QDP++ Data Parallel Interface for QCD

Version 1.24.1

Robert G. Edwards
SciDAC Software Coordinating Committee

June 15, 2007

1 Introduction

This is a user's guide for the C++ binding for the QDP Data Parallel Applications Programmer Interface developed under the auspices of the U.S. Department of Energy Scientific Discovery through Advanced Computing (SciDAC) program.

The QDP Level 2 API has the following features:

- Provides data parallel operations (logically SIMD) on all sites across the lattice or subsets of these sites.
- Operates on lattice objects, which have an implementation-dependent data layout that is not visible above this API.
- Hides details of how the implementation maps onto a given architecture, namely how the logical problem grid (i.e. lattice) is mapped onto the machine architecture.
- Allows asynchronous (non-blocking) shifts of lattice level objects over any permutation map of sites onto sites. However, from the user's view these instructions appear blocking and in fact may be so in some implementation.
- Provides broadcast operations (filling a lattice quantity from a scalar value(s)), global reduction operations, and lattice-wide operations on various data-type primitives, such as matrices, vectors, and tensor products of matrices (propagators).
- Operator syntax that support complex expression constructions.

2 Datatypes

The N_d dimensional lattice consists of all the space-time sites in the problem space. Lattice data are fields on these sites. A data primitive describes data on a single site. The lattice fields consist of the primitives over all sites. We do not define data types restricted to a subset of the lattice — rather, lattice fields occupy the entire lattice. The primitive types at each site are represented as the (tensor) product space of, for example, a vector space over color components with a vector space over spin components and complex valued elements.

2.1 Type Structure

Generically objects transform under different spaces with a tensor product structure as shown below:

	Lattice		*Color*		*Spin*		*Complexity*
Gauge fields :	`Lattice`	$\otimes$	`Matrix(Nc)`	$\otimes$	`Scalar`	$\otimes$	`Complex`
Fermions :	`Lattice`	$\otimes$	`Vector(Nc)`	$\otimes$	`Vector(Ns)`	$\otimes$	`Complex`
Scalars :	`Scalar`	$\otimes$	`Scalar`	$\otimes$	`Scalar`	$\otimes$	`Scalar`
Propagators :	`Lattice`	$\otimes$	`Matrix(Nc)`	$\otimes$	`Matrix(Ns)`	$\otimes$	`Complex`
Gamma :	`Scalar`	$\otimes$	`Scalar`	$\otimes$	`Matrix(Ns)`	$\otimes$	`Complex`

`Nd` is the number of space-time dimensions
`Nc` is the dimension of the color vector space
`Ns` is the dimension of the spin vector space

Gauge fields can left-multiply fermions via color matrix times color vector but is diagonal in spin space (spin scalar times spin vector). A gamma matrix can right-multiply a propagator (spin matrix times spin matrix) but is diagonal in color space (color matrix times color scalar).

Types in the QDP interface are parameterized by a variety of types including:

- *Word type*: int, float, double, bool. Basic machine types.
- *Reality type*: complex or scalar. This is where the idea of a complex number lives.
- *Primitive type*: scalar, vector, matrix, etc. This is where the concept of a gauge or spin field lives. There can be many more types here.
- *Inner grid type*: scalar or lattice. Supports vector style architectures.
- *Outer grid type*: scalar or lattice. Supports super-scalar style architectures. In combination with Inner grid can support a mixed mode like a super-scalar architecture with short length vector instructions.

There are template classes for each of the type variants listed above. The interface relies heavily on templates for composition - there is very little inheritance. The basic objects are constructed (at the users choice) by compositions like the following:

```
typedef OLattice<PScalar<PColorMatrix<RComplex<float>, Nc> > > LatticeColorMatrix
typedef OLattice<PSpinVector<PColorVector<RComplex<float>, Nc>, Ns> > LatticeFermion
```

The classes PScalar, PSpinVector, PColorMatrix, PColorVector are all subtypes of a primitive type. The relative ordering of the classes is important. It is simply a user convention that spin is used as the second index (second level of type composition) and color is the third. The ordering of types can be changed. From looking at the types one can immediately decide what operations among objects makes sense.

2.2 Generic Names

The linear algebra portion of the QDP API is designed to resemble the functionality that is available in the Level 1 QLA API and the C Level QDP API. Thus the datatypes and function naming conventions are similar. Predefined names for some generic lattice field datatypes are listed in the table below. Because the API is based heavily on templates, the possible types allowed is much larger than listed below.

name	description
`LatticeReal`	real
`LatticeComplex`	complex
`LatticeInt`	integer
`LatticeColorMatrix`	$N_c \times N_c$ complex matrix
`LatticeFermion`	N_s spin, N_c color spinor
`LatticeHalfFermion`	two-spin, N_c color spinor
`LatticeDiracFermion`	four-spin, N_c color spinor
`LatticeStaggeredFermion`	one-spin, N_c color spinor
`LatticeDiracPropagator`	$4N_c \times 4N_c$ complex matrix
`LatticeStaggeredPropagator`	$N_c \times N_c$ complex matrix
`LatticeSeed`	implementation dependent

Single site (lattice wide constant fields) versions of types exist without the `Lattice` preprended. All types and operations defined for QDP live within a C++ namespace called `QDP` thus ensuring no type conflicts with other namespaces.

2.3 Specific Types for Color and Precision

According to the chosen color and precision, names for specific floating point types are constructed from names for generic types. Thus `LatticeColorMatrix` becomes `LatticeColorMatrix`PC, where the precision P is `D` or `F` according to the table below

abbreviation	description
`D`	double precision
`F`	single precision

and C is `2`, `3`, or some arbitrary `N`, if color is a consideration. Note, the value of `N` is an arbitrary compile time constant.

If the datatype carries no color, the color label is omitted. Also, if the number of color components is the same as the compile time constant, then the color label can be omitted. Integers also have no precision label. The specification of precision and number of colors is not needed for functions because of overloading.

For example, the type

```
LatticeDiracFermionF3
```

describes a lattice quantity of single-precision four-spin, three-color spinor field.

2.4 Color and Precision Uniformity

The only place that the number of color or spin components occur is through instance of the global constant variables N_c, and N_s. These are only directly used in the typedef constructions of user defined types. Nothing restricts a user from constructing types for other number of colors. In fact, the use of N_c in the construction of user defined types is simply a convenience for the user, and as such a user can use any integer that is reasonable. The API merely requires that the types used in operations are conforming.

However, in standard coding practice it is assumed that a user keeps one of the precision, color, and spin options in force throughout the compilation. So as a rule all functions in the interface take operands of the same precision, color, and number of spin components. As with data type names, function names come in generic color-, spin- and precision-specific forms, as described in the next section. Exceptions to this rule are functions that explicitly convert from double to single precision and vice versa. If the user choose to adopt color and precision uniformity, then all variables can be defined with generic types and all functions accessed through generic names. The prevailing color is defined through the compile time constant N_c. The interface automatically translates data type names and function names to the appropriate specific type names through typedefs. With such a scheme and careful coding, changing only the compile time N_c and the QDP library converts code from one color and precision choice to another.

2.5 Breaking Color and Precision Uniformity

It is permissible for a user to mix precision and color choices. This is done by declaring variables with specific type names, using functions with specific names, and making appropriate precision conversions when needed.

3 QDP Functions

The QDP functions are grouped into the following categories:

1. Entry and exit from QDP
2. Layout utilities
3. Data parallel functions
4. Data management utilities
5. Subset definition
6. Shift creation
7. I/O utilities
8. Temporary exit and reentry

3.1 Entry and exit from QDP

QDP must be initialized before any other routine can be used. The initialization is broken into two steps – initializing the underlying hardware and initializing the layout.

Initialization of QDP

Prototype	`void QDP_initialize(int *argc, char ***argv)`
Purpose	Places the hardware into a known state.
Example	`QDP_initialize();`

This routine will be responsible for initializing any hardware like the physical layer of the message passing system. For compatibility with QMP, the addresses of the main programs `argc` and `argv` must be passed. They may be modified.

Shutdown of QDP

Prototype	`void QDP_finalize()`
Purpose	Shutdown QDP.
Example	`QDP_finalize();`

This call provides for an orderly shutdown of QDP. It is called by all nodes. It concludes all communications, does housekeeping, if needed and performs a barrier wait for all nodes. Then it returns control to the calling process.

Panic exit from QDP

Prototype	`void QDP_abort(int status)`
Purpose	Panic shutdown of the process.
Example	`QDP_abort(1);`

This routine may be called by one or more nodes. It sends kill signals to all nodes and exits with exit status `status`.

Entry into QDP

Prototype	`void Layout::create()`
Purpose	Starts QDP with layout parameters in `Layout`.
Example	`Layout::create();`

The routine `Layout::create()` is called once by all nodes and starts QDP operations. It calls the layout routine with the parameters set in the namespace `Layout` specifying the layout. The layout is discussed in Section 3.2.

This step is separated from the `QDP_initialize()` above so layout parameters can be read and broadcasted to the nodes. Otherwise the layout parameters have to be set from the environment or fixed in the compilation.

Exit from QDP

Prototype	`void Layout::destroy()`
Purpose	Exits QDP.
Example	`Layout::destroy();`

This call provides for an orderly exit from QDP. It is called by all nodes. It concludes all communications, does housekeeping, if needed and performs a barrier wait for all nodes. The communication layer is not finalized.

3.2 Layout utilities

Routines for constructing the layout are collected in the namespace `Layout`. The *set*ter and *get*ter routines provide a way to set parameters like the lattice size.

The layout creation function determines which nodes get which lattice sites and in what linear order the sites are stored. The `Layout` namespace has entry points that allow a user to inquire about the lattice layout to facilitate accessing single site data from a QDP lattice field. For code written entirely with other QDP calls, these routines may be ignored by the user, with the exception of the useful routine `latticeCoordinate`. However, if a user removes data from a QDP lattice object (see `expose` or `extract`) and wishes to manipulate the data on a site-by-site basis, the global entry points provided here are needed to locate the site data.

Some implementations may have a built-in tightly constrained layout. In flexible implementations there may be several layout choices, thereby allowing the user the freedom to select one that works best with a given application. Furthermore, such implementations may allow the user to create a custom layout to replace one of the standard layouts. As long as the custom layout procedure provides the entry points and functionality described here, compatibility with the remainder of the QDP library is assured.

3.2.1 QDP setup

Layout creation
The layout creation routine `Layout::create()` defined in Section 3.1 generates predefined lattice subsets for specifying even, odd, and global subsets of the lattice. The `rb` set can be dereferenced to produce the `even` and `odd` subsets:

```
Subset even, odd, all,  rb[0], rb[1],  mcb[0], ..., mcb[1 << (Nd+1)]
```

It also creates the nearest-neighbor shifts for each coordinate direction.

