Input/Output for QCD

Version 2.3

SciDAC Software Coordinating Committee

May 25, 2008

1   Introduction

This document describes the Input/Output Applications Programmer Interface developed under the auspices of the U.S. Department of Energy Scientific Discovery through Advanced Computing (SciDAC) program.

Although the QIO I/O system was developed to support the data parallel lattice-QCD API called QDP/C and QDP++, it is designed to function independently of QDP, requiring only the lower level QMP message passing package. Three data models are treated: full volume lattice fields, consisting of data of the same format residing on each site of a hypercubic lattice, subvolume lattice fields, consisting of data of the same format residing on a hypercubic subset of a hypercubic lattice (e.g. a 3D time slice), and global data, constant across all lattice sites. Lattice field data is distributed among multiple nodes.

The file format consists of a series of logical records. Each record contains user-controlled metadata and binary data. An arbitrary combination of logical records is permitted. The physical file format is based on a custom SciDAC LIME standard, (Lattice QCD Interchange Message Encapsulation), which views the file as a series of LIME messages, each, in turn, consisting of a series of LIME records. Details of the physical format are hidden from the user. The LIME package is included with QIO.

2   Overview of Binary File Format

2.1   Introduction

The binary file format has been designed with flexibility in mind. For archiving purposes, the allowable logical records, metadata, and binary content may be further restricted. Here we describe the unrestricted format. The archivable International Lattice Data Group format is described at the end of this section.

Three classes of file volumes are supported: single-file volumes, partition-file volumes and multiple-file volumes. Single files are read and written through a single master node; partition files are read and written through a set of designated I/O nodes; and multiple files, through each node. With partition-file

and multiple-file formats the binary data is split into separate files, one for each node that participates in I/O. With the single-file format, data is contained in a single file. Single-processor utilities are provided for converting between the partition-file and single-file formats.

For singlefile format, two modes of file reading and writing are supported, namely serial and parallel. With serial reading or writing, all data is funneled through the master I/O node, which is the only node opening the file. With parallel reading or writing, all nodes open the same file and participate in reading or writing their local data. Whether written serially or in parallel, the resulting file is exactly the same. Thus a file can be written serially and read in parallel or vice versa, if so desired. Currently parallel reading/writing from/to partition-file formats is not supported.

2.1.1 Single file format

Single binary files are composed of a series of one or more logical application records. A single logical record encodes a single lattice field, an array of lattice fields of the same data type, or an array of global data. Physics metadata, managed at the convenience of the applications programmer can be inserted in the file header and separately with each logical record header. The applications programmer views the file as follows:

- File physics metadata

- Record 1 physics metadata and binary data

- Record 2 physics metadata and binary data

- etc.

For example, a file might record a series of staggered fermion eigenvectors for a gauge field configuration. Each record would then map to a single field for a color vector. The file metadata might include information about the parent 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 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 logical record containing the array of four color matrices.

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 private QIO metadata and physics metadata and each logical application record consists of four LIME records. Within QIO the file is viewed as a series of LIME 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. The byte layout of the site data is determined by user-supplied "factory" functions. Physical byte ordering of IEEE numeric data (integers and floating point) is big-endian, regardless of the architecture of the processor that creates the file. To achieve this result, numeric site data within a given logical record must consist of a series of words of the same size, such as an $SU(3)$ matrix represented entirely by single precision words (or entirely double precision). Mixed precision structures are excluded, but 32-bit integers and floats may coexist.

2.2   Partition file format

With cluster computers consisting of many hundreds of processors it may prove impractical to provide NFS mounts from each processor to a common file system. Instead processors can be grouped into I/O sets, each with a single I/O node and disk. Input files are fragmented and staged to disks attached to these nodes and output files are reassembled from them. Single-processor utilities are provided for file disassembly and assembly. It is intended that the installation and implementation hide these details from the user, so the user's view of the file is the same as with single file format. Ideally the local software environment is designed with portability in mind, so that user code calling for file input or output will get the same result in the end without any change in the code, whether or not the intermediate volume format happens to be single file at one installation and partitioned at another.

Each partition I/O node processes data only for sites stored on that partition. The data for the binary field is divided accordingly, so each partition file holds field data only for the nodes on its partition. For ease in conversion to and from single file format, the site data for a file is always arranged in a standard "lexicographic" order according to the lattice coordinate. The lexicographic

rank identifies the site. A binary site list is placed at the beginning of every file to identify its contents. The master I/O node, which must also be a partition I/O node, handles all of the global data in the file, including file and record metadata and any global binary data. Its file format is identical to the single file format, except for the addition of a sitelist record. The other partition files contain only site lists and the binary data for the relevant partition.

All component files are given unique names, constructed by attaching the file extension `.vol`*nnnn*, where *nnnn* is the number of the node that reads the file (with leading zeros). The file is known to the user by its unextended name.

The principal file read by the master node contains most of the metadata.

- Private file QIO metadata

- User file physics metadata

- Binary index of sites

- 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 secondary files contain these records:

- Binary index of sites

- Record 1 binary data

- Record 2 binary data

- etc.

The site index in each case is a table of contents, that is, a list of the lexicographic ranks of all sites contained in the file in the order of appearance.

2.3 Multifile format

The API provides for rapid temporary writing of data to scratch disks and reading from scratch disks. In this case it is assumed that the files are not intended for longer term storage. The file formats are identical to the partition file formats with one exception: the site order is internal storage order, rather than lexicographic order. This choice is made to reduce cache-misses during I/O.

2.4 ILDG format

The recently adopted ILDG standard for SU(3) gauge configuration files is closely compatible with the standard SciDAC file format. The flexibility of the standard permits the creation of files that meet both SciDAC and ILDG requirements. Provision is made within QIO to produce and read files that meet the standard, even if they were not generated by QIO. ILDG version 1.0 compatible files created by QIO have only one lattice field and contain the following LIME records.

- SciDAC Private file QIO metadata

- SciDAC User file physics metadata

- SciDAC Binary index of sites

- SciDAC private record QIO metadata

- SciDAC user physics metadata

- ILDG format record (see standard)

- ILDG LFN record (see standard)

- ILDG binary data record containing the gauge field (see standard)

- SciDAC private checksum

The content of the ILDG binary data record is identical to the SciDAC binary data record for a field with four SU(3) color matrices per site.

3 Metadata Standard and Manipulation

The QIO implementation uses an XML encoding for its private file and record metadata. It is hidden above the QIO API. The data is available to the user through a C structure with accessor functions for retrieving and setting values.

Since QIO processes the user file and record metadata blindly as a character string, QIO places no restrictions on the format of the user metadata.

4 QIO API

This section describes the QIO interface.

