the program to distinguish different types of messages. The range of valid tag values is 0,...,UB, where the value of UB is implementation dependent. It can be found by querying the value of the attribute MPI_TAG_UB, as described in Chapter 8. MPI requires that UB be no less than 32767.

The comm argument specifies the **communicator** that is used for the send operation. Communicators are explained in Chapter 6; below is a brief summary of their usage.

A communicator specifies the communication context for a communication operation. Each communication context provides a separate "communication universe:" messages are always received within the context they were sent, and messages sent in different contexts do not interfere.

The communicator also specifies the set of processes that share this communication context. This **process group** is ordered and processes are identified by their rank within this group. Thus, the range of valid values for dest is 0, ... , n-1, where n is the number of processes in the group. (If the communicator is an inter-communicator, then destinations are identified by their rank in the remote group. See Chapter 6.)

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.

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.*)

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 subsections "Problems Due to Data Copying and Sequence Association," and "A Problem with Register Optimization" in Section 16.2.2 on pages 512 and 515. (*End of advice to users.*)

The following auxillary [function provides]functions provide useful information on derived datatypes. ticket265.

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)
    INTEGER DATATYPE, SIZE, IERROR
```

{`int MPI::Datatype::Get_size() const` *(binding deprecated, see Section 15.2)* } ticket265.

MPI_TYPE_SIZE_X(datatype, size)

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

```
int MPI_Type_size_x(MPI_Datatype datatype, MPI_Count *size)

MPI_TYPE_SIZE_X(DATATYPE, SIZE, IERROR)
    INTEGER DATATYPE, IERROR
    INTEGER (KIND=MPI_COUNT_KIND) SIZE
```

ticket265. MPI_TYPE_SIZE and MPI_TYPE_SIZE_X [returns]set the value of size to the total size, ticket265. 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 ticket265. times in the datatype are counted with their multiplicity.

If the total size of the datatype can not be expressed by the size parameter, then MPI_TYPE_SIZE and MPI_TYPE_SIZE_X set the value of size to MPI_UNDEFINED.

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 97. 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 overide the alignment rules that are used to compute upper bounds and extents. E.g., a C compiler may allow the user to overide 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 ($extent$(MPI_LB) = $extent$(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 = (MPI_LB, MPI_INT, 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 {(lb, -3), (int, 0), (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 {(lb, -3), (int, 0), (int,9), (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

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

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

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

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 $\max_i k_i$.

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

4.1.7 Extent and Bounds of Datatypes

ticket265.

[The following function replaces]MPI_TYPE_GET_EXTENT and MPI_TYPE_GET_EXTENT_X replace the three functions MPI_TYPE_UB, MPI_TYPE_LB and MPI_TYPE_EXTENT. [It also returns]and also return address sized integers[,] in the Fortran binding. The use of MPI_TYPE_UB, MPI_TYPE_LB and MPI_TYPE_EXTENT is deprecated.

ticket265.

ticket265.

MPI_TYPE_GET_EXTENT(datatype, lb, extent)

IN	datatype	datatype to get information on (handle)
OUT	lb	lower bound of datatype (integer)
OUT	extent	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)
    INTEGER DATATYPE, IERROR
    INTEGER(KIND = MPI_ADDRESS_KIND) LB, EXTENT
```

```
{void MPI::Datatype::Get_extent(MPI::Aint& lb, MPI::Aint& extent)
              const(binding deprecated, see Section 15.2) }
```

ticket265.

MPI_TYPE_GET_EXTENT_X(datatype, lb, extent)

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

```
int MPI_Type_get_extent_x(MPI_Datatype datatype, MPI_Count *lb,
              MPI_Count *extent)
```

```
MPI_TYPE_GET_EXTENT_X(DATATYPE, LB, EXTENT, IERROR)
    INTEGER DATATYPE, IERROR
    INTEGER(KIND = MPI_COUNT_KIND) LB, EXTENT
```

Returns the lower bound and the extent of datatype (as defined in Section 4.1.6 on page 96).

MPI allows one to change the extent of a datatype, using lower bound and upper bound markers (MPI_LB and MPI_UB). This is useful, as it allows to control the stride of successive datatypes that are replicated by datatype constructors, or are replicated by the count argument in a send or receive call. However, the current mechanism for achieving it is painful; also it is restrictive. MPI_LB and MPI_UB are "sticky": once present in a datatype, they cannot be overridden (e.g., the upper bound can be moved up, by adding a new MPI_UB marker, but cannot be moved down below an existing MPI_UB marker). A new type constructor is provided to facilitate these changes. The use of MPI_LB and MPI_UB is deprecated.

MPI_TYPE_CREATE_RESIZED(oldtype, lb, extent, newtype)

IN	oldtype	input datatype (handle)
IN	lb	new lower bound of datatype (integer)
IN	extent	new extent of datatype (integer)
OUT	newtype	output datatype (handle)

```
int MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb, MPI_Aint
              extent, MPI_Datatype *newtype)

MPI_TYPE_CREATE_RESIZED(OLDTYPE, LB, EXTENT, NEWTYPE, IERROR)
    INTEGER OLDTYPE, NEWTYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT

{MPI::Datatype MPI::Datatype::Create_resized(const MPI::Aint lb,
              const MPI::Aint extent) const(binding deprecated, see Section 15.2) }
```

Returns in newtype a handle to a new datatype that is identical to oldtype, except that the lower bound of this new datatype is set to be lb, and its upper bound is set to be lb + extent. Any previous **lb** and **ub** markers are erased, and a new pair of lower bound and upper bound markers are put in the positions indicated by the lb and extent arguments. This affects the behavior of the datatype when used in communication operations, with count > 1, and when used in the construction of new derived datatypes.

Advice to users. It is strongly recommended that users use these two new functions, rather than the old MPI-1 functions to set and access lower bound, upper bound and extent of datatypes. (*End of advice to users.*)

4.1.8 True Extent of Datatypes

Suppose we implement gather (see also Section 5.5 on page 140) as a spanning tree implemented on top of point-to-point routines. Since the receive buffer is only valid on the root process, one will need to allocate some temporary space for receiving data on intermediate nodes. However, the datatype extent cannot be used as an estimate of the amount of space that needs to be allocated, if the user has modified the extent using the MPI_UB and MPI_LB values. [A function is] MPI_TYPE_GET_TRUE_EXTENT and ticket265.

ticket265. MPI_TYPE_GET_TRUE_EXTENT_X are provided which [returns]return the true extent of the datatype.

MPI_TYPE_GET_TRUE_EXTENT(datatype, true_lb, true_extent)

IN	datatype	datatype to get information on (handle)
OUT	true_lb	true lower bound of datatype (integer)
OUT	true_extent	true size of datatype (integer)

```
int MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint *true_lb,
              MPI_Aint *true_extent)
```

```
MPI_TYPE_GET_TRUE_EXTENT(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)
    INTEGER DATATYPE, IERROR
    INTEGER(KIND = MPI_ADDRESS_KIND) TRUE_LB, TRUE_EXTENT
```

```
{void MPI::Datatype::Get_true_extent(MPI::Aint& true_lb,
              MPI::Aint& true_extent) const(binding deprecated, see Section 15.2) }
```

ticket265.

MPI_TYPE_GET_TRUE_EXTENT_X(datatype, true_lb, true_extent)

IN	datatype	datatype to get information on (handle)
OUT	true_lb	true lower bound of datatype (integer)
OUT	true_extent	true size of datatype (integer)

```
int MPI_Type_get_true_extent_x(MPI_Datatype datatype, MPI_Count *true_lb,
              MPI_Count *true_extent)
```

```
MPI_TYPE_GET_TRUE_EXTENT_X(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)
    INTEGER DATATYPE, IERROR
    INTEGER(KIND = MPI_COUNT_KIND) TRUE_LB, TRUE_EXTENT
```

true_lb returns the offset of the lowest unit of store which is addressed by the datatype, i.e., the lower bound of the corresponding typemap, ignoring MPI_LB markers. true_extent returns the true size of the datatype, i.e., the extent of the corresponding typemap, ignoring MPI_LB and MPI_UB markers, and performing no rounding for alignment. If the typemap associated with datatype is

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

Then

$$true\_lb(Typemap) = min_j\{disp_j \ : \ type_j \neq \mathbf{lb}, \mathbf{ub}\},$$

$$true\_ub(Typemap) = max_j\{disp_j + sizeof(type_j) \ : \ type_j \neq \mathbf{lb}, \mathbf{ub}\},$$

and

$$true\_extent(Typemap) = true\_ub(Typemap) - true\_lb(typemap).$$

(Readers should compare this with the definitions in Section 4.1.6 on page 96 and Section 4.1.7 on page 97, which describe the function MPI_TYPE_GET_EXTENT.)

The true_extent is the minimum number of bytes of memory necessary to hold a datatype, uncompressed.