Defining the layout
There are set/accessor functions to specify the lattice geometry used in the layout. Generically, the accessors have the form:

Generic	`void Layout::set<something>(<param>)`
Purpose	Set one of the site data layout configurations.
Example	`Layout::setLattSize(size);`

The type of input information needed by the layout is as follows:

1. Number of dimensions N_d. Must be the compile time dimensions.
2. Lattice size (e.g., L_0, L_1, ..., L_{N_d-1})
3. SMP flag

These parameters are accessed and set with the following functions:

Generic	`void Layout::setLattSize(const multi1d<int>& size)`
Purpose	Set the lattice size for the data layout.
Default	No default value. Must always be set.
Example	`Layout::setLattSize(size);`

Generic	`void Layout::setSMPFlag(bool)`
Purpose	Turn on using multi-processor/threading
Default	Default value is false - single thread of execution.
Example	`Layout::setSMPFlag(true);`

Generic	`void Layout::setNumProc(int N)`
Purpose	In a multi-threaded implementation, use `N` processors.
Default	Default value is 1 - single thread of execution.
Example	`Layout::setNumProc(2);`

3.2.2 Generic layout information

The following global entry points are provided in the `Layout` namespace. They provide generic user information.

Returning the spacetime coordinates

Prototype	`LatticeInt Layout::latticeCoordinate(int d)`
Purpose	The `d`th spacetime coordinate.
Example	`LatticeInt coord = Layout::latticeCoordinate(2);`

The call `Layout::latticeCoordinate(d)` returns an integer lattice field with a value on each site equal to the integer value of the `d`th space-time coordinate on that site.

Lattice volume

Prototype	`int Layout::vol()`
Purpose	Return the total lattice volume
Example	`int vol = Layout::vol();`

3.2.3 Entry points specific to the layout

The additional global entry points are provided in the `Layout` namespace. They reveal some information specific to the implementation.

Node number of site

Prototype	`int Layout::nodeNumber(const multi1d<int>& x)`
Purpose	Returns logical node number containing site `x`.
Example	`node = Layout::nodeNumber(x);`

Linear index of site

Prototype	`int Layout::linearSiteIndex(const multi1d<int>& x)`
Purpose	Returns the linearized index for the lattice site `x`.
Example	`int k = Layout::linearSiteIndex(x);`

Map node and linear index to coordinate

Prototype	`multi1d<int> Layout::siteCoords(int node, int index)`
Purpose	Returns site coordinate `x` for the given node `node` and linear index `index`.
Example	`multi1d<int> lc = Layout::siteCoords(n, i);`

Number of sites on a node

Prototype	`int Layout::sitesOnNode()`
Purpose	Returns number of sites assigned to a node.
Example	`int num = Layout::sitesOnNode();`

The linear index returned by `Layout::linearSiteIndex()` ranges from 0 to `Layout:sitesOnNode()`-1

3.3 Data Parallel Functions

Data parallel functions are described in detail in Sec. 9. In the C++ API, there are overloaded functions that can be applied to site or lattice wide objects. Arbitrarily complicated expressions can be built from these functions. The design of the API describes that all operations are to be performed site-wise. The only connection between sites is via a map or shift function.

The class of operations are generically described by site-wise operations (the "linear algebra" part of the API), and shift (or map) versions. The latter generically involves communications among processors in a parallel implementation.

The operator style provided by the API thus allows operations like the following:

```
LatticeFermion A, B;
LatticeColorMatrix U;
B = U * A;
```

From the type declarations

```
typedef OLattice<PScalar<PColorMatrix<RComplex<float>, Nc> > > LatticeColorMatrix
typedef OLattice<PSpinVector<PColorVector<RComplex<float>, Nc>, Ns> > LatticeFermion
```

one can see a `OLattice` multiplies a `OLattice`. At each site, the `U` field is a scalar in spin space, thus a `PScalar` multiplies a `PSpinVector` - a vector in spin space. For each spin component, there is a `PColorMatrix` multipling a `PColorVector`. The multplications involve complex numbers.

Thus we see that mathematically the expression carries out the product

$$\mathrm{B}^i_\alpha(x) = \mathrm{U}^{ij}(x) * \mathrm{A}^j_\alpha(x)$$

for all lattice coordinates `x` belonging to the subset `all`. Here `A` and `B` are objects of lattice Dirac fermion fields and `U` is an onject of type lattice gauge field. The superscripts i, j refer to the color indices and the subscript α refers to the spin index. For each spin and color component, the multiplication is over complex types.

This tensor product factorization of types allows for potentially a huge variety of mathematical objects. The operations between the objects is determined by their tensor product structure.

The API allows for operations to be narrowed to a subset of sites. The infix notation does not allow for extra arguments to be passed to an operation, so the subset is fixed via the target. The API mandates that there is in use in even a complex operation, namely the target specifies the subset to use. To narrow an operation to a specific subset, one specifies the subset in the target as follows:

```
chi[even] = u * psi;
```

which will store the result of the multiplication on only the *even* subset.

The C++ API differs from the C API signficantly in the name of functions. In C++, there is no need for naming conventions for the functions since one can overload the function name on the types of its arguments. More significantly, the C API uses a functional style where the destination of an operation is part of the arguments for an operation, and all functions return void. The C++ API uses an operator/infix style allowing complex expressions to be built.

3.3.1 Constant Arguments

In some cases it is desirable to keep an argument constant over the entire subset. For example the function

```
Complex z;
LatticeFermion c, b;
c[s] = z * b;
```

multiplies a lattice field of color vectors by a complex constant as in

```
c[x] = z*b[x]
```

for `x` in subset `s`.

3.3.2 Functions

In the C++ API all operations are functions that act on their argument and most functions return their results. Except for explicit shift functions and global reductions, these functions are point-wise. The C++ API differs from the C API in that there are no combined operations like adjoint with a multiply. Instead, one simply calls the adjoint function. Thus

```
c = adj(u)*b
```

carries out the product

```
c[x] = adj(u[x])*b[x]
```

for all sites `x` in subset `all`.

3.3.3 Gamma matrices

Multiplication of spin vectors and matrices by γ-matrices is provided.

```
LatticeDiracFermion c, b;
int n;
c = Gamma(n) * b;
```

Where the multiplciation is spin matrix times a spin vector. Right multiplication is also supported.

```
LatticePropagator q, r;
int n;
q = r * Gamma(n);
```

The `Gamma(n)` provides an enumeration of all possible γ-matrix combinations. See the Section 7 for more details.

3.3.4 Shift

A shift function is a type of map that maps sites from one lattice site to another. In general, maps can be permutation maps but there are nearest neighbor shift functions provided by default. See the discussion of shifts below in Section 3.8. Thus

```
  c[s] = shift(b,sign,dir)
```

shifts an object along the direction specified by `dir` and `sign` for all sites `x` in destination subset `s`. Here `sign` $= \pm 1$ and `dir` $= 0, \ldots, Nd - 1$.

3.3.5 Aliasing

The API does not specify the behavior when a source in a data-parallel expression is simultaneously used in the left-hand side of that expression. An example is as follows:

```
  // DO NOT DO THIS
  LatticeColorMatrix a;
  a = a * a;
  LatticeComplex b;
  b = shift(b,sign,dir);
```

This is an instance of the "self-reference" problem.

3.4 Creating and destroying lattice fields

The declaration of an object of some type say `LatticeReal` will call a constructor. The implementation guarantees the object is fully created and all memory needed for it is allocated. Thus, there is no need for the user to use `new` to create an object. The use of pointers is discouraged. When an object goes out of scope, a destructor is called which will guarantee all memory associated with the object is released.

There is no aliasing or referencing of two objects with the same internal data storage. Each object a user can construct has its own unique storage.

3.5 Array container objects

For convenience, the API provides array container classes with much limited facility compared to the Standard Template Library. In particular, one, two, three, and four dimensional array container classes are available. The benefit of two and higher dimension classes is that they can be allocated after they are declared. This is in contrast to the STL technique, which builds multi-dimensional arrays out of nested one-dimensional array, and one must allocate a nested array of array classes by looping over the individual elements allocating each one.

An array of container classes is constructed as follows:

```
multi1d<LatticeComplex> r(Nd); // a 1-D array of LatticeComplex
multi2d<Real> foo(2,3);        // a 2-D array of Real with first index slowest
```

3.6 Function objects

Function objects are used in the constructions of Sets/subsets and maps/shifts. The objects created by maps are themselves function objects. They serve the role as functions, but because of their class structure can also carry state.

A function object has a struct/class declaration. The key part is the function call operator. A generic declaration is something like:

```
struct MyFunction
{
  MyFunction(int dir) : mu(dir) {}
  Real operator()(const int& x)
    {\* operates on x using state held in mu and returns a Real *\}

  int mu;
}
```

A user can then use an object of type MyFunction like a function:

```
MyFunction  foo(37); // hold 37 within foo
int x;
Real boo = foo(x);  // applies foo via operator()
```

3.7 Subsets

It is sometimes convenient to partition the lattice into multiple disjoint subsets (e.g. time slices or checkerboards). Such subsets are defined through a user-supplied function that returns a range of integers $0, 1, 2, \ldots, n-1$, so that if $f(x) = i$, then site x is in partition i. A single subset may also be defined by limiting the range of return values to a single value (i.e. 0). This procedure may be called more than once, and sites may be assigned to more than one subset. Thus, for example an even site may also be assigned to a time slice subset and one of the subsets in a 32-level checkerboard scheme. A subset definition remains valid until is destructor is called.

The layout creation routine `Layout::create()` defined in Section 3.1 generates predefined lattice subsets for specifying even, odd, and global subsets of the lattice. The `rb` set can be dereferenced to produce the `even` and `odd` subsets:

```
Subset even, odd, all,  rb[0], rb[1],  mcb[0], ..., mcb[1 << (Nd+1)]
```

Defining a set Subsets are first defined using a function object through the construction of an object of type `OrderedSet` and `UnorderedSet` whose parent type is `Set`. This function object is a derived type of `SetFunc`. Function objects are described in Section 3.6. Subsets are defined through the parent data type `Subset`. There are two derived useable (concrete) types called `UnorderedSubset` and `OrderedSubset`. The latter type is an optimization that assumes (and run-time enforces) that the subset of sites for a given site layout must be contiguous. It is an error if they are not. Clearly, this assumption is layout dependent and is used mainly by the system wide supplied `even`, `odd`, etc. subsets under compile time flags. A general user subset should be declared to be `UnorderedSubset`. In both ordered and unordered subsets, they are constructed from the corresponding `OrderedSet` and `UnorderedSet`.

Prototype	`UnorderedSet::make(const SetFunc& func)`
	`OrderedSet::make(const SetFunc& func)`
	`int SetFunc::operator()(const multi1d<int>& x)`
	`int SetFunc::numSubsets()`

Purpose	Creates a Set that holds `numSubsets` subsets based on `func`.
Requirements	The `func` is a derived type of `SetFunc` and maps lattice coordinates to a partition number. The function in `func.numSubsets()` returns number of partitions.