The QIO system provides for binary file operation for writing and reading lattice fields and global data. Lattice fields consist of any data type homogeneous over lattice sites or an array of such data types. Global data consists of an array of data types or of strings. The storage of lattice data on the nodes is described in a `QIO_Layout` structure, and the information required for presenting field data in the correct byte order is encapsulated in "factory" functions.

4.1 The layout structure

The structure is defined as follows:

```
typedef struct {
  /* Data distribution */
  int (*node_number)(const int coords[]);
  int (*node_index)(const int coords[]);
  void (*get_coords)(int coords[], int node, int index);
  size_t (*num_sites)(int node);
  int *latsize;
  int latdim;
  size_t volume;
  size_t sites_on_node;
  int this_node;
  int number_of_nodes;
} QIO_Layout;
```

The data distribution (layout) structure has nine members. The `node_number` member is an implementer-supplied function returning the number of the node that has the specified lattice coordinate. The `node_index` member returns the storage order index for the site on its node. The `get_coords` member maps the node number and index values to lattice coordinates. The `num_sites` member returns the number of sites on the specified node. The next two members specify the lattice coordinate extent and spacetime dimensionality. The seventh member specifies the full spacetime volume. The eighth, the number of sites on the current node, the ninth, the number of the present node, and the ninth, the total number of nodes.

Here is an illustration of how the layout structure is loaded from the data in our implementation of the QDP/C API prior to a `QIO_open_read` or `QIO_open_write` call:

```
QIO_Layout layout;

layout.node_number = QDP_node_number;
layout.node_index  = QDP_index;
layout.get_coords = QDP_get_coords;
```

```
layout.num_sites = QDP_num_sites;
layout.latdim = QDP_ndim();
layout.latsize = (int *)malloc(layout->latdim*sizeof(int));
QDP_latsize(layout.latsize);
layout.volume = QDP_volume();
layout.sites_on_node = QDP_sites_on_node;
layout.this_node = QDP_this_node;
layout.number_of_nodes = QDP_numnodes();
```

4.2   Private Record Metadata

Field data is described by a private QIO record metadata structure. On output the application must create and populate the structure. On input, the structure is populated from the file.

The private QIO record metadata is used for consistency checking and for providing the user a standard tool for recording and discovering the data type being stored. Semantically, it serves the same purpose as a BinX record. It carries enough information to completely define the binary record format. The record metadata is held in an opaque `QIO_RecordInfo` structure. Elements are accessed and manipulated through the following functions.

Create and populate the private record metadata structure   Before writing a record the calling program must create the private record metadata structure. Before reading a record, the calling program must allocate space for the private record metadata structure using the same calling procedure.

Prototype	`QIO_RecordInfo *QIO_create_record_info(int globaltype,`
	`int lower[], int upper[], int n,`
	`char *datatype, char *precision, int colors,`
	` int spins, int typesize,`
	` int datacount);`

Example	`rec_info = QIO_create_record_info(QIO_FIELD,"QDP_F_Real","F",` `0,0,0,0,0,size,1);`
Example	`rec_info = QIO_create_record_info(0, "", "", 0,0,0, 0, 0, 0, 0);`

The first example is appropriate for output. The second, for input.

The `globaltype` parameter distinguishes between a record containing a lattice field and a record containing a lattice constant array.

   `QIO_FIELD, QIO_HYPER, QIO_GLOBAL`

for field (full volume), hypercube (subvolume), and global (constant) record types, respectively.

For a hypercube record, the `lower` and `upper` parameters are integer arrays specifying the coordinate lower bounds and coordinate upper bounds of the hypercube. For example, for the contents of time slice 4 on a lattice of dimension $32^3 \times 48$, use

```
int lower[4] = {0, 0, 0, 4}
int upper[4] = {31, 31, 31, 4}
```

The parameter n gives the number of spacetime dimensions of the full lattice volume (in this example, 4).

The datatype string is not interpreted by QIO. It allows the applications programmer a standard way to identify the data type. For that purpose the name should be unique. For QDP/C we use the datatype name of the QDP field. For USQCD standard formats, there are special names. For global data we use the name of one of the QLA datatypes.

The precision string is one of these:

F	single
D	double
S	random number generator state consisting of 32-bit floats and ints
I	integer (currently only 32-bit is supported)

This string is interpreted by the host file conversion utility.

The colors and spins arguments give the working value for these quantities, if they apply to the datatype. Otherwise, they should be zero. They are not interpreted by QIO.

The typesize specifies the number of bytes per site item and the datacount specifies the number of such items per site. The product is the total number of bytes per site. For example, for a single precision SU(3) gauge field with four color matrices per site, the typesize is 72 and the datacount is 4.

It is not an error to create a structure with zeros for integer values and null string pointers. Those data items are tagged as "missing". However, QIO_write and QIO_read return an error condition, if the total byte count per site is inconsistent with the values in this structure.

Destroy the private record metadata structure

Prototype	`void QIO_destroy_record_info(QIO_RecordInfo *record_info);`
Example	`QIO_destroy_record_info(rec_info);`

Compare two private record metadata structures   To allow for verification that a record being read matches what is expected, the calling program may create the record information structure that it expects and compare it with the structure that was read from the file.

Prototype	`int QIO_compare_record_info(QIO_RecordInfo *found,` `QIO_RecordInfo *expect);`
Example	`int ok = QIO_compare_record_info(rec_info, cmp_info);`

The arguments are *not* symmetric. Only those fields that are non-empty in the expect structure are compared with fields in the found structure.

Extract values from the file reader structure The following accessors perform self-evident functions:

Prototype	`int QIO_get_reader_latdim(QIO_Reader *in);`
	`int *QIO_get_reader_latsize(QIO_Reader *in);`
	`uint32_t QIO_get_reader_last_checksuma(QIO_Reader *in);`
	`uint32_t QIO_get_reader_last_checksumb(QIO_Reader *in);`

Extract values from the file writer structure The following accessors perform self-evident functions:

Prototype	`uint32_t QIO_get_writer_last_checksuma(QIO_Writer *out);`
	`uint32_t QIO_get_writer_last_checksumb(QIO_Writer *out);`

Extract values from the private record metadata structure The following accessors perform self-evident functions:

Prototype	`int QIO_get_recordtype(QIO_RecordInfo *record_info);`
	`int *QIO_get_hyperlower(QIO_RecordInfo *record_info);`
	`int *QIO_get_hyperupper(QIO_RecordInfo *record_info);`
	`char *QIO_get_datatype(QIO_RecordInfo *record_info);`
	`char *QIO_get_precision(QIO_RecordInfo *record_info);`
	`int QIO_get_colors(QIO_RecordInfo *record_info);`
	`int QIO_get_spins(QIO_RecordInfo *record_info);`
	`int QIO_get_typesize(QIO_RecordInfo *record_info);`
	`int QIO_get_datacount(QIO_RecordInfo *record_info);`
	`char *QIO_get_record_date(QIO_RecordInfo *record_info);`

4.3 Opening and closing binary files

The file opening procedures differ, depending on whether the file is opened for reading or writing.

