QMP: LQCD Message Passing API

Jie Chen, Robert Edwards, William Watson

Version 1.2

13-Feb-03

Recent Changes:

There is no longer a logical node number, only a node number which does not change as the logical machine is defined. Thus there are two styles of messaging:

- messages are sent to a node by node number, or
- messages are sent to a relative (logical) node.

Methods related to node numbers have been changed (some dropped, some added).

Introduction

This note presents

1. the requirements for message passing within Lattice QCD applications
2. a draft message API for both C and C++
3. implementation design ideas

The API is intended to be sufficiently flexible to be used by all Lattice QCD applications, and execute efficiently on all existing and anticipated platforms, so that there is no need to directly call non-portable message passing routines.

Because of the highly regular grid communications within LQCD, MPI calls (which are more general) impose some additional overhead that is predicted to be non-negligible for large machines. Depending upon demand, a subset of MPI could be implemented above this new API so that legacy codes which use MPI could function on the new architectures which implement (only) the new API. Further, the new API has been implemented atop MPI so that new applications using this new API can still be run on older machines for which only MPI is available.

Interspersed with the API description are some descriptions for how the API could be implemented for myrinet clusters and the QCDOC machine. These are meant to more fully illustrate the functionality, and are not intended as the final design.

At the time of writing, the following implementations exist:

1. QMP-GM Uses GM
2. QMP-MPI Uses MPI; tested above MPICH-GM, MPICH-SM (shared memory), and MPICH-P4 (sockets)

Capability Requirements

1. Barrier call (synchronize all nodes).

2. Send a contiguous message to a given node (identified by a single number, application manages lexicographic ordering of nodes in a grid).
3. Send a message to a neighboring node (direction identified by a number, with, for example, magnitude of the number indicating dimension 1,2,3,..., sign of the number indicating left/right; library manages mapping onto physical node).
4. Send (receive) a non-contiguous message consisting of a set of strided blocks (for each element in the set, specify base, blocksize, stride and number of blocks). This capability will allow sending hypersurfaces and other complicated sub-sets of data, and will map efficiently onto underlying hardware that supports chained strided access, particularly the QCDOC.
5. Send to a machine specified by a communications map (1-to-1; machine i sends to $\text{map}(i)$).
6. Send to a machine specified by its physical node number (to support "special" nodes).
7. Broadcast a message to all nodes.
8. Global sum for 32 bit & 64 bit floats, ints, and arrays of same. Calls to do the same for complex may also be provided, perhaps as a convenience routine above the array calls.
9. Global max for float & double; global exclusive OR for int and long.
10. Arbitrary binary function global reduction.
11. Configuration discovery and control (how physical nodes of machine are arranged into a logical grid).

API Design: Performance Requirements

1. Design must allow for overlapping of computation and communications. Hence, initiating the send (or receive) of a message must be decoupled from testing for or waiting for its completion.
2. Design must allow for issuing multiple sends without waiting for the first to complete. Example, send in all 4 positive directions. This is important for myrinet so that the overhead of "filling the pipe" is not incurred for each message.
3. Design must allow initiating sends in all directions in a single call (preserve hardware capability of QCDOC).
4. Design must avoid forcing the use of barrier calls across the whole machine when all that is really needed is to wait for a single neighboring node. Therefore one must be able to poll or wait for receipt of a particular message, instead of using a global barrier.
5. Design must allow expensive operations involved in defining a communication to be done ahead of invoking the communication multiple times. Example: locking virtual pages in memory for use by a PCI DMA engine.
6. Attempts should be made to minimize bookkeeping overhead on host and any intelligent interfaces.

Hardware Issues

A design (probably not the only one possible) that addresses these performance constraints is something along the lines of a zero copy (where possible) channel oriented I/O library. Ignoring the scatter-gather issue, i.e. restricting the design to contiguous messages for a moment, consider the following behavior:

1. Node A declares an intent to (repetitively) to receive messages from node B into buffer Q. At this point pages are locked in memory, and physical addresses for Q are determined (translate virtual addresses, as needed). This defines one endpoint of a message channel B->A.
2. Node B declares an intent to (repetitively) send a buffer R to node A. At this point R is locked in memory, and physical addresses are determined. Also, whatever work is necessary to compute the target destination (network address and perhaps also remote memory physical address) is done. This defines the sending endpoint of a message channel.
3. Node A initiates a receive operation for channel B->A. This enables the channel and declares the buffer can be written into (semaphore).
4. Node B initiates the send on channel B->A.
5. At some later time, Node A tests to see if the channel B->A has received new data.