Example	`UnorderedSet timeslice;`
	`class timesliceFunc : public SetFunc;`
	`timeslice.make(timesliceFunc);`

Here is an explicit example for a timeslice:

```
struct TimeSliceFunc : public SetFunc
{
  TimeSliceFunc(int dir): mu(dir) {}

  // Simply return the mu'th coordinate
  int operator()(const multi1d<int>& coord)
  {return coord[mu];}

  // The number of subsets is the length of the lattice
  // in direction mu
  int numSubsets() {return Layout::lattSize()[mu];}

  int mu; // state
}

UnorderedSet timeslice;
timeslice.make(TimeSliceFunc(3)) // makes timeslice in direction 3
```

It is permissible to call `UnorderedSet.make()` with a function object having only 1 subset. In this case the partition function must return zero if the site is in the subset and nonzero if not. (Note, this is opposite the “true”, “false” convention in C).

Extracting a subset A subset is returned from indexing a `UnorderedSet` or `OrderedSet` object.

Prototype	`OrderedSubset Set::operator[](int i)`
	`UnorderedSubset Set::operator[](int i)`

Purpose	Returns the `i`-th subset from a `Set` object.

Example	`UnorderedSet timeslice;`
	`Subset origin = timeslice[0];`

The `Set::make()` functions allocates all memory associated with a `Set`. A `Subset` holds a reference info to the original `Set`. A destructor call on a `Set` frees all memory.

Using a subset A subset can be used in an assignment to restrict sites involved in a computation:

```
LatticeComplex r, a, b;
UnorderedSubset s;
r[s] = 17 * a * b;
```

will multiply `17 * a * b` onto `r` only on sites in the subset `s`.

3.8 Maps and shifts

Shifts are general communication operations specified by any permutation of sites. Nearest neighbor shifts are a special case. Thus, for example,

```
LatticeHalfFermion a, r;
r[s] = shift(a,sign,dir);
```

shifts the half fermion field `a` along direction `dir`, forward or backward according to `sign`, placing the result in the field `r`. Nearest neighbor shifts are specified by values of `dir` in the range $[0, N_d - 1]$. The sign is $+1$ for shifts from the positive direction, and -1 for shifts from the negative direction. That is, for `sign`$= +1$ and `dir`$= \mu$, $r(x) = a(x + \hat{\mu})$. For more general permutations, `dir` is missing and `sign` specifies the permutation or its inverse.

The subset restriction applies to the destination field `r`. Thus a nearest neighbor shift operation specifying the even subset shifts odd site values from the source `a` and places them on even site values on the destination field `r`.

Creating shifts for arbitrary permutations The user must first create a function object for use in the map creation as described in Section 3.6. Thus to use the make a map one uses a function object in the map creation:

Prototype	`Map::make(const MapFunc& func)`
Purpose	Creates a map specified by the permutation map function object `func`.
Requirements	The `func` is a derived type of `MapFunc` and must have a `multi1d<int> operator()(const multi1d<int>& d)` member function that maps a source site to `d`.
Result	Creates an object of type map which has a function call `template<class T> T Map::operator()(const T& a)`

Example	`Map naik;`
	 `LatticeReal r,a;`
	 `r = naik(a);`

The coordinate map function object `func` above that is handed to the map creation function `Map::make()` maps lattice coordinates of the the destination to the source lattice coordinates. After construction, the function object of type `Map` can be used like any function via the `operator()`. It can be applied to all QDP objects in an expression.

The function object has an operator that given a coordinate will return the source site coordinates. An example is as follows:

```
struct naikfunc : public MapFunc
{
  naik(int dir) : mu(dir) {}
  multi1d<int> operator()(const multi1d<int>& x)
    {\* maps x to x + 3*mu  where mu is direction vector *\}

  int mu;
}
```

For convenience, there are predefined Map functions named `shift` that can shift by 1 unit backwards or forwards in any lattice direction. They have the form

```
shift(const QDPType& source, int sign, int dir);
```

The construction of a `Map` object allocates all the necessary memory needed for a shift. Similarly, a destructor call on a `Map` object frees memory.

3.9 Temporary entry and exit from QDP

For a variety of reasons it may be necessary to remove data from QDP structures. Conversely, it may be necessary to reinsert data into QDP structures. For example, a highly optimized linear solver may operate outside QDP. The operands would need to be extracted from QDP fields and the eventual solution reinserted. It may also be useful to suspend QDP communications temporarily to gain separate access to the communications layer. For this purpose function calls are provided to put the QDP implementation and/or QDP objects into a known state, extract values, and reinsert them.

Extracting QDP data

Prototype	`void QDP_extract(multi1d<`*Type2*`>& dest, const` *Type1*`& src, const Subset& s)`
Purpose	Copy data values from field `src` to array `dest`.
Type1 *Type2*	All lattice types All corresponding scalar lattice types

Example	`LatticeFermion a;`
	`multi1d<Fermion> r(Layout::sitesOnNode());`
	`QDP_extract(r,a,even);`

The user must allocate the space of size `Layout::sitesOnNode()` for the destination array before calling this function, regardless of the size of the subset.

This function copies the data values contained in the QDP field `src` to the destination field. Only values belonging to the specified subset are copied. Any values in the destination array not associated with the subset are left unmodified. The order of the data is given by `Layout::linearSiteIndex`. Since a copy is made, QDP operations involving the source field may proceed without disruption.

Inserting QDP data

Prototype	`void QDP_insert(`*Type1*`& dest, const multi1d<`*Type2*`>& src, const Subset& s)`
Purpose	Inserts data values from array `src`.
Type1 *Type2*	All lattice types All corresponding scalar lattice types

Example	`multi1d<Fermion> a(Layout::sitesOnNode());`
	`LatticeFermion r;`
	`QDP_insert(r,a,odd);`

Only data associated with the specified subset are inserted. Other values are unmodified. The data site order must conform to `Layout::linearSiteIndex`. This call, analogous to a fill operation, is permitted at any time and does not interfere with QDP operations.

Suspending QDP communications If a user wishes to suspend QDP communications temporarily and carry on communications by other means, it is first necessary to call `QDP_suspend`.

Prototype	`void QDP_suspend(void)`
Purpose	Suspends QDP communications.
Example	`QDP_suspend();`

No QDP shifts can then be initiated until `QDP_resume` is called. However QDP linear algebra operations without shifts may proceed.

Resuming QDP communications To resume QDP communications one uses

Prototype	`void QDP_resume(void)`
Purpose	Restores QDP communications.
Example	`QDP_resume();`

4 Simple I/O utilities

4.1 Basic structure

There are three main types of user accessible classes for *simple* file I/O - Text, XML and Binary. For each of these classes there is a Reader and a Writer version. Each support I/O for *any* QDP defined scalar and lattice quantity as well as the standard C++ builtin types like `int` and `float`. These classes all read/write to one primary node in the computer, namely `Layout::primaryNode()` or node 0. Lattice quantities are read/written lexicographically as one contiguous field with the first index in the lattice size `Layout::lattSize()` varying the fastest. The XML reader functions utilize C++ *exceptions*.

A record structure format is available to store both metadata and binary data. The metadata uses the XML format. The binary I/O functions support more advanced I/O mechanisms and is described in Section 5.

The C++ standard IO streams `cout`, `cerr` and `cin` are, of course, provided by the language but will not work as expected. Namely, the output functions will write on all nodes, and `cin` will try to read from all nodes and fail. QDP predefined glbal objects `QDPIO::cout`, `QDPIO::cerr`, and `QDPIO::cin` are provided as replacements, and will write/read from only the primary node. Output can be selected from any 1 or all nodes for debugging. The QDP implementation does not destroy the standard IO streams.

4.2 Text Reading and Writing

Standard IO streams The global predefined objects `QDPIO::cout`, `QDPIO::cerr`, and `QDPIO::cin` are used like their C++ standard IO streams counterparts. All QDP scalar site fields (e.g., non-lattice) fields can be read and written with these streams. For example, one can read data and be assured the data is appropriately distributed to all nodes.

```
multi1d<int> my_array(4);
QDPIO::cin >> my_array;  // broadcasted to all nodes
Real x;
random(x);
QDPIO::cout << "QDP is GREAT: x = " << x << std::endl; // one copy on output
```

The default behavior is for only the primary node to print on output. Also provided are C++ Standard Library-like IO manipulators that can be used to change this behavior. Namely, IO can be directed from any node which can aid debugging. **Implementation note: this IO manipulator for changing node output is not yet implemented**

TextFileReader member functions and global functions

Open to read	`TextFileReader::TextFileReader(const string& filename)`
	`void TextFileReader::open(const string& filename)`

Close	`TextFileReader::~TextFileReader()`
	`void TextFileReader::close()`

Open?	`bool TextFileReader::is_open()`
Any IO errors?	`bool TextFileReader::fail()`
Input a type `T`	`TextFileReader& operator>>(TextFileReader&, T&)`

TextFileWriter member functions and global functions

Open to write	`TextFileWriter::TextFileWriter(const string& filename)`
	`void TextFileWriter::open(const string& filename)`

Close	`TextFileWriter::~TextFileWriter()`
	`void TextFileWriter::close()`

Open?	`bool TextFileWriter::is_open()`
Any IO errors?	`bool TextFileWriter::fail()`
Output a type `T`	`TextFileWriter& operator<<(TextFileWriter&, const T&)`

To read and write ascii text from the file, use the standard operators familiar in the C++ Standard Library. An example is as follows:

```
TextFileWriter out("foo");
Real a = 1.2;
Complex b = cmplx(Real(-1.1), Real(2.2));
out << a << endl << b << endl;
close(out);

TextFileReader in("foo");
Real a;
Complex b;
in >> a >> b;
close(in);
```

The `TextFileWriter` functions would produce a file "foo" that looks like

```
1.2
-1.1 2.2
```

4.3 XML Reading and Writing

XML is intended as the standard format for user produced human readable data as well as metadata. The XML format is always a tree of key/value pairs with arbitrarily deep nesting. Here, the keys are variable names. The semantics imposed by QDP is that no key can be repeated twice at the same nesting level. Also, the values are considered one of three types – a simple type, a structure, or an array of one of these three types including array.

The XML reader functions utilize C++ *exceptions*.

The path specification for the XML reader functions is XPath. Namely, QDP only *requires* a simple unix like path to reference a tag. With a simple path and nested reading, all data can be read from a document. However, more complicated queries are possible allowing the user to read individual pieces of a document - e.g., a single array element. See the XPath language specification for additional information: http://www.w3.org/TR/xpath.html .

Further details on the document format of various types is given in Section 4.4.

