QMP: LQCD Message Passing API

Version 2.0

Oct 29, 2004

This note presents (1) the requirements for message passing within Lattice QCD applications, and (2) an overview of the library and application programming interface (API) designed to implement these requirements. More details can be found online at http://www.lqcd.org/QMP/

Changes between version 1.x and version 2.0:

1. A number of API clarifications have been made in the behavior of the library.
2. A small number of API fixes have been made in rarely used routines.
3. A small number of new routines have been added (mostly convenience routines).
4. Type definitions in the implementation have mostly reverted to standard types, wherever feasible instead of QMP specific types.

Introduction

Lattice QCD (LQCD) is an application domain strongly characterized by regular, repetitive communications between adjacent nodes in an N-dimensional torus (periodic boundary conditions). QMP (QCD Message Passing) is an application programming interface (API) optimized for this style of communications.

The QMP 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 platform specific, 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 (small problem size per node). This is particularly so for the QCDOC custom machine, which has communications hardward capable of supporting a light weight message passing system, but not the full symantics of MPI. These observations are the primary motivation to create an application domain specific message passing abstraction and layer.

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 (albeit at somewhat reduced efficiency). 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, with negligible overhead.

Interspersed with the API description below are some descriptions of how the API could be implemented for switched clusters with programmable network interfaces, and for the QCDOC machine or similar custom machines or custom links. These are meant to more fully illustrate the functionality, and are not intended as the final implementation design.

At the time of writing, the following implementations exist:

1. QMP-MPI — Uses MPI; tested above MPICH-GM, MPICH-SM (shared memory), and MPICH-P4 (sockets)
2. QMP-QCDOC — for the custom QCDOC machine
3. QMP-MVIA — for gigE mesh clusters (or any VIA supported link)

Capability Requirements

1. Barrier call (synchronize all nodes).
2. Send a contiguous message to a given node (the node is identified by a single number, where the application manages lexicographic ordering of nodes in a grid).
3. Send a message to a neighboring node (axis and direction identified by integer numbers; library manages mapping this onto the physical network).
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 of an N-dimensional volume, and other complicated sub-sets of data, and will map efficiently onto any 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 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.
9. Global max for float & double; global exclusive OR for int and long.
10. Arbitrary binary function global reduction.
11. Machine configuration discovery and control (how physical nodes of the machine are arranged into a logical grid).

Library and Interface 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 switched networks with buffering interfaces 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 and performance of the 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 (message). 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 DMA engine.
6. Attempts should be made to minimize bookkeeping overhead on host and any intelligent interfaces.

Hardware Issues

The selected design (certainly not the only one possible) that addresses these performance constraints is 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) 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 to physical 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 overwritten (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 which is gated by A being ready to receive. This is an important distinction in that it is NOT a simple remote memory write. The data won't be written to Q until A is ready.

Hardware notes: this type of synchronized remote memory write is implemented directly in hardware on the QCDOC (which is custom designed for lattice QCD). 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. The desired synchronization could be handled completed by the NIC if sufficient programmability existed (as it does for myrinet). Also, send requests for myrinet are queued in a FIFO, enabling one to satisfy performance requirement 2.

Simplified API Example

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

```
Host A:
    opaqueFromB = QMP_declare_receive_from (remoteNodeB, buffer,  nbytes);
    …
    QMP_start (opaqueFromB);
    …
    QMP_wait (opaqueFromB);

Host B:
    opaqueToA = QMP_declare_send_to (remoteNodeA, buffer, nbytes);
    …
    QMP_start (opaqueToA);
```

For myrinet, this send operation could (via custom code) 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. This is a conceptual design only, however, and would require writing custom myrinet firmware – justified only if a large enough myrinet installation existed. Similar designs could be made for programmable Infiniband NICSs.

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 implemented as 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 summarized in the subsequent section. This C binding hides all intermediate structures as opaque types through typedef's (not shown).

Initialization and Layout

There are two ways to view the multi-node machine, or in other words there are two models for communications. In one model, there is simply a set of nodes numbered 0 to N-1. In the second model the multi-node machine is an N-dimensional grid or torus. Applications are free to use either model, although there are performance optimizations for the torus model on some machines (especially the QCDOC).

Some machines (such as switched clusters) are capable of being configured into a logical torus at run-time. In this discussion, the term "allocated machine" refers to the set of nodes allocated to the job, and the communications fabric tying those nodes together. The term "logical machine" refers to an N-dimensional torus machine in which each node appears to have direct links to each adjacent (in N-space) node.

