QDP/C User Manual

Version 1.3.1

This document provides a detailed description of the C implementation of the SciDAC Level 2 QDP Data Parallel interface.

C. DeTar and J.C. Osborn

with the SciDAC Software Committee

Table of Contents

1 Introduction

This is the detailed user's guide for the C binding of 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 on 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 some combined shift and linear algebra instructions for convenience and optimization.
- Provides fill 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).

2 Compilation with QDP

2.1 Generic header and macros

As described above, normally the user selects a prevailing color and precision for the entire calculation. In that case it is permissible to use the generic function names and datatypes, making it possible to change colors and precision with a simple recompilation, if desired. For this purpose the generic header file is `qdp.h`. The following macros must be defined by the user prior to including this header file:

Required Macro	Choices
`QDP_Precision`	'D', 'F'
`QDP_Nc`	number of colors

Single quotes are required around nonnumeric values. In the C implementation of QDP the number of spins is set to 4 when the library is built.

An additional macro is available to the user, but is normally set to its default value according to the macro `QDP_Nc`.

Optional Macro	Choices
`QDP_Colors`	2, 3, 'N'

This is the macro that determines the color namespace for the generic functions. Its default value is *2* when `QDP_Nc` is *2*, *3* when *3*, and *N* otherwise. If for some reason a users wishes to work with the *N* namespace for two or three colors, rather than the more efficient *2* or *3* namespace, he/she may do so by setting this macro to *'N'* explicitly.

A sample preamble for double precision SU(3) reads

```
#define QDP_Precision 'D'
#define QDP_Nc 3
#include <qdp.h>
```

with the include search path set to '`QDP_HOME/include`' and '`QDP_HOME`' set to the home directory for QDP. With such a preamble the generic function names and datatypes are automatically mapped to the appropriate specific types. Of course the precision and color macros can also be defined through a compiler flag, as in

```
gcc -DQDP_Precision=\'D\' -DQDP_Nc=3 ...
```

The single quotes are required and they must each be preceded by a backslash to keep them from being eaten by the shell.

For SU(4) one might do

```
#define QDP_Precision 'F'
#define QDP_Nc 4
#include <qdp.h>
```

The maximum number of colors is determined by the QLA library.

2.2 Libraries

Normally, it is necessary to link five QDP libraries for a given choice of color and precision. Other libraries may be required by the implementation. Routines involving only integers and the random state are common to all choices. Routines involving only real or complex numbers are common to all colors. Thus for single precision SU(3) the required libraries are linked through

```
-lqdp_common -lqdp_int -lqdp_f -lqdp_f3 -lm
```

with the library search path set to 'QDP_HOME/lib'. A complete list of the QDP libraries is given below. Each library will depend on the corresponding QLA library. Also since global sums are done in a higher precision, using them requires linking to QLA libraries of the next higher precision in addition to the corresponding conversion library.

Name	Purpose
'libqdp_common.a'	QDP utilities
'libqdp_int.a'	integer and boolean
'libqdp_f.a'	real and complex single precision
'libqdp_d.a'	real and complex double precision
'libqdp_df.a'	real and complex precision conversion
'libqdp_f2.a'	2 color single precision
'libqdp_d2.a'	2 color double precision
'libqdp_df2.a'	2 color precision conversion
'libqdp_f3.a'	3 color single precision
'libqdp_d3.a'	3 color double precision
'libqdp_df3.a'	3 color precision conversion
'libqdp_fn.a'	N color single precision
'libqdp_dn.a'	N color double precision
'libqdp_dfn.a'	N color precision conversion

2.3 Nonuniform color and precision

Users wishing to vary color and precision within a single calculation must use specific type names and function names whenever these types and names differ from the prevailing precision and color. For example, if an SU(3) calculation is done primarily in single precision, but has double precision components, the following preamble is appropriate:

```
#define QDP_Precision 'F'
#define QDP_Nc 3
#include <qdp.h>
#include <qdp_d.h>
#include <qdp_df.h>
#include <qdp_d3.h>
#include <qdp_df3.h>
```

and the following linkage to get the corresponding libraries:

`-lqdp_common -lqdp_int -lqdp_f -lqdp_f3 -lqdp_d -lqdp_d3 -lqdp_df -lqdp_df3 -lm`

As in the previous example, the single precision components for SU(3) are automatically included through '`qdp.h`'. Then we need the corresponding double precision components. And we also need the `DF` libraries to do conversions between single and double precision. They, too have colored and noncolored members.

The following table lists all the QDP headers.

Name	Purpose
'`qdp.h`'	Master header and QDP utilities
'`qdp_int.h`'	integers, boolean
'`qdp_f.h`'	real, complex, single precision
'`qdp_d.h`'	real, complex, double precision
'`qdp_df.h`'	real, complex, precision conversion
'`qdp_f2.h`'	SU(2), single precision
'`qdp_d2.h`'	SU(2), double precision
'`qdp_df2.h`'	SU(2), precision conversion
'`qdp_f3.h`'	SU(3), single precision
'`qdp_d3.h`'	SU(3), double precision
'`qdp_df3.h`'	SU(3), precision conversion
'`qdp_fn.h`'	SU(N), single precision
'`qdp_dn.h`'	SU(N), double precision
'`qdp_dfn.h`'	SU(N), precision conversion

3 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.

3.1 Generic Names

The linear algebra portion of the QDP API is designed to resemble the Level 1 QLA API. Thus the datatypes and function naming conventions are similar. As with QLA there are two levels of generic naming: fully generic in which both color and precision may be controlled globally through two macros and color-generic in which precision is explicit but not color. Generic naming applies to datatypes, module names, and accessor macros and follows similar rules.

Names for fully generic datatypes are listed in the table below.

Name	Abbreviation	Description
`QDP_RandomState`	`S`	implementation dependent
`QDP_Int`	`I`	integer
`QDP_Real`	`R`	real
`QDP_Complex`	`C`	complex
`QDP_ColorVector`	`V`	one-spin, N_c color spinor
`QDP_HalfFermion`	`H`	two-spin, N_c color spinor
`QDP_DiracFermion`	`D`	four-spin, N_c color spinor
`QDP_ColorMatrix`	`M`	$N_c \times N_c$ complex matrix
`QDP_DiracPropagator`	`P`	$4N_c \times 4N_c$ complex matrix

The name for the corresponding primitive type, also known as the QLA type, is obtained by replacing the `QDP` prefix with a `QLA` prefix. Thus `QLA_ColorMatrix` is the primitive (QLA) type associated with the field `QDP_ColorMatrix`.

Names for color-generic datatypes are obtained by inserting `_F` for single precision or `_D` for double precision after `QDP` where appropriate. Thus `QDP_D_ColorMatrix` specifies a double precision color matrix with color to be set through a global macro.