Effectively, this defines a channel from B's R to A's Q, or allows B to remotely write to A's memory in a way gated by A being ready to receive.

Hardware notes: on a myrinet system, the receiving network interface card (NIC) can autonomously write into the receiving host's memory, at an address determined by the NIC with no receiving host intervention. Also, send requests are queued in a FIFO, enabling one to satisfy performance requirement 2. On the QCDOC, each wire has hardware for both send and receive.

Simplified API Example

To see how this might look, below are a few representative calls, written as C code.

Host A:

```
opaqueFromB = declareReceiveFrom (remoteNodeB, buffer, nbytes);
...
startIO (opaqueFromB);
...
testCompletion (opaqueFromB);
```

Host B:

```
opaqueToA = declareSendTo (remoteNodeA, buffer, nbytes);
...
startIO(opaqueToA);
```

For myrinet, this send operation could be implemented as a single move instruction into a control fifo of the myrinet NIC, where the value moved (opaqueToA.myri) could be a pointer to a structure previously created in the NIC's memory, or an index into an array of such structures. That structure could have a pre-digested set of values (resident on the myrinet card) to be moved into the PCI DMA engine, plus other necessary values.

For the QCDOC, opaqueToA could be a pointer to a structure containing all values needed to be moved into the corresponding link's transfer engine. Making some assumptions about application behavior, the send could also be triggered by a single move instruction to a control register, selecting one of several (32) possible pre-digested DMA operations.

C Message API

The following presents a C binding of the API; a C++ binding is presented in the subsequent section. This C binding hides all intermediate structures as opaque types through typedef's (not shown).

Initialization and Layout

The following set of calls are used by the application to

1. discover the configuration of the allocated machine and discover which node the current machine is (0 to N-1)
2. configure the logical layout of the machine (number of boxes in each direction) subject to the constraints of the underlying allocated machine, and determine which node the current node is in the logical grid of boxes
3. optimally partition the total lattice onto the logical machine

The difference between the allocated layout of the machine and the logical layout of the machine is that the logical layout of the machine is meant to present to the application programmer a simple grid machine, convenient for use in grid applications such as lattice QCD. If, for example, the machine nodes are connected by a switch, in which all nodes are "adjacent", the creation of a logical "view" of that machine enables one to give meaning to sending to the nearest neighbor in the positive X direction.

Defining the logical machine may on some platforms also "rotate" the machine. For example, if the allocated machine is 8x4x4x4, but you want more segmentation in the 4th dimension, the logical view could be converted to 4x4x4x8 by rotating the allocated machine. This should not be necessary in practice for the QCDOC, as the operating system's allocation mechanism will give the requested shape as specified in the job's parameters (prior to job start). Similarly, defaults for a switched machine could also be passed from the environment to this library, but in the initial implementation, it will be required that the application specify the desired logical machine prior to using any of the nearest-neighbor messaging routines.

Laying out the logical machine will not change the number (0 to N-1) of a node, so that node 0 can be treated as a special node. This implies that the node number can not be assumed to be the lexicographic position within the logical grid.

QMP C API own data types

In order to have better portability for QMP C implementation, the following data types are defined:

- typedef unsigned char QMP_u8_t
- typedef unsigned short QMP_u16_t
- typedef unsigned int QMP_u32_t
- typedef char QMP_s8_t
- typedef short QMP_s16_t
- typedef int QMP_s32_t
- typedef int QMP_bool_t

- `#ifdef QMP_64BIT_LONG`
`typedef unsigned long       QMP_u64_t`
`typedef long                QMP_s64_t`
`#else`
`typedef unsigned long long   QMP_u64_t`
`typedef long long            QMP_s64_t`
`#endif`
- `typedef float               QMP_float_t`
- `typedef double             QMP_double_t`
- `typedef void*              QMP_msgmem_t`
- `typedef void*              QMP_msghandle_t`
- `typedef int                QMP_status_t`