4.1.9 Commit and Free

A datatype object has to be **committed** before it can be used in a communication. As an argument in datatype constructors, uncommitted and also committed datatypes can be used. There is no need to commit basic datatypes. They are "pre-committed."

MPI_TYPE_COMMIT(datatype)

INOUT	datatype	datatype that is committed (handle)

```
int MPI_Type_commit(MPI_Datatype *datatype)

MPI_TYPE_COMMIT(DATATYPE, IERROR)
    INTEGER DATATYPE, IERROR
```

{`void MPI::Datatype::Commit()` *(binding deprecated, see Section 15.2)* }

The commit operation commits the datatype, that is, the formal description of a communication buffer, not the content of that buffer. Thus, after a datatype has been committed, it can be repeatedly reused to communicate the changing content of a buffer or, indeed, the content of different buffers, with different starting addresses.

> *Advice to implementors.* The system may "compile" at commit time an internal representation for the datatype that facilitates communication, e.g. change from a compacted representation to a flat representation of the datatype, and select the most convenient transfer mechanism. (*End of advice to implementors.*)

MPI_TYPE_COMMIT will accept a committed datatype; in this case, it is equivalent to a no-op.

Example 4.10 The following code fragment gives examples of using MPI_TYPE_COMMIT.

```
INTEGER type1, type2
CALL MPI_TYPE_CONTIGUOUS(5, MPI_REAL, type1, ierr)
             ! new type object created
CALL MPI_TYPE_COMMIT(type1, ierr)
             ! now type1 can be used for communication
type2 = type1
             ! type2 can be used for communication
             ! (it is a handle to same object as type1)
CALL MPI_TYPE_VECTOR(3, 5, 4, MPI_REAL, type1, ierr)
             ! new uncommitted type object created
CALL MPI_TYPE_COMMIT(type1, ierr)
             ! now type1 can be used anew for communication
```

MPI_TYPE_FREE(datatype)

INOUT	datatype	datatype that is freed (handle)

```
int MPI_Type_free(MPI_Datatype *datatype)
```

```
MPI_TYPE_FREE(DATATYPE, IERROR)
    INTEGER DATATYPE, IERROR
```

{`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 then 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

MPI_TYPE_DUP(type, newtype)

IN	type	datatype (handle)
OUT	newtype	copy of type (handle)

```
int MPI_Type_dup(MPI_Datatype type, MPI_Datatype *newtype)
```

```
MPI_TYPE_DUP(TYPE, NEWTYPE, IERROR)
    INTEGER TYPE, NEWTYPE, IERROR
```

{`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 type and any copied cached information, see Section 6.7.4 on page 263. 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 type.

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,

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

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 $n \cdot$ count entries, where entry $i \cdot n + j$ is at location $addr_{i,j} =$ buf $+ extent \cdot i + disp_j$ and has type $type_j$, for $i = 0, ...,$ count $- 1$ and $j = 0, ..., n - 1$. 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 $n \cdot$ 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,

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

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 $n \cdot$ count entries, where entry $i \cdot n + j$ is at location buf $+ extent \cdot i + disp_j$ and has type $type_j$. If the incoming message consists of k elements, then we must have $k \leq n \cdot$ 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.

Example 4.11 This example shows that type matching is defined in terms of the basic types that a derived type consists of.

```
...
CALL MPI_TYPE_CONTIGUOUS( 2, MPI_REAL, type2, ...)
CALL MPI_TYPE_CONTIGUOUS( 4, MPI_REAL, type4, ...)
CALL MPI_TYPE_CONTIGUOUS( 2, type2, type22, ...)
...
CALL MPI_SEND( a, 4, MPI_REAL, ...)
CALL MPI_SEND( a, 2, type2, ...)
CALL MPI_SEND( a, 1, type22, ...)
CALL MPI_SEND( a, 1, type4, ...)
...
CALL MPI_RECV( a, 4, MPI_REAL, ...)
CALL MPI_RECV( a, 2, type2, ...)
CALL MPI_RECV( a, 1, type22, ...)
CALL MPI_RECV( a, 1, type4, ...)
```

Each of the sends matches any of the receives.

A datatype may specify overlapping entries. The use of such a datatype in a receive operation is erroneous. (This is erroneous even if the actual message received is short enough not to write any entry more than once.)

Suppose that MPI_RECV(buf, count, datatype, dest, tag, comm, status) is executed, where datatype has type map,

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

The received message need not fill all the receive buffer, nor does it need to fill a number of locations which is a multiple of n. Any number, k, of basic elements can be received, where $0 \leq k \leq \text{count} \cdot n$. The number of basic elements received can be retrieved from status using the query function MPI_GET_ELEMENTS.

MPI_GET_ELEMENTS(status, datatype, count)

IN	status	return status of receive operation (Status)
IN	datatype	datatype used by receive operation (handle)
OUT	count	number of received basic elements (integer)

```
int MPI_Get_elements(MPI_Status *status, MPI_Datatype datatype, int *count)
```

```
MPI_GET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
```

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

ticket265.

MPI_GET_ELEMENTS_X(status, datatype, count)

IN	status	return status of receive operation (Status)
IN	datatype	datatype used by receive operation (handle)
OUT	count	number of received basic elements (integer)

```
int MPI_Get_elements_x(MPI_Status *status, MPI_Datatype datatype,
              MPI_Count *count)

MPI_GET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR
    INTEGER (KIND=MPI_COUNT_KIND) COUNT
```

The previously defined function MPI_GET_COUNT (Section 3.2.5), has a different behavior. It returns the number of "top-level entries" received, i.e. the number of "copies" of type datatype. In the previous example, MPI_GET_COUNT may return any integer value k, where $0 \leq k \leq$ count. If MPI_GET_COUNT returns k, then the number of basic elements received (and the value returned by MPI_GET_ELEMENTS or MPI_GET_ELEMENTS_X) is $n \cdot k$. If the number of basic elements received is not a multiple of n, that is, if the receive operation has not received an integral number of datatype "copies," then MPI_GET_COUNT [returns]sets the value of count to MPI_UNDEFINED. The datatype argument should match the argument provided by the receive call that set the status variable.

If the number of basic elements received can not be expressed by the count parameter, then MPI_GET_ELEMENTS and MPI_GET_ELEMENTS_X set the value of count to MPI_UNDEFINED.

Example 4.12 Usage of MPI_GET_COUNT and MPI_GET_ELEMENTS.

```
...
CALL MPI_TYPE_CONTIGUOUS(2, MPI_REAL, Type2, ierr)
CALL MPI_TYPE_COMMIT(Type2, ierr)
...
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
      CALL MPI_SEND(a, 2, MPI_REAL, 1, 0, comm, ierr)
      CALL MPI_SEND(a, 3, MPI_REAL, 1, 0, comm, ierr)
ELSE IF (rank.EQ.1) THEN
      CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
      CALL MPI_GET_COUNT(stat, Type2, i, ierr)     ! returns i=1
      CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr)  ! returns i=2
      CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
      CALL MPI_GET_COUNT(stat, Type2, i, ierr)     ! returns i=MPI_UNDEFINED
      CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr)  ! returns i=3
END IF
```

The [function]functions MPI_GET_ELEMENTS and MPI_GET_ELEMENTS_X can also be used after a probe to find the number of elements in the probed message. Note that the [two] functions MPI_GET_COUNT [and], MPI_GET_ELEMENTS and MPI_GET_ELEMENTS_X return the same values when they are used with basic datatypes.

Rationale. The extension given to the definition of MPI_GET_COUNT seems natural: one would expect this function to return the value of the count argument, when the receive buffer is filled. Sometimes datatype represents a basic unit of data one wants to transfer, for example, a record in an array of records (structures). One should be able to find out how many components were received without bothering to divide by the number of elements in each component. However, on other occasions, datatype is used to define a complex layout of data in the receiver memory, and does not represent a basic unit of data for transfers. In such cases, one needs to use the function MPI_GET_ELEMENTS or MPI_GET_ELEMENTS_X . (*End of rationale.*) ticket265.

Advice to implementors. The definition implies that a receive cannot change the value of storage outside the entries defined to compose the communication buffer. In particular, the definition implies that padding space in a structure should not be modified when such a structure is copied from one process to another. This would prevent the obvious optimization of copying the structure, together with the padding, as one contiguous block. The implementation is free to do this optimization when it does not impact the outcome of the computation. The user can "force" this optimization by explicitly including padding as part of the message. (*End of advice to implementors.*)

4.1.12 Correct Use of Addresses

Successively declared variables in C or Fortran are not necessarily stored at contiguous locations. Thus, care must be exercised that displacements do not cross from one variable to another. Also, in machines with a segmented address space, addresses are not unique and address arithmetic has some peculiar properties. Thus, the use of **addresses**, that is, displacements relative to the start address MPI_BOTTOM, has to be restricted.

