QG8 Specification and Programmer's Reference

Version 1.0, July 18, 2021

F. Rayment and S. Whitlock[1]

Centre Européen des Sciences Quantiques (CESQ) and Institut de Science et d'Ingénierie Supramoléculaires (ISIS, UMR7006), University of Strasbourg and CNRS

QG8 is a lightweight format for storing and exchanging numerical tensor data and data flow graphs in a binary format. It was created as way to store quantum models and data in a single file, including quantum gate and quantum circuit specifications, control pulse definitions, time independent and time-dependent model Hamiltonians, device characteristics, calibration data and noise models, user defined control flow instructions and simulation or measurement results. This facilitates numerical simulation and optimization of quantum problems as well as interfacing with quantum computers.

A QG8 file consists of a header and a collection of chunks that encode a directed acyclic graph. Each chunk can represent either a *tensor node* (containing numerical tensor data and an associated header) acting as an input or placeholder, an *op node* which does not contain data but specifies operations that should be performed on the data, or an *adjacency tensor* indicating how information flows through the graph. The advantage of this approach is that it allows to store all the data as well as the instructions on how to perform computations on the data in a single object which can then easily be exchanged between different systems.

There is a large and growing number of software packages which use directed acyclic graphs and tensors that should be compatible with QG8 for representing quantum problems, including TensorFlow, Tensorflow Quantum, Pytorch, Jax, Boulder Opal, Qiskit Terra, and Pennylane.

Key features:

- Efficient storage of real or complex valued scalars, vectors, matrices and higher rank tensors using sparse packing and dynamically assigned index data types.
- Uncompressed binary format for fast file I/O, which can be further compressed to save space and allow for transmission of large problems physically or over networks.
- Possibility to load individual nodes or to iteratively read a graph without necessarily loading the entire file into the computer's memory or RAM.
- Tensor data can be stored as integers, single or double precision floating point numbers and complex numbers.
- Practically no storage limits. QG8 can in principle store an unlimited number of tensors with rank $< 2^{16}$, each containing up to 2^{64} nonzero elements.
- Accommodates any tensor format that can be exported as a list of indices and values, including custom packing formats, e.g., to efficiently store symmetric or Hermitian matrices.
- Chunks or graph nodes can be additionally specified by flags and an optional string label to facilitate integration and use.
- Open format with libraries for fast and convenient integration into Python and C/C++ code.

[1]Electronic mail: whitlock@unistra.fr

The first section of this document defines the QG8 file format and details of the core implementation. The second section contains a reference for programming within the QG8 system and how to use the file format to store data graphs.

The syntax used in this document is that of C code, and as such, `*` refers to a pointer, which typically refers to an array of elements or a struct such as `qg8_file` or `qg8_tensor`. In the case where `**` or further is displayed, one may assume the data to be an array of arrays and so on.

Version history

Table 1:

Specification version	`version`
v1.0	1

Contents

1 File format

1.1 Definition

QG8 specifies an exchange data format for tensors of numerical data and data flow graphs. QG8 files can be identified by the suffix `.qg8` and consist of one or more chunks encoding a graph that can be saved in binary format. These binaries can then be further compressed to save space and allow for transmission of large problems physically or over the net.

All data structures defined in this section are written in C syntax. Any data types beginning with `u` imply an unsigned type (without which all types are implicitly implied as being signed), and any number after the type name designates the number of bits the type occupies. An example of this is `uint16_t` which signifies an unsigned integer type capable of containing 16-bit numbers. All data types are defined in `little-endian` order except for strings which are defined below.

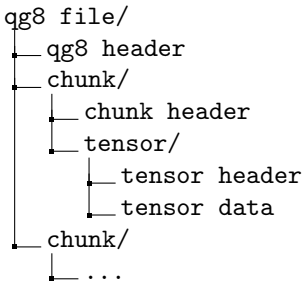
To implement the QG8 format, one must at a minimum include the base functionality specified in this document. The implementer must also ensure all file data structures are packed tightly. That is to say, byte boundaries must be respected, no padding may be applied to the written file and all portions of each structure must be written and read in the exact order that they appear in the specification.

Any time the type `dynamic` occurs the field is not necessarily defined by one single type, but will be assigned dynamically depending on one or more preceding bytes.

Finally, any time a "string" is mentioned in quotation marks, the implementer must ensure that the data is written to and read from left to right into ASCII where the first character corresponds to the first byte, and so on. The string comprises the characters inside the quotation marks, and the marks themselves are to be ignored.

1.2 QG8 file structure

A QG8 file is a serialized binary file based on the following structure which is detailed further below



1.2.1 qg8 header

The first component of a QG8 file is the `qg8 header` which only appears once. This header contains the following elements:

```
typedef struct
{
    uint8_t signature[8];
    uint16_t version;
    uint8_t _reserved[6];
} qg8_file_header;
```

`signature` specifies an 8 byte string which starts with `"QG8"`. The last 5 bytes can be chosen by the implementer. This signature is always the first 8 bytes to appear in the file and is used internally to identify valid `.qg8` files.