The following set of calls are used by the application to

1. discover the configuration of the allocated machine and discover the node number of the current node (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 the coordinates of the current node in the logical grid of nodes
3. optimally partition the total lattice (problem) 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 4^{th} 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 cluster implementations, it is 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 changed 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 data types

Certain opaque and special datatypes are defined:

- typedef int QMP_bool_t
- typedef struct QMP_mem_struct_t QMP_mem_t
- typedef void* QMP_msgmem_t
- typedef void* QMP_msghandle_t

In addition the following data types are enumerated data types:

- QMP_status_t (status codes)
- QMP_ictype_t (interconnect type)
  QMP_SWITCH = 0,
  QMP_GRID = 1,
  QMP_MESH = 1,
  QMP_FATTREE = 2
- QMP_thread_level_t (thread safe indication)
  QMP_THREAD_SINGLE,
  QMP_THREAD_FUNNELED,
  QMP_THREAD_SERIALIZED,
  QMP_THREAD_MULTIPLE

See the qmp.h header file in the implementation, or the end of this document, for enumeration values for status codes.

Initialization & Finalization

The following routines are used to initialize and finalize (free resources within) the QMP library:

QMP_status_t QMP_init_msg_passing (int* argc, char*** argv,
QMP_thread_level_t required, QMP_thread_level_t *provided);

initialize communications hardware (if necessary), and retrieve information from the environment such as number of nodes, and ID's of the other nodes;

thread level may be:

QMP_THREAD_SINGLE = the process (application) only uses a single thread per node

QMP_THREAD_FUNNELED = process uses multiple threads per node, but only one will make calls to the QMP library

QMP_THREAD_SERIALIZED = process uses multiple threads to access QMP, but only one at a time per node will use QMP (in time)

QMP_THREAD_MULTIPLE = process uses multiple threads, and multiple threads may make calls to QMP simultaneously

(note: the provided thread level may be higher than the requested)

returns QMP_SUCCESS if success, else an error number; error string obtained via QMP_get_error_string()

QMP_bool_t QMP_is_initialized(void);

test if library is initialized

void QMP_finalize_msg_passing (void);

free any allocated resources

Allocated Machine

The allocated machine routines allow the application to discover the capabilities of the machine at run time:

QMP_ictype_t QMP_get_msg_passing_type (void);

return enum QMP_SWITCH, QMP_GRID, QMP_FATTREE, …

int QMP_get_number_of_nodes (void);

return number of nodes allocated to this job

int QMP_get_node_number(void);

return the node number (0 to N-1) of this node

QMP_bool_t QMP_is_primary_node(void);

int 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 int * QMP_get_allocated_dimensions (void);

size of the allocated grid machine (returns null for switch)

const int * 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 a 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. switched clusters) 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 int * dimensions, int 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 int 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 int * 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 int * 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 int QMP_get_node_number_from (const int node_coordinates[]);

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

const int * QMP_get_logical_coordinates_from (int node_number);

returns coordinates within the logical machine grid of the specified node

The implementation should re-order dimensions and support logical machines with a number of dimensions different from the allocated machine if the hardware can support it (example, unfolding higher dimensions to make a lower dimensionality logical machine. If re-ordering of dimensions occurs, the implementation must do the appropriate mapping for send relative operations.

Note: On the QCDOC it is expected that only the logical machine declared in the environment will give success on this call; sizes must match exactly (implementation constraint).

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.

QMP_bool_t QMP_layout_grid (int latticeDimensions[], int 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 1x4x4x8 nodes, yielding sub-grids of 24x6x6x4, thus collapsing one dimension into the box, and minimizing the communicated sub-grid surface area. The optimization algorithm will likely be machine dependent.