Variables belong to the same **sequential storage** if they belong to the same array, to the same COMMON block in Fortran, or to the same structure in C. Valid addresses are defined recursively as follows:

1. The function MPI_GET_ADDRESS returns a valid address, when passed as argument a variable of the calling program.
2. The buf argument of a communication function evaluates to a valid address, when passed as argument a variable of the calling program.
3. If v is a valid address, and i is an integer, then v+i is a valid address, provided v and v+i are in the same sequential storage.
4. If v is a valid address then MPI_BOTTOM + v is a valid address.

A correct program uses only valid addresses to identify the locations of entries in communication buffers. Furthermore, if u and v are two valid addresses, then the (integer) difference u - v can be computed only if both u and v are in the same sequential storage. No other arithmetic operations can be meaningfully executed on addresses.

The rules above impose no constraints on the use of derived datatypes, as long as they are used to define a communication buffer that is wholly contained within the same sequential storage. However, the construction of a communication buffer that contains variables that are not within the same sequential storage must obey certain restrictions. Basically, a communication buffer with variables that are not within the same sequential

storage can be used only by specifying in the communication call buf = MPI_BOTTOM, count = 1, and using a datatype argument where all displacements are valid (absolute) addresses.

Advice to users. It is not expected that MPI implementations will be able to detect erroneous, "out of bound" displacements — unless those overflow the user address space — since the MPI call may not know the extent of the arrays and records in the host program. (*End of advice to users.*)

Advice to implementors. There is no need to distinguish (absolute) addresses and (relative) displacements on a machine with contiguous address space: MPI_BOTTOM is zero, and both addresses and displacements are integers. On machines where the distinction is required, addresses are recognized as expressions that involve MPI_BOTTOM. (*End of advice to implementors.*)

4.1.13 Decoding a Datatype

MPI datatype objects allow users to specify an arbitrary layout of data in memory. There are several cases where accessing the layout information in opaque datatype objects would be useful. The opaque datatype object has found a number of uses outside MPI. Furthermore, a number of tools wish to display internal information about a datatype. To achieve this, datatype decoding functions are provided. The two functions in this section are used together to decode datatypes to recreate the calling sequence used in their initial definition. These can be used to allow a user to determine the type map and type signature of a datatype.

MPI_TYPE_GET_ENVELOPE(datatype, num_integers, num_addresses, num_datatypes, combiner)

IN	datatype	datatype to access (handle)
OUT	num_integers	number of input integers used in the call constructing combiner (non-negative integer)
OUT	num_addresses	number of input addresses used in the call constructing combiner (non-negative integer)
OUT	num_datatypes	number of input datatypes used in the call constructing combiner (non-negative integer)
OUT	combiner	combiner (state)

```
int MPI_Type_get_envelope(MPI_Datatype datatype, int *num_integers,
             int *num_addresses, int *num_datatypes, int *combiner)

MPI_TYPE_GET_ENVELOPE(DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES,
             COMBINER, IERROR)
    INTEGER DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES, COMBINER,
    IERROR

{void MPI::Datatype::Get_envelope(int& num_integers, int& num_addresses,
             int& num_datatypes, int& combiner) const(binding deprecated, see
             Section 15.2) }
```

For the given datatype, MPI_TYPE_GET_ENVELOPE returns information on the number and type of input arguments used in the call that created the datatype. The number-of-arguments values returned can be used to provide sufficiently large arrays in the decoding routine MPI_TYPE_GET_CONTENTS. This call and the meaning of the returned values is described below. The combiner reflects the MPI datatype constructor call that was used in creating datatype.

Rationale. By requiring that the combiner reflect the constructor used in the creation of the datatype, the decoded information can be used to effectively recreate the calling sequence used in the original creation. One call is effectively the same as another when the information obtained from MPI_TYPE_GET_CONTENTS may be used with either to produce the same outcome. C calls MPI_Type_hindexed and MPI_Type_create_hindexed are always effectively the same while the Fortran call MPI_TYPE_HINDEXED will be different than either of these in some MPI implementations. This is the most useful information and was felt to be reasonable even though it constrains implementations to remember the original constructor sequence even if the internal representation is different.

The decoded information keeps track of datatype duplications. This is important as one needs to distinguish between a predefined datatype and a dup of a predefined datatype. The former is a constant object that cannot be freed, while the latter is a derived datatype that can be freed. (*End of rationale.*)

The list below has the values that can be returned in combiner on the left and the call associated with them on the right.

If combiner is MPI_COMBINER_NAMED then datatype is a named predefined datatype.

For deprecated calls with address arguments, we sometimes need to differentiate whether the call used an integer or an address size argument. For example, there are two combiners for hvector: MPI_COMBINER_HVECTOR_INTEGER and MPI_COMBINER_HVECTOR. The former is used if it was the MPI-1 call from Fortran, and the latter is used if it was the MPI-1 call from C or C++. However, on systems where MPI_ADDRESS_KIND = MPI_INTEGER_KIND (i.e., where integer arguments and address size arguments are the same), the combiner MPI_COMBINER_HVECTOR may be returned for a datatype constructed by a call to MPI_TYPE_HVECTOR from Fortran. Similarly, MPI_COMBINER_HINDEXED may be returned for a datatype constructed by a call to MPI_TYPE_HINDEXED from Fortran, and MPI_COMBINER_STRUCT may be returned for a datatype constructed by a call to MPI_TYPE_STRUCT from Fortran. On such systems, one need not differentiate constructors that take address size arguments from constructors that take integer arguments, since these are the same. The preferred calls all use address sized arguments so two combiners are not required for them.

Rationale. For recreating the original call, it is important to know if address information may have been truncated. The deprecated calls from Fortran for a few routines could be subject to truncation in the case where the default `INTEGER` size is smaller than the size of an address. (*End of rationale.*)

The actual arguments used in the creation call for a datatype can be obtained from the call:

MPI_COMBINER_NAMED	a named predefined datatype
MPI_COMBINER_DUP	MPI_TYPE_DUP
MPI_COMBINER_CONTIGUOUS	MPI_TYPE_CONTIGUOUS
MPI_COMBINER_VECTOR	MPI_TYPE_VECTOR
MPI_COMBINER_HVECTOR_INTEGER	MPI_TYPE_HVECTOR from Fortran
MPI_COMBINER_HVECTOR	MPI_TYPE_HVECTOR from C or C++ and in some case Fortran or MPI_TYPE_CREATE_HVECTOR
MPI_COMBINER_INDEXED	MPI_TYPE_INDEXED
MPI_COMBINER_HINDEXED_INTEGER	MPI_TYPE_HINDEXED from Fortran
MPI_COMBINER_HINDEXED	MPI_TYPE_HINDEXED from C or C++ and in some case Fortran or MPI_TYPE_CREATE_HINDEXED
MPI_COMBINER_INDEXED_BLOCK	MPI_TYPE_CREATE_INDEXED_BLOCK
MPI_COMBINER_STRUCT_INTEGER	MPI_TYPE_STRUCT from Fortran
MPI_COMBINER_STRUCT	MPI_TYPE_STRUCT from C or C++ and in some case Fortran or MPI_TYPE_CREATE_STRUCT
MPI_COMBINER_SUBARRAY	MPI_TYPE_CREATE_SUBARRAY
MPI_COMBINER_DARRAY	MPI_TYPE_CREATE_DARRAY
MPI_COMBINER_F90_REAL	MPI_TYPE_CREATE_F90_REAL
MPI_COMBINER_F90_COMPLEX	MPI_TYPE_CREATE_F90_COMPLEX
MPI_COMBINER_F90_INTEGER	MPI_TYPE_CREATE_F90_INTEGER
MPI_COMBINER_RESIZED	MPI_TYPE_CREATE_RESIZED

Table 4.1: combiner values returned from MPI_TYPE_GET_ENVELOPE

MPI_TYPE_GET_CONTENTS(datatype, max_integers, max_addresses, max_datatypes, array_of_integers, array_of_addresses, array_of_datatypes)

IN	datatype	datatype to access (handle)
IN	max_integers	number of elements in array_of_integers (non-negative integer)
IN	max_addresses	number of elements in array_of_addresses (non-negative integer)
IN	max_datatypes	number of elements in array_of_datatypes (non-negative integer)
OUT	array_of_integers	contains integer arguments used in constructing datatype (array of integers)
OUT	array_of_addresses	contains address arguments used in constructing datatype (array of integers)
OUT	array_of_datatypes	contains datatype arguments used in constructing datatype (array of handles)

```
int MPI_Type_get_contents(MPI_Datatype datatype, int max_integers,
              int max_addresses, int max_datatypes, int array_of_integers[],
              MPI_Aint array_of_addresses[],
              MPI_Datatype array_of_datatypes[])

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(*)

{void MPI::Datatype::Get_contents(int max_integers, int max_addresses,
              int max_datatypes, int array_of_integers[],
              MPI::Aint array_of_addresses[],
              MPI::Datatype array_of_datatypes[]) const(binding deprecated, see
              Section 15.2) }
```

datatype must be a predefined unnamed or a derived datatype; the call is erroneous if datatype is a predefined named datatype.