XMLReader member functions and global functions

Read file	`XMLReader::XMLReader(const string& filename)`
	`void XMLReader::open(const string& filename)`

Read stream	`XMLReader::XMLReader(std::istream&)`
	`void XMLReader::open(std::istream&)`

Read buffer	`XMLReader::XMLReader(const XMLBufferWriter&)`
	`void XMLReader::open(const XMLBufferWriter&)`

Close	`XMLReader::~XMLReader()`
	`void XMLReader::close()`

Open?	`bool XMLReader::is_open()`
Any IO errors?	`bool XMLReader::fail()`
Input a type `T`	`void read(XMLReader&, const string& path, T&)`

An example of reading a file is

```
XMLReader xml_in("foo");
int a;
Complex b;
multi1d<Real> c;
multi1d<Complex> d;
read(xml_in, "/bar/a", a);  // primitive type reader
read(xml_in, "/bar/b", b);  // QDP defined struct reader
read(xml_in, "/bar/c", d);  // array of primitive type reader
try {   // try to do the following code, if an exception catch it
  read(xml_in, "/bar/d", d);  // calls read(XMLReader, string, Complex)
} catch( const string& error) {
  cerr << "Error reading /bar/d : " << error << endl;
}
```

The file “foo” might look like the following:

```
<?xml version="1.0"?>
<bar>
 <!-- A simple primitive type -->
 <a>17</d>
 <!-- Considered a structure -->
 <b>
   <re>1.0</re>
   <im>2.0</im>
 </b>

 <!-- A length 3 array of primitive types -->
 <c>1 5.6 7.2</c>

 <!-- A length 2 array of non-simple types -->
 <d>
   <elem>
     <re>1.0</re>
     <im>2.0</im>
   </elem>
   <elem>
     <re>3.0</re>
     <im>4.0</im>
   </elem>
 </d>
</bar>
```

The user can defined their own reader functions following the same overloaded syntax as the predefined ones. This allows one to nest readers.

```
struct MyStruct {int a; Real b;};

void read(XMLReader& xml_in, const string path&, MyStruct& input)
{
  read(xml_in, path + "/a", input.a);
  read(xml_in, path + "/b", input.b);
}

XMLReader xml_in;  // user should initialize here
multi1d<MyStruct> foo;  // array size will be allocated in array reader
read(xml_in, "/root", foo); // will call user defined read above for each element
```

As stated before, the path specification for a read is actually an XPath query. For example, the user can read only one array element in the file “foo” above via an XPath query:

```
XMLReader xml_in("foo");
Complex dd;
read(xml_in, "/bar/d/elem[2]", dd);  // read second array elem of d, e.g. d[1]
```

XMLWriter base class global functions

Start a group
End a group
	`void push(XMLWriter&, const string& name)`
	`void pop(XMLWriter&, const string& name)`

Output a T
Output
	`void write(XMLWriter&, const string& path, const T&)`
	`void write(XMLWriter&, const string& path, const XMLBufferWriter&)`
	`XMLWriter& operator<<(XMLWriter&, const XMLBufferWriter&)`
	`void write(XMLWriter&, const string& path, const XMLReader&)`
	`XMLWriter& operator<<(XMLWriter&, const XMLReader&)`

The `XMLWriter` is an abstract base class for three concrete classes which allow to write into a memory buffer, a file, or write an array of objects in a series of steps.

XMLBufferWriter derived class member functions

Return entire buffer
Return only root element
	`string XMLBufferWriter::str()`
	`string XMLBufferWriter::printRoot()`

XMLFileWriter derived class member functions

File to write	`XMLFileWriter::XMLFileWriter(const string& filename)`
	`void XMLFileWriter::open(const string& filename)`

Close	`XMLFileWriter::~XMLFileWriter()`
	`void XMLFileWriter::close()`

Open?	`bool XMLFileWriter::is_open()`
Any IO errors?	`bool XMLFileWriter::fail()`
Flush	`void XMLFileWriter::flush()`

Similar to the read case, the user can also create a tower of writer functions. In addition, the user can create memory held buffered output that can be used for metadata. Similarly, a user can go back and forth from readers to writers.

```
XMLBufferWriter xml_buf;
push(xml_buf, "bar");
write(xml_buf, "a", 1);  // write /bar/a = 1
pop(xml_buf);

XMLReader xml_in(xml_buf);  // re-parse the xml_buf
int a;
read(xml_in, "/bar/a", a); // now have 1 in a

XMLFileWriter xml_out("foo");
xml_out << xml_in;     // will have ‘‘bar’’ as the root tag
xml_out.close();
```

XMLArrayWriter derived class member and global functions

Constructor	`XMLArrayWriter::XMLArrayWriter(XMLWriter&, int size=-1)`

Close	`XMLArrayWriter::~XMLArrayWriter()`
	`void XMLArrayWriter::close()`

Size	`int XMLArrayWriter::size()`

Start an array
End an array
	`void push(XMLArrayWriter&)`
	`void pop(XMLArrayWriter&)`

The Array class allows one to break writing an array into multiple pieces.

```
XMLFileWriter xml_out("foo");
XMLArrayWriter xml_array(xml_out, 350000);  // Note: a big array size here
push(xml_array, "the_name_of_my_array");
for(int i=0; i < xml_array.size(); ++i)
{
  push(xml_array);  // start next array element - name of tag already defined
  Real foo = i;
  write(xml_array, "foo", foo);
  pop(xml_array);   // finish this array element
}
```

4.3.1 Using Array Containers in Reading and Writing

Array sizes present a special problem in IO. However, within XML the `read(XMLReader&, string path` function can deduce the number of elements an array is expected to hold and will always `resize` the array to the appropriate size. Hence, there is no need to record the array size in the output of a `write(XMLWriter&, string path, const multi1d<T>&)` function call since the corresponding `read` can deduce the size.

This behavior is unlike the `BinaryReader` and `BinaryWriter` functions `read` and `write` of `multi1d`. There, the length of the array is always read/written unless the C-like behavior varieties are used.

4.4 XML document structure

QDP regards the structure of a document as composed of structures, simple types, or arrays of structures or simple types. Simple types are the usual builtin type of C and C++, namely `int`, `float`, `double`, `bool`, etc. In addition, the QDP scalar equivalents `Integer`, `Real`, `Double`, `Boolean`, etc. are also consider simple types. For instance, the code snippet

```
int a = 3;
write(xml_out, "a", a);
```

would produce

```
<a>3</a>
```

indentities the name of a variable of a type with some values. Following the XML Schema specifications, arrays of these simple types have a simple form

```
<!-- produced from writing a multid<int> -->
<a>3 3 4 5</a>
```

Again, following the XML Schema specifications all other objects are considered complex (e.g., complicated) types. Hence, the document snippet

```
<?xml version="1.0"?>
<!-- Considered a structure of simple types, arrays and other structures -->
<bar>
 <!-- A simple primitive type -->
 <a>17</d>

 <!-- Considered a structure -->
 <b>
   <re>1.0</re>
   <im>2.0</im>
 </b>

 <!-- A length 3 array of primitive types -->
 <c>1 5.6 7.2</c>
```

is viewed as a structure of other types:

```
struct bar_t
{
  int a;
  Complex b;
  multi1d<Real> c;
} bar;
```

Hence, one views the `push`/`pop` semantics as a way of dynamically constructing structures.

XML document format

Integer,Real,RealD	<a>3</a>
Boolean	<a>yes</a>
string	<a>hello world</a>
multi1d<int>	<a>1 2 3</a>
Complex	<a> <re>1.2</re> <im>2.0</im> </a>
multi1d<*Type*>	<a> <elem>*Type* </elem> <elem>*Type* </elem> </a>
multi1d<Complex>	<a> <elem> <re>1.2</re> <im>2.0</im> </elem> <elem> <re>3</re> <im>5.0</im> </elem> </a>
ColorVector	<a> <ColorVector> <elem row="0"> <re>0</re> <im>1</im> </elem> <elem row="1"> <re>2</re> <im>3</im> </elem> ... </ColorVector> </a>
ColorMatrix	<a> <ColorMatrix> <elem row="0" col="0"> <re>0</re> <im>1</im> </elem> <elem row="1" col="0"> <re>2</re> <im>3</im> </elem> ... </ColorMatrix> </a>
DiracPropagator	<a> <SpinMatrix> <elem row="0" col="0"> <ColorMatrix> <elem row="0" col="0"> <re>0</re> <im>1</im> </elem> ... </ColorMatrix> </elem> </SpinMatrix> </a>
Lattice*Type*	<a> <OLattice> <elem site="0">*Type* </elem> <elem site="1">*Type* </elem> ... </OLattice> </a>
LatticeReal	<a> <OLattice> <elem site="0">1 </elem> <elem site="1">2 </elem> ... </OLattice> </a>
LatticeColorVector	<a> <OLattice> <elem site="0"> <ColorVector> <elem row="0"> <re>1</re> <im>2</im> </elem> ... </ColorVector> </elem> </OLattice> </a>

A table of the document format for a variable "a" of various types.

4.5 Binary Reading and Writing

BinaryReader base class member functions and global functions

Any IO errors?	`bool BinaryReader::fail()`
Checksum	`BinaryReader::getChecksum()`

Input a type `T`	`BinaryReader& operator>>(BinaryReader&, T&)`
	`void read(BinaryReader&, T&)`
	`void read(BinaryReader&, multi1d<T>&)`
	`void read(BinaryReader&, multi1d<T>&, int num)`
	`void read(BinaryReader&, multi2d<T>&)`
	`void read(BinaryReader&, multi2d<T>&, int num1, int num2)`

BinaryBufferReader member functions and global functions

Constructors	`BinaryBufferReader::BinaryBufferReader(const string& input)`
	`void BinaryBufferReader::open(const string& input)`

Contents	`string BinaryBufferReader::str()`

BinaryFileReader member functions and global functions

Open to read	`BinaryFileReader::BinaryFileReader(const string& filename)`
	`void BinaryFileReader::open(const string& filename)`

Close	`BinaryFileReader::~BinaryFileReader()`
	`void BinaryFileReader::close()`

Open?	`bool BinaryFileReader::is_open()`

BinaryWriter base class member functions and global functions

Any IO errors?	`bool BinaryWriter::fail()`
Flush	`bool BinaryWriter::flush()`
Checksum	`BinaryWriter::getChecksum()`

Output a type `T`	`BinaryWriter& operator<<(BinaryWriter&, const T&)`
	`void write(BinaryWriter&, const T&)`
	`void write(BinaryWriter&, const multi1d<T>&)`
	`void write(BinaryWriter&, const multi1d<T>&, int num)`

BinaryBufferWriter member functions and global functions

Construct	`BinaryBufferWriter::BinaryBufferWriter(const string& input)`
	`void BinaryBufferWriter::open(const string& input)`

Contents	`string BinaryBufferWriter::str()`

BinaryFileWriter member functions and global functions

Open to write	`BinaryFileWriter::BinaryFileWriter(const string& filename)`
	`void BinaryFileWriter::open(const string& filename)`

Close	`BinaryFileWriter::~BinaryFileWriter()`
	`void BinaryFileWriter::close()`

Open?	`bool BinaryFileWriter::is_open()`

Output a type `T`	`BinaryFileWriter& operator<<(BinaryFileWriter&, const T&)`
	`void write(BinaryFileWriter&, const T&)`
	`void write(BinaryFileWriter&, const multi1d<T>&)`
	`void write(BinaryFileWriter&, const multi1d<T>&, int num)`

To read and write ascii text from the file, use the standard operators familiar in the C++ Standard Library. E.g.,

```
BinaryFileWriter out("foo");
Real a;
LatticeColorMatrix b
write(out, a);  // can write this way
out << b;       // or can write this way - have choice of style
close(out);

BinaryFileReader in("foo");
Real a;
LatticeColorMatrix b;
read(in, a);
in >> b;
close(in);
```

4.5.1 Using Arrays Containers in Reading and Writing

The `read` and `write` functions using `BinaryReader` and `BinaryWriter` are special since *metadata* (in this case the length of the array) is read/written along with an object of type `multi1d`.