A long double precision type with abbreviation `Q` is defined for QLA, but currently not for QDP.

3.2 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 QDP_ColorMatrixbecomes `QDP_PC_ColorMatrix`, where the precision `P` is `F` or `D` according to the table below

Abbreviation	Description
`F`	single precision

D	double precision

and C is 2, 3, or N, if color is a consideration, as listed below.

Abbreviation	Description
2	SU(2)
3	SU(3)
N	SU(N)

If the datatype carries no color, the color label is omitted. Integers also have no precision label. Likewise for functions, if none of the arguments carry color, the color label is omitted, and if all numeric arguments are integers, the precision label is omitted. For example, the type `QDP_F3_DiracFermion` describes a single-precision four-spin, three-color spinor field. The general color choice N can also be used for specialized SU(2) or SU(3) at the cost of degrading performance.

3.3 Color and Precision Uniformity

In standard coding practice it is assumed that a user keeps one of the precision and color options in force throughout the compilation. So as a rule all functions in the interface take operands of the same precision and color. As with data type names, function names come in generic and color- 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. These and functions that do not depend on color or precision are divided among thirteen separate libraries. If the user chooses 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 and precision is then defined through macros. The interface automatically translates data type names and function names to the appropriate specific type names through typedefs and macros. With such a scheme and careful coding, changing only two macros and the QDP library converts code from one color and precision choice to another.

3.4 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. In this case it may be necessary to link against a larger set of libraries.

4 QDP Functions

The QDP functions are grouped into the following categories:

Entry and exit from QDP

Layout utilities

Data parallel functions

Data management utilities

Subset definition

Shift creation

I/O utilities

Temporary exit and reentry

Optimization calls

4.1 Entry and exit from QDP

QDP can be started at any time, however field operations cannot be used until the layout is created. In the mean time it will be possible for the user to read input parameters and broadcast them to all nodes. It may also be possible to get parameters from the environment. This procedure is stil under development so, for now, you're on your own.

[**The startup procedure needs more thought - CD**]

4.1.1 Entry to QDP

Syntax	`void QDP_initialize(int *argc, char ***argv);`
Meaning	Starts QDP.
Example	`QDP_initialize(&argc, &argv);`

The routine `QDP_initialize` is called once by all nodes and starts QDP operations. It initializes message passing, but does not setup the site layout (see `QDP_create_layout` and related functions). It also defines the global variable `int QDP_this_node;` specifying the logical node number of the current node.

4.1.2 Exit from QDP

Syntax	`void QDP_finalize(void);`
Meaning	Exits QDP.
Example	`QDP_finalize();`

This call provides for an orderly shutdown. 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.

4.1.3 Panic exit from QDP

Syntax	`void QDP_abort(int status);`
Meaning	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`.

4.1.4 Check for initialization

Syntax	`int QDP_initialized(void);`
Meaning	Checks if QDP is initialized.
Example	`if(!QDP_initialized()) QDP_initialize(&argc, &argv);`

4.1.5 Control profiling

Syntax	`int QDP_profcontrol(int new);`
Meaning	Controls profiling.
Example	`old = QDP_profcontrol(new);`

Profiling results are only available for code that is compiled with the macro `QDP_PROFILE` defined. This must be defined before the qdp.h header is included. When `QDP_finalize` is called a list of QDP function and call times will be sent to stdio on node 0. This function toggles the accumulation of the profiling statistics within the profiled sections of code. A value of zero turns the collection of timing info off and nonzero values turn it on. It is on by default.

4.1.6 Control checking communications

Syntax	`int QDP_check_comm(int new);`
Meaning	Controls checking communications.
Example	`old = QDP_check_comm(new);`

A nonzero value turns the internal communications checksums on and zero turns it off. It is off by default.

4.2 Layout utilities

The layout routine determines which nodes get which lattice sites and in what linear order the sites are stored. It has entry points that allow a user to access single site data extracted from a QDP lattice field. The layout must be created before any operations on QDP field objects are allowed. If a user removes data from a QDP lattice object (see `QDP_expose` or `QDP_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.

4.2.1 Defining the layout

Prior to creating the layout the layout parameters must be defined. This is done through function calls.

Syntax	`void QDP_set_latsize(int nd, int size[]);`
Meaning	Sets number of spacetime dimensions and lattice size. No default. Must always be set.
Example	`QDP_set_latsize(4, size);`

Syntax	`void QDP_create_layout(void);`
Meaning	Lays out the sites.
Example	`QDP_create_layout();`

All layout parameters must be initialized through the `set` function calls prior to creating the layout.

After creating the layout the following global variables are accessible. The predefined lattice subsets for specifying even, odd, and global subsets of the lattice:

`QDP_Subset QDP_even, QDP_odd, QDP_all;`

The even and odd subsets are elements of a two-element subset array `QDP_even_odd`, such that

```
QDP_even = QDP_even_odd[0];
QDP_odd  = QDP_even_odd[1];
```

It also creates the nearest-neighbor shifts

`QDP_shift QDP_neighbor[];`

for each coordinate direction. And finally the variable

`int QDP_sites_on_node;`

gives the number of sites assigned to a node by the layout utility. Note that this may vary between nodes.

The following global entry points are provided by the `QDP_create_layout` procedure:

4.2.2 Number of dimensions

Syntax	`int QDP_ndim(void);`
Meaning	Returns the number of dimensions.
Example	`ndim = QDP_ndim();`

4.2.3 Length of lattice in a given direction

Syntax	`int QDP_coord_size(int i);`
Meaning	Returns length of lattice in direction `i`.
Example	`nx = QDP_coord_size(0);`

4.2.4 Length of lattice in all directions

Syntax	`void QDP_latsize(int latsize[]);`
Meaning	Returns lattice dimensions into array `latsize`.
Example	`QDP_latsize(latsize);`

4.2.5 Length of lattice in all directions

Syntax	`size_t QDP_volume(void);`
Meaning	Returns lattice volume.
Example	`vol = QDP_volume();`

4.2.6 Node number of site

Syntax	`int QDP_node_number(int x[]);`
Meaning	Returns logical node number containing site `x`.
Example	`node = QDP_node_number(x);`

4.2.7 Linear index of site

Syntax	`int QDP_index(int x[]);`
Meaning	Returns the linearized index for the lattice site `x`.
Example	`k = QDP_index(x);`

The linear index returned by `QDP_index` ranges from 0 to `QDP_sites_on_node` - 1.