Open a file for writing

Prototype	`QIO_Writer *QIO_open_write(QIO_String *xml_file,`
	`  char *filename, int volfmt, QIO_Layout *layout,`
	`  QIO_Filesystem *fs, QIO_Oflag *oflag);`
Purpose	Opens a named file for writing and writes the file metadata.
Example	`QIO_Writer *outfile;`
	`QIO_Layout layout;`
	`outfile = QIO_open_write(xml_file_out, filename,`
	`  QIO_SINGLEFILE, &layout, &fs,`
	`  &oflag);`

The `QIO_Writer *` return value points to the file handle used in subsequent references to the file. The first argument is the user file XML. To create the `QIO_String` structure, starting from a plain character array, use the command

Prototype	`void QIO_string_set(QIO_String *qs, const char *const string)`
Example	`QIO_String *xml_file = QIO_string_create();`
	`QIO_string_set(xml_file, xmlstring);`

The third-to-last argument is the `layout` structure. It is assumed that the user has prepared it as described above.

The next-to-last argument specifies the I/O-nodes in use. Here are the members that require definition:

```
typedef struct {
  int (*my_io_node)(const int node);   /* Which node does I/O for a node */
  int (*master_io_node)(void);         /* Which node is the master */
} QIO_Filesystem;
```

For example

```
    QIO_Filesystem fs;
    fs.my_io_node = io_node;
    fs.master_io_node = master_io_node;
```

where the `io_node` function `io_node(node)` returns the number of the node that does I/O for node `node` and the `master_io_node` function returns the number of the master I/O node. If the `fs` parameter is NULL (zero) in the call to `QIO_open_write`, QIO assumes each node is its own I/O node, and the master node is node 0. The same defaults apply to the separate members if the structure pointer is non-null, but a member is a null function pointer.

The `QIO_Oflag` structure is defined as follows:

```
typedef struct {
  int serpar;             /* QIO_SERIAL or QIO_PARALLEL */
  int mode;               /* QIO_TRUNC or QIO_APPEND */
  int ildgstyle;          /* QIO_ILDGNO or QIO_ILDGLAT */
  QIO_String *ildgLFN;    /* NULL if unknown */
} QIO_Oflag;
```

The `serpar` member specifies whether the component file(s) is(are) to be written in parallel (many nodes writing to the same component file) or serially (only one writer for each component file). The `mode` member specifies whether the file is to be truncated or data is to be appended. The `ildgstyle` member specifies whether the file (currently only a lattice file) is to be written with additional LIME records for ILDG compatibility. If so, a pointer to the ILDG logical file name (LFN) must be supplied through the `ildgLFN` member.

The structure is initialized as in the following example:

```
    QIO_Oflag oflag;
    oflag.serpar   = QIO_SERIAL;
    oflag.mode     = QIO_TRUNC;
    oflag.ildgstye = QIO_ILDGLAT;
    oflag.ildgLFN  = QIO_string_create();
    QIO_string_set(oflag.ildgLFN,"MILC.ks_imp_3flav.4096f21b708m0031m031b.696");
```

(Please note, this illustrative LFN is not valid.)

When the &oflag parameter is passed as a null pointer, the default values are serial mode, truncate, non-ILDG, and null LFN. If the LFN pointer is null, the ILDG LFN record is not written. It must then be appended later to produce a file that is fully ILDG compatible.

Parallel I/O is supported only in singlefile format. If parallel mode is requested for other formats, the request is currently ignored. Of course, in a sense multifile and partfile formats are parallel formats, but the component files are opened by only one node. So we say each component file is accessed serially. It is conceivable in future versions of QIO that one could have a subset of nodes on a partition open the same partition file. We would call that parallel I/O of a partition file.

Caution: If a file is opened for appending, QIO presently does not verify that the fields being appended conform to the lattice dimensions and layout of the fields already present.

Open a file for reading

Prototype	`QIO_Reader *QIO_open_read(QIO_String *xml_file,`
	`    char *filename, QIO_Layout *layout,`
	`    QIO_Filesystem *fs, QIO_Iflag *iflag);`
Purpose	Opens a named file for reading and reads the file metadata.
Example	`QIO_Reader *infile;`
	`QIO_Layout layout;`
	`infile = QIO_open_read(xml_file_in, filename,`
	`    &layout, QIO_SERIAL);`

The `QDP_Reader` return value is the file handle used in subsequent references to the file. A null return value signals an error. It is assumed the user has created the file metadata structure with address `xml_file`, so it can be read from the head of the file and inserted. Space for the string within the structure is reallocated to a sufficient size by QIO. The other arguments have the same meaning as with `QIO_open_write`. The volume format is auto-detected so is not specified by the calling program. It is assumed that the user has prepared the `layout` argument as described above.

The `QIO_Iflag` structure is defined as follows:

```
typedef struct {
  int serpar;    /* QIO_SERIAL or QIO_PARALLEL */
  int volfmt;    /* QIO_UNKNOWN, QIO_SINGLEFILE, QIO_PARTFILE,
```

```
                        QIO_MULTIFILE */
} QIO_Iflag;
```

A file is usually opened with automatic detection of the file format. However, confusion arises when the file appears in both formats in the same directory. In that case the `volfmt` member is needed to specify a preference. Otherwise, the parameter can be safely passed as `QIO_UNKNOWN` or `QIO_SINGLEFILE`, regardless of the file format, and the format will be set according to the existing file. The structure also has a placeholder for future use for specifying whether the file is to be read in parallel or serially. The structure is initialized as in the following example:

```
    QIO_Iflag iflag;
    iflag.serpar = QIO_SERIAL;
    iflag.mode   = QIO_UNKNOWN;
```

These are the default values used when the `&iflag` parameter is passed as a null pointer.