The standard C behavior is when writing an array, only write whatever number of elements is desired. The problem occurs when reading since number of elements is not known beforehand. The default `write` behavior is to also write the number of elements, and the `read` expects to find this length. The standard C behavior (reading/writing a fixed number of elements) is obtained through an an explicit argument to the call. Specifically:

```
BinaryFileWriter out("foo");
multi1d<Real> a(17);
write(out, a);     // will write an int=a.size() along with a.size() Real elements
write(out, a, 4);  // only writes 4  Real  elements
close(out);
```

```
BinaryFileReader in("foo");
multi1d<Real> a;
read(in, a);  // reads an int=a.size(), a is resized, and reads a.size() elements
read(in, a, 4);  // reads precisely 4 elements, no resizing.
in >> b;
close(in);
```

5 QDP Record I/O utilities

5.1 Overview of File Format

5.1.1 Binary QDP Files

The binary file format has been designed with flexibility in mind. For archiving purposes, the allowable file organization may be further restricted. Here we described the unrestricted format.

Two classes of file volumes are supported: single-file volumes and multiple-file volumes. In the latter case lattice data is scattered among several files for distributed reading and writing. In the former case all the lattice data is contained in a single file.

Single file format Single binary QDP files are composed of a series of one or more application records. A single application record encodes a single QDP field or an array of QDP fields of the same data type. Physics metadata, managed at the convenience of the applications programmer, is associated with the file itself and with each application record as well. Above the API the QDP file is viewed as follows:

- File physics metadata
- Record 1 physics metadata and data
- Record 2 physics metadata and data
- etc.

For example, a file might record a series of staggered fermion eigenvectors for a gauge field configuration. Each record would map to a single field of type `LatticeColorVector`. The file metadata might include information about the gauge field configuration and the record metadata might encode the eigenvalue and an index for the eigenvector.

For another example, the gauge field configuration in four dimensions is represented in QDP as an array of four color matrix fields. The configuration is conventionally written so that the four color matrices associated with each site appear together. A file containing a single gauge field configuration would then consist of a single record containing the array of four color matrices.

The API permits mixing records of different datatypes in the same file. While this practice may be convenient for managing projects, it may be forbidden for archival files.

Additional metadata is automatically managed by QIO (without requiring intervention by the applications programmer) to facilitate the implementation and to check data integrity. Thus the file actually begins with QIO metadata and physics metadata and each application record consists of five logical records. Within QIO the file is viewed as a series of logical records as follows:

- Private file QIO metadata

- User file physics metadata
- Record 1 private QIO metadata
- Record 1 user physics metadata
- Record 1 binary data
- Record 1 private checksum
- Record 2 private QIO metadata
- Record 2 user physics metadata
- Record 2 binary data
- Record 2 private checksum
- etc.

The site order of the binary data is lexicographic according to the site coordinate r_i with the first coordinate r_0 varying most rapidly.

A new format called LIME (Lattice-QCD Interchange Message Encapsulation) is used for packaging the logical records. A feature of this format is the maximum record size is quite - $2^{64} - 1$ bytes. However, file system limitations may require splitting a single file into multiple physical files. No provision is provided for such splitting. It is expected that in time file systems will evolve to allow much larger file sizes. In the interim, facilities like `cpio` and `tar` can be used for file splitting.

Multifile format The API provides for rapid temporary writing of data to scratch disks and reading from scratch disks. This same format may be used for staging files for access by many compute nodes. In this case it is assumed that the files are not intended for longer term storage. Thus the file format in this case is implementation-dependent and not standardized. A specific choice of format is described in the Appendix.

5.1.2 ASCII Metadata Files

The API also provides for reading and writing global values in a standard metadata format from or to a file or a stream buffer. Startup parameters for controlling a simulation could be read in this way. Results of a computation could be written in this way for post processing and analysis.

The XML I/O facilities described in Section 4.3 are used for manipulating the metadata.

5.2 QDP/C++ Record API

As with standard Unix, a file must be opened before reading or writing. However, we distinguish file handles for both cases. If the system provides a parallel file system, it is possible for several processors to read and write a single file. We call this mode "parallel". Otherwise the file is read by a single processor and the data delivered according to the distributed memory layout. The reverse occurs upon writing. We call this mode "serial". To allow user choice where the architecture permits, we provide for requesting either mode. However, the request may be overridden if the system permits only one mode. Upon writing, we allow appending to an existing file.

QDPFileWriter class member functions and global functions

Open	`QDPFileWriter::QDPFileWriter(`
	`    const XMLBufferWriter& file_xml, const string& path,)`
	`    QDP_volfmt_t volfmt, QDPIO_serialparallel_t serpar,`
	`    QDP_filemode_t mode)`
	`void QDPFileWriter::open(const XMLBufferWriter& file_xml,`
	`    const string& path,`
	`    QDP_volfmt_t volfmt, QDPIO_serialparallel_t serpar,`
	`    QDP_filemode_t mode)`

Close	`QDPFileWriter::~QDPFileWriter()`
	`void QDPFileWriter::close()`

Open?	`bool QDPFileWriter::is_open()`
Errors?	`bool QDPFileReader::bad()`

Write a `T`
Array of `T`
	`void write(QDPFileWriter&, XMLBufferWriter& rec_xml, const T&)`
	`void write(QDPFileWriter&, XMLBufferWriter& rec_xml,`
	`    const multi1d<T>&)`

Concrete class for all QDPIO write operations. Here, `write` writes the sites as the slowest varying index and the array indices (muli1d) inside of them. `bad` states if any fatal errors have occurred. The `volfmt` argument is one of

```
QDPIO_SINGLEFILE, QDPIO_MULTIFILE
```

The `serpar` argument is one of

```
QDPIO_SERIAL, QDPIO_PARALLEL
```

and the `mode` argument is one of

```
QDPIO_CREATE, QDPIO_OPEN, QDPIO_APPEND
```

where `QDPIO_CREATE` fails if the file already exists, `QDPIO_OPEN` overwrites the file if it already exists and creates it if not, and `QDPIO_APPEND` fails if the file does not exist and otherwise appends at the end of the file. When appending, the file metadata argument is ignored, since it should already exist.

QDPFileReader class member functions and global functions

Open	`QDPFileReader::QDPFileReader(XMLReader& file_xml,`
	`    const string& path, QDP_serialparallel_t serpar)`
	`void QDPFileReader::open(XMLReader& file_xml,`
	`    const string& path, QDP_serialparallel_t serpar)`

Close	`QDPFileReader::~QDPFileReader()`
	`void QDPFileReader::close()`

Open?	`bool QDPFileReader::is_open()`
EOF?	`bool QDPFileReader::eof()`
Errors?	`bool QDPFileReader::bad()`

Read a `T`
Array of `T`
	`void read(QDPFileReader&, XMLReader& rec_xml, T&)`
	`void read(QDPFileReader&, XMLReader& rec_xml, multi1d<T>&)`

Only xml	`void peek(QDPFileReader&, XMLReader& rec_xml)`
Next record	`void skip(QDPFileReader&)`

Concrete class for all QDPIO read operations. Here, `read(\ldots,multi1d<T>)` expects in the binary file that the sites are the slowest varying index and the array indices (muli1d) inside of them. The `QDP_volfmt_t volfmt` argument is not needed - a file in either `QDPIO_SINGLEFILE` or `QDPIO_MULTIFILE` format will be automatically detected and read appropriately. `peek` returns only the metadata and repositions back to the beginning of the record. `skip` skips to the next logical record - it may read the data and discard it. `bad` states if any fatal errors have occurred. There are no user functions that position within a logical record.

Here is an example of how to use the record I/O facility.

```
XMLBufferWriter file_xml;
QDPFileWriter out(file_xml, "foo", QDPIO_SINGLEFILE,
                  QDPIO_SERIAL, QDPIO_OPEN);
XMLBufferWriter rec_xml;
LatticeColorMatrix a;
write(out, rec_xml, a);
write(out, rec_xml, a);  // for fun, write field twice
close(out);

QDPFileReader in(file_xml, "foo", QDPIO_SERIAL);
read(in, rec_xml, a);
skip(in);
close(in);
```

6 Compilation with QDP

6.1 Generic header and macros

The compilation parameters:
Nd – the number of space-time dimensions
Nc – the dimension of the color vector space
Ns – the dimension of the spin vector space
are defined in `qdp++/include/params.h` . There are macros `ND`, `NC`, `NS` that are used to set the above parameters via

```
const int Nd = ND;
const int Nc = NC;
const int Ns = NS;
```

They are set in the build directories file `include/qdp_config.h` during configuration.

6.2 How to configure QDP++

QDP++ uses the GNU autoconf and automake systems for builds. Help on configuration parameters can be found with

```
% cd qdp++
% configure --help
```

The most important flag is the `--enable-parallel-arch=<some arch>` with the architectural choices `scalar`, `parscalar`, `scalarvec`, `parscalarvec`.