4.2.8 Number of sites on a node

Syntax	`int QDP_numsites(int node);`
Meaning	Return the number of sites on a node. Same as `QDP_sites_on_node` if `node = QDP_this_node`
Example	`k = QDP_numsites(i);`

4.2.9 Map node and linear index to coordinate

Syntax	`void QDP_get_coords(int x[], int node, int index);`
Meaning	Returns site coordinates `x` for the given node `node` and linear index `index`.
Example	`QDP_get_coords(x, 0, 31);`

4.2.10 Defining the spacetime coordinate

Syntax	`void QDP_I_eq_coord(QDP_Int *r, int i);`
Meaning	The `i`th spacetime coordinate.
Example	`QDP_Int *coord_z;` `QDP_I_eq_coord(coord_z, 2);`

The call `QDP_I_eq_coord(&coord[i],i)` fills an integer lattice field `coord[i]` with a value on each site equal to the integer value of the ith space-time coordinate on that site.

4.3 Naming Conventions for Data Parallel Functions

Data parallel functions are described in detail in Chapter 5 [Function Details], page 19. Here we desribe the naming conventions. Data parallel function names are constructed with a pattern that suggests their functionality. Thus the function

```
QDP_V_eq_M_times_V(c, u, b, s);
```

carries out the product

```
c[x] = u[x]*b[x];
```

for all lattice coordinates `x` belonging to the subset `s`. where c and b are pointers to lattice staggered fermion vector fields and u is a pointer to a lattice color matrix field. The elements of the function name are separated by an underscore (`_`) for readability. All function names in this interface begin with `QDP`. The specific name continues with a precision and color label as in `QDP_F3_V_eq_M_times_V` for single precision *SU(3)*. Then comes a string of elements that mimics the algebraic expression. The next character `V` abbreviates the type for the destination operand, in this case the argument `c`. The abbreviations are listed in Chapter 3 [Datatypes], page 5. The next string `eq` specifies the assignment operator. In this case it is a straight replacement, but modifications are also supported, as described below. Then comes the first rhs operand type `M` followed by a string `times` specifying the operation and a character `V` specifying the second rhs operand type.

Supported variants of the assignment operator are tabulated below.

Abbreviation	Meaning
eq	$=$
peq	$+=$
meq	$-=$
eqm	$=-$

Some functions allow all of these and some take only a simple replacement (eq).

4.3.1 Constant Arguments

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

```
QDP_V_eq_c_times_V(c,z,b,s)
```

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

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

for `x` in subset `s`. In this case we specify that the argument is constant (coordinate-independent) by writing the type abbreviation in lower case: `c`.

4.3.2 Color argument for *SU(N)*

For the general color case *SU(N)* the specific function requires an extra argument giving the number of colors. It always comes first. Thus in the above example we would write

```
QDP_FN_V_eq_c_times_V(nc, c, z, b, s);
```

where `nc` specifies the number of colors. In normal practice, the variable `nc` should be replaced by the required user-defined macro `QDP_Nc`specifying the prevailing number of colors. The generic function is actually a macro and is automatically converted to this usage with `QDP_Nc`for the first argument. However, if the specific name is used, the user must supply the argument.

4.3.3 Adjoint

The adjoint of an operand is specified by a suffix `a` after the type abbreviation. Thus

```
QDP_V_eq_Ma_times_V(c, u, b, s);
```

carries out the product

```
c[x] = adjoint(u[x])*b[x];
```

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

4.3.4 Shift

A shift in an operand is specified by a prefix lowercase `s` before the type abbreviation. (See the discussion of shifts below.) Thus

```
QDP_V_eq_sV(c, b, dir, sign, s);
```

shifts staggered fermion data along the direction specified by `dir` and `sign` for all sites `x` in destination subset `s`.

4.3.5 Operations on arrays of fields

Some of the routines can operate on multiple fields at a time. These functions are designated by placing a `v` in front of the *eqop* operator. The allowed *eqop*'s are then `veq`, `vpeq`, `vmeq` and `veqm`. All arguments to the function are then made into arrays of the type the original argument was *except* for the subset. Even scalar values (QLA types) and other parameters are turned into arrays. The length of the arrays is then given as the last argument. For example the function

```
QDP_F3_V_peq_M_times_V( QDP_F3_ColorVector *r, QDP_F3_ColorMatrix *a,
  QDP_F3_ColorVector *b, QDP_Subset s );
```

becomes

```
QDP_F3_V_peq_M_times_V( QDP_F3_ColorVector *r[], QDP_F3_ColorMatrix *a[],
  QDP_F3_ColorVector *b[], QDP_Subset s, int n );
```

This has the same effect as the following code

```
for(i=0; i<n; i++) QDP_F3_V_peq_M_times_V( r[i], a[i], b[i], s );
```

however it may be implemented in a more efficient manner. It is permissible to have multiple array elements point to the same field. The result will always agree with that of the above `for` loop.

4.4 Creating and destroying lattice fields

All QDP functions expect that lattice field arguments point to fields that have already been created. The sole exception to this rule is the creation utility itself, which returns a pointer

4.4.1 Creating a lattice field

Syntax	`Type * QDP_create_T(void);`
Meaning	Creates lattice field of type *Type*.
Type	`S, I, R, C, V, H, D, M, P`
Example	`u = QDP_create_M();`

In prototype specifications throughout this document the notation *Type* specifies the generic or specific datatype name matching the abbreviation T according to the table in Chapter 3 [Datatypes], page 5.

4.4.2 Destroying a lattice field

Syntax	`void QDP_destroy_T(Type *a);`
Meaning	Frees memory associated with field `a`.
Type	`S, I, R, C, V, H, D, M, P`
Example	`QDP_destroy_M(u);`

4.5 Subsets

All QDP linear algebra and shift operations require specifying the subset of the lattice on which the operation is performed. The subset may be the entire lattice. When defining subsets, it is often 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,...,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 `QDP_destroy_subset` is called.

4.5.1 Defining a subset

Subsets are defined through the data type `QDP_Subset`

Syntax	`QDP_Subset * QDP_create_subset(int (*func)(int x[], void *args), void *args, int argsize, int n);`
Meaning	Creates an array of `n` subsets based on `func`.
Example	`QDP_Subset ts[nt];` `ts = QDP_create_subset(timeslice, NULL, 0, nt);`

The extra arguments `args` are passed directly to the function and saved in case the subset function is needed again when doing shifts involving the subset. Therefore the function should not depend on any other global parameters that may change later in the program. It is permissible to call `QDP_create_subset` with n = 1. In this case the 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).

4.5.2 Destroying subsets

Syntax	`void QDP_destroy_subset(QDP_subset s[]);`
Meaning	Destroys all subsets created with `s`.
Example	`QDP_destroy_subset(ts);`

This procedure frees all memory associated with the subset object *s*. The `QDP_subset *` value *s* should be the object returned by `QDP_create_subset`. All subsets in the array *s* are destroyed.