`version` refers to the version of the specification that the file adheres to. **The current version of this specification is** `1 = \x01`. Table 1 at the front of this reference contains a list valid version numbers.

`_reserved` contains exactly 6 bytes. Reserved bytes are not currently used and their values can be ignored in the implementation.

1.2.2 Chunk

The basic data unit of a QG8 file is a chunk. QG8 supports an unlimited number of chunks in a single file. Each chunk must start with a chunk header.

1.2.2.1 Chunk header

The chunk header contains identifying information about the chunk and its contents defined as follows:

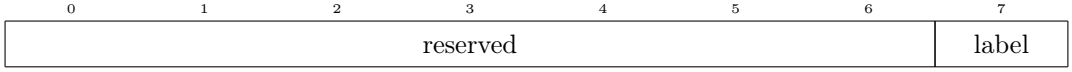
```
typedef struct
{
    uint16_t type;
    uint8_t flags;
    uint8_t string_id[16];
    uint8_t _reserved[5];
    uint64_t skip;
} qg8_chunk_header;
```

`type` specifies the type of data that is stored in the chunk or the operation that is to be performed on input data.

The only chunk type explicitly defined in this specification is `QG8_TYPE_ADJACENCY = 1`. In most cases, a graph will include a single adjacency chunk containing an adjacency matrix (rank 2 tensor) where non-zero entries specify (optionally weighted) directed edges connecting pairs of graph nodes. Adjacency matrices are stored as QG8 tensors in coordinate format as arrays of row indices, column indices and values. The indices refer to nodes (other chunks) in the order that they appear in the file. If provided, the implementation should use this chunk to assign each node with a set of input nodes which determines the order of operations to be performed when computing the graph.

A table of other recommended types is in section 2.1.4. But for now their implementation and functionality should be defined by the implementer. Further standardization will be considered for future versions of the QG8 format. For the moment we recommend that custom types defined by the implementer be assigned values > 32.

`flags` is an 8-bit field that contains 8 boolean values. The layout is as follows:



`label` should be set to `1 = True` if `string_id` contains a 16 character string (described below), or `0 = False` if it is omitted. The remaining 7 bits are currently undefined in the specification.

`string_id` is a 16 byte character array that contains user defined identifying information about the chunk. If the user wishes to use a `string_id` that is less than 16 bytes, the implementer must ensure that the remaining bytes are set to the byte value `0 = \x00`. Each string character

can take any value defined in ASCII except for `\x00`.

`_reserved` 5 bytes for future versions of the specification.

`skip` contains the number of bytes occupied by the remaining chunk data immediately following this field until the last byte of the current chunk or the end of the file. If `skip` is `0` then the chunk does not contain a tensor, otherwise it specifies the exact number of bytes reserved for the tensor (see next section). This number can be used in the implementation of the `qg8_file_next` function to advance to the next chunk and seek through an open file without needing to read the entire chunk.

1.2.3 Tensor

Chunks may optionally contain a tensor which will be used as input data or to store results of computations performed on the graph. A tensor is defined if the `skip` field is > 0, in which case the tensor will comprise of a header and tensor data with at least one element. QG8 does not currently support ragged tensors (tensors with dimensions whose slices may have different lengths) so these are assumed to be padded out with extra zeros.

1.2.3.1 Tensor header

The tensor header contains metadata specifying the user defined encoding and data types used to represent the tensor data. It includes the following:

```
typedef struct
{
    uint8_t packing;
    uint8_t itype_id;
    uint8_t dtype_id;
    uint16_t rank;
    uint8_t _reserved[3];
    dynamic *dims;
    uint64_t num_elements;
} qg8_tensor_header;
```

`packing` is an 8-bit integer that specifies the format used to store the tensor data. The reserved `packing` codes can be found in section 2.1.6.

`itype_id` and `dtype_id` indicate the data type used to store the indices and the data respectively for the tensor. The following data types may be used:

Data types
QG8_DTYPE_UINT8
QG8_DTYPE_UINT16
QG8_DTYPE_UINT32
QG8_DTYPE_UINT64
QG8_DTYPE_INT8
QG8_DTYPE_INT16
QG8_DTYPE_INT32
QG8_DTYPE_INT64
QG8_DTYPE_FLOAT32
QG8_DTYPE_FLOAT64
QG8_DTYPE_COMPLEX64
QG8_DTYPE_COMPLEX128

For a full list of all data types, see section 2.1.5.

Currently the only data types accepted for `itype_id` are unsigned integer types. `itype_id` should be set automatically in the implementation in order to accommodate the maximum dimensions of the tensor data. **This field is dynamically assigned**.