6.3 Nonuniform color and precision

Users wishing to vary color and precision within a single calculation must use specific type names whenever these types and names differ from the prevailing precision and color. Type declarations can be found in `qdp++/include/defs.h` . A convenient definition of a LatticeColorMatrixand LatticeDiracFermionis as follows:

```
typedef OLattice<PScalar<ColorMatrix<Complex<float>, Nc> > > LatticeColorMatrix
typedef OLattice<SpinVector<ColorVector<Complex<float>, Nc>, Ns> > LatticeFermion
```

However, for the user to choose a specific number of colors:

```
const int NN = 17  // work in SU(17)
typedef OLattice<PScalar<ColorMatrix<Complex<float>, NN> > > LatticeColorMatrix17
```

7 Spin Conventions

The following set of γ-matrices are used in four dimensions:

$$\begin{array}{cccc}
\gamma_0 & \begin{pmatrix} 0 & 0 & 0 & i \\ 0 & 0 & i & 0 \\ 0 & -i & 0 & 0 \\ -i & 0 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & i\sigma^1 \\ -i\sigma^1 & 0 \end{pmatrix} & -\sigma^2 \otimes \sigma^1 \\
\gamma_1 & \begin{pmatrix} 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & -i\sigma^2 \\ i\sigma^2 & 0 \end{pmatrix} & \sigma^2 \otimes \sigma^2 \\
\gamma_2 & \begin{pmatrix} 0 & 0 & i & 0 \\ 0 & 0 & 0 & -i \\ -i & 0 & 0 & 0 \\ 0 & i & 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & i\sigma^3 \\ -i\sigma^3 & 0 \end{pmatrix} & -\sigma^2 \otimes \sigma^3 \\
\gamma_3 & \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & \mathbf{1} \\ \mathbf{1} & 0 \end{pmatrix} & \sigma^1 \otimes 1
\end{array}$$

The basis is chiral. All the possible gamma matrix products are represented via

$$\Gamma(n) = \gamma_0^{n_0} \gamma_1^{n_1} \gamma_2^{n_2} \gamma_3^{n_3}$$

where n_i are single bit fields. Since γ_0 comes first the bit for it must come first. So, $\gamma_5 = \gamma_0\gamma_1\gamma_2\gamma_3$ is represented as 1111b = 15d, and $\gamma_0\gamma_1\gamma_3$ is represented as 1011b = 11d (note the ordering). The conventional γ-matrices are

$$\begin{aligned}
\Gamma(1) &= \gamma_0 \\
\Gamma(2) &= \gamma_1 \\
\Gamma(4) &= \gamma_2 \\
\Gamma(8) &= \gamma_3
\end{aligned}$$

This enumeration is γ-basis independent.

8 Implementation Details

The following table lists some of the QDP headers.

name	purpose
`qdp.h`	Master header and QDP utilities
`qdptype.h`	Main class definition
`qdpexpr.h`	Expression class definition
`primitive.h`	Main header for all primitive types
`primscalar.h`	Scalar primitive class and operations
`primmatrix.h`	Matrix primitive and operations
`primvector.h`	Vector primitive and operations
`primseed.h`	Seed (random number) primitive
`reality.h`	Complex number internal class
`simpleword.h`	Machine word-type operations

9 Supported Operations

This section describes in some detail the names and functionality for all functions in the interface involving linear algebra with and without shifts.

All QDP objects are of type QDPType, and QDP functions act on objects of this base class type. Unless otherwise indicated, operations occur on all sites in the specified subset of the target, often an assignment statement or object definition. The indexing of a QDPType returns an lvalue suitable for assignment (but not object definition). It is also used to narrow the lattice sites participating in a global reduction since the result of such a reduction is a lattice scalar, hence are independent of lattice sites.

Supported operations are listed below. Convention: protoyypes are basically of the form:

```
QDPType  unary_function(const QDPType&)
QDPType  binary_function(const QDPType&, const QDPType&)
```

9.1 Subsets and Maps

`Set::make(const SetFunc&)`	Set construction of ordinality num subsets. func maps coordinates to a coloring in [0,num)
`Map::make(const MapFunc&)`	Construct a map function from source sites to the dest site.

9.2 Infix operators

Unary infix (e.g., `operator-`*):*

`-`	:	negation
`+`	:	unary plus
`~`	:	bitwise not
`!`	:	boolean not

Binary infix (e.g., `operator+`*):*

`+`	:	addition
`-`	:	subtraction
`*`	:	multiplication
`/`	:	division
`%`	:	mod
`&`	:	bitwise and
`|`	:	bitwise or
`^`	:	bitwise exclusive or
`<<`	:	left-shift
`>>`	:	right-shift

Comparisons (returning booleans, e.g., `operator<`*):*

```
<, <=, >, >=, ==, !=
&& : and of 2 booleans
|| : or of 2 boolean
```

Assignments (e.g., `operator+=`*):*

```
=, +=, -=, *=, /=, %=, |=, &=, ^=, <<=, >>=
```

Trinary:

```
where(bool,arg1,arg2) : the C trinary "?" operator -> (bool) ? arg1 : arg2
```

9.3 Functions (standard C math lib)

Unary:

```
cos, sin, tan, acos, asin, atan, cosh, sinh, tanh,
exp, log, log10, sqrt,
ceil, floor, fabs
```

Binary:

```
ldexp, pow, fmod, atan2
```

9.4 Additional functions (specific to QDP)

Unary:

`adj`	:	hermitian conjugate (adjoint)
`conj`	:	complex conjugate
`transpose`	:	matrix tranpose, on a scalar it is a nop
`transposeColor`	:	color matrix tranpose, on a scalar it is a nop
`transposeSpin`	:	spin matrix tranpose, on a scalar it is a nop
`trace`	:	matrix trace
`real`	:	real part
`imag`	:	imaginary part
`traceColor`	:	trace over color indices
`traceSpin`	:	trace over spin indices
`timesI`	:	multiplies argument by imag "i"
`localNorm2`	:	on fibers computes trace(adj(source)*source)

Binary:

`cmplx`	:	returns complex object arg1 + i*arg2
`localInnerProduct`	:	at each site computes trace(adj(arg1)*arg2)
`outerProduct`	:	at each site constructs $(\mathrm{arg1}_i * \mathrm{arg2}_j^*)_{ij}$

9.5 In place functions

`random(dest)`	:	uniform random numbers - all components
`gaussian(dest)`	:	uniform random numbers - all components
`copymask(dest,mask,src)`	:	copy src to dest under boolean mask

9.6 Broadcasts

Broadcasts via assignments (via, `operator=`*):*

`<LHS> = <constant>`	:	globally set conforming LHS to constant
`<LHS> = zero`	:	global always set LHS to zero

9.7 Global reductions

`sum(arg1)`	:	sum over lattice indices returning object of same fiber type
`norm2(arg1)`	:	sum(localNorm2(arg1))
`innerProduct(arg1,arg2)`	:	sum(localInnerProduct(arg1,arg2))
`sumMulti(arg1,Set)`	:	sum over each subset of Set returning #subset objects of same fiber type

9.8 Global comparisons

`globalMax(arg1)`	:	maximum across the lattice (simple scalars)
`globalMin(arg1)`	:	minimum across the lattice (simple scalars)

9.9 Accessors

Peeking and poking (accessors) into various component indices of objects.

`peekSite(arg1,multi1d<int> coords)`	:	return object located at lattice coords
`peekColor(arg1,int row,int col)`	:	return color matrix elem row and col
`peekColor(arg1,int row)`	:	return color vector elem row
`peekSpin(arg1,int row,int col)`	:	return spin matrix elem row and col
`peekSpin(arg1,int row)`	:	return spin vector elem row
`pokeSite(dest,src,multi1d<int> coords)`	:	insert into site given by coords
`pokeColor(dest,src,int row,int col)`	:	insert into color matrix elem row and col
`pokeColor(dest,src,int row)`	:	insert into color vector elem row
`pokeSpin(dest,src,int row,int col)`	:	insert into spin matrix elem row and col
`pokeSpin(dest,src,int row)`	:	insert into spin vector elem row

9.10 More exotic functions:

- `spinProject(QDPType psi, int dir, int isign)`
  Applies spin projection $(1+isign*\gamma_\mu)$*`psi` returning a half spin vector or matrix

- `spinReconstruct(QDPType psi, int dir, int isign)`
  Applies spin reconstruction of $(1+isign*\gamma_\mu)$*`psi` returning a full spin vector or matrix

- `quarkContract13(a,b)`
  Epsilon contract 2 quark propagators and return a quark propagator. This is used for diquark constructions. Eventually, it could handle larger Nc. The numbers represent which spin index to sum over.

  The sources and targets must all be propagators but not necessarily of the same lattice type. Effectively, one can use this to construct an anti-quark from a di-quark contraction. In explicit index form, the operation `quarkContract13` does

  $$target^{k'k}_{\alpha\beta} = \epsilon^{ijk}\epsilon^{i'j'k'} * source1^{ii'}_{\rho\alpha} * source2^{jj'}_{\rho\beta}$$

  and is (currently) only appropriate for Nc=3 (or SU(3)).

- `quarkContract14(a,b)`
  Epsilon contract 2 quark propagators and return a quark propagator.

  $$target^{k'k}_{\alpha\beta} = \epsilon^{ijk}\epsilon^{i'j'k'} * source1^{ii'}_{\rho\alpha} * source2^{jj'}_{\beta\rho}$$

- `quarkContract23(a,b)`
  Epsilon contract 2 quark propagators and return a quark propagator.

  $$target^{k'k}_{\alpha\beta} = \epsilon^{ijk}\epsilon^{i'j'k'} * source1^{ii'}_{\alpha\rho} * source2^{jj'}_{\rho\beta}$$

- `quarkContract24(a,b)`
  Epsilon contract 2 quark propagators and return a quark propagator.

  $$target^{k'k}_{\alpha\beta} = \epsilon^{ijk}\epsilon^{i'j'k'} * source1^{ii'}_{\rho\alpha} * source2^{jj'}_{\beta\rho}$$

- `quarkContract12(a,b)`
  Epsilon contract 2 quark propagators and return a quark propagator.

  $$target^{k'k}_{\alpha\beta} = \epsilon^{ijk}\epsilon^{i'j'k'} * source1^{ii'}_{\rho\rho} * source2^{jj'}_{\alpha\beta}$$

- `quarkContract34(a,b)`
  Epsilon contract 2 quark propagators and return a quark propagator.

  $$target^{k'k}_{\alpha\beta} = \epsilon^{ijk}\epsilon^{i'j'k'} * source1^{ii'}_{\alpha\beta} * source2^{jj'}_{\rho\rho}$$

- `colorContract(a,b,c)`
  Epsilon contract 3 color primitives and return a primitive scalar. The sources and targets must all be of the same primitive type (a matrix or vector) but not necessarily of the same lattice type. In explicit index form, the operation colorContract does

  $$target = \epsilon^{ijk}\epsilon^{i'j'k'} * source1^{ii'} * source2^{jj'} * source3^{kk'}$$

  or

  $$target = \epsilon^{ijk} * source1^{i} * source2^{j} * source3^{k}$$

  and is (currently) only appropriate for Nc=3 (or SU(3)).

9.11 Operations on subtypes

Types in the QDP interface are parameterized by a variety of types, and can look like the following:

```
typedef OLattice<PScalar<PColorMatrix<RComplex<float>, Nc> > > LatticeColorMatrix
typedef OLattice<PSpinVector<PColorVector<RComplex<float>, Nc>, Ns> > LatticeFermion
```

- *Word type*: int, float, double, bool. Basic machine types.
- *Reality type*: RComplex or RScalar.
- *Primitive type*: PScalar, PVector, PMatrix, PSeed.
- *Inner grid type*: IScalar or ILattice.
- *Outer grid type*: OScalar or OLattice.