4.5.3 Reductions on subsets

Reduction operations (norms, inner products, global sums) come in two variants according to whether the result is computed on a single subset of the lattice or on multiple subsets. Thus the operation

```
QLA_Complex z;
QDP_ColorVector *a, *b;
QDP_c_eq_V_dot_V(&z, a, b, QDP_even);
```

sums the dot product of the lattice staggered fermion fields *a* and *b* on the even sites and stores the result in *z*. The operation

```
QLA_Complex z[nt];
QDP_ColorVector *a, *b;
QDP_c_eq_V_dot_V_multi(z, a, b, ts, nt);
```

with the timeslice subsets illustrated above computes the dot product summed separately on each timeslice and stores the sums in the array *z*, so that the value in *z[i]* results from the sum on the subset *ts[i]*.

4.6 Shifts

Shifts are general communication operations specified by any permutation of sites. Nearest neighbor shifts are a special case and are preinitialized by `QDP_initialize`. Arbitrary displacement shifts are an intermediate generalization and are created with `QDP_create_shift`. Arbitrary permutations are created with `QDP_create_map`. However they are created, all shifts are specified by a *direction* label *dir* of type `QDP_Shift` and a sign *sign* of type `QDP_ShiftDir` that takes one of two predefined values `QDP_forward` and `QDP_backward`.

Shifts are treated syntactically as a modification of a QDP argument and are specified with a prefix *s* before the type abbreviation for the shifted field. Thus, for example,

```
QDP_H_eq_sH(r, a, dir, sign, s);
```

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 the global shift `QDP_neighbor[mu]` with `mu` in the range [0, Ndim-1]. The sign is `QDP_forward` for shifts from the positive direction, and `QDP_backward` for shifts from the negative direction. That is, for `QDP_forward` and `dir = QDP_neighbor[mu]`, r(x) = a(x+mu). For more general shifts, the direction *dir* is specified by the object returned by `QDP_create_shift` or `QDP_create_map` and *sign* must be either `QDP_forward` or `QDP_backward` to specify the permutation or its inverse, respectively.

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*.

4.6.1 Creating displacement shifts

Syntax	`QDP_Shift QDP_create_shift(int d[]);`
Meaning	Creates a shift defined by the displacement vector `d`.

Example	`int d[4] = {0,1,2,0};`
	`QDP_Shift knight[4][4];`
	`knight[2][3] = QDP_create_shift(d);`

Calling with a displacement vector {1,0,0,0} would reproduce the shift `QDP_neighbor[0]`.

4.6.2 Creating arbitrary permutations

Syntax	`QDP_Shift QDP_create_map(int *(func*)(int coord[Nd], void *args),` `void *args, int argsize);`
Meaning	Creates a shift specified by the permutation map `func`.

Example	`int mu = 1;`
	`QDP_Shift mirror[4];`
	`mirror[mu] = QDP_create_map(reflect, &mu, sizeof(mu));`

The return value is used in the various linear algebra calls involving shifts. The arguments *args* are passed through to the calling function. The *argsize* parameter specifies the byte length of the argument array or structure.

The implementation may choose to postpone construction of a shift. Thus it is required that the callback function *func* be static and invariant, i.e. a function call with the same arguments must give the same result, even if the call is postponed. The parameters *args* are copied at the moment the shift is created, however, so they may be volatile. The size argument *argsize* makes copying possible.

4.6.3 Destroying a shift

The corresponding destruction function is `QDP_destroy_shift`.

Syntax	`void QDP_destroy_shift(QDP_Shift dir);`
Meaning	Frees memory associated with the map *dir*.
Example	`QDP_destroy_shift(dir);`

4.7 I/O utilities

[Under development.]

4.8 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.

4.8.1 Exposing QDP data

Syntax	`QLA_Type * QDP_expose_T(Type *src);`
Meaning	Deliver data values from field `src`.
Type	`I, R, C, V, H, D, M, P`
Example	`r = QDP_expose_M(a);`

This function grants direct access to the data values contained in the QDP field *src*. The return value is a pointer to an array of QLA data *dest* of type *T*. The order of the data is given by `QDP_index`. No QDP operations are permitted on exposed data until `QDP_reset` is called. (See next.)

4.8.2 Returning control of QDP data

Syntax	`void QDP_reset_T(Type *field);`
Meaning	Returns control of data values to QDP.
Type	`I, R, C, V, H, D, M, P`
Example	`QDP_reset_M(r);`

This call signals to QDP that the user is ready to resume QDP operations with the data in the specified field.

4.8.3 Extracting QDP data

Syntax	`void QDP_extract_T(QLA_Type *dest, Type *src);`
Meaning	Copy data values from field `src` to array `dest`.
Type	`I, R, C, V, H, D, M, P`
Example	`QDP_extract_M(r, a, QDP_even);`

The user must allocate space of size `QDP_sites_on_node*sizeof(QLA_Type)` 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 `QDP_index`. Since a copy is made, QDP operations involving the source field may proceed without disruption.

4.8.4 Inserting QDP data

Syntax	`void QDP_insert_`*T*`(`*Type* `*dest, `*QLA_Type* `*src);`
Meaning	Inserts data values from QLA array `src`.
Type	`I, R, C, V, H, D, M, P`
Example	`QDP_insert_M(a, r);`

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

4.8.5 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_comm`.

Syntax	`void QDP_suspend_comm(void);`
Meaning	Suspends QDP communications.
Example	`QDP_suspend_comm();`

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

4.8.6 Resuming QDP communications

To resume QDP communications one uses

Syntax	`void QDP_resume_comm(void);`
Meaning	Restores QDP communications.
Example	`QDP_resume_comm();`

4.9 Optimization Calls

The following procedure is included to aid in optimization of the QDP implementation

4.9.1 Marking discarded data

Syntax	`void QDP_discard_`*T*`(`*Type* `*a);`
Meaning	Indicates data in `a` is no longer needed.
Type	`I, R, C, V, H, D, M, P`
Example	`QDP_discard_M(utemp);`

The field is not destroyed and memory is not released. For that purpose, see `QDP_destroy`. This call allows the implementation to cancel the deferred resolution of a lazy shift. It is a runtime error to attempt to use discarded data as an rvalue (source operand or incremented destination) in any subsequent operation. However, once the field is used as an lvalue (fully replaced destination), data integrity is automatically reinstated.

5 Function Details

This section describes in some detail the names and functionality for all functions in the interface involving linear algebra with and without shifts. Because of the variety of datatypes, and assignment operations, there are a few hundred names altogether. However, there are only a couple dozen categories. It is hoped that the construction of the names is sufficiently natural that with only a little practice, the user can guess the name of any function and determine its functionality without consulting a list.