The values given for max_integers, max_addresses, and max_datatypes must be at least as large as the value returned in num_integers, num_addresses, and num_datatypes, respectively, in the call MPI_TYPE_GET_ENVELOPE for the same datatype argument.

> *Rationale.* The arguments max_integers, max_addresses, and max_datatypes allow for error checking in the call. (*End of rationale.*)

The datatypes returned in array_of_datatypes are handles to datatype objects that are equivalent to the datatypes used in the original construction call. If these were derived datatypes, then the returned datatypes are new datatype objects, and the user is responsible for freeing these datatypes with MPI_TYPE_FREE. If these were predefined datatypes, then the returned datatype is equal to that (constant) predefined datatype and cannot be freed.

The committed state of returned derived datatypes is undefined, i.e., the datatypes may or may not be committed. Furthermore, the content of attributes of returned datatypes is undefined.

Note that MPI_TYPE_GET_CONTENTS can be invoked with a datatype argument that was constructed using MPI_TYPE_CREATE_F90_REAL, MPI_TYPE_CREATE_F90_INTEGER, or MPI_TYPE_CREATE_F90_COMPLEX (an unnamed predefined datatype). In such a case, an empty array_of_datatypes is returned.

> *Rationale.* The definition of datatype equivalence implies that equivalent predefined datatypes are equal. By requiring the same handle for named predefined datatypes, it is possible to use the == or .EQ. comparison operator to determine the datatype involved. (*End of rationale.*)

> *Advice to implementors.* The datatypes returned in array_of_datatypes must appear to the user as if each is an equivalent copy of the datatype used in the type constructor call. Whether this is done by creating a new datatype or via another mechanism such as a reference count mechanism is up to the implementation as long as the semantics are preserved. (*End of advice to implementors.*)

Rationale. The committed state and attributes of the returned datatype is deliberately left vague. The datatype used in the original construction may have been modified since its use in the constructor call. Attributes can be added, removed, or modified as well as having the datatype committed. The semantics given allow for a reference count implementation without having to track these changes. (*End of rationale.*)

In the deprecated datatype constructor calls, the address arguments in Fortran are of type INTEGER. In the preferred calls, the address arguments are of type INTEGER(KIND=MPI_ADDRESS_KIND). The call MPI_TYPE_GET_CONTENTS returns all addresses in an argument of type INTEGER(KIND=MPI_ADDRESS_KIND). This is true even if the deprecated calls were used. Thus, the location of values returned can be thought of as being returned by the C bindings. It can also be determined by examining the preferred calls for datatype constructors for the deprecated calls that involve addresses.

Rationale. By having all address arguments returned in the array_of_addresses argument, the result from a C and Fortran decoding of a datatype gives the result in the same argument. It is assumed that an integer of type INTEGER(KIND=MPI_ADDRESS_KIND) will be at least as large as the INTEGER argument used in datatype construction with the old MPI-1 calls so no loss of information will occur. (*End of rationale.*)

The following defines what values are placed in each entry of the returned arrays depending on the datatype constructor used for datatype. It also specifies the size of the arrays needed which is the values returned by MPI_TYPE_GET_ENVELOPE. In Fortran, the following calls were made:

```
      PARAMETER (LARGE = 1000)
      INTEGER TYPE, NI, NA, ND, COMBINER, I(LARGE), D(LARGE), IERROR
      INTEGER(KIND=MPI_ADDRESS_KIND) A(LARGE)
!     CONSTRUCT DATATYPE TYPE (NOT SHOWN)
      CALL MPI_TYPE_GET_ENVELOPE(TYPE, NI, NA, ND, COMBINER, IERROR)
      IF ((NI .GT. LARGE) .OR. (NA .GT. LARGE) .OR. (ND .GT. LARGE)) THEN
        WRITE (*, *) "NI, NA, OR ND = ", NI, NA, ND, &
        " RETURNED BY MPI_TYPE_GET_ENVELOPE IS LARGER THAN LARGE = ", LARGE
        CALL MPI_ABORT(MPI_COMM_WORLD, 99, IERROR)
      ENDIF
      CALL MPI_TYPE_GET_CONTENTS(TYPE, NI, NA, ND, I, A, D, IERROR)
```

or in C the analogous calls of:

```
#define LARGE 1000
int ni, na, nd, combiner, i[LARGE];
MPI_Aint a[LARGE];
MPI_Datatype type, d[LARGE];
/* construct datatype type (not shown) */
MPI_Type_get_envelope(type, &ni, &na, &nd, &combiner);
if ((ni > LARGE) || (na > LARGE) || (nd > LARGE)) {
  fprintf(stderr, "ni, na, or nd = %d %d %d returned by ", ni, na, nd);
```

```
  fprintf(stderr, "MPI_Type_get_envelope is larger than LARGE = %d\n",
          LARGE);
  MPI_Abort(MPI_COMM_WORLD, 99);
};
MPI_Type_get_contents(type, ni, na, nd, i, a, d);
```

The C++ code is in analogy to the C code above with the same values returned.

In the descriptions that follow, the lower case name of arguments is used.

If combiner is MPI_COMBINER_NAMED then it is erroneous to call MPI_TYPE_GET_CONTENTS.

If combiner is MPI_COMBINER_DUP then

Constructor argument	C & C++ location	Fortran location
oldtype	d[0]	D(1)

and ni = 0, na = 0, nd = 1.

If combiner is MPI_COMBINER_CONTIGUOUS then

Constructor argument	C & C++ location	Fortran location
count	i[0]	I(1)
oldtype	d[0]	D(1)

and ni = 1, na = 0, nd = 1.

If combiner is MPI_COMBINER_VECTOR then

Constructor argument	C & C++ location	Fortran location
count	i[0]	I(1)
blocklength	i[1]	I(2)
stride	i[2]	I(3)
oldtype	d[0]	D(1)

and ni = 3, na = 0, nd = 1.

If combiner is MPI_COMBINER_HVECTOR_INTEGER or MPI_COMBINER_HVECTOR then

Constructor argument	C & C++ location	Fortran location
count	i[0]	I(1)
blocklength	i[1]	I(2)
stride	a[0]	A(1)
oldtype	d[0]	D(1)

and ni = 2, na = 1, nd = 1.

If combiner is MPI_COMBINER_INDEXED then

Constructor argument	C & C++ location	Fortran location
count	i[0]	I(1)
array_of_blocklengths	i[1] to i[i[0]]	I(2) to I(I(1)+1)
array_of_displacements	i[i[0]+1] to i[2*i[0]]	I(I(1)+2) to I(2*I(1)+1)
oldtype	d[0]	D(1)

and ni = 2*count+1, na = 0, nd = 1.

If combiner is MPI_COMBINER_HINDEXED_INTEGER or MPI_COMBINER_HINDEXED then

Constructor argument	C & C++ location	Fortran location
count	i[0]	I(1)
array_of_blocklengths	i[1] to i[i[0]]	I(2) to I(I(1)+1)
array_of_displacements	a[0] to a[i[0]-1]	A(1) to A(I(1))
oldtype	d[0]	D(1)

and ni = count+1, na = count, nd = 1.

If combiner is MPI_COMBINER_INDEXED_BLOCK then

Constructor argument	C & C++ location	Fortran location
count	i[0]	I(1)
blocklength	i[1]	I(2)
array_of_displacements	i[2] to i[i[0]+1]	I(3) to I(I(1)+2)
oldtype	d[0]	D(1)

and ni = count+2, na = 0, nd = 1.

If combiner is MPI_COMBINER_STRUCT_INTEGER or MPI_COMBINER_STRUCT then

Constructor argument	C & C++ location	Fortran location
count	i[0]	I(1)
array_of_blocklengths	i[1] to i[i[0]]	I(2) to I(I(1)+1)
array_of_displacements	a[0] to a[i[0]-1]	A(1) to A(I(1))
array_of_types	d[0] to d[i[0]-1]	D(1) to D(I(1))

and ni = count+1, na = count, nd = count.

If combiner is MPI_COMBINER_SUBARRAY then

Constructor argument	C & C++ location	Fortran location
ndims	i[0]	I(1)
array_of_sizes	i[1] to i[i[0]]	I(2) to I(I(1)+1)
array_of_subsizes	i[i[0]+1] to i[2*i[0]]	I(I(1)+2) to I(2*I(1)+1)
array_of_starts	i[2*i[0]+1] to i[3*i[0]]	I(2*I(1)+2) to I(3*I(1)+1)
order	i[3*i[0]+1]	I(3*I(1)+2]
oldtype	d[0]	D(1)

and ni = 3*ndims+2, na = 0, nd = 1.