In addition the following data types are enumerated data types:

- `QMP_ictype_t`
- `QMP_smpaddr_t`

Allocated Machine

`QMP_status_t QMP_init_msg_passing (int* argc, char*** argv, QMP_smpaddr_t options);`
 initialize communications hardware (if necessary), and retrieve information from the environment such as number of nodes, and ID's of the other nodes;
 returns `QMP_SUCCESS` if success, else an error number; error string obtained via `QMP_get_error_string()`

options may be:

`QMP_SMP_ONE_ADDRESS` specifies that the multiple processors of an SMP are to be treated as a single node for addressing purposes. In this case, the application is responsible for using the multiple processors (the SMP node will have only one logical address).

`QMP_SMP_MULTIPLE_ADDRESS` specifies that the multiple processors of an SMP are to have multiple addresses. This mode is used when multiple copies of the application are executing on the SMP (no threads).

`void QMP_finalize_msg_passing (void);`

free any allocated resources

`QMP_u32_t QMP_get_SMP_count(void);`

returns the number of processors on this node (for use by applications managing the SMP capabilities of the node)

`QMP_ictype_t QMP_get_msg_passing_type (void);`

return enum `QMP_SWITCH`, `QMP_GRID`, `QMP_FATTREE`, ...

`QMP_u32_t QMP_get_number_of_nodes (void);`

return number of nodes allocated to this job

QMP_u32_t QMP_get_node_number(void);
 return the node number (0 to N-1) of this node

const QMP_u32_t QMP_get_allocated_number_of_dimensions (void);
 for a grid machine, returns number of dimensions in the grid for the allocated nodes;
 for a switched machine, returns 0

const QMP_u32_t * QMP_get_allocated_dimensions (void);
 size of the allocated grid machine (returns null for switch)

const QMP_u32_t * QMP_get_allocated_coordinates (void);
 returns coordinates within machine grid (null for switch)

Logical Machine

The logical machine is a view of the allocated machine. This view can be created explicitly by the call to QMP_declare_logical_topology (below), or implicitly by a call to QMP_layout_grid, in the next section. Defining the logical machine is necessary on non-grid allocated machines (e.g. myrinet cluster) if the application intends to use any nearest neighbor message passing calls. (This requirement may be lifted when the ability to define a default logical machine via the job environment is implemented).

QMP_bool_t QMP_declare_logical_topology (const QMP_u32_t * dimensions, QMP_u32_t ndims);

forces the logical topology to be a simple grid of the given dimensions, if possible;
 returns false if it fails (if allocated machine topology constraints can't support the request); this routine can only be called once

Note: It is considered an error to declare a logical topology which does not map one-to-one to the allocated machine. In particular, the number of nodes must match.

QMP_bool_t QMP_logical_topology_is_declared (void);
 returns true if QMP_declare_logical_topology has been called (explicitly or implicitly)

const QMP_u32_t QMP_get_logical_number_of_dimensions (void);
 dimensionality of the logical machine, not the allocated machine
 if no logical topology has been forced, returns info from allocated machine

const QMP_u32_t * QMP_get_logical_dimensions (void);
 returns the dimensions of the logical machine, as set by QMP_declare_logical_topology
 if no logical topology has been forced, returns info from physical machine

const QMP_u32_t* QMP_get_logical_coordinates (void);
 returns coordinates within the logical machine grid; if no logical topology has been forced, returns info from the allocated machine

const QMP_u32_t QMP_get_node_number_from (QMP_u32_t * nodecoordinates);

returns the node number for messaging, given the logical coordinates of the node

```
const QMP_u32_t * QMP_get_logical_coordinates_from (QMP_u32_t nodenumber);
```

returns coordinates within the logical machine grid of the specified node

Problem Specification

The following routines are convenience routines for subdividing the lattice onto the set of available nodes. They are not required for the message passing library to function.

```
void QMP_layout_grid (QMP_u32_t * latticeDimensions, QMP_u32_t ndims);
```

computes optimal layout, and calls `QMP_declare_logical_topology` if it has not already been set (forced) by the application; if the logical layout has been forced, this routine uses it to subdivide the lattice. Note that this function may "rotate" the allocated machine (via the call to declare logical topology) to better line up with the lattice dimensions. I.e. the original machine's x direction may change. This routine can only be called once.