In prototype specifications throughout this document the notation *Type* specifies the generic or specific datatype name matching the abbreviation *T* according to the table in Chapter 3 [Datatypes], page 5. We also introduce the shorthand

`#define` *`subset`* `QDP_Subset subset`

Unless otherwise indicated, operations occur on all sites in the specified subset.

5.1 Functions involving shifts

Shifting

Syntax	`QDP_`*`T`*`_eq_s`*`T`*`(`*`Type`*` *r, `*`Type`*` *a, QDP_Shift s, QDP_ShiftDir d, `*`subset`*`);` `QDP_`*`T`*`_veq_s`*`T`*`(`*`Type`*` *r[], `*`Type`*` *a[], QDP_Shift s[], QDP_ShiftDir d[], `*`subset`*`, int n);`
Meaning	r = shift(a)
Type	`I, R, C, V, H, D, M, P`

Left multiplication by shifted color matrix

Syntax	`QDP_`*`T`*`_eq_sM_times_`*`T`*`(`*`Type`*` *r, QDP_ColorMatrix *a, `*`Type`*` *b, QDP_Shift s, QDP_ShiftDir d, `*`subset`*`);`
Meaning	r = shift(a) * b
Type	`V, H, D, M, P`

Left multiplication of shifted field by color matrix

Syntax	`QDP_`*`T`*`_eq_M_times_s`*`T`*`(`*`Type`*` *r, QDP_ColorMatrix *a, `*`Type`*` *b, QDP_Shift s, QDP_ShiftDir d, `*`subset`*`);`
Meaning	r = a * shift(b)
Type	`V, H, D, M, P`

Left multiplication by color matrix then shift

Syntax	`QDP_`*`T`*`_eq_sM_times_s`*`T`*`(`*`Type`*` *r, QDP_ColorMatrix *a, `*`Type`*` *b, QDP_Shift s, QDP_ShiftDir d, `*`subset`*`);`
Meaning	r = shift(a * b)
Type	`V, H, D, M, P`

5.2 Fills and random numbers

Zero fills

Syntax	`QDP_`*T*`_eq_zero(`*Type* `*r, `*subset*`);`
Meaning	r = 0
Type	`I, R, C, V, H, D, M, P`

Constant fills

Syntax	`QDP_`*T*`_eq_`*t*`(`*Type* `*r, `*QLA_Type* `*a, `*subset*`);`
Meaning	r = a
Type	`I, R, C, V, H, D, M, P`

Fill color matrix with constant times identity

Syntax	`QDP_M_eq_c(QDP_ColorMatrix *r, QLA_Complex *a, `*subset*`);`
Meaning	r = a I

Seeding the random number generator field from an integer field

Syntax	`QDP_S_eq_seed_i_I(QDP_RandomState *r, QLA_Int c, QDP_Int *a, `*subset*`);`
Meaning	seed r from constant c and field a

Uniform random number fills

Syntax	`QDP_R_eq_random_S(QDP_Real *r, QDP_RandomState *a, `*subset*`);`
Meaning	r = uniform random number in (0,1) from seed a

Gaussian random number fills

Syntax	`QDP_`*T*`_eq_gaussian_S(`*Type* `*r, QDP_RandomState *a, `*subset*`);`
Meaning	r = normal gaussian from seed a
Type	`R, C, V, H, D, M, P`

Function fills

Syntax	`QDP_`*T*`_eq_func(`*Type* `*r, void (*func)(`*QLA_Type* `*dest, int coords[]), `*subset*`);`
Meaning	calls func(&r[x], x) for all coordinates x in subset
Type	`I, R, C, V, H, D, M, P`

5.3 Unary Operations

Bitwise not

Syntax	`QDP_I_eq_not_I(QDP_Int *r, QDP_Int *a,` *`subset`*`);`
Meaning	r = not(a)

Elementary unary functions on reals

Syntax	`QDP_R_eq_`*`func`*`_R(QDP_Real *r, QDP_Real *a,` *`subset`*`);`
Meaning	r = *func*(a)
func	`sin, cos, tan, asin, acos, atan, sqrt, fabs, exp, log, sign, ceil, floor, sinh, cosh, tanh, log10`

Elementary unary functions real to complex

Syntax	`QDP_C_eq_cexpi_R(QDP_Complex *r, QDP_Real *a,` *`subset`*`);`
Meaning	r = exp(ia)

Elementary unary functions complex to real

Syntax	`QDP_R_eq_`*`func`*`_C(QDP_Real *r, QDP_Complex *a,` *`subset`*`);`
Meaning	r = *func*(a)
func	`norm, arg`

Elementary unary functions on complex values

Syntax	`QDP_C_eq_`*`func`*`_C(QDP_Complex *r, QDP_Complex *a,` *`subset`*`);`
Meaning	r = *func*(a)
func	`cexp, csqrt, clog`

Copying

Syntax	`QDP_`*`T`*`_eq_`*`T`*`(`*`Type`*` *r,` *`Type`*` *a,` *`subset`*`);` `QDP_`*`T`*`_veq_`*`T`*`(`*`Type`*` *r[],` *`Type`*` *a[],` *`subset`*`, int n);`
Meaning	r = a
Type	`S, I, R, C, V, H, D, M, P`

Incrementing

Syntax	`QDP_`*T*`_`*eqop*`_`*T*`(`*Type* `*r, `*Type* `*a, `*subset*`);` `QDP_`*T*`_v`*eqop*`_`*T*`(`*Type* `*r[], `*Type* `*a[], `*subset*`, int n);`
Meaning	r *eqop* a
Type	`I, R, C, V, H, D, M, P`
eqop	`eqm, peq, meq`

Transpose

Syntax	`QDP_`*T*`_`*eqop*`_transpose_`*T*`(`*Type* `*r, `*Type* `*a, `*subset*`);`
Meaning	r *eqop* transpose(a)
Type	`M, P`
eqop	`eq, peq, meq, eqm`

Complex conjugate

Syntax	`QDP_`*T*`_`*eqop*`_conj_`*T*`(`*Type* `*r, `*Type* `*a, `*subset*`);`
Meaning	r *eqop* conjugate(a)
Type	`C, V, H, D, M, P`
eqop	`eq, peq, meq, eqm`

Hermitian conjugate

Syntax	`QDP_`*T*`_`*eqop*`_`*T*`a(`*Type* `*r, `*Type* `*a, `*subset*`);`
Meaning	r *eqop* adjoint(a)
Type	`C, M, P`
eqop	`eq, peq, meq, eqm`

Local squared norm: uniform precision

Syntax	`QDP_R_eq_norm2_`*T*`(QDP_Real *r, `*Type* `*a, `*subset*`);`
Meaning	r = norm2(a)
Type	`C, V, H, D, M, P`