Return value indicates success of operation.

const int * QMP_get_subgrid_dimensions (void);

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

int QMP_get_number_of_subgrid_sites (void);

get the local problem subgrid volume

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

QMP messaging 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 for all but very short messages). The following functions are used to declare buffers and declare message operations:

Declare Memory Addresses for Messages:

QMP_mem_t* QMP_allocate_memory (size_t nbytes);

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

QMP_mem_t* QMP_allocate_aligned_memory (size_t nbytes, size_t alignment, int flags);

allocates a buffer for messaging, aligned to the specified number of bytes, and with memory type specified by flags (architecture dependent); may include:

QMP_MEM_NONCACHE 0x01
QMP_MEM_COMMS 0x02
QMP_MEM_FAST 0x04
QMP_MEM_DEFAULT (QMP_MEM_COMMS|QMP_MEM_FAST)

void* QMP_get_memory_pointer (QMP_mem_t* mem);

get a pointer to the memory buffer allocated by QMP

void QMP_free_memory(QMP_mem_t* mem);

QMP_msgmem_t QMP_declare_msgmem (const void * buffer, size_t nbytes);

QMP_msgmem_t QMP_declare_strided_msgmem (void * base, size_t blksize, int nblocks, ptrdiff_t stride);

QMP_msgmem_t QMP_declare_strided_array_msgmem (void * base[], size_t blksize[], int nblocks[], ptrdiff_t stride[]);

void QMP_free_msgmem (QMP_msgmem_t mem);

Declare (free) a Receive or Send Operation:

Declare an endpoint for a message channel:

QMP_msghandle_t QMP_declare_receive_relative (QMP_msgmem_t mm, int dimension, int sign, int 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

Returns null if it fails to allocate the necessary resources.

QMP_msghandle_t QMP_declare_receive_from (QMP_msgmem_t mm, int sourceNode, int 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, int dimension, int sign, int 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, int remoteHost, int 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[], int nhandles);

Use of declare_multiple invalidates the individual I/O handles and subsequent operations on the individual I/O handles are undefined. QMP_declare_multiple will free the individual I/O handles so that the user will not have to do this.

If a multiple operation is defined and started, it is not valid to wait on an individual operation using the QMP_msghandle_t used to construct the multiple operation.

Recursive use of declare_multiple is allowed (but invalidates the inputs) and the ordering of individual messages if multiple messages are going to / coming from a single destination is the ordering implied by doing all of the individual operations in the declared order.

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 mh);

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 mh);

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. 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. 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 and the data has been received into the hidden buffer.

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.

QMP_status_t QMP_wait_all (QMP_msghandle_t msgh[], int nhandles);

Wait on all I/O handles in the array. May be a mixture of sends and receives.

Restrictions

Sending from overlapping buffers is allowed. Sending to overlapping buffers is undefined (considered an undetected error).

Starting a second send (separate handle) to the same adjacent node before the first completes, or a second receive before the first completes, is allowed. The second send/receive start function is allowed to block. That is, the implementation is not required to implement an I/O queue to support this behavior.

Starting a handle twice before the first operation completes is undefined.

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 (int* i);

QMP_status_t QMP_sum_float (float* x);

QMP_status_t QMP_sum_double (double* x);

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

QMP_status_t QMP_sum_float_array (float x[], int length);
operation is done "in place"

QMP_status_t QMP_sum_double_array (double x[], int length);

QMP_status_t QMP_max_float (float* x);

QMP_status_t QMP_max_double (double* x);

QMP_status_t QMP_min_float (float* x);

QMP_status_t QMP_min_double (double* x);

QMP_status_t QMP_xor_ulong (unsigned long* value);

QMP_status_t QMP_binary_reduction (void * localvalue, size_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_broadcast (void *buf, size_t nbytes);
broadcast from node 0

QMP_status_t QMP_barrier (void);
Wait for a barrier with no timeout.

Profiling, Error Handling and Printing

int QMP_verbose (int level);

Set the error reporting level of the implementation, returning the previous level. Level 0 is the least verbose.

int QMP_profcontrol (int level);

Set the profiling level for implementations containing profiling.

int QMP_printf (const char* format, ...);

int QMP_fprintf (FILE* stream, const char* format, ...);

Like printf and fprintf, with pre-pended rank and host information

int QMP_info (const char *format, ...);

int QMP_error (const char *format, ...);

Printing to stdout, stderr.

void QMP_abort(void);

Kills all processes in the job, used to abnormally terminate a job. Like exit() but attempts to do some cleanup.