For example: if the problem size (lattice size) is 24x24x24x32, and the machine is a switched machine of 128 nodes, then this routine might create a logical machine of 4x4x1x8 nodes, yielding sub-grids of 6x6x24x4, thus collapsing one dimension into the box, and minimizing the communicated sub-grid surface area. The optimization algorithm will likely be machine dependent.

```
const QMP_u32_t* QMP_get_subgrid_dimensions (void);
```

get size of lattice for this node; only valid if `QMP_layout_grid` has been called

These convenience routines will typically be used in one of two fashions:

A:

1. `QMP_layout_grid(..)` // to optimally partition the lattice
2. `QMP_get_subgrid_dimensions()` // to get the sub-lattice size
3. `QMP_get_logical_dimensions()` // to detect if any dimensions collapsed to 1 box, so as to avoid communications in that dimension, as in the example above
4. `QMP_get_node_number()` // to find out who I am

B:

1. `QMP_declare_logical_topology()` // to force a particular logical machine (e.g. a ring with all nodes in the time dimension, to facilitate FFT's)
2. `QMP_layout_grid(..)` // constrained, for example, by the ring topology
3. `QMP_get_subgrid_dimensions()` // (or compute them assuming the ring)
4. `QMP_get_node_number()` // find position within the ring

Communication Declarations

Currently QMP only supports one flavor of message passing. This flavor is meant to be highly repetitive and high performance, and uses a gated message channel paradigm. In this case messaging is done by first declaring the source and destination buffers and node ID (expensive part), then executing the pre-computed I/O operation on demand as rapidly as possible. Destinations are always known – pre-allocated buffers are used (no queuing and so no extra copy). The following functions are used to declare buffers and declare message operations:

Declare Memory Addresses for Messages:

`void* QMP_allocate_aligned_memory (QMP_u32_t nbytes );`

allocates a buffer for messaging, optimally aligned (quadword, page, as appropriate for the machine); enhanced version of “malloc”

`void QMP_free_aligned_memory(void *);`

`QMP_msgmem_t QMP_declare_msgmem (const void * buffer, QMP_u32_t nbytes);`

`QMP_msgmem_t QMP_declare_strided_msgmem (void * base, QMP_u32_t blksize, QMP_u32_t nblocks, QMP_u32_t stride);`

`QMP_msgmem_t QMP_declare_strided_array_msgmem (void ** base, QMP_u32_t* blksize, QMP_u32_t *nblocks, QMP_u32_t* stride);`

`void QMP_free_msgmem (QMP_msgmem_t mm);`

Declare (free) a Receive or Send Operation:

`QMP_msghandle_t QMP_declare_receive_relative (QMP_msgmem_t mm, QMP_s32_t dimension, QMP_s32_t sign, QMP_s32_t priority);`

Declares an endpoint for a message channel operation using the remote node's direction. dimension is an integer, 0, 1, ..., Ndimensions-1, etc., and sign is +1 for forward and backwards. Priority is used to guide underlying resource allocations, where priority = 0 is highest priority

`QMP_msghandle_t QMP_declare_receive_from (QMP_msgmem_t mm, QMP_u32_t sourceNode, QMP_s32_t priority);`

Declares an endpoint for a message channel operation using the remote node's node number.

`QMP_msghandle_t QMP_declare_send_relative (QMP_msgmem_t mm, QMP_s32_t dimension, QMP_s32_t sign, QMP_s32_t priority);`

Declares an endpoint (or a starting point) for a message channel operation using the remote node's direction. dimension is an integer, 0, 1, ..., Ndimensions-1, etc., and sign

is +1 for forward and backwards. Priority is used to guide underlying resource allocations, where priority = 0 is highest priority

`QMP_msghandle_t QMP_declare_send_to (QMP_msgmem_t mm, QMP_s32_t remoteHost, QMP_s32_t priority);`

remoteHost is an integer [0,#nodes-1]

If possible, the receive_from and send_to methods will use the same optimal communications used by QMP_xxx_relative routines. So, for example if the machine is a switched machine, or if the addressed node is in fact an adjacent node on a grid machine, the non-relative methods will have the same effect as the relative methods.