If combiner is MPI_COMBINER_DARRAY then

Constructor argument	C & C++ location	Fortran location
size	i[0]	I(1)
rank	i[1]	I(2)
ndims	i[2]	I(3)
array_of_gsizes	i[3] to i[i[2]+2]	I(4) to I(I(3)+3)
array_of_distribs	i[i[2]+3] to i[2*i[2]+2]	I(I(3)+4) to I(2*I(3)+3)
array_of_dargs	i[2*i[2]+3] to i[3*i[2]+2]	I(2*I(3)+4) to I(3*I(3)+3)
array_of_psizes	i[3*i[2]+3] to i[4*i[2]+2]	I(3*I(3)+4) to I(4*I(3)+3)
order	i[4*i[2]+3]	I(4*I(3)+4)
oldtype	d[0]	D(1)

and ni = 4*ndims+4, na = 0, nd = 1.

If combiner is MPI_COMBINER_F90_REAL then

Constructor argument	C & C++ location	Fortran location
p	i[0]	I(1)
r	i[1]	I(2)

and ni = 2, na = 0, nd = 0.

If combiner is MPI_COMBINER_F90_COMPLEX then

Constructor argument	C & C++ location	Fortran location
p	i[0]	I(1)
r	i[1]	I(2)

and ni = 2, na = 0, nd = 0.

If combiner is MPI_COMBINER_F90_INTEGER then

Constructor argument	C & C++ location	Fortran location
r	i[0]	I(1)

and ni = 1, na = 0, nd = 0.

If combiner is MPI_COMBINER_RESIZED then

Constructor argument	C & C++ location	Fortran location
lb	a[0]	A(1)
extent	a[1]	A(2)
oldtype	d[0]	D(1)

and ni = 0, na = 2, nd = 1.

4.1.14 Examples

The following examples illustrate the use of derived datatypes.

Example 4.13 Send and receive a section of a 3D array.

```
      REAL a(100,100,100), e(9,9,9)
      INTEGER oneslice, twoslice, threeslice, sizeofreal, myrank, ierr
      INTEGER status(MPI_STATUS_SIZE)

C      extract the section a(1:17:2, 3:11, 2:10)
C      and store it in e(:,:,:).

      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

      CALL MPI_TYPE_EXTENT( MPI_REAL, sizeofreal, ierr)

C     create datatype for a 1D section
      CALL MPI_TYPE_VECTOR( 9, 1, 2, MPI_REAL, oneslice, ierr)

C     create datatype for a 2D section
      CALL MPI_TYPE_HVECTOR(9, 1, 100*sizeofreal, oneslice, twoslice, ierr)

C     create datatype for the entire section
      CALL MPI_TYPE_HVECTOR( 9, 1, 100*100*sizeofreal, twoslice,
                             threeslice, ierr)
```

```
      CALL MPI_TYPE_COMMIT( threeslice, ierr)
      CALL MPI_SENDRECV(a(1,3,2), 1, threeslice, myrank, 0, e, 9*9*9,
                        MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
```

Example 4.14 Copy the (strictly) lower triangular part of a matrix.

```
      REAL a(100,100), b(100,100)
      INTEGER  disp(100), blocklen(100), ltype, myrank, ierr
      INTEGER status(MPI_STATUS_SIZE)

C     copy lower triangular part of array a
C     onto lower triangular part of array b

      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

C     compute start and size of each column
      DO i=1, 100
        disp(i) = 100*(i-1) + i
        blocklen(i) = 100-i
      END DO

C     create datatype for lower triangular part
      CALL MPI_TYPE_INDEXED( 100, blocklen, disp, MPI_REAL, ltype, ierr)

      CALL MPI_TYPE_COMMIT(ltype, ierr)
      CALL MPI_SENDRECV( a, 1, ltype, myrank, 0, b, 1,
                         ltype, myrank, 0, MPI_COMM_WORLD, status, ierr)
```

Example 4.15 Transpose a matrix.

```
      REAL a(100,100), b(100,100)
      INTEGER row, xpose, sizeofreal, myrank, ierr
      INTEGER status(MPI_STATUS_SIZE)

C     transpose matrix a onto b

      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

      CALL MPI_TYPE_EXTENT( MPI_REAL, sizeofreal, ierr)

C     create datatype for one row
      CALL MPI_TYPE_VECTOR( 100, 1, 100, MPI_REAL, row, ierr)

C     create datatype for matrix in row-major order
      CALL MPI_TYPE_HVECTOR( 100, 1, sizeofreal, row, xpose, ierr)
```

```
      CALL MPI_TYPE_COMMIT( xpose, ierr)

C     send matrix in row-major order and receive in column major order
      CALL MPI_SENDRECV( a, 1, xpose, myrank, 0, b, 100*100,
                MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
```

Example 4.16 Another approach to the transpose problem:

```
      REAL a(100,100), b(100,100)
      INTEGER  disp(2), blocklen(2), type(2), row, row1, sizeofreal
      INTEGER  myrank, ierr
      INTEGER status(MPI_STATUS_SIZE)

      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

C     transpose matrix a onto b

      CALL MPI_TYPE_EXTENT( MPI_REAL, sizeofreal, ierr)

C     create datatype for one row
      CALL MPI_TYPE_VECTOR( 100, 1, 100, MPI_REAL, row, ierr)

C     create datatype for one row, with the extent of one real number
      disp(1) = 0
      disp(2) = sizeofreal
      type(1)  = row
      type(2)  = MPI_UB
      blocklen(1)  = 1
      blocklen(2)  = 1
      CALL MPI_TYPE_STRUCT( 2, blocklen, disp, type, row1, ierr)

      CALL MPI_TYPE_COMMIT( row1, ierr)

C     send 100 rows and receive in column major order
      CALL MPI_SENDRECV( a, 100, row1, myrank, 0, b, 100*100,
                MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
```

Example 4.17 We manipulate an array of structures.

```
struct Partstruct
   {
      int    class;  /* particle class */
      double d[6];   /* particle coordinates */
      char   b[7];   /* some additional information */
   };

struct Partstruct    particle[1000];
```

```
1
2   int          i, dest, rank, tag;
3   MPI_Comm     comm;
4
5
6   /* build datatype describing structure */
7
8   MPI_Datatype Particletype;
9   MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
10  int          blocklen[3] = {1, 6, 7};
11  MPI_Aint     disp[3];
12  MPI_Aint     base;
13
14
15  /* compute displacements of structure components */
16
17  MPI_Address( particle, disp);
18  MPI_Address( particle[0].d, disp+1);
19  MPI_Address( particle[0].b, disp+2);
20  base = disp[0];
21  for (i=0; i < 3; i++) disp[i] -= base;
22
23  MPI_Type_struct( 3, blocklen, disp, type, &Particletype);
24
25     /* If compiler does padding in mysterious ways,
26     the following may be safer */
27
28  MPI_Datatype type1[4] = {MPI_INT, MPI_DOUBLE, MPI_CHAR, MPI_UB};
29  int          blocklen1[4] = {1, 6, 7, 1};
30  MPI_Aint     disp1[4];
31
32  /* compute displacements of structure components */
33
34  MPI_Address( particle, disp1);
35  MPI_Address( particle[0].d, disp1+1);
36  MPI_Address( particle[0].b, disp1+2);
37  MPI_Address( particle+1, disp1+3);
38  base = disp1[0];
39  for (i=0; i < 4; i++) disp1[i] -= base;
40
41  /* build datatype describing structure */
42
43  MPI_Type_struct( 4, blocklen1, disp1, type1, &Particletype);
44
45
46                /* 4.1:
47           send the entire array */
48
```

```
MPI_Type_commit( &Particletype);
MPI_Send( particle, 1000, Particletype, dest, tag, comm);


             /* 4.2:
        send only the entries of class zero particles,
        preceded by the number of such entries */

MPI_Datatype Zparticles;   /* datatype describing all particles
                              with class zero (needs to be recomputed
                              if classes change) */
MPI_Datatype Ztype;

MPI_Aint     zdisp[1000];
int          zblock[1000], j, k;
int          zzblock[2] = {1,1};
MPI_Aint     zzdisp[2];
MPI_Datatype zztype[2];

/* compute displacements of class zero particles */
j = 0;
for(i=0; i < 1000; i++)
   if (particle[i].class == 0)
      {
        zdisp[j] = i;
        zblock[j] = 1;
        j++;
      }

/* create datatype for class zero particles  */
MPI_Type_indexed( j, zblock, zdisp, Particletype, &Zparticles);

/* prepend particle count */
MPI_Address(&j, zzdisp);
MPI_Address(particle, zzdisp+1);
zztype[0] = MPI_INT;
zztype[1] = Zparticles;
MPI_Type_struct(2, zzblock, zzdisp, zztype, &Ztype);

MPI_Type_commit( &Ztype);
MPI_Send( MPI_BOTTOM, 1, Ztype, dest, tag, comm);


       /* A probably more efficient way of defining Zparticles */

/* consecutive particles with index zero are handled as one block */
j=0;
for (i=0; i < 1000; i++)
```

```
   if (particle[i].index == 0)
      {
         for (k=i+1; (k < 1000)&&(particle[k].index == 0) ; k++);
         zdisp[j] = i;
         zblock[j] = k-i;
         j++;
         i = k;
      }
MPI_Type_indexed( j, zblock, zdisp, Particletype, &Zparticles);


               /* 4.3:
          send the first two coordinates of all entries */

MPI_Datatype Allpairs;      /* datatype for all pairs of coordinates */

MPI_Aint sizeofentry;

MPI_Type_extent( Particletype, &sizeofentry);

     /* sizeofentry can also be computed by subtracting the address
        of particle[0] from the address of particle[1] */

MPI_Type_hvector( 1000, 2, sizeofentry, MPI_DOUBLE, &Allpairs);
MPI_Type_commit( &Allpairs);
MPI_Send( particle[0].d, 1, Allpairs, dest, tag, comm);

      /* an alternative solution to 4.3 */

MPI_Datatype Onepair;   /* datatype for one pair of coordinates, with
                           the extent of one particle entry */
MPI_Aint disp2[3];
MPI_Datatype type2[3] = {MPI_LB, MPI_DOUBLE, MPI_UB};
int blocklen2[3] = {1, 2, 1};

MPI_Address( particle, disp2);
MPI_Address( particle[0].d, disp2+1);
MPI_Address( particle+1, disp2+2);
base = disp2[0];
for (i=0; i<2; i++) disp2[i] -= base;

MPI_Type_struct( 3, blocklen2, disp2, type2, &Onepair);
MPI_Type_commit( &Onepair);
MPI_Send( particle[0].d, 1000, Onepair, dest, tag, comm);
```