In normal operation the user specifies the lattice dimension in the `QIO_Layout` structure, and an error condition occurs, if the dimensions in the file do not match the dimensions in the layout structure. Provision is made to operate in discovery mode. If the layout `latdim` member is zero when `QIO_open_read` is called, no checking takes place and the lattice dimensions are taken from the file and kept with the `QIO_Reader` structure. The user's `QIO_layout` structure is not altered by QIO. Instead, it works with an updated internal copy of that structure, kept in the opaque `QIO_Reader`. Two accessor functions are provided for extracting the dimensions from the reader:

Get the number of spacetime dimensions

Prototype	`int QIO_get_reader_latdim(QIO_Reader *in);`
Purpose	Returns the number of spacetime dimensions.
Example	`int latdim = QIO_get_reader_latdim(qio_in);`

Get the lattice size in each direction

Prototype	`int *QIO_get_reader_latsize(QIO_Reader *in);`
Purpose	Returns a pointer to an integer array of sizes for each dimension.
Example	`int *latsize = QIO_get_reader_latsize(qio_in);`

Allocation of the array is controlled by QIO. The array storage is released by the `QIO_close_read` call.

Close an output file

Prototype	`int QIO_close_write(QIO_Writer *out);`
Example	`QIO_close_write(outfile);`

Close an input file

Prototype	`int QIO_close_read(QIO_Reader *in);`
Example	`QIO_close_read(infile);`

In both cases the integer return value is 0 for success and 1 for failure.

4.4 Writing and reading fields, arrays of fields, or arrays of global data

Prototype	`int QIO_write(QIO_Writer *out,`
	`  QIO_RecordInfo *record_info,QIO_String *xml_record,`
	`  void (*get)(char *buf, size_t index, size_t count, void *arg),`
	`  int datum_size, int word_size, void *arg);`

Example	`QIO_RecordInfo *rec_info;`
	`rec_info = QIO_create_record_info(QDP_FIELD,"QDP_F_Real","F",`
	`0,0,0,0,QLA_Ns,size,1);`
	`QIO_write(outfile, rec_info, xml_record, QDP_F_get_R,`
	`  sizeof(QLA_Real), sizeof(QLA_Real), (void *)field);`

The integer return value is 0 for success and 1 for failure. It is assumed the user has prepared the record metadata and the field data in advance.

The input arguments are as follows:

`out`	The `QIO_Writer` handle returned by `QIO_open_write`.
`record_info`	The private metadata for the record (see below).
`xml_record`	The user-constructed metadata for the record.
`get`	Factory function (see below).
`datum_size`	The total number of bytes required to serialize the datum.
`word_size`	The number of bytes in a datum word.
`arg`	Pass-through parameters for the factory function.

The second argument, the `record_info` structure, contains information about the data format, as described in Sec. 4.2. It must be created by the caller in all cases. For output, the caller must set its values. For input, the values are returned from the file.

The fourth argument is a factory function that, in this example, is invoked by QIO like this:

```
QDP_F_get_R(buf, index, count, field);
```

It is expected to fill the QIO-supplied buffer `buf` with a byte-serialized copy of the field datum at site index `index`. The parameter `count` specifies the array length of the field datum at that site. The datum size parameter `datum_size` gives the total number of bytes to be delivered as the product of the count parameter and the byte length of the array element on that site.

It is up to the applications programmer to insure that the data base-type (int, float, double) word order produced by the factory function follows the

SciDAC convention for the specified datatype. However byte ordering within a word (big endian or little endian) processed by the factory functions should be in the native order of the architecture. Any byte rearrangement needed to convert to and from standard file endianness is the responsibility of QIO. To this end the user must specify the base-type word length of the data in bytes through the parameter `word_size`. All numeric SciDAC data types are homogeneous in word size, so a single parameter suffices.

For example for an array of four single precision color vector fields, each consisting of three complex numbers, there are $4 \times 3 \times 2 = 24$ real values per site, each of them single-precision floating point numbers. The word size for the IEEE float datatype is 4 (bytes). The factory function must produce the standard word order: real part of the first color component of the first color vector, followed by the imaginary part of the same component, followed by the real and then imaginary parts of the second color component of the first color vector, etc. The count is 4 (color vectors), and the datum size is $4 \times 24 = 96$ (total bytes per call). [The type size of $3 \times 2 \times 4 = 12$ (bytes) and the count of 4 (array elements) were specified when creating the `QIO_record_info` structure.]

The same factory function signature is used for global and field data, even though for global data the site `index` parameter has no meaning. The applications programmer would doubtless provide different functions for the two cases. For field data, QIO calls the factory function once per lattice site. For global data, QIO calls only once and expects to take all the data in that call. It is the responsibility of the applications programmer to provide the appropriate factory function for each case.

Since the open operation has already registered a `node_number` function, QIO knows to ask only for a site on the present node. The factory function is not required to fetch data from a different node.

The seventh argument of `QIO_write` is passed through as the fourth argument of the `get` function. It can be used to identify the field from which the data is required. In this way only one factory function is needed for each QDP and QLA datatype.

Read a field, array of fields, or array of global data

Prototype	`int QIO_read(QIO_Reader *in,`
	`    QIO_RecordInfo *record_info, QIO_String *xml_record,`
	`    void (*put)(char *buf, int coords[], void *arg),`
	`    int datum_size, void *arg);`

Example	`QIO_read(infile, rec_info, xml_record, QDP_F_put_r,` `    sizeof(QLA_Real), (void *)field);`

The integer return value is 0 for success and 1 for failure. It is assumed the user has prepared the record metadata and the field data in advance. This operation is the inverse of the write operation described. The `put` factory function does the reverse of the `get` function.

Read only the record metadata This utility makes it possible to examine only the header of the record in order to decide whether to continue reading. The state of the file is remembered, so a subsequent call to `QIO_read` reads the full record as though this call had not been made.

Prototype	`int QIO_read_record_info(QIO_Reader *in,` `  QIO_RecordInfo *record_info, QIO_String *xml_record);`
Example	`QIO_read_record_info(infile, rec_info, xml_record);`

Skip to the next record

Prototype	`int QIO_next_record(QIO_Reader *in);`
Example	`QIO_next_record(infile);`

Set and determine the verbosity level A user can control the verbosity of QIO. Choices in increasing degree of chatter are

```
QIO_VERB_OFF
QIO_VERB_LOW
QIO_VERB_MED
QIO_VERB_REG
QIO_VERB_DEBUG
```

Prototype	`int QIO_verbose(int level);`
Example	`oldlevel = QIO_verbose(QIO_OFF);`

A user can also inquire about the current verbosity level with the following function.

Prototype	`level = QIO_verbosity();`

4.5   File format conversion - utilities

The following single-processor utilities are generated when the package is built for a single processor:

- `qio-convert-mesh-singlefs` Converts files from single file to partition file format and vice versa. Optionally, the conversion from partition file to single file is done with ILDG compatibility. The partition files are produced (found) in the same directory. There is one file per node.

- `qio-convert-mesh-pfs` Same as above, except that the partition files are scattered among multiple file systems, so a path table must be supplied to locate them.