QMP Status Code

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

QMP_SUCCESS = 0	*QMP_HOSTNAME_ERR*	*QMP_NONEIGHBOR_INFO*
QMP_ERROR = 0x1001	*QMP_INITSVC_ERR*	*QMP_MEMSIZE_TOOBIG*
QMP_NOT_INITED	*QMP_TOPOLOGY_EXISTS*	*QMP_BAD_MEMORY*
QMP_RTENV_ERR	*QMP_CH_TIMEOUT*	*QMP_NO_PORTS*
QMP_CPUINFO_ERR	*QMP_NOTSUPPORTED*	*QMP_NODE_OUTRANGE*
QMP_NODEINFO_ERR	*QMP_SVC_BUSY*	*QMP_CHDEF_ERR*
QMP_NOMEM_ERR	*QMP_BAD_MESSAGE*	*QMP_MEMUSED_ERR*
QMP_MEMSIZE_ERR	*QMP_INVALID_ARG*	*QMP_INVALID_OP*
	QMP_INVALID_TOPOLOGY	*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 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      initialize(int* argc, char ***argv,
                         QMP_thread_level_t required, QMP_thread_level_t *provided);
  QMP_bool_t        isInitialized(void);
  void              finalize(void);

  QMP_ictype_t      getMsgPassingType (void);
  int               getNumberOfNodes (void);
  int               getNodeNumber(void);
  QMP_bool_t        isPrimaryNode(void);

  const int         getAllocatedNumberOfDimensions(void);
  const int*        getAllocatedDimensions(void);
  int*              getAllocatedCoordinates(void);

  QMP_bool_t        declareLogicalTopology (const int*dims, int ndim);
  QMP_bool_t        logicalTopologyIsDeclared(void);
  const int         getLogicalNumberOfDimensions(void);
  const int*        getLogicalDimensions(void);
  const int*        getLogicalCoordinates(void);

  const int         getNodeNumberFrom (const int  *coor);
  const int *       getLogicalCoordinatesFrom (int  nodenum);

  QMP_bool_t        layoutGrid (int  *lattDim, int  ndims);
  const int *       getSubgridDimensions(void);
  int               getNumberOfSubgridSites(void);
```

Profiling, Error Handling, Printing

```
int             verbose (int level);
int             profcontrol (int level);

int             printf (const char* format, ...);
int             fprintf (FILE* stream, const char* format, ...);
int             info (const char* format, ...);
int             error (const char* format, ...);

void            abort (void);

const char*     errorString(QMP_status_t code);
```

Message Passing

```
class AlignedMemory {

  AlignedMemory (void){};
  ~AlignedMemory (void);

  AlignedMemory (size_t nbytes, size_t alignment = sizeof (ptrdiff_t*), int flags =
        QMP_MEM_DEFAULT);

  void * getBuffer(void);
};

class MessageMemory {

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

  MessageMemory (AlignedMemory am);

  MessageMemory (const void *buffer, size_t  blksize,
                 int  nblocks = 1,  ptrdiff_t  stride = 0);

  MessageMemory (const void **buffer, size_t  *blksize,
                 int  *nblocks ,  ptrdiff_t  *stride,  int  n);

  void init     (const void *buffer, int  blksize, int  nblocks = 1, int  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;

  waitAll (MessageOperation [] ops, n);
};

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

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

why not replace these initialization methods with ReceiveOperation / SendOperation constructors???

```
  QMP_status_t declareSend (MessageMemory *mm,
                            int dimension, int sign,
                            int priority = DEFAULT_PRIORITY);

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

  QMP_status_t declareReceive (MessageMemory *mm, int dimension,
                               int sign,  int priority  = DEFAULT_PRIORITY);
  QMP_status_t declareReceive (MessageMemory *mm, int remoteHos,
                               int 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 two SingleOperation's should have same wire or node.
   *  Receiving and Sending operation is considered separate.  */
class MultiOperation : public MessageOperation{

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

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

  void init (SingleOperation *msgops, int  nmsgops);
  void init (SingleOperation **msgops, int  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 (int *i);
QMP_status_t sumFloat (float *x);
QMP_status_t sumDouble (double *x);
QMP_status_t sumDoubleExtended (double *x);
QMP_status_t sumFloatArray (float *x, int  length);
QMP_status_t sumDoubleArray (double *x, int  length);

QMP_status_t maxInt(int *i);
QMP_status_t maxFloat(float *x);
QMP_status_t maxDouble(double *x);
QMP_status_t minInt(int *i);
QMP_status_t minFloat(float *x);
QMP_status_t minDouble(double *x);

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

QMP_status_t xorUlong(ulong *lval);
QMP_status_t broadcast(void *buf, int  nbytes);
QMP_status_t barrier(void);
}
```