`void QMP_free_msghandle(QMP_msghandle_t mh);`

If the QMP_msgmem_t is a strided memory declaration (for scatter or gather), and the communications hardware cannot directly support strided access, then the creation of the QMP_msghandle_t will also create an appropriately sized temporary buffer, and scatter/gather operations will then be performed by the CPU, with communications then being done to/from this hidden buffer.

In some cases, performance in initiating multiple sends can be improved by collapsing them into a single call. For this purpose, the following function is available and recommended (note that the actual implementation may simply be a loop over the individual calls):

`QMP_msghandle_t QMP_declare_multiple (QMP_msghandle_t * msgh, QMP_u32_t nhandles);`

QMP_msghandle_t's referenced by a multiple style QMP_msghandle_t should not be deleted before the multiple QMP_msghandle_t is deleted (they may be referenced by the implementation of the multiple).

If a multiple operation is started, it is permissible and valid to wait on an individual operation using the QMP_msghandle_t used to construct the multiple operation.

Any errors in declaring a send or receive will return a null pointer, and error info is retrieved via a separate calls:

`const char * QMP_get_error_string (QMP_msghandle_t);`

If the argument is null, returns a global error string from the last operation.

`const QMP_status_t QMP_get_error_number (QMP_msghandle_t);`

If the argument is null, returns a global error number from the last operation.

`const char* QMP_error_string (QMP_status_t status);`

Return an error string for an error code.

Communication Operations

`QMP_status_t QMP_start (QMP_msghandle_t msgh);`

returns `QMP_SUCCESS` if success; can ignore return value and test for completion later
Implementation probably clears a flag which can be tested later. A Send operation is defined as complete when the data has been copied out of the user's buffer, i.e. when the user is free to overwrite the data.

`QMP_bool_t QMP_is_complete (QMP_msghandle_t msgh);`

Implementation probably tests a flag which is set by the underlying library when an operation actually completes. This routine may also do a scatter operation if the receive buffer is strided and the underlying I/O hardware does not directly support strided access.

`QMP_status_t QMP_wait (QMP_msghandle_t msgh);`

This routine will internally attempt to detect and recover from lost messages, and time out after a very long time (e.g. 10 minutes), returning false if the I/O was not completed.

Possible implementation idea for myrinet is to have `QMP_start` set a flag in memory, which is cleared by the NIC on operation completion; `QMP_is_complete` then just tests this memory location. For QCDOC this could operate on control registers.

For strided access on a non-strided access hardware machine, the `QMP_is_complete` call will detect that the hidden internal buffer contains the received data, and will then expand it out into the users strided memory. It will be necessary (for strided calls on a non-strided machine) for the user to call one of these two routines to finish the receive operation.

Global Operations

The following operations are optimized for the hardware, and not necessarily built upon the message passing routines above. All routines return a status code (0 if success), and do operations "in place".

`QMP_status_t QMP_sum_int (QMP_s32_t * i);`

`QMP_status_t QMP_sum_float (QMP_float_t * x);`

`QMP_status_t QMP_sum_double (QMP_double_t * x);`

`QMP_status_t QMP_sum_double_extended (QMP_double_t * x);`
intermediate values kept in extended precision if possible

`QMP_status_t QMP_sum_float_array (QMP_float_t * x, QMP_u32_t length);`
operation is done "in place"

`QMP_status_t QMP_sum_double_array (QMP_double_t * x, QMP_u32_t length);`

QMP_status_t QMP_binary_reduction (void * localvalue, QMP_u32_t nbytes,
QMP_binary_func funcptr);

The binary function has a syntax like:

typedef void (*QMP_binary_func) (void* inout, void* in);

QMP_status_t QMP_max_float (QMP_float_t * x);

QMP_status_t QMP_max_double (QMP_double_t * x);

QMP_status_t QMP_min_float (QMP_float_t * x);

QMP_status_t QMP_min_double (QMP_double_t * x);

QMP_status_t QMP_global_xor (long * lval);

QMP_status_t QMP_broadcast (void *buf, QMP_u32_t nbytes);
broadcast from node 0

QMP_status_t QMP_wait_for_barrier (QMP_s32_t milliseconds);

Wait for a barrier up to timeout value in milliseconds. Return either success or timeout.