- `qio-convert-mesh-ppfs` This utility groups nodes into I/O families with one I/O node (hence one file) per family.

- `qio-copy-mesh-ppfs` Utility for copying files from source directories to local file systems on the appropriate I/O nodes.

- `qio-convert-nersc` Utility for converting a file in NERSC archive format to SciDAC file format. Optionally, the resulting file also made ILDG compatible.

4.6   File format conversion - API

The API provides subroutines for converting between single file and partition file format. Since the partition file format depends on which nodes are I/O nodes and it depends on the data layout as it appears on the compute nodes, the complete code for carrying out file conversion requires an implementation suited to the locale.

The file conversion utilities require information about the data layout on the compute nodes. This information is provided by the `QIO_Layout` structure as described above. Furthermore, it requires information about the file system and the identity of the I/O nodes. This information is encapsulated in a `QIO_FileSystem` structure, which must be completed by the applications programmer.

```
typedef struct {
  int number_io_nodes;
  int type;
  int (*my_io_node)(const int node);
  int (*master_io_node)(void);
  int *io_node;
  char **node_path;
} QIO_Filesystem;
```

The `number_io_nodes` member specifies the number of I/O nodes. If it is the same as the `number_of_nodes` member of the layout structure, each node does its own I/O.

The `type` member is either `QIO_SINGLE_PATH` or `QIO_MULTI_PATH`. In single-path mode, all files are found in the same directory. In multi-path mode, a separate directory is specified for each I/O node.

The `my_io_node` function maps a node to its I/O node, based on the logical node number (rank). The `master_io_node` function returns the number of the master node.

The `io_node` table lists the numbers of the I/O nodes. If the number of I/O nodes is the same as the number of nodes, this table is not required, since each node does its own I/O. In that case the `my_io_node` function should be the identity map.

The `node_path` table is required only in multi-path mode. It lists the directories where the files for the I/O nodes are to be placed. The table has one entry for each I/O node. The entries must correlate with the entries in the `io_node` table.

Convert single file to partition file format

Prototype	`int QIO_single_to_part( const char filename[],`
	`    QIO_Filesystem *fs, QIO_Layout *layout);`
Purpose	Convert an existing file from single to partition format.
Example	`QIO_single_to_part(filename, fs, mpp_layout);`

When converting a non-SciDAC, but ILDG-compatible, file to partfile format, the resulting partfiles are written in SciDAC format. Non-ILDG LIME records are ignored. Currently, the ILDG LFN record is also ignored. When converting a SciDAC ILDG-compatible file to partfile format, the ILDG records, including the ILDG LFN, are also converted.

Convert partition file to single file format

Prototype	`int QIO_part_to_single( const char filename[],`
	`    QIO_Filesystem *fs, QIO_Layout *layout);`
Purpose	Convert an existing file from single to partition format.
Example	`QIO_part_to_single(filename, fs, mpp_layout);`

As a matter of convenience, the file conversion application may be designed so that the code gets the lattice dimension and size from the file. The file should be opened by `QIO_open_read` with the layout `latdim` member set to zero. The lattice dimensions are then taken from the file and kept with the `QIO_Reader` structure. Two accessor utilities are provided for extracting the dimensions from the opaque structure, as described above.

4.7 String Handling with QIO

A few utilities are provided for manipulating the QIO string type `QIO_String` required by the API.

Creating an empty QIO String

Prototype	`QIO_String *QIO_string_create(void);`
Purpose	Creates an empty string.
Example	`fileinfo = QIO_string_create();`

Filling a QIO string from a null-terminated character array

Prototype	`QIO_String *QIO_string_set(QIO_String — *qs,`
	`    const char *const string);`
Purpose	Inserts the null-terminated character array `string` into the string `qs`.
Example	`QIO_string *recinfo = QIO_string_create();`
	`QIO_string_set(recinfo,string);`

Copying a QIO string

Prototype	`QIO_String *QIO_string_copy(QIO_String *dest, QIO_String *src);`
Purpose	Copies the string.
Example	`QIO_string_copy(newxml,oldxml);`

Resizing a string

Prototype	`QIO_String *QIO_string_realloc(QIO_String *dest, int length);`
Purpose	Change the length of the string with truncation if necessary.
Example	`QIO_string_realloc(xml,32);`

Appending to a string

Prototype	`QIO_String *QIO_string_append(QIO_String *dest, const char *string);`
Purpose	Append "string" to the end of the QIO string "dest".
Example	`QIO_string_append(xml,''<info>'';`

Accessing the string length

Prototype	`size_t QIO_string_bytes(const QIO_String *const xml);`
Purpose	Returns a pointer to the null-terminated character array in the string.
Example	`printf("%s\n", QIO_string_bytes(xml));`

Accessing the string character array

Prototype	`char *QIO_string_ptr(const QIO_String *const xml);`
Purpose	Returns the length of the string.
Example	`  length = QIO_string_length(xml);`

Destroying a QIO string

Prototype	`void QIO_string_destroy(QIO_String *xml);`
Purpose	Frees storage.
Example	`QIO_string_destroy(xml);`

4.8   Compilation with QIO

There is a single top-level header file `qio.h` and a single library `libqio.a`.
The QIO package is currently built in conjunction with the independent LIME
package through `configure`, `make` and `make install`.

A   Creating USQCD Standard Files

QIO provides some support for reading and writing USQCD standard file for-
mats. These standard files conform to the generic SciDAC file format, but
the order and content of records and the user record and file XML strings are
standardized. QIO does not enforce the record order or content. This responsi-
bility is left to the applications programmer. But it supports the encoding and
decoding of the standard record and file XML strings.

This section describes the standard USQCD file formats and the QIO API
for constructing and parsing the standard XML strings.

A.1   USQCD Lattice Format

This format is consistent with the ILDG standard.

There is one logical record, namely the gauge field. The user file XML is not
specified in this standard. The user record XML has the following format:

```
<?xml version="1.0" encoding="UTF-8"?>
<usqcdInfo>
<version>1.0</version>
<plaq>(plaquette)</plaq>
<linktr>(link trace)</linktr>
<info>(information)</info>
</usqcdInfo>
```

where the plaquette is the average plaquette normalized to unit trace for the
unit matrix and the link trace is the real part of the average of the trace of the
link matrices. These values are presented in standard floating point notation
with precision appropriate to the precision of the stored field. The information
field can be any string, including an XML substring. The current string limit is
1023 bytes.

This XML string can be constructed by any means before converting it to
a `QIO_String` type and passing it to `QIO_write`. However, QIO provides a
convenience utility for constructing it. Construction takes two steps. First the
user record data structure is created. Then the data structure is encoded as an
XML string.