Supported operations for each type level as follows:

Grid type: *OScalar, OLattice, IScalar, ILattice*
All operations listed in Sections 9.2–9.10

Primitive type:

PScalar: All operations listed in Sections 9.2–9.10

PMatrix<N>:

Unary:	`-(PMatrix)`, `+(PMatrix)`
Binary:	`-(PMatrix,PMatrix)`, `+(PMatrix,PMatrix)`, `*(PMatrix,PScalar)`, `*(PScalar,PMatrix)`, `*(PMatrix,PMatrix)`
Comparisons:	none
Assignments:	`=(PMatrix)`, `=(PScalar)`, `-=(PMatrix)`, `+=(PMatrix)`, `*=(PScalar)`
Trinary:	`where`
C-lib funcs:	none
QDP funcs:	all
In place funcs:	all
Reductions:	all

PVector<N>:

Unary:	`-(PVector)`, `+(PVector)`
Binary:	`-(PVector,PVector)`, `+(PVector,PVector)`, `*(PVector,PScalar)`, `*(PScalar,PVector)`, `*(PMatrix,PVector)`
Comparisons:	none
Assignments:	`=(PVector)`, `-=(PVector)`, `+=(PVector)`, `*=(PScalar)`
Trinary:	`where`
C-lib funcs:	none
QDP funcs:	`real`, `imag`, `timesI`, `localNorm2`, `cmplx`, `localInnerProduct`, `outerProduct`
In place funcs:	all
Broadcasts:	`=(Zero)`
Reductions:	all

PSpinMatrix<N>: Inherits same operations as PMatrix

Unary:	`traceSpin`, `transposeSpin`
Binary:	`*(PSpinMatrix,Gamma)`, `*(Gamma,PSpinMatrix)`
Exotic:	`peekSpin`, `pokeSpin`, `spinProjection`, `spinReconstruction`

PSpinVector<N>: Inherits same operations as PVector

Binary:	`*(Gamma,PSpinVector)`
Exotic:	`peekSpin`, `pokeSpin`, `spinProjection`, `spinReconstruction`

PColorMatrix<N>: Inherits same operations as PMatrix

Unary:	`traceColor`, `transposeColor`
Binary:	`*(PColorMatrix,Gamma)`, `*(Gamma,PColorMatrix)`
Exotic:	`peekColor`, `pokeColor`

PColorVector<N>: Inherits same operations as PVector

Binary:	`*(Gamma,PColorVector)`
Exotic:	`peekColor`, `pokeColor`

Reality: *RScalar, RComplex*
All operations listed in Sections 9.2–9.10

Word: *int, float, double, bool*
All operations listed in Sections 9.2–9.10. Only boolean ops allowed on bool.

10 Detailed function description

The purpose of this section is to show some explicit prototypes and usages for the functions described in Section 9. In that section, all the functions are shown with complete information on which operations and their meaning are supported on some combination of types. The purpose of this section is something like the inverse - namely show all the functions and what are some (selected) usages.

10.1 Unary Operations

Elementary unary functions on reals

Syntax	*Type* *func*`(const` *Type*`& a)`
Meaning	$r = \text{func}(a)$
func	cos, sin, tan, acos, asin, atan, sqrt, abs, exp, log, sign
Type	Real, LatticeReal

Elementary unary functions on complex values

Syntax	*Type* *func*`(const` *Type*`& a)`
Meaning	$r = \text{func}(a)$
func	exp, sqrt, log
Type	Complex, LatticeComplex

Assignment operations

Syntax	*Type* `operator=(const` *Type*`& r, const` *Type*`& a)`
Meaning	$r = a$
Type	All numeric types

Shifting

Syntax	*Type* `shift(const` *Type*`& a, int sign, int dir)`
Meaning	$r = a$
Type	All numeric types

Hermitian conjugate

Syntax	*Type* `adj(const` *Type*`& a)`
Meaning	$r = a^\dagger$
Type	Real, Complex, ColorMatrix, DiracPropagator Also corresponding lattice variants

Transpose

Syntax	*Type* `transpose(const` *Type*`& a)`
Meaning	$r = \text{transpose}(a)$
Type	Real, Complex, ColorMatrix, DiracPropagator Also corresponding lattice variants

Transpose of a color matrix

Syntax	*Type* `transposeColor(const` *Type*`& a)`
Meaning	$r^{ij} = a^{ji}$
Type	Real, Complex, ColorMatrix, DiracPropagator Also corresponding lattice variants

Transpose of a spin matrix

Syntax	*Type* `transposeSpin(const` *Type*`& a)`
Meaning	$r_{\alpha\beta} = a_{\beta\alpha}$
Type	Real, Complex, SpinMatrix, DiracPropagator Also corresponding lattice variants

Complex conjugate

Syntax	*Type* `conj(const` *Type*`& a)`
Meaning	$r = a^*$
Type	Real, Complex, ColorMatrix, DiracFermion, DiracPropagator Also corresponding lattice variants

10.2 Type conversion

Types can be precision converted via a conversion function of the destination class.

Convert integer or float to double

Syntax	*Type2 Type2*(`const` *Type1*`& a)`

Example	`LatticeReal a; LatticeRealD r = LatticeRealD(a)`
	`LatticeColorMatrix a; LatticeColorMatrixD r = LatticeColorMatrixD(a)`

Type1 *Type2*	All single precision numeric types All conforming double precision numeric types

Convert double to float

Syntax	*Type2 Type2*(`const` *Type1*`& a)`

Example	`LatticeRealD a; LatticeReal r = LatticeReal(a)`
	`LatticeColorMatrixD a; LatticeColorMatrix r = LatticeColorMatrix(a)`

Type1 *Type2*	All double precision numeric types All conforming single precision numeric types

Integer to real

Syntax	*Type2 Type2*(`const` *Type1*`& a`)
Example	`LatticeInt a; LatticeReal r = LatticeReal(a)`
Type1	All integer precision numeric types
Type2	All conforming real precision numeric types

Real to integer

Syntax	*Type2 Type2*(`const` *Type1*`& a`)
Example	`LatticeReal a; LatticeInt r = LatticeInt(a)`

QDP Type to underlying wordtype

Syntax	*Type2 Type2*(`const` *Type1*`& a`)
Meaning	`r = bool(a);`

Example	`Integer a; int r = toWordType(a);`
	`Boolean a; bool r = toWordType(a);`
	`Real32 a; float r = toWordType(a);`
	`Real64 a; double r = toWordType(a);`
	`Real a; float r = toWordType(a);` *for single precision build*
	`Real a; double r = toWordType(a);` *for double precision build*

Type1	All QDP simple scalar types, like Real
Type2	The underlying word type. In the case of floating point types, the underlying base precision.

QDP types like Real, Double, Integer, Boolean are not primitive types, so an explicit conversion is provided.

Real to float

Syntax	`float toFloat(const Real& a)`
Meaning	`r = float(a);`
Example	`Real a; float r = toFloat(a);`

The QDP type Real is not a primitive type, so an explicit conversion is provided.

Double to double

Syntax	`double toDouble(const RealD& a)`
Meaning	`r = double(a);`
Example	`RealD a; double r = toDouble(a);`

The QDP type RealD is not a primitive type, so an explicit conversion is provided.

Bool to bool

Syntax	`bool toBool(const Boolean& a)`
Meaning	`r = bool(a);`
Example	`Boolean a; bool r = toBool(a);`

The QDP type Boolean is not a primitive type, so an explicit conversion is provided.

10.3 Operations on complex arguments

Convert real and imaginary to complex

Syntax	*Type* `cmplx(const` *Type1*`& a, const` *Type2*`& b)`
Meaning	$\mathrm{Re}\, r = a$, $\mathrm{Im}\, r = b$
Type1 *Type2*	constant, Real, Also corresponding lattice variants constant, Real, Also corresponding lattice variants
Type	Complex, Also corresponding lattice variants

Example	`Reala;`
	`Complex = cmplx(a, 0);`

Real part of complex

Syntax	*Type* `real(const` *Type*`& a)`
Meaning	$r = \mathrm{Re}\, a$

Imaginary part of complex

Syntax	*Type* `imag(const` *Type*`& a)`
Meaning	$r = \mathrm{Im}\, a$

10.4 Component extraction and insertion

Accessing a site object

Syntax	*Type* `peekSite(const Lattice`*Type*`& a, const multi1d<int>& c)`
Meaning	$r = a[x]$

Accessing a color matrix element

Syntax	`LatticeComplex peekColor(const LatticeColorMatrix& a,`
	`int i, int j)`
	`LatticeSpinMatrix peekColor(const LatticeDiracPropagator& a,`
	`int i, int j)`

Meaning	$r = a_{i,j}$

Inserting a color matrix element

Syntax	`LatticeColorMatrix& pokeColor(LatticeColorMatrix& r, const LatticeComplex& a, int i, int j)`
Meaning	$r_{i,j} = a$

Accessing a color vector element

Syntax	`LatticeComplex peekColor(const LatticeColorVector& a, int i)`
	 `LatticeSpinVector peekColor(const LatticeDiracFermion& a, int i)`

Meaning	$r = a_i$

This function will extract the desired color component with all the other indices unchanged.

A lattice color vector is another name (typedef) for a LatticeStaggeredFermion. Namely, an object that is vector in color spin and a scalar in spin space. Together with spin accessors, one can build a LatticeDiracFermion.

Inserting a color vector element

Syntax	`LatticeColorVector& pokeColor(LatticeColorVector& r, const LatticeComplex& a, int i)`
Meaning	$r_i = a$

This function will extract the desired color component with all the other indices unchanged.

A lattice color vector is another name (typedef) for a LatticeStaggeredFermion. Namely, an object that is vector in color spin and a scalar in spin space. Together with spin accessors, one can build a LatticeDiracFermion or a LatticeDiracPropagator.

Accessing a spin matrix element

Syntax	`LatticeComplex peekSpin(const LatticeSpinMatrix& a, int i, int j)`
	 `LatticeColorMatrix peekSpin(const LatticeDiracPropagator& a, int i, int j)`

Meaning	$r = a_{i,j}$

Inserting a spin matrix element

Syntax	`LatticeSpinMatrix& pokeSpin(LatticeSpinMatrix& r, const LatticeComplex& a, int i, int j)`
Meaning	$r_{i,j} = a$

Accessing a spin vector element

Syntax	`LatticeComplex peekSpin(const LatticeSpinVector& a, int i)`
	 `LatticeColorVector peekSpin(const LatticeDiracFermion& a, int i)`

Meaning	$r = a_i$

This function will extract the desired spin component with all the other indices unchanged.