5.4 Type conversion and component extraction and insertion

Convert float to double

Syntax	`QDP_`*T*`_eq_`*T*`(`*Type* `*r, `*Type* `*a, `*subset*`);`
Meaning	r = a
Type	`R, C, V, H, D, M, P`

Convert double to float

Syntax	`QDP_`*T*`_eq_`*T*`(`*Type* `*r,` *Type* `*a,` *subset*`);`
Meaning	r = a
Type	`R, C, V, H, D, M, P`

Convert real to complex (zero imaginary part)

Syntax	`QDP_C_eq_R(QDP_Complex *r, QDP_Real *a,` *subset*`);`
Meaning	r = a + i0

Convert real and imaginary to complex

Syntax	`QDP_C_eq_R_plus_i_R(QDP_Complex *r, QDP_Real *a, QDP_Real *b,` *subset*`);`
Meaning	r = a + i b

Real/Imaginary part of complex

Syntax	`QDP_R_eq_`*func*`_C(QDP_Real *r, QDP_Complex *a,` *subset*`);`
Meaning	r = *func*(a)
func	`re, im`

Integer to real

Syntax	`QDP_R_eq_I(QDP_Real *r, QDP_Int *a,` *subset*`);`
Meaning	r = a

Real to integer (truncate/round)

Syntax	`QDP_I_eq_`*func*`_R(QDP_Int *r, QDP_Real *a,` *subset*`);`
Meaning	r = *func*(a)
func	`trunc, round`

Accessing a color matrix element

Syntax	`QDP_C_eq_elem_M(QDP_Complex *r, QDP_ColorMatrix *a, int i, int j,` *subset*`);`
Meaning	r = a[i, j]

Inserting a color matrix element

Syntax	`QDP_M_eq_elem_C(QDP_ColorMatrix *r, QDP_Complex *a, int i, int j,` *subset*`);`
Meaning	r[i, j] = a

Accessing a half fermion or Dirac fermion spinor element

Syntax	`QDP_C_eq_elem_`*`T`*`(QDP_Complex *r, `*`Type`*` *a, int color, int spin, `*`subset`*`);`
Meaning	r = a[color, spin]
Type	`H, D`

Inserting a half fermion or Dirac fermion spinor element

Syntax	`QDP_`*`T`*`_eq_elem_C(`*`Type`*` *r, QDP_Complex *a, int color, int spin, `*`subset`*`);`
Meaning	r[color, spin] = a
Type	`H, D`

Accessing a color vector element

Syntax	`QDP_C_eq_elem_V(QDP_Complex *r, QDP_ColorVector *a, int i, `*`subset`*`);`
Meaning	r = a[i]

Inserting a color vector element

Syntax	`QDP_V_eq_elem_C(QDP_ColorVector *r, QDP_Complex *a, int i, `*`subset`*`);`
Meaning	r[i] = a

Accessing a Dirac propagator matrix element

Syntax	`QDP_C_eq_elem_P(QDP_Complex *r, QDP_DiracPropagator *a, int ic, int is, int jc, int js, `*`subset`*`);`
Meaning	r = a[ic, is, jc, js]

Inserting a Dirac propagator matrix element

Syntax	`QDP_P_eq_elem_C(QDP_DiracPropagator *r, QDP_Complex *a, int ic, int is, int jc, int js, `*`subset`*`);`
Meaning	r[ic, is, jc, js] = a

Extracting a color vector from a color matrix column

Syntax	`QDP_V_eq_colorvec_M(QDP_ColorVector *r, QDP_ColorMatrix *a, int j, `*`subset`*`);`
Meaning	r[i] = a[i, j] (for all i)

Inserting a color vector into a color matrix column

Syntax	`QDP_M_eq_colorvec_V(QDP_ColorMatrix *r, QDP_ColorVector *a, int j, subset);`
Meaning	r[i, j] = a[i] (for all i)

Extracting a color vector from a half fermion or Dirac fermion

Syntax	`QDP_V_eq_colorvec_T(QDP_ColorVector *r, Type *a, int spin, subset);`
Meaning	r[color] = a[color, spin] (for all color)
Type	`H, D`

Inserting a color vector into a half fermion or Dirac fermion

Syntax	`QDP_T_eq_colorvec_V(Type *r, QDP_ColorVector *a, int spin, subset);`
Meaning	r[color, spin] = a[color] (for all color)
Type	`H, D`

Extracting a Dirac vector from a Dirac propagator matrix column

Syntax	`QDP_D_eq_diracvec_P(QDP_DiracFermion *r, QDP_DiracPropagator *a, int jc, int js, subset);`
Meaning	r[ic, is] = a[ic, is, jc, js] (for all ic, is)

Inserting a Dirac vector into a Dirac propagator matrix column

Syntax	`QDP_P_eq_diracvec_D(QDP_DiracPropagator *r, QDP_DiracFermion *a, int jc, int js, subset);`
Meaning	r[ic, is, jc, js] = a[ic, is] (for all ic, is)

Trace of color matrix

Syntax	`QDP_C_eq_trace_M(QDP_Complex *r, QDP_ColorMatrix *a, subset);`
Meaning	r = trace(a)

Real/Imaginary part of trace of color matrix

Syntax	`QDP_R_eq_func_M(QDP_Real *r, QDP_ColorMatrix *a, subset);`
Meaning	r = *func*(a)
func	`re_trace, im_trace`

Traceless antihermitian part of color matrix

Syntax	`QDP_M_eq_antiherm_M(QDP_ColorMatrix *r, QDP_ColorMatrix *a, subset);`
Meaning	r = (a - a^\dagger)/2 - i Im Tr a/n_c

Spin trace of Dirac propagator

Syntax	`QDP_M_eq_spintrace_P(QDP_ColorMatrix *r, QDP_DiracPropagator *a,` *`subset`*`);`
Meaning	r[ic, jc] = Sum_is a[ic, is, jc, is]

Dirac spin projection

Syntax	`QDP_H_eq_spproj_D(QDP_HalfFermion *r, QDP_DiracFermion *a, int dir, int sign,` *`subset`*`);`
	`QDP_H_veq_spproj_D(QDP_HalfFermion *r[], QDP_DiracFermion *a[], int dir[], int sign[],` *`subset`*`, int n);`
Meaning	r = spin project(a, dir, sign)

Dirac spin reconstruction

Syntax	`QDP_D_`*`eqop`*`_sprecon_H(QDP_DiracFermion *r, QDP_HalfFermion *a, int dir, int sign,` *`subset`*`);`
	`QDP_D_v`*`eqop`*`_sprecon_H(QDP_DiracFermion *r[], QDP_HalfFermion *a[], int dir[], int sign[],` *`subset`*`, int n);`
Meaning	r = spin reconstruct(a, dir, sign)
eqop	`eq`, `peq`, `meq`, `eqm`