Example 4.18 The same manipulations as in the previous example, but use absolute

addresses in datatypes.

```
struct Partstruct
   {
      int class;
      double d[6];
      char b[7];
   };

struct Partstruct particle[1000];

          /* build datatype describing first array entry */

MPI_Datatype Particletype;
MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
int          block[3] = {1, 6, 7};
MPI_Aint     disp[3];

MPI_Address( particle, disp);
MPI_Address( particle[0].d, disp+1);
MPI_Address( particle[0].b, disp+2);
MPI_Type_struct( 3, block, disp, type, &Particletype);

/* Particletype describes first array entry -- using absolute
   addresses */

                /* 5.1:
            send the entire array */

MPI_Type_commit( &Particletype);
MPI_Send( MPI_BOTTOM, 1000, Particletype, dest, tag, comm);


               /* 5.2:
         send the entries of class zero,
         preceded by the number of such entries */

MPI_Datatype Zparticles, Ztype;

MPI_Aint     zdisp[1000];
int          zblock[1000], i, j, k;
int          zzblock[2] = {1,1};
MPI_Datatype zztype[2];
MPI_Aint     zzdisp[2];

j=0;
for (i=0; i < 1000; i++)
   if (particle[i].index == 0)
```

```
      {
         for (k=i+1; (k < 1000)&&(particle[k].index == 0) ; k++);
         zdisp[j] = i;
         zblock[j] = k-i;
         j++;
         i = k;
      }
MPI_Type_indexed( j, zblock, zdisp, Particletype, &Zparticles);
/* Zparticles describe particles with class zero, using
   their absolute addresses*/

/* prepend particle count */
MPI_Address(&j, zzdisp);
zzdisp[1] = MPI_BOTTOM;
zztype[0] = MPI_INT;
zztype[1] = Zparticles;
MPI_Type_struct(2, zzblock, zzdisp, zztype, &Ztype);

MPI_Type_commit( &Ztype);
MPI_Send( MPI_BOTTOM, 1, Ztype, dest, tag, comm);
```

Example 4.19 Handling of unions.

```
union {
   int     ival;
   float   fval;
      } u[1000];

int     utype;

/* All entries of u have identical type; variable
   utype keeps track of their current type */

MPI_Datatype   type[2];
int            blocklen[2] = {1,1};
MPI_Aint       disp[2];
MPI_Datatype   mpi_utype[2];
MPI_Aint       i,j;

/* compute an MPI datatype for each possible union type;
   assume values are left-aligned in union storage. */

MPI_Address( u, &i);
MPI_Address( u+1, &j);
disp[0] = 0; disp[1] = j-i;
type[1] = MPI_UB;
```

```
type[0] = MPI_INT;
MPI_Type_struct(2, blocklen, disp, type, &mpi_utype[0]);

type[0] = MPI_FLOAT;
MPI_Type_struct(2, blocklen, disp, type, &mpi_utype[1]);

for(i=0; i<2; i++) MPI_Type_commit(&mpi_utype[i]);

/* actual communication */

MPI_Send(u, 1000, mpi_utype[utype], dest, tag, comm);
```

Example 4.20 This example shows how a datatype can be decoded. The routine printdatatype prints out the elements of the datatype. Note the use of MPI_Type_free for datatypes that are not predefined.

```
/*
  Example of decoding a datatype.

  Returns 0 if the datatype is predefined, 1 otherwise
 */
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
int printdatatype( MPI_Datatype datatype )
{
    int *array_of_ints;
    MPI_Aint *array_of_adds;
    MPI_Datatype *array_of_dtypes;
    int num_ints, num_adds, num_dtypes, combiner;
    int i;

    MPI_Type_get_envelope( datatype,
                           &num_ints, &num_adds, &num_dtypes, &combiner );
    switch (combiner) {
    case MPI_COMBINER_NAMED:
        printf( "Datatype is named:" );
        /* To print the specific type, we can match against the
           predefined forms. We can NOT use a switch statement here
           We could also use MPI_TYPE_GET_NAME if we prefered to use
           names that the user may have changed.
         */
        if      (datatype == MPI_INT)    printf( "MPI_INT\n" );
        else if (datatype == MPI_DOUBLE) printf( "MPI_DOUBLE\n" );
        ... else test for other types ...
        return 0;
        break;
```

```
    case MPI_COMBINER_STRUCT:
    case MPI_COMBINER_STRUCT_INTEGER:
        printf( "Datatype is struct containing" );
        array_of_ints   = (int *)malloc( num_ints * sizeof(int) );
        array_of_adds   =
                    (MPI_Aint *) malloc( num_adds * sizeof(MPI_Aint) );
        array_of_dtypes = (MPI_Datatype *)
            malloc( num_dtypes * sizeof(MPI_Datatype) );
        MPI_Type_get_contents( datatype, num_ints, num_adds, num_dtypes,
                        array_of_ints, array_of_adds, array_of_dtypes );
        printf( " %d datatypes:\n", array_of_ints[0] );
        for (i=0; i<array_of_ints[0]; i++) {
            printf( "blocklength %d, displacement %ld, type:\n",
                    array_of_ints[i+1], array_of_adds[i] );
            if (printdatatype( array_of_dtypes[i] )) {
                /* Note that we free the type ONLY if it
                   is not predefined */
                MPI_Type_free( &array_of_dtypes[i] );
            }
        }
        free( array_of_ints );
        free( array_of_adds );
        free( array_of_dtypes );
        break;
        ... other combiner values ...
    default:
        printf( "Unrecognized combiner type\n" );
    }
    return 1;
}
```

4.2 Pack and Unpack

Some existing communication libraries provide pack/unpack functions for sending noncontiguous data. In these, the user explicitly packs data into a contiguous buffer before sending it, and unpacks it from a contiguous buffer after receiving it. Derived datatypes, which are described in Section 4.1, allow one, in most cases, to avoid explicit packing and unpacking. The user specifies the layout of the data to be sent or received, and the communication library directly accesses a noncontiguous buffer. The pack/unpack routines are provided for compatibility with previous libraries. Also, they provide some functionality that is not otherwise available in MPI. For instance, a message can be received in several parts, where the receive operation done on a later part may depend on the content of a former part. Another use is that outgoing messages may be explicitly buffered in user supplied space, thus overriding the system buffering policy. Finally, the availability of pack and unpack operations facilitates the development of additional communication libraries layered on top of MPI.

MPI_PACK(inbuf, incount, datatype, outbuf, outsize, position, comm)

IN	inbuf	input buffer start (choice)
IN	incount	number of input data items (non-negative integer)
IN	datatype	datatype of each input data item (handle)
OUT	outbuf	output buffer start (choice)
IN	outsize	output buffer size, in bytes (non-negative integer)
INOUT	position	current position in buffer, in bytes (integer)
IN	comm	communicator for packed message (handle)

```
int MPI_Pack(void* inbuf, int incount, MPI_Datatype datatype, void *outbuf,
             int outsize, int *position, MPI_Comm comm)
```

```
MPI_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION, COMM, IERROR)
    <type> INBUF(*), OUTBUF(*)
    INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM, IERROR
```

```
{void MPI::Datatype::Pack(const void* inbuf, int incount, void *outbuf,
             int outsize, int& position, const MPI::Comm &comm)
             const(binding deprecated, see Section 15.2) }
```

Packs the message in the send buffer specified by inbuf, incount, datatype into the buffer space specified by outbuf and outsize. The input buffer can be any communication buffer allowed in MPI_SEND. The output buffer is a contiguous storage area containing outsize bytes, starting at the address outbuf (length is counted in bytes, not elements, as if it were a communication buffer for a message of type MPI_PACKED).