Creating the USQCD gauge field record information structure

Prototype	`QIO_USQCDLatticeInfo *QIO_create_usqcd_lattice_info` `(char *plaq, char *linktr, char *info);`
Example	`rec_info = QIO_create_usqcd_lattice_info("0.8941325","0.0314259",myXML);`

Prototype	`void QIO_encode_usqcd_lattice_info(QIO_String *record_string,` `QIO_USQCDLatticeInfo *record_info);`
Example	`QIO_encode_usqcd_lattice_info(rec_string, rec_info);`

The resulting string is then passed to `QIO_write`.

Destroying the record information structure

Prototype	`void QIO_destroy_usqcd_lattice_info(QIO_USQCDLatticeInfo *rec_info);`
Purpose	Frees storage.
Example	`QIO_destroy_usqcd_lattice_info(rec_info);`

When the file is read, the user record XML can be parsed by converting the XML string to a data structure and then calling accessors for the data items.

Prototype	`int QIO_decode_usqcd_lattice_info(QIO_USQCDLatticeInfo *record_info,` `  QIO_String *record_string);`
Example	`status = QIO_decode_usqcd_lattice_info(rec_info, rec_string);`

The return value is zero for success and nonzero if errors are encountered.

It may be useful to know whether the field was found during parsing. A set of utilities provides that capability.

Determining whether the field occurs

Prototype	`int QIO_defined_plaq(QIO_USQCDLatticeInfo *rec_info);`
Prototype	`int QIO_defined_linktr(QIO_USQCDLatticeInfo *rec_info);`
Prototype	`int QIO_defined_info(QIO_USQCDLatticeInfo *rec_info);`
Purpose	Returns 1 if the field was found and 0 if not.

Accessing the values

Prototype	`char *QIO_get_plaq(QIO_USQCDLatticeInfo *rec_info);`
Prototype	`char *QIO_get_linktr(QIO_USQCDLatticeInfo *rec_info);`
Prototype	`char *QIO_get_info(QIO_USQCDLatticeInfo *rec_info);`
Purpose	Returns the value as a pointer to the character string.
Example	`sscanf(QIO_get_plaq(rec_info),"%f",&plaq);`

The gauge field byte order conforms to the ILDG standard. The site order is lexicographic with the 0 (x) coordinate varying most rapidly. The data for each lattice site consists of four SU(3) link matrices. Floating points values are written bigendian, with each matrix presented as three rows of three complex numbers. Single and double precision are permitted.

The data type string is `USQCD_F3_ColorMatrix`, a synonym for `QDP_F3_ColorMatrix` in older formats, or `USQCD_D3_ColorMatrix` for double precision.

Here is an example of a call to create the private record XML:

```
QIO_RecordInfo *rec_info;

rec_info = QIO_create_record_info(QIO_FIELD, 0, 0, 0, "USQCD_F3_ColorMatrix",
            "F", 3, 72, 4);
```

20

A.2 USQCD Dirac Propagator Format

There are four standard propagator file formats. Each file includes the source field or fields as a complex scalar or Dirac field as well as the solution fields.

1. `C1D12`: One complex scalar source record and twelve solution records, one for each source spin and color. The solution records correspond to each source spin and color. The order of source spin and color should be sequential with color varying most rapidly.

2. `CD_PAIRS`: Alternating source and solution for any number of pairs. The source in each case is a complex field.

3. `DD_PAIRS`: Alternating source and solution for any number of pairs. The source in each case is a Dirac field.

4. `LHPC`: [USQCD standard under development.]

In all cases the source can be specified either on a time slice or as a full field. The `CD_PAIRS` and `DD_PAIRS` formats could be used for a series of random source/solution pairs, or they could be used for a series of sequential sources plus their solutions. Thus in some, but not all cases, the file contains twelve solutions, one for each source color and spin. When it does, the order of the pairs should be the same as for the `C1D12` format, namely, sequential with color varying most rapidly.

A.2.1 File information

In all cases the user file information is prescribed as follows. It is passed as the `xml_file` parameter to `QIO_open_write` and returned as the `xml_file` parameter by `QIO_open_read`.

```
<?xml version="1.0" encoding="UTF-8"?>
   <usqcdPropFile>
       <version>1.0</version>
       <type>(type string)</type>
       <info>(information)</info>
   </usqcdPropFile>
```

where the file type string is one of

```
"USQCD_DiracFermion_ScalarSource_TwelveSink"
"USQCD_DiracFermion_Source_Sink_Pairs"
"USQCD_DiracFermion_ScalarSource_Sink_Pairs"
"LHPC_DiracPropagator"
```

and the information field is at the user's discretion.

There are convenience function for constructing this string. The first step is to create the file info data structure:

Creating the USQCD propagator file information data structure

Prototype	QIO_USQCDPropFileInfo *QIO_create_usqcd_propfile_info (int type, char *info);
Example	file_info = QIO_create_usqcd_propfile_info (QIO_USQCDPROPFILETYPE_C1D12, myXML);

The type parameter is an integer (not a string) taking on one of these values:

```
QIO_USQCDPROPFILETYPE_C1D12
QIO_USQCDPROPFILETYPE_DD_PAIRS
QIO_USQCDPROPFILETYPE_CD_PAIRS
QIO_USQCDPROPFILETYPE_LHPC
```

The data structure for the file information is then converted to an XML string:

Encoding the file information

Prototype	void QIO_encode_usqcd_propfile_info(QIO_String *file_string, QIO_USQCDPropFileInfo *file_info);
Example	QIO_encode_usqcd_propfile_info(file_string, file_info);

The resulting string is then passed to `QIO_open_write`.

Destroying the file information structure

Prototype	void QIO_destroy_usqcd_propfile_info(QIO_USQCDPropFileInfo *file_info);
Purpose	Frees storage.
Example	QIO_destroy_usqcd_propfile_info(file_info);

Conversely, after obtaining the string from `QIO_open_read`, it can be decoded (converted to a data structure) as follows:

Prototype	int QIO_decode_usqcd_propfile_info(QIO_USQCDPropfileInfo *file_info, QIO_String *file_string);
Example	status = QIO_decode_usqcd_propfile_info(file_info, file_string);

after which the information can be extracted with the accessors.

Determining whether the field occurs

Prototype	int QIO_defined_propfile_type(QIO_USQCDPropFileInfo *file_info);
Prototype	int QIO_defined_propfile_info(QIO_USQCDPropFileInfo *file_info);
Purpose	Returns 1 if the field was found and 0 if not.

Accessing the values

Prototype	`int QIO_get_propfile_type(QIO_USQCDPropFileInfo *file_info);`
Prototype	`char *QIO_get_propfile_info(QIO_USQCDPropFileInfo *file_info);`

The returned integer file type is one of the values listed above for creating the file info data structure or `QIO_ERR_FILE_INFO` if the type is unrecognized.