Matrix multiply and Dirac spin reconstruction

Syntax	`QDP_D_`*`eqop`*`_M_times_H(QDP_DiracFermion *r, QDP_ColorMatrix *a, QDP_HalfFermion *b, int dir, int sign,` *`subset`*`);`
	`QDP_D_v`*`eqop`*`_M_times_H(QDP_DiracFermion *r[], QDP_ColorMatrix *a[], QDP_HalfFermion *b[], int dir[], int sign[],` *`subset`*`, int n);`
Meaning	r = spin reconstruct(a*b, dir, sign)
eqop	`eq_sprecon`, `peq_sprecon`, `meq_sprecon`, `eqm_sprecon`

Matrix multiply and Wilson spin multiplication

Syntax	`QDP_D_`*`eqop`*`_M_times_D(QDP_DiracFermion *r, QDP_ColorMatrix *a, QDP_DiracFermion *b, int dir, int sign,` *`subset`*`);`
	`QDP_D_v`*`eqop`*`_M_times_D(QDP_DiracFermion *r[], QDP_ColorMatrix *a[], QDP_DiracFermion *b[], int dir[], int sign[],` *`subset`*`, int n);`
Meaning	r = Wilson spin(a*b, dir, sign)
eqop	`eq_wilsonspin`, `peq_wilsonspin`, `meq_wilsonspin`, `eqm_wilsonspin`

5.5 Binary operations with constants

Multiplication by integer constant

Syntax	`QDP_I_eqop_i_times_I(QDP_Int *r, QLA_Int *a, QDP_Int *b, subset);`
Meaning	r *eqop* a * b
eqop	`eq, peq, meq, eqm`

Multiplication by real constant

Syntax	`QDP_T_eqop_r_times_T(Type *r, QLA_Real *a, Type *b, subset);` `QDP_T_veqop_r_times_T(Type *r[], QLA_Real *a[], Type *b[], subset, int n);`
Meaning	r *eqop* a * b
Type	`R, C, V, H, D, M, P`
eqop	`eq, peq, meq, eqm`

Multiplication by complex constant

Syntax	`QDP_T_eqop_c_times_T(Type *r, QLA_Complex *a, Type *b, subset);` `QDP_T_veqop_c_times_T(Type *r[], QLA_Complex *a[], Type *b[], subset, int n);`
Meaning	r *eqop* a * b
Type	`C, V, H, D, M, P`
eqop	`eq, peq, meq, eqm`

Multiplication by i

Syntax	`QDP_T_eqop_i_T(Type *r, Type *a, subset);`
Meaning	r *eqop* i a
Type	`C, V, H, D, M, P`
eqop	`eq, peq, meq, eqm`

Left multiplication by gamma matrix

Syntax	`QDP_T_eq_gamma_times_T(Type *r, Type *a, int i, subset);`
Meaning	r = gamma(i) * a
Type	`D, P`

Right multiplication by gamma matrix

Syntax	`QDP_P_eq_P_times_gamma(QDP_DiracPropagator *r, QDP_DiracPropagator *a, int i, subset);`
Meaning	r = a * gamma(i)

5.6 Binary operations with fields

Elementary binary functions on integers

Syntax	`QDP_I_eq_I_`*func*`_I(QDP_Int *r, QDP_Int *a, QDP_Int *b,` *subset*`);`
Meaning	r = a *func* b
func	`lshift, rshift, mod, max, min, or, and, xor`

Elementary binary functions on reals

Syntax	`QDP_R_eq_R_`*func*`_R(QDP_Real *r, QDP_Real *a, QDP_Real *b,` *subset*`);`
Meaning	r = a *func* b
func	`mod, max, min, pow, atan2`

Multiplying real by integer power of 2

Syntax	`QDP_R_eq_R_ldexp_I(QDP_Real *r, QDP_Real *a, QDP_Int *b,` *subset*`);`
Meaning	r = a * 2^b

Addition

Syntax	`QDP_`*T*`_eq_`*T*`_plus_`*T*`(`*Type* `*r,` *Type* `*a,` *Type* `*b,` *subset*`);`
Meaning	r = a + b
Type	`I, R, C, V, H, D, M, P`

Subtraction

Syntax	`QDP_`*T*`_eq_`*T*`_minus_`*T*`(`*Type* `*r,` *Type* `*a,` *Type* `*b,` *subset*`);`
Meaning	r = a - b
Type	`I, R, C, V, H, D, M, P`

Multiplication: uniform types

Syntax	`QDP_`*T*`_`*eqop*`_`*T*`_times_`*T*`(`*Type* `*r,` *Type* `*a,` *Type* `*b,` *subset*`);`
Meaning	r *eqop* a * b
Type	`I, R, C, P`
eqop	`eq, peq, meq, eqm`

Division of integer, real, and complex fields

Syntax	`QDP_`*T*`_eq_`*T*`_divide_`*T*`(`*Type* `*r,` *Type* `*a,` *Type* `*b,` *subset*`);`
Meaning	r = a / b
Type	`I, R, C`

Left multiplication by color matrix

Syntax	QDP_*T*_*eqop*_M_times_*T*(*Type* *r, QDP_ColorMatrix *a, *Type* *b, *subset*); QDP_*T*_v*eqop*_M_times_*T*(*Type* *r[], QDP_ColorMatrix *a[], *Type* *b[], *subset*, int n);
Meaning	r *eqop* a * b
Type	V, H, D, M, P
eqop	eq, peq, meq, eqm

Left multiplication by adjoint of color matrix

Syntax	QDP_*T*_*eqop*_Ma_times_*T*(*Type* *r, QDP_ColorMatrix *a, *Type* *b, *subset*); QDP_*T*_v*eqop*_Ma_times_*T*(*Type* *r[], QDP_ColorMatrix *a[], *Type* *b[], *subset*, int n);
Meaning	r *eqop* adjoint(a) * b
Type	V, H, D, M, P
eqop	eq, peq, meq, eqm

Right multiplication by color matrix

Syntax	QDP_P_*eqop*_P_times_M(QDP_DiracPropagator *r, QDP_DiracPropagator *a, QDP_ColorMatrix *b, *subset*);
Meaning	r *eqop* a * b
eqop	eq, peq, meq, eqm

Right multiplication by adjoint of color matrix

Syntax	QDP_*T*_*eqop*_*T*_times_Ma(*Type* *r, *Type* *a, QDP_ColorMatrix *b, *subset*);
Meaning	r *eqop* a * adjoint(b)
Type	M, P
eqop	eq, peq, meq, eqm