Note: not yet use the timeout value in QMP GM implementation.

Error Handling

Some routines may detect potentially fatal errors (e.g. QMP_wait_for). A mechanism similar to signal handling is provided for these errors. The application may declare a message handler; if no handler is declared a default handler is used, which generally prints an error message and exits.

ERRFUNCPTR_t QMP_set_error_function (ERRFUNCPTR_t funcptr);

QMP Status Code

The following QMP status code are suggestions and currently used in our sample implementations.

QMP_SUCCESS = 0

QMP_ERROR = 0x1001

QMP_NOT_INITED

QMP_RTENV_ERR

QMP_CPUINFO_ERR

QMP_NODEINFO_ERR

QMP_NOMEM_ERR

QMP_MEMSIZE_ERR

QMP_HOSTNAME_ERR

QMP_INITSVC_ERR

QMP_TOPOLOGY_EXISTS
QMP_CH_TIMEOUT
QMP_NOTSUPPORTED
QMP_SVC_BUSY
QMP_BAD_MESSAGE
QMP_INVALID_ARG
QMP_INVALID_TOPOLOGY
QMP_NONEIGHBOR_INFO
QMP_MEMSIZE_TOOBIG
QMP_BAD_MEMORY
QMP_NO_PORTS
QMP_NODE_OUTRANGE
QMP_CHDEF_ERR
QMP_MEMUSED_ERR
QMP_INVALID_OP
QMP_TIMEOUT

A QMP status returned by a QMP function may be the status code defined above or may be status codes defined by underlying services such as GM or MPI. Nevertheless the QMP_error_string (QMP_status_t status) will return corresponding error string for all status codes.

C++ Message API

The following presents a C++ binding of the proposed API.

- All classes are defined within the namespace QMP::
- Object definitions are not shown, and may be implementation dependent.
- Names of methods are similar to the C binding (with different naming convention), and discussion details are omitted (see section above).

Machine Initialization and Layout

These operations will be handled by a single class with methods, 1-to-1 mapped onto the corresponding functions in the C API:

```
enum SIGN {PLUS = 1, MINUS = -1};
```

```
namespace QMP {
```

```
    QMP_status_t    init(int* argc, char ***argv,  
                        QMP_smpaddr_type_t option = QMP_SMP_ONE_ADDRESS);
```

```
    void            finalize(void);
```

```
    QMP_u32_t       getSMPCount (void);
```

```
    QMP_ictype_t    getMsgPassingType (void);
```

```
    QMP_u32_t       getNumberOfNodes (void);
```

```
    QMP_u32_t       getNodeNumber(void);
```

```
    const QMP_u32_t getAllocatedNumberOfDimensions(void);
```

```
    const QMP_u32_t* getAllocatedDimensions(void);
```

```
    const QMP_u32_t* getAllocatedCoordinates(void);
```

```
    QMP_bool_t      declareLogicalTopology (const QMP_u32_t *dims, QMP_u32_t ndim);
```

```
    QMP_bool_t      logicalTopologyIsDeclared(void);
```

```
    const QMP_u32_t getLogicalNumberOfDimensions(void);
```

```
    const QMP_u32_t* getLogicalDimensions(void);
```

```
    const QMP_u32_t* getLogicalCoordinates(void);
```

```
    const QMP_u32_t getNodeNumberFrom (const QMP_u32_t *coor);
```

```
    const QMP_u32_t* getLogicalCoordinatedFrom (QMP_u32_t nodenum);
```

```
    QMP_bool_t      layoutGrid (QMP_u32_t *lattDim, QMP_u32_t ndims);
```

```
    const QMP_u32_t* getSubgridDimensions(void);
```

```

const char*      errorString(QMP_status_t code);

void*            allocateAlignedMemory(QMP_u32_t nbytes);
void             freeAlignedMemory(void *ab);

class MessageMemory {

    MessageMemory (void){};
    ~MessageMemory (void);

    MessageMemory (const void *buffer, QMP_u32_t blksize,
                    QMP_u32_t nblocks = 1, QMP_u32_t stride = 0);

    MessageMemory (const void **buffer, QMP_u32_t *blksize,
                    QMP_u32_t *nblocks , QMP_u32_t *stride, QMP_u32_t n);

    void Init      (const void *buffer, QMP_u32_t blksize, QMP_u32_t nblocks = 1,
                    QMP_u32_t stride = 0);

};

class MessageOperation{

    virtual QMP_status_t start(void) = 0;
    virtual QMP_status_t isComplete(void) = 0;
    virtual QMP_status_t wait(void) = 0;
    virtual QMP_status_t getErrorNumber (void) const = 0;
    virtual const char * getErrorString (void) = 0;
};

/**
 * SingleOperation refers to a (possibly multiple strided)
 * MessageOperation to a specific wire or a node.
 */
class SingleOperation : public MessageOperation{

    SingleOperation (void){};
    ~SingleOperation(void){};