The input value of position is the first location in the output buffer to be used for packing. position is incremented by the size of the packed message, and the output value of position is the first location in the output buffer following the locations occupied by the packed message. The comm argument is the communicator that will be subsequently used for sending the packed message.

MPI_UNPACK(inbuf, insize, position, outbuf, outcount, datatype, comm)

IN	inbuf	input buffer start (choice)
IN	insize	size of input buffer, in bytes (non-negative integer)
INOUT	position	current position in bytes (integer)
OUT	outbuf	output buffer start (choice)
IN	outcount	number of items to be unpacked (integer)
IN	datatype	datatype of each output data item (handle)
IN	comm	communicator for packed message (handle)

```
int MPI_Unpack(void* inbuf, int insize, int *position, void *outbuf,
             int outcount, MPI_Datatype datatype, MPI_Comm comm)
```

```
MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, COMM,
              IERROR)
    <type> INBUF(*), OUTBUF(*)
    INTEGER INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM, IERROR

{void MPI::Datatype::Unpack(const void* inbuf, int insize, void *outbuf,
              int outcount, int& position, const MPI::Comm& comm)
              const(binding deprecated, see Section 15.2) }
```

Unpacks a message into the receive buffer specified by outbuf, outcount, datatype from the buffer space specified by inbuf and insize. The output buffer can be any communication buffer allowed in MPI_RECV. The input buffer is a contiguous storage area containing insize bytes, starting at address inbuf. The input value of position is the first location in the input buffer occupied by the packed message. position is incremented by the size of the packed message, so that the output value of position is the first location in the input buffer after the locations occupied by the message that was unpacked. comm is the communicator used to receive the packed message.

> *Advice to users.* Note the difference between MPI_RECV and MPI_UNPACK: in MPI_RECV, the count argument specifies the maximum number of items that can be received. The actual number of items received is determined by the length of the incoming message. In MPI_UNPACK, the count argument specifies the actual number of items that are unpacked; the "size" of the corresponding message is the increment in position. The reason for this change is that the "incoming message size" is not predetermined since the user decides how much to unpack; nor is it easy to determine the "message size" from the number of items to be unpacked. In fact, in a heterogeneous system, this number may not be determined *a priori*. (*End of advice to users.*)

To understand the behavior of pack and unpack, it is convenient to think of the data part of a message as being the sequence obtained by concatenating the successive values sent in that message. The pack operation stores this sequence in the buffer space, as if sending the message to that buffer. The unpack operation retrieves this sequence from buffer space, as if receiving a message from that buffer. (It is helpful to think of internal Fortran files or sscanf in C, for a similar function.)

Several messages can be successively packed into one **packing unit**. This is effected by several successive **related** calls to MPI_PACK, where the first call provides position = 0, and each successive call inputs the value of position that was output by the previous call, and the same values for outbuf, outcount and comm. This packing unit now contains the equivalent information that would have been stored in a message by one send call with a send buffer that is the "concatenation" of the individual send buffers.

A packing unit can be sent using type MPI_PACKED. Any point to point or collective communication function can be used to move the sequence of bytes that forms the packing unit from one process to another. This packing unit can now be received using any receive operation, with any datatype: the type matching rules are relaxed for messages sent with type MPI_PACKED.

A message sent with any type (including MPI_PACKED) can be received using the type MPI_PACKED. Such a message can then be unpacked by calls to MPI_UNPACK.

A packing unit (or a message created by a regular, "typed" send) can be unpacked into several successive messages. This is effected by several successive related calls to

MPI_UNPACK, where the first call provides position = 0, and each successive call inputs the value of position that was output by the previous call, and the same values for inbuf, insize and comm.

The concatenation of two packing units is not necessarily a packing unit; nor is a substring of a packing unit necessarily a packing unit. Thus, one cannot concatenate two packing units and then unpack the result as one packing unit; nor can one unpack a substring of a packing unit as a separate packing unit. Each packing unit, that was created by a related sequence of pack calls, or by a regular send, must be unpacked as a unit, by a sequence of related unpack calls.

Rationale. The restriction on "atomic" packing and unpacking of packing units allows the implementation to add at the head of packing units additional information, such as a description of the sender architecture (to be used for type conversion, in a heterogeneous environment) (*End of rationale.*)

The following call allows the user to find out how much space is needed to pack a message and, thus, manage space allocation for buffers.

MPI_PACK_SIZE(incount, datatype, comm, size)

IN	incount	count argument to packing call (non-negative integer)
IN	datatype	datatype argument to packing call (handle)
IN	comm	communicator argument to packing call (handle)
OUT	size	upper bound on size of packed message, in bytes (non-negative integer)

```
int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm,
              int *size)
```

```
MPI_PACK_SIZE(INCOUNT, DATATYPE, COMM, SIZE, IERROR)
    INTEGER INCOUNT, DATATYPE, COMM, SIZE, IERROR
```

```
{int MPI::Datatype::Pack_size(int incount, const MPI::Comm& comm)
              const(binding deprecated, see Section 15.2) }
```

A call to MPI_PACK_SIZE(incount, datatype, comm, size) returns in size an upper bound on the increment in position that is effected by a call to MPI_PACK(inbuf, incount, datatype, outbuf, outcount, position, comm). The value returned as the size argument of MPI_PACK_SIZE for a datatype larger than what can be represented by a C int or Fortran INTEGER is MPI_UNDEFINED. ticket265.

Rationale. The call returns an upper bound, rather than an exact bound, since the exact amount of space needed to pack the message may depend on the context (e.g., first message packed in a packing unit may take more space). (*End of rationale.*)

Example 4.21 An example using MPI_PACK.

5.9.2 Predefined Reduction Operations

The following predefined operations are supplied for MPI_REDUCE and related functions MPI_ALLREDUCE, MPI_REDUCE_SCATTER, MPI_SCAN, and MPI_EXSCAN. These operations are invoked by placing the following in op.

Name	Meaning
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical exclusive or (xor)
MPI_BXOR	bit-wise exclusive or (xor)
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

The two operations MPI_MINLOC and MPI_MAXLOC are discussed separately in Section 5.9.4. For the other predefined operations, we enumerate below the allowed combinations of op and datatype arguments. First, define groups of MPI basic datatypes in the following way.

C integer:	MPI_INT, MPI_LONG, MPI_SHORT, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_LONG_LONG_INT, MPI_LONG_LONG (as synonym), MPI_UNSIGNED_LONG_LONG, MPI_SIGNED_CHAR, MPI_UNSIGNED_CHAR, MPI_INT8_T, MPI_INT16_T, MPI_INT32_T, MPI_INT64_T, MPI_UINT8_T, MPI_UINT16_T, MPI_UINT32_T, MPI_UINT64_T
Fortran integer:	MPI_INTEGER, MPI_AINT, MPI_COUNT, MPI_OFFSET, and handles returned from MPI_TYPE_CREATE_F90_INTEGER, and if available: MPI_INTEGER1, MPI_INTEGER2, MPI_INTEGER4, MPI_INTEGER8, MPI_INTEGER16
Floating point:	MPI_FLOAT, MPI_DOUBLE, MPI_REAL, MPI_DOUBLE_PRECISION MPI_LONG_DOUBLE and handles returned from

ticket265.

12.3 Associating Information with Status

MPI supports several different types of requests besides those for point-to-point operations. These range from MPI calls for I/O to generalized requests. It is desirable to allow these calls [use]to use the same request [mechanism. This]mechanism, which allows one to wait or test on different types of requests. However, MPI_{TEST|WAIT}{ANY|SOME|ALL} returns a status with information about the request. With the generalization of requests, one needs to define what information will be returned in the status object.

Each MPI call fills in the appropriate fields in the status object. Any unused fields will have undefined values. A call to MPI_{TEST|WAIT}{ANY|SOME|ALL} can modify any of the fields in the status object. Specifically, it can modify fields that are undefined. The fields with meaningful [value]values for a given request are defined in the sections with the new request.

Generalized requests raise additional considerations. Here, the user provides the functions to deal with the request. Unlike other MPI calls, the user needs to provide the information to be returned in status. The status argument is provided directly to the callback function where the status needs to be set. Users can directly set the values in 3 of the 5 status values. The count and cancel fields are opaque. To overcome this, these calls are provided:

MPI_STATUS_SET_ELEMENTS(status, datatype, count)

INOUT	status	status with which to associate count (Status)
IN	datatype	datatype associated with count (handle)
IN	count	number of elements to associate with status (integer)

```
int MPI_Status_set_elements(MPI_Status *status, MPI_Datatype datatype,
              int count)

MPI_STATUS_SET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR

{void MPI::Status::Set_elements(const MPI::Datatype& datatype, int
              count) (binding deprecated, see Section 15.2) }
```

ticket265.