Adjoint of color matrix times adjoint of color matrix

Syntax	QDP_M_*eqop*_Ma_times_Ma(QDP_ColorMatrix *r, QDP_ColorMatrix *a, QDP_ColorMatrix *b, *subset*);
Meaning	r *eqop* adjoint(a) * adjoint(b)
eqop	eq, peq, meq, eqm

Local inner product

Syntax	QDP_C_eq_*T*_dot_*T*(QDP_Complex *r, *Type* *a, *Type* *b, *subset*);
Meaning	r = Tr adjoint(a) * b
Type	C, V, H, D, M, P

Real part of local inner product

Syntax	QDP_R_eq_re_*T*_dot_*T*(QDP_Real *r, *Type* *a, *Type* *b, *subset*);
Meaning	r = Re Tr adjoint(a) * b
Type	C, V, H, D, M, P

Color matrix from outer product

Syntax	QDP_M_*eqop*_V_times_Va(QDP_ColorMatrix *r, QDP_ColorVector *a, QDP_ColorVector *b, *subset*);
	QDP_M_v*eqop*_V_times_Va(QDP_ColorMatrix *r[], QDP_ColorVector *a[], QDP_ColorVector *b[], *subset*, int n);
Meaning	r[i, j] *eqop* a[i] * b[j]
eqop	eq, peq, meq, eqm

5.7 Ternary operations with fields

Addition or subtraction with integer scalar multiplication

Syntax	QDP_I_eq_i_times_I_*func*_I(QDP_Int *r, QLA_Int *c, QDP_Int *a, QDP_Int *b, *subset*);
Meaning	r = c * a +/- b
func	plus, minus

Addition or subtraction with real scalar multiplication

Syntax	QDP_*T*_eq_r_times_*T*_*func*_*T*(*Type* *r, QLA_Real *c, *Type* *a, *Type* *b, *subset*);
Meaning	r = c * a +/- b
Type	R, C, V, H, D, M, P
func	plus, minus

Addition or subtraction with complex scalar multiplication

Syntax	`QDP_`*T*`_eq_c_times_`*T*`_`*func*`_`*T*`(`*Type* `*r, QLA_Complex *c,` *Type* `*a,` *Type* `*b,` *subset*`);`
Meaning	r = c * a +/- b
Type	`C, V, H, D, M, P`
func	`plus, minus`

5.8 Boolean operations

Comparisons of integers and reals

Syntax	`QDP_I_eq_`*T*`_`*func*`_`*T*`(QDP_Int *r,` *Type* `*a,` *Type* `*b,` *subset*`);`
Meaning	r = a *func* b
Type	`I, R`
func	`eq, ne, gt, lt, ge, le`

Copy under a mask

Syntax	`QDP_`*T*`_eq_`*T*`_mask_I(`*Type* `*r,` *Type* `*a, QDP_Int *b,` *subset*`);`
Meaning	r = a (if b is not 0)
Type	`I, R, C, V, H, D, M, P`

5.9 Reductions

Global squared norm: uniform precision

Syntax	`QDP_r_eq_norm2_`*T*`(QLA_Real *r,` *Type* `*a,` *subset*`);` `QDP_r_veq_norm2_`*T*`(QLA_Real r[],` *Type* `*a[],` *subset*`, int n);`
Meaning	r = Sum norm2(a)
Type	`I, R, C, V, H, D, M, P`

Global inner product

Syntax	`QDP_r_eq_`*T*`_dot_`*T*`(QLA_Real *r,` *Type* `*a,` *Type* `*b,` *subset*`);` `QDP_r_veq_`*T*`_dot_`*T*`(QLA_Real r[],` *Type* `*a[],` *Type* `*b[],` *subset*`, int n);`
Meaning	r = Sum Tr a * b
Type	`I, R`

Syntax	`QDP_c_eq_T_dot_T(QLA_Complex *r, Type *a, Type *b, subset);` `QDP_c_veq_T_dot_T(QLA_Complex r[], Type *a[], Type *b[], subset, int n);`
Meaning	r = Sum Tr adjoint(a) * b
Type	`C, V, H, D, M, P`

Real part of global inner product

Syntax	`QDP_r_eq_re_T_dot_T(QLA_Real *r, Type *a, Type *b, subset);` `QDP_r_veq_re_T_dot_T(QLA_Real r[], Type *a[], Type *b[], subset, int n);`
Meaning	r = Sum Re Tr adjoint(a) * b
Type	`C, V, H, D, M, P`

Global sums

Syntax	`QDP_r_eq_sum_I(QLA_Real *r, QDP_Int *a, subset);`
Meaning	r = Sum a
Syntax	`QDP_t_eq_sum_T(QLA_Type *r, Type *a, subset);`
Meaning	r = Sum a
Type	`R, C, V, H, D, M, P`

Multisubset Norms

Syntax	`QDP_r_eq_norm2_T_multi(QLA_Real r[], Type *a, QDP_Subset subset[], int n);` `QDP_r_veq_norm2_T_multi(QLA_Real r[], Type *a[], QDP_Subset subset[], int n);`
Meaning	r[i] = Sum_subset[i] norm2(a)
Type	`I, R, C, V, H, D, M, P`

Multisubset inner products

Syntax	`QDP_r_eq_T_dot_T_multi(QLA_Real r[], Type *a, Type *b, QDP_Subset subset[], int n);` `QDP_r_veq_T_dot_T_multi(QLA_Real r[], Type *a[], Type *b[], QDP_Subset subset[], int n);`
Meaning	r[i] = Sum_subset[i] a * b
Type	`I, R`
Syntax	`QDP_c_eq_T_dot_T_multi(QLA_Complex r[], Type *a, Type *b, QDP_Subset subset[], int n);` `QDP_c_veq_T_dot_T_multi(QLA_Complex r[], Type *a[], Type *b[], QDP_Subset subset[], int n);`
Meaning	r[i] = Sum_subset[i] adjoint(a) * b
Type	`C, V, H, D, M, P`

Multisubset real part of global inner product

Syntax	`QDP_r_eq_re_T_dot_T_multi(QLA_Real r[], Type *a, Type *b, QDP_Subset subset[], int n);`
	`QDP_r_veq_re_T_dot_T_multi(QLA_Real r[], Type *a[], Type *b[], QDP_Subset subset[], int n);`
Meaning	r = Sum Re Tr adjoint(a) * b
Type	C, V, H, D, M, P

Multisubset global sums

Syntax	`QDP_r_eq_sum_I_multi(QLA_Real r[], QDP_Int *a, QDP_Subset subset[], int n);`
Meaning	r[i] = Sum_subset[i] a
Syntax	`QDP_t_eq_sum_T_multi(QLA_Type r[], Type *a, QDP_Subset subset[], int n);`
Meaning	r[i] = Sum_subset[i] a
Type	R, C, V, H, D, M, P