A.2.2   Source information

For each of the formats there are one or more source records. The user record XML is prescribed as follows.

The record information string for the source record is also prescribed.

```
<?xml version="1.0" encoding="UTF-8"?>
<usqcdSourceInfo>
  <version>1.0</version>
  <info> collaboration use </info>
</usqcdSourceInfo>
```

The operations for creating, encoding, decoding, and accessing values follow the same pattern as with the file information, so we simply list them:

Convenience functions for the propagator source record

Prototype	`QIO_USQCDPropSourceInfo *QIO_create_usqcd_propsource_info`
	`(char *info);`
Prototype	`void QIO_destroy_usqcd_propsource_info`
	`(QIO_USQCDPropSourceInfo *rec_info);`
Prototype	`void QIO_encode_usqcd_propsource_info`
	`(QIO_String *record_string,`
	`  QIO_USQCDPropSourceInfo *record_info);`
Prototype	`int QIO_decode_usqcd_propsource_info`
	`(QIO_USQCDPropSourceInfo *record_info,`
	`  QIO_String *record_string);`
Prototype	`char *QIO_get_usqcd_propsource_info`
	`(QIO_USQCDPropSourceInfo *record_info);`
Prototype	`int QIO_defined_usqcd_propsource_info`
	`(QIO_USQCDPropSourceInfo *record_info);`

As with all QIO records, in addition to providing the user record information, it is necessary to supply the private record information. The following parameters are passed to `QIO_create_record_info` before writing the record:

```
QIO_RecordInfo *QIO_create_record_info(int recordtype, int lower[],
        int upper[], int n,
```

```
        char *datatype, char *precision,
        int colors, int spins, int typesize,
        int datacount);
```

The record type field is `QIO_FIELD` or `QIO_HYPER`. In the former case it is permissible to pass null pointers for `lower` and `upper` and a zero value for `n`. For a source specified on a single time slice, these arrays specify the bounds of the time slice, as illustrated in Sec. 4.2 above. For complex source fields, the data type field is `"USQCD_F_Complex"` or `"USQCD_D_Complex"` and for Dirac vector source fields, the data type field is `"USQCD_F3_DiracFermion"` or `"USQCD_D3_DiracFermion"`. The precision field in either case is `"F"` or `"D"`. The colors and spins parameters apply to a Dirac spinor field and should be zero for a complex source field. The type size specifies the byte count for the site data, and the data count field should always be 1.

A.2.3 Dirac solution fields

For each of the above file formats there are Dirac solution fields. The user record information is prescribed as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
    <usqcdPropInfo>
        <version>1.0</version>
        <spin>(spin)</spin>
        <color>(color)</color>
        <info>(information)</info>
    </usqcdPropInfo>
```

The spin and color values are required for the `C1D12` format and are optional for the other formats. QIO does not enforce this requirement, but provides two convenience functions for creating the data structure for this string depending on whether the spin and color are to be encoded.

Convenience functions for the propagator source record

Prototype	`QIO_USQCDPropRecordInfo *QIO_create_usqcd_proprecord_info` `(char *info);`
Prototype	`QIO_USQCDPropRecordInfo *QIO_create_usqcd_proprecord_sc_info` `(int spin, int color, char *info);`
Prototype	`void QIO_destroy_usqcd_proprecord_info` `(QIO_USQCDPropRecordInfo *rec_info);`

Prototype	`void QIO_encode_usqcd_proprecord_info`
	  `(QIO_String *record_string,`
	  `QIO_USQCDPropRecordInfo *record_info);`

Prototype	`int QIO_decode_usqcd_proprecord_info`
	  `(QIO_USQCDPropRecordInfo *record_info,`
	  `QIO_String *record_string);`

Prototype	`int QIO_defined_usqcd_proprecord_spin` `(QIO_USQCDPropRecordInfo *record_info);`
Prototype	`int QIO_defined_usqcd_proprecord_color` `(QIO_USQCDPropRecordInfo *record_info);`
Prototype	`int QIO_defined_usqcd_proprecord_info` `(QIO_USQCDPropRecordInfo *record_info);`
Prototype	`int QIO_get_usqcd_proprecord_spin` `(QIO_USQCDPropRecordInfo *record_info);`
Prototype	`int QIO_get_usqcd_proprecord_color` `(QIO_USQCDPropRecordInfo *record_info);`
Prototype	`char *QIO_get_usqcd_proprecord_info` `(QIO_USQCDPropRecordInfo *record_info);`

For the private record information structure, the following fields are used:
The data type is either `"USQCD_F3_DiracFermion"` or `"USQCD_D3_DiracFermion"`
for single or double precision, respectively, and the precision is likewise either
`"F"` or `"D"`.

A.3   USQCD Staggered Propagator Format

The staggered propagator formats follow the same pattern as the first three
Dirac propagator formats. Each file includes the source field or fields as a
complex scalar or color vector field as well as the solution fields.

1. `C1V3`: One complex scalar source record and three solution records, one
   for each source color.

2. `CV_PAIRS`: Alternating source and solution for any number of pairs. The
   source in each case is a complex field.

3. `VV_PAIRS`: Alternating source and solution for any number of pairs. The
   source in each case is a color vector field.

In all cases the source can be specified either on a time slice or as a full field.
The `CV_PAIRS` and `VV_PAIRS` formats could be used for a series of random

source/solution pairs, or they could be used for a series of sequential sources plus their solutions. Thus in some, but not all cases, the file contains sets of three solutions, one for each source color.

A.3.1 File information

In all cases the user file information is prescribed as follows. It is passed as the `xml_file` parameter to `QIO_open_write` and returned as the `xml_file` parameter by `QIO_open_read`.