```

```

QMP_status_t declareSend (MessageMemory *mm,
                           QMP_s32_t dimension, SIGN sign,
                           QMP_s32_t priority = DEFAULT_PRIORITY);

QMP_status_t declareSend (MessageMemory *mm, QMP_s32_t remoteHost,
                           QMP_s32_t priority = DEFAULT_PRIORITY);

QMP_status_t declareReceive (MessageMemory *mm, QMP_s32_t dimension,
                             SIGN sign, QMP_s32_t priority = DEFAULT_PRIORITY);
QMP_status_t declareReceive (MessageMemory *mm, QMP_s32_t remoteHos,
                             QMP_s32_t priority = DEFAULT_PRIORITY);

/**
 * It is NOT allowed to "re-use" MessageOperations
 */
QMP_status_t start (void);
QMP_status_t isComplete (void);
QMP_status_t wait (void);
QMP_status_t getErrorNumber (void) const;
const char * getErrorString (void);
};

/**
 * MultiOperation refers to a collection of SingleOperation's
 * No 2 SingleOperation's should have same wire or the node.
 * Receiving and Sending operation is considered separate.
 */
class MultiOperation : public MessageOperation{

    MultiOperation (void){};
    ~MultiOperation(void){};

    MultiOperation(SingleOperation *msgops, QMP_u32_t nmsgops);
    MultiOperation(SingleOperation **msgops, QMP_u32_t nmsgops);

    void init(SingleOperation *msgops, QMP_u32_t nmsgops);
    void init(SingleOperation **msgops, QMP_u32_t nmsgops);

```

```

QMP_status_t start(void);
QMP_status_t isComplete(void);
QMP_status_t wait(void);
QMP_status_t getErrorNumber (void) const;
const char * getErrorString (void);
};

```

```

QMP_status_t sumInt(QMP_s32_t *i);
QMP_status_t sumFloat(QMP_float_t *x);
QMP_status_t sumDouble(QMP_double_t *x);

```

```

QMP_status_t sumFloatArray(QMP_float_t *x, QMP_u32_t length);
QMP_status_t sumDoubleArray(QMP_double_t *x, QMP_u32_t length);

```

```

QMP_status_t maxInt(QMP_s32_t *i);
QMP_status_t maxFloat(QMP_float_t *x);
QMP_status_t maxDouble(QMP_double_t *x);

```

```

QMP_status_t minInt(QMP_s32_t *i);
QMP_status_t minFloat(QMP_float_t *x);
QMP_status_t minDouble(QMP_double_t *x);

```

```

QMP_status_t binaryReduction( void *localvalue, QMP_u32_t nbytes,
                             QMP_binary_func funcptr);
typedef void (*QMP_binary_func) (void* inoutvec, void* invec);

```

```

/**

```

```

 * Maybe using QMP_u64_t is more consistent instead of using long as in C binding?
 */

```

```

QMP_status_t globalXor(QMP_u64_t *lval);
QMP_status_t broadcast(void *buf, QMP_u32_t nbytes);
QMP_status_t waitBarrier(QMP_s32_t milliseconds);
}

```