A lattice spin vector is an object that is a vector in spin space and a scalar in color space. Together with color accessors, one can build a LatticeDiracFermion or a LatticeDiracPropagator.

Inserting a spin vector element

Syntax	`LatticeSpinVector& pokeSpin(LatticeSpinVector& r, const LatticeComplex& a, int i)`
Meaning	$r_i = a$

This function will extract the desired spin component with all the other indices unchanged.

A lattice spin vector is an object that is a vector in spin space and a scalar in color space. Together with color accessors, one can build a LatticeDiracFermion or a LatticeDiracPropagator.

Trace of matrix

Syntax	*Type2* `trace(const` *Type1*`& a)`
Meaning	$r = \mathrm{Tr}\, a$
Type1 *Type2*	ColorMatrix, DiracPropagator, Also corresponding lattice variants Complex, Complex, Also corresponding lattice variants

Example	`LatticeColorMatrix a;`
	 `LatticeComplex r = trace(a);`

Traces over all matrix indices. It is an error to trace over a vector index. It will trivially trace a scalar variable.

Color trace of matrix

Syntax	*Type2* `traceColor(const` *Type1*`& a)`
Meaning	$r = \mathrm{Tr}\, a$
Type1 *Type2*	SpinMatrix, Also corresponding lattice variants Complex, Also corresponding lattice variants

Example	`LatticeDiracPropagator a;`
	 `LatticeSpinMatrix r = traceColor(a);`

Traces only over color matrix indices. It is an error to trace over a color vector index. All other indices are left untouched. It will trivially trace a scalar variable.

Spin trace of matrix

Syntax	*Type2* `traceSpin(const` *Type1*`& a)`
Meaning	$r = \mathrm{Tr}\, a$
Type1 *Type2*	DiracPropagator, Also corresponding lattice variants ColorMatrix, Also corresponding lattice variants

Example	`LatticeDiracPropagator a;`
	`LatticeColorMatrix r = traceSpin(a);`

Traces only over spin matrix indices. It is an error to trace over a spin vector index. All other indices are left untouched. It will trivially trace a scalar variable.

Dirac spin projection

Syntax	*Type2* `spinProject(const` *Type1*`& a, int d, int p)`
Meaning	$r = (1 + p\gamma_d)a$
Type1 *Type2*	DiracFermion, Also corresponding lattice variants HalfFermion, Also corresponding lattice variants

Dirac spin reconstruction

Syntax	*Type2* `spinReconstruct(const` *Type1*`& a, int d, int p)`
Meaning	$r = \mathrm{recon}\,(p, d, a)$
Type1 *Type2*	HalfFermion, Also corresponding lattice variants DiracFermion, Also corresponding lattice variants

10.5 Binary Operations with Constants

Multiplication by real constant

Syntax	*Type* `operator*(const Real& a, const` *Type*`& b)` *Type* `operator*(const` *Type*`& b, const Real& a)`
Meaning	$r = a * b$ (a real, constant)
Type	All floating types

Multiplication by complex constant

Syntax	*Type* `operator*(const Real& a, const` *Type*`& b)` *Type* `operator*(const` *Type*`& b, const Real& a)`
Meaning	$r = a * b$ (a complex, constant)
Type	All numeric types

Left multiplication by gamma matrix

Syntax	*Type* `operator*(const` *Gamma*`& a, const` *Type*`& b)`
Meaning	$r = \gamma_d * a$
Gamma	Gamma constructed from an explicit integer in $[0, N_s^2 - 1]$
Type	SpinVector, SpinMatrix, HalfFermion, DiracFermion, DiracPropagator, and similar lattice variants
Example	`r = Gamma(7) * b;`

See Section 7 for details on γ-matrix conventions.

Right multiplication by gamma matrix

Syntax	*Type* `operator*(const` *Type*`& a, const` *Gamma*`& b)`
Meaning	$r = a * \gamma_d$
Gamma	Gamma constructed from an explicit integer in $[0, N_s^2 - 1]$
Type	SpinMatrix, DiracPropagator, and similar lattice variants
Example	`r = a * Gamma(15);`

See Section 7 for details on γ-matrix conventions.

10.6 Binary Operations with Fields

Division of real fields

Syntax	*Type* `operator/(const` *Type*`& a, const` *Type*`& b)`
Meaning	$r = a/b$

Addition

Syntax	*Type* `operator+(const` *Type*`& a, const` *Type*`& b)`
Meaning	$r = a + b$
Type	All numeric types

Subtraction

Syntax	*Type* `operator-(const` *Type*`& a, const` *Type*`& b)`
Meaning	$r = a - b$
Type	All numeric types

Multiplication: uniform types

Syntax	*Type* `operator*(const` *Type*`& a, const` *Type*`& b)`
Meaning	$r = a * b$
Type	constant, Real, Complex, Integer, ColorMatrix, SpinMatrix, DiracPropagator

ColorMatrix matrix from outer product

Syntax	*Type* `outerProduct(const` *Type1*`& a, const` *Type2*`& b)`
Meaning	$r_{i,j} = a_i * b_j^*$
Type1,2	ColorVector, LatticeColorVector
Type	ColorMatrix, LatticeColorMatrix

Left multiplication by gauge matrix

Syntax	*Type* `operator*(const` *Type1*`& a, const` *Type*`& b)`
Meaning	$r = a * b$
Type1	ColorMatrix, LatticeColorMatrix
Type	constant, Complex, ColorMatrix, ColorVector, SpinVector, DiracPropagator, and similar lattice variants

Right multiplication by gauge matrix

Syntax	*Type* `operator*(const` *Type*`& a, const` *Type1*`& b)`
Meaning	$r = a * b$
Type1	ColorMatrix, LatticeColorMatrix
Type	ColorMatrix, SpinMatrix, DiracPropagator, and similar lattice variants

10.7 Boolean and Bit Operations

Comparisons

Syntax	*Type2* `op(const` *Type*`& a, const` *Type1*`& b)`
Meaning	$r = a \, \mathrm{op} \, b$ or $r = \mathrm{op}(a, b)$
op	`<`, `>`, `!=`, `<=`, `>=`, `==`
Type1	Integer, Real, RealD, and similar lattice variants
Type2	Boolean or LatticeBoolean (result is lattice if any arg is lattice)

Elementary binary operations on integers

Syntax	*Type2* `op(const` *Type*`& a, const` *Type1*`& b)`
Meaning	$r = a \, \mathrm{op} \, b$ or $r = \mathrm{op}(a, b)$
op	`<<`, `>>`, `&` (and), `|` (or), `^` (xor), `mod, max, min`

Elementary binary operations on reals

Syntax	*Type* `op(const` *Type1*`& a, const` *Type2*`& b)`
Meaning	$r = a \, \mathrm{op} \, b$ or $r = \mathrm{op}(a, b)$
op	`mod, max, min`
Type	Real, RealD, and similar lattice variants

Boolean Operations

Syntax	*Type* `op(const` *Type*`& a, const` *Type*`& b)`
Meaning	$r = a \,\mathrm{op}\, b$
op	`|` (or), `&` (and), `^` (xor)
Type	Boolean, LatticeBoolean

Syntax	*Type* `op(const` *Type*`& a)`
Meaning	$r = \mathrm{not}\, a$
op	`!` (not)
Type	Boolean, LatticeBoolean

Copymask

Syntax	`void copymask(const` *Type2*`& r, const` *Type1*`& a, const` *Type1*`& b)`
Meaning	$r = b$ if a is true
Type	All numeric types

10.8 Reductions

Global reductions sum over all lattice sites in the subset specified by the left hand side of the assignment.

Norms

Syntax	`Real norm2(`*Type*`& a)`
Meaning	$r = \sum \|a\|^2$
Type	All numeric types

Inner products

Syntax	`Complex innerProduct(`*Type*`& a, const` *Type*`& b)`
Meaning	$r = \sum a^\dagger \cdot b$
Type	All numeric types

Global sums

Syntax	*Type* `sum(const Lattice`*Type*`& a)`
Meaning	$r = \sum a$
Type	All numeric non-lattice scalar types

10.9 Global comparisons

Find the maximum or minimum of a quantity across the lattice. These operations do not have subset variants.

Syntax	*Type* `globalMax(const Lattice`*Type*`& a)`
Meaning	$r = max_{lattice} a(x)$
Type	LatticeRealF, LatticeRealD

Syntax	*Type* `globalMin(const Lattice`*Type*`& a)`
Meaning	$r = min_{lattice} a(x)$
Type	LatticeRealF, LatticeRealD

10.10 Fills

Coordinate function fills

Syntax	`LatticeInt Layout::latticeCoordinate(int d)`
Meaning	$r = f(d)$ for direction `d`.
Purpose	Return the lattice coordinates in direction `d`

The call `Layout::latticeCoordinate(d)` returns an integer lattice field with a value on each site equal to the integer value of the `d`th space-time coordinate on that site.

Constant fills

Syntax	`Lattice`*Type* `operator=(Lattice`*Type*`& r, const` *Type*`& a)`
Meaning	$r = a$ for all sites
Type	All non-lattice objects

Example	`Real a = 2.0;`
	`LatticeReal r = a;`

Constant (or lattice global) fills are always defined for lattice scalar objects broadcasting to all lattice sites. These are broadcasts of a lattice scalar type to a conforming lattice type.

NOTE, one can not fill a LatticeColorVector with a Real.

Syntax	`Lattice`*Type* `operator=(Lattice`*Type*`& r, const` *Type*`& a)`
Meaning	$r = \text{diag}(a, a, \dots)$ (constant a)
Type	Complex, ColorMatrix, SpinMatrix

Example	`Real a = 2.0;`
	`LatticeColorMatrix r = a;`

Only sets the diagonal part of a field to a constant `a` times the identity.

This fill can only be used on primitive types that are scalars or matrices. E.g., it can not be used for a *vector* field since there is no meaning of diagonal. NOTE, a zero cannot be distinguished from a constant like 1. To initialize to zero the `zero` argument must be used.

Zero fills

Syntax	*Type* `operator=(`*Type*`& r, const Zero& zero)`
Meaning	$r = 0$
Type	All numeric types
Example	`LatticeDiracFermion r = zero;`

This is the only way to fill a vector field with a constant (like zero).

Uniform random number fills

Syntax	`void random(`*Type*`& r)`
Meaning	r random, uniform on $[0, 1]$
Type	All floating types

Gaussian random number fills

Syntax	`void gaussian(`*Type*`& r)`
Meaning	r normal Gaussian
Type	All floating types

Seeding the random number generator

Syntax	`void RNG::setrn(const Seed& a)`
Meaning	Initialize the random number generator with seed state `a`

For details see the discussion of the corresponding scalar function `random.h`.

Extracting the random number generator seed

Syntax	`void RNG::savern(Seed& r)`
Meaning	Extract the random number generator into seed state `r`

For details see the discussion of the corresponding scalar function `random.h`.