MPI_STATUS_SET_ELEMENTS_X(status, datatype, count)

INOUT	status	status with which to associate count (Status)
IN	datatype	datatype associated with count (handle)
IN	count	number of elements to associate with status (integer)

```
int MPI_Status_set_elements_x(MPI_Status *status, MPI_Datatype datatype,
              MPI_Count count)

MPI_STATUS_SET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR
    INTEGER (KIND=MPI_COUNT_KIND) COUNT
```

This call modifies the opaque part of status so that a call to MPI_GET_ELEMENTS or MPI_GET_ELEMENTS_X will return count. MPI_GET_COUNT will return a compatible value. ticket265.

Rationale. The number of elements is set instead of the count because the former can deal with a nonintegral number of datatypes. (*End of rationale.*)

A subsequent call to MPI_GET_COUNT(status, datatype, count) [or to], MPI_GET_ELEMENTS(status, datatype, count) , or MPI_GET_ELEMENTS_X(status, datatype, count) must use a datatype argument that has the same type signature as the datatype argument that was used in the call to MPI_STATUS_SET_ELEMENTS or MPI_STATUS_SET_ELEMENTS_X. ticket265. ticket265. ticket265.

Rationale. [This]The requirement of matching type signatures for these calls is similar to the restriction that holds when count is set by a receive operation: in that case, the calls to MPI_GET_COUNT and MPI_GET_ELEMENTS must use a datatype with the same signature as the datatype used in the receive call. (*End of rationale.*) ticket0.

MPI_STATUS_SET_CANCELLED(status, flag)

INOUT	status	status with which to associate cancel flag (Status)
IN	flag	if true indicates request was cancelled (logical)

```
int MPI_Status_set_cancelled(MPI_Status *status, int flag)

MPI_STATUS_SET_CANCELLED(STATUS, FLAG, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), IERROR
    LOGICAL FLAG

{void MPI::Status::Set_cancelled(bool flag) (binding deprecated, see Section 15.2)
              }
```

If flag is set to true then a subsequent call to MPI_TEST_CANCELLED(status, flag) will also return flag = true, otherwise it will return false.

Advice to users. Users are advised not to reuse the status fields for values other than those for which they were intended. Doing so may lead to unexpected results when using the status object. For example, calling MPI_GET_ELEMENTS may cause an error if the value is out of range or it may be impossible to detect such an error. The extra_state argument provided with a generalized request can be used to return information that does not logically belong in status. Furthermore, modifying the values in a status set internally by MPI, e.g., MPI_RECV, may lead to unpredictable results and is strongly discouraged. (*End of advice to users.*)

12.4 MPI and Threads

This section specifies the interaction between MPI calls and threads. The section lists minimal requirements for **thread compliant** MPI implementations and defines functions

Assorted Constants

C type: `const int` (or unnamed `enum`) Fortran type: `INTEGER`	C++ type: `const int` (or unnamed `enum`)
MPI_PROC_NULL	MPI::PROC_NULL
MPI_ANY_SOURCE	MPI::ANY_SOURCE
MPI_ANY_TAG	MPI::ANY_TAG
MPI_UNDEFINED	MPI::UNDEFINED
MPI_BSEND_OVERHEAD	MPI::BSEND_OVERHEAD
MPI_KEYVAL_INVALID	MPI::KEYVAL_INVALID
MPI_LOCK_EXCLUSIVE	MPI::LOCK_EXCLUSIVE
MPI_LOCK_SHARED	MPI::LOCK_SHARED
MPI_ROOT	MPI::ROOT

Status size and reserved index values (Fortran only)

Fortran type: `INTEGER`	
MPI_STATUS_SIZE	Not defined for C++
MPI_SOURCE	Not defined for C++
MPI_TAG	Not defined for C++
MPI_ERROR	Not defined for C++

Variable Address Size (Fortran only)

Fortran type: `INTEGER`	
MPI_ADDRESS_KIND	Not defined for C++
[ticket265.] MPI_COUNT_KIND	Not defined for C++
MPI_INTEGER_KIND	Not defined for C++
MPI_OFFSET_KIND	Not defined for C++

Error-handling specifiers

C type: `MPI_Errhandler` Fortran type: `INTEGER`	C++ type: `MPI::Errhandler`
MPI_ERRORS_ARE_FATAL	MPI::ERRORS_ARE_FATAL
MPI_ERRORS_RETURN	MPI::ERRORS_RETURN
	MPI::ERRORS_THROW_EXCEPTIONS

Maximum Sizes for Strings

C type: `const int` (or unnamed `enum`) Fortran type: `INTEGER`	C++ type: `const int` (or unnamed `enum`)
MPI_MAX_PROCESSOR_NAME	MPI::MAX_PROCESSOR_NAME
MPI_MAX_ERROR_STRING	MPI::MAX_ERROR_STRING
MPI_MAX_DATAREP_STRING	MPI::MAX_DATAREP_STRING
MPI_MAX_INFO_KEY	MPI::MAX_INFO_KEY
MPI_MAX_INFO_VAL	MPI::MAX_INFO_VAL
MPI_MAX_OBJECT_NAME	MPI::MAX_OBJECT_NAME
MPI_MAX_PORT_NAME	MPI::MAX_PORT_NAME

Named Predefined Datatypes		C/C++ types
C type: `MPI_Datatype`	C++ type: `MPI::Datatype`	
Fortran type: `INTEGER`		
MPI_CHAR	MPI::CHAR	`char` (treated as printable character)
MPI_SHORT	MPI::SHORT	`signed short int`
MPI_INT	MPI::INT	`signed int`
MPI_LONG	MPI::LONG	`signed long`
MPI_LONG_LONG_INT	MPI::LONG_LONG_INT	`signed long long`
MPI_LONG_LONG	MPI::LONG_LONG	`long long` (synonym)
MPI_SIGNED_CHAR	MPI::SIGNED_CHAR	`signed char` (treated as integral value)
MPI_UNSIGNED_CHAR	MPI::UNSIGNED_CHAR	`unsigned char` (treated as integral value)
MPI_UNSIGNED_SHORT	MPI::UNSIGNED_SHORT	`unsigned short`
MPI_UNSIGNED	MPI::UNSIGNED	`unsigned int`
MPI_UNSIGNED_LONG	MPI::UNSIGNED_LONG	`unsigned long`
MPI_UNSIGNED_LONG_LONG	MPI::UNSIGNED_LONG_LONG	`unsigned long long`
MPI_FLOAT	MPI::FLOAT	`float`
MPI_DOUBLE	MPI::DOUBLE	`double`
MPI_LONG_DOUBLE	MPI::LONG_DOUBLE	`long double`
MPI_WCHAR	MPI::WCHAR	`wchar_t` (defined in `<stddef.h>`) (treated as printable character)
MPI_C_BOOL	(use C datatype handle)	`_Bool`
MPI_INT8_T	(use C datatype handle)	`int8_t`
MPI_INT16_T	(use C datatype handle)	`int16_t`
MPI_INT32_T	(use C datatype handle)	`int32_t`
MPI_INT64_T	(use C datatype handle)	`int64_t`
MPI_UINT8_T	(use C datatype handle)	`uint8_t`
MPI_UINT16_T	(use C datatype handle)	`uint16_t`
MPI_UINT32_T	(use C datatype handle)	`uint32_t`
MPI_UINT64_T	(use C datatype handle)	`uint64_t`
MPI_AINT	(use C datatype handle)	`MPI_Aint`
[ticket265.] MPI_COUNT	(use C datatype handle)	`MPI_Count`
MPI_OFFSET	(use C datatype handle)	`MPI_Offset`
MPI_C_COMPLEX	(use C datatype handle)	`float _Complex`
MPI_C_FLOAT_COMPLEX	(use C datatype handle)	`float _Complex`
MPI_C_DOUBLE_COMPLEX	(use C datatype handle)	`double _Complex`
MPI_C_LONG_DOUBLE_COMPLEX	(use C datatype handle)	`long double _Complex`
MPI_BYTE	MPI::BYTE	(any C/C++ type)
MPI_PACKED	MPI::PACKED	(any C/C++ type)

Named Predefined Datatypes		Fortran types
C type: MPI_Datatype	C++ type: MPI::Datatype	
Fortran type: INTEGER		
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)
[ticket265.] MPI_COUNT	(use C datatype handle)	INTEGER (KIND=MPI_COUNT_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		
MPI_DOUBLE_COMPLEX	MPI::F_DOUBLE_COMPLEX	DOUBLE COMPLEX
MPI_INTEGER1	MPI::INTEGER1	INTEGER*1
MPI_INTEGER2	MPI::INTEGER2	INTEGER*8
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