```
<?xml version="1.0" encoding="UTF-8"?>
   <usqcdKSPropFile>
       <version>1.0</version>
       <type>(type string)</type>
       <info>(information)</info>
   </usqcdKSPropFile>
```

where the file type string is one of

```
"USQCD_ColorVector_ScalarSource_ThreeSink"
"USQCD_ColorVector_Source_Sink_Pairs"
"USQCD_ColorVector_ScalarSource_Sink_Pairs"
```

and the information field is at the user's discretion.

As with the Dirac propagator, there are convenience function for constructing this string. The first step is to create the file info data structure:

Creating the USQCD propagator file information data structure

Prototype	`QIO_USQCDKSPropFileInfo *QIO_create_usqcd_kspropfile_info` `(int type, char *info);`
Example	`file_info = QIO_create_usqcd_kspropfile_info` `(QIO_USQCDKSPROPFILETYPE_C1V3, myXML);`

The type parameter is an integer (not a string) taking on one of these values:

```
QIO_USQCDKSPROPFILETYPE_C1V3
QIO_USQCDKSPROPFILETYPE_VV_PAIRS
QIO_USQCDKSPROPFILETYPE_CV_PAIRS
```

The data structure for the file information is then converted to an XML string:

Encoding the file information

Prototype	`void QIO_encode_usqcd_kspropfile_info(QIO_String *file_string,` `QIO_USQCDKSPropFileInfo *file_info);`
Example	`QIO_encode_usqcd_kspropfile_info(file_string, file_info);`

The resulting string is then passed to `QIO_open_write`.

Destroying the file information structure

Prototype	`void QIO_destroy_usqcd_kspropfile_info(QIO_USQCDKSPropFileInfo *file_info);`
Purpose	Frees storage.
Example	`QIO_destroy_usqcd_kspropfile_info(file_info);`

Conversely, after obtaining the string from `QIO_open_read`, it can be decoded (converted to a data structure) as follows:

Prototype	`int QIO_decode_usqcd_kspropfile_info(QIO_USQCDKSPropfileInfo *file_info,` `QIO_String *file_string);`
Example	`status = QIO_decode_usqcd_kspropfile_info(file_info, file_string);`

after which the information can be extracted with the accessors.

Determining whether the field occurs

Prototype	`int QIO_defined_kspropfile_type(QIO_USQCDKSPropFileInfo *file_info);`
Prototype	`int QIO_defined_kspropfile_info(QIO_USQCDKSPropFileInfo *file_info);`
Purpose	Returns 1 if the field was found and 0 if not.

Accessing the values

Prototype	`int QIO_get_kspropfile_type(QIO_USQCDKSPropFileInfo *file_info);`
Prototype	`char *QIO_get_kspropfile_info(QIO_USQCDKSPropFileInfo *file_info);`

The returned integer file type is one of the values listed above for creating the file info data structure or `QIO_ERR_FILE_INFO` if the type is unrecognized.

A.3.2 Source information

For each of the formats there are one or more source records. The user record XML is prescribed as follows.

The record information string for the source record is also prescribed.

```
<?xml version="1.0" encoding="UTF-8"?>
<usqcdSourceInfo>
  <version>1.0</version>
  <info> collaboration use </info>
</usqcdSourceInfo>
```

The operations for creating, encoding, decoding, and accessing values follow the same pattern as with the file information, so we simply list them:

Convenience functions for the kspropagator source record

Prototype	`QIO_USQCDKSPropSourceInfo *QIO_create_usqcd_kspropsource_info` `(char *info);`
Prototype	`void QIO_destroy_usqcd_kspropsource_info` `(QIO_USQCDKSPropSourceInfo *rec_info);`
Prototype	`void QIO_encode_usqcd_kspropsource_info` `(QIO_String *record_string,` `  QIO_USQCDKSPropSourceInfo *record_info);`
Prototype	`int QIO_decode_usqcd_kspropsource_info` `(QIO_USQCDKSPropSourceInfo *record_info,` `  QIO_String *record_string);`
Prototype	`char *QIO_get_usqcd_kspropsource_info` `(QIO_USQCDKSPropSourceInfo *record_info);`
Prototype	`int QIO_defined_usqcd_kspropsource_info` `(QIO_USQCDKSPropSourceInfo *record_info);`

As with all QIO records, in addition to providing the user record information, it is necessary to supply the private record information. The following parameters are passed to `QIO_create_record_info` before writing the record:

```
QIO_RecordInfo *QIO_create_record_info(int recordtype, int lower[],
       int upper[], int n,
       char *datatype, char *precision,
       int colors, int spins, int typesize,
       int datacount);
```

The record type field is `QIO_FIELD` or `QIO_HYPER`. In the former case it is permissible to pass null pointers for `lower` and `upper` and a zero value for `n`. For a source specified on a single time slice, these arrays specify the bounds of the time slice, as illustrated in Sec. 4.2 above. For complex source fields, the data type field is `"USQCD_F_Complex"` or `"USQCD_D_Complex"` and for color vector source fields, the data type field is `"USQCD_F3_ColorVector"` or `"USQCD_D3_ColorVector"`. The precision field in either case is `"F"` or `"D"`. The spins parameter is zero. The colors parameter applies to a color vector field and should be zero for a complex source field. The type size specifies the byte count for the site data, and the data count field should always be 1.

A.3.3   Color vector solution fields

For each of the above file formats there are color vector solution fields. The user record information is prescribed as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
   <usqcdKSPropInfo>
      <version>1.0</version>
      <color>(color)</color>
```

```
    <info>(information)</info>
  </usqcdKSPropInfo>
```

The color value is required for the `C1V3` format and is optional for the other
formats. QIO does not enforce this requirement, but provides two convenience
functions for creating the data structure for this string depending on whether
the color is to be encoded.

Convenience functions for the propagator source record

Prototype	`QIO_USQCDKSPropRecordInfo *QIO_create_usqcd_ksproprecord_info` ` (char *info);`
Prototype	`QIO_USQCDKSPropRecordInfo *QIO_create_usqcd_ksproprecord_c_info` `(int color, char *info);`
Prototype	`void QIO_destroy_usqcd_ksproprecord_info` ` (QIO_USQCDKSPropRecordInfo *rec_info);`

Prototype	`void QIO_encode_usqcd_ksproprecord_info`
	` (QIO_String *record_string,`
	` QIO_USQCDKSPropRecordInfo *record_info);`

Prototype	`int QIO_decode_usqcd_ksproprecord_info`
	` (QIO_USQCDKSPropRecordInfo *record_info,`
	` QIO_String *record_string);`
	` (QIO_USQCDKSPropRecordInfo *record_info);`

Prototype	`int QIO_defined_usqcd_ksproprecord_color` `(QIO_USQCDKSPropRecordInfo *record_info);`

Prototype	`int QIO_defined_usqcd_ksproprecord_info`
	`(QIO_USQCDKSPropRecordInfo *record_info);`
	`(QIO_USQCDKSPropRecordInfo *record_info);`

Prototype	`int QIO_get_usqcd_ksproprecord_color` `(QIO_USQCDKSPropRecordInfo *record_info);`
Prototype	`char *QIO_get_usqcd_ksproprecord_info` `(QIO_USQCDKSPropRecordInfo *record_info);`

For the private record information structure, the following fields are used:
The data type is either **"USQCD_F3_ColorVector"** or **"USQCD_D3_ColorVector"**
for single or double precision, respectively, and the precision is likewise either
"F" or **"D"**.