The dynamic index types supported by `itype_id` are:

Index types
QG8_DTYPE_UINT8
QG8_DTYPE_UINT16
QG8_DTYPE_UINT32
QG8_DTYPE_UINT64

`rank` indicates how many dimensions the tensor has. A vector is a tensor of rank 1, a matrix is a tensor of rank 2, and so on. The minimum rank allowed is `rank=1`, which means that scalars should be stored as length 1 vectors. The maximum rank supported is $2^{16} - 1$.

`_reserved` 3 bytes.

`num_elements` defines the number of tensor elements that are stored in the file for the given tensor. Each element corresponds to a single stored value, i.e a dense 5×5 matrix will have 25 elements.

`dims` holds the lengths of each tensor dimension. The type of the field is specified by `itype_id`.

1.2.3.2 Tensor data

The final level of the file is the tensor data block.

```
typedef struct
{
    dynamic **indices;
    dynamic *re;
    dynamic *im;
} qg8_tensor_element;
```

Tensor data is stored sequentially starting with `num_elements` indices for the first tensor dimension, then the second dimension (if `rank>1`), up to `rank` dimensions. This is then followed by a length `num_elements` array of real values and a length `num_elements` array of imaginary values (if `dtype_id` is one of the complex types). Each block is stored sequentially in this order with no byte padding between elements. Tensor values (and corresponding indices) can be stored in any order upon creation, but the order must be preserved upon file read or write.

`indices` points to an array of arrays which specify the position of each tensor element. This field consists of `num_elements` $\times$ `rank` unsigned integers which will be serialized upon writing to file, dimension-by-dimension (i.e. `indices`$[0,0]$, `indices`$[1,0]$, ..., `indices`$[m,0]$, `indices`$[0,1]$, `indices`$[1,1]$, ..., `indices`$[m,n]$ for a $m \times n$ dimensional matrix/tensor). Each element of `indices` has a `dynamic` type set by `itype_id` in the tensor header.

`re` is the real part of the tensor values which must contain `num_elements` elements. This field is `dynamic` with the type specified by `dtype_id` in the tensor header.

`im` is the imaginary part of the tensor values which must contain `num_elements` elements. This block is only set if `dtype_id` is set to value `QG8_DTYPE_COMPLEX64` or `QG8_DTYPE_COMPLEX128` otherwise `im` is not used or set to `NULL`. This field is `dynamic` with the type specified by `dtype_id` in the tensor header.

2 QG8 Library and functions

2.1 Constant defines

The following list of defines must be specified by the library. All constants of the same group must be hardcoded with unique values.

2.1.1 File header data

Header data	Value
QG8_VERSION	1
QG8_MAGIC	`"QG8XXXXX"`

2.1.2 File modes

File modes influence the way a file is opened and the behaviour of QG8 functions.

File Modes
QG8_MODE_READ
QG8_MODE_WRITE

2.1.3 Chunk Flags

Chunk flags can be used to set modifiers or special conditions on QG8 chunks.

Chunk Flags
QG8_FLAG_LABEL

2.1.4 Chunk types

Constant	Value	File version
QG8_TYPE_ADJACENCY	1	1
QG8_TYPE_INPUT	2	1
QG8_TYPE_CONSTANT	3	1
QG8_TYPE_KET	4	1
QG8_TYPE_OPERATOR	5	1
QG8_TYPE_OBSERVABLE	6	1
QG8_TYPE_TIME	7	1
QG8_TYPE_TRACK	8	1
QG8_TYPE_NOISESPEC	9	1
QG8_TYPE_ADD	10	1
QG8_TYPE_SUBTRACT	11	1
QG8_TYPE_MATMUL	12	1
QG8_TYPE_JOIN	13	1
QG8_TYPE_SOLVE	14	1
QG8_TYPE_EXPECTATIONVALUE	15	1
QG8_TYPE_SAMPLE	16	1

2.1.5 Data types

Constant	Value	File version
QG8_DTYPE_BOOL	1	1
QG8_DTYPE_CHAR	2	1
QG8_DTYPE_UINT8	3	1
QG8_DTYPE_UINT16	4	1
QG8_DTYPE_UINT32	5	1
QG8_DTYPE_UINT64	6	1
QG8_DTYPE_INT8	7	1
QG8_DTYPE_INT16	8	1
QG8_DTYPE_INT32	9	1
QG8_DTYPE_INT64	10	1
QG8_DTYPE_FLOAT32	11	1
QG8_DTYPE_FLOAT64	12	1
QG8_DTYPE_COMPLEX64	13	1
QG8_DTYPE_COMPLEX128	14	1

2.1.6 Tensor packing formats

Constant	Value	File version
QG8_PACKING_FULL	1	1
QG8_PACKING_SPARSE_COO	2	1
QG8_PACKING_HALF_HERMITIAN	3	1

`QG8_PACKING_FULL` indicates that all elements of the tensor data are stored, including zeros.

`QG8_PACKING_SPARSE_COO` indicates that only non-zero elements of the tensor data are stored in coordinate format.

`QG8_PACKING_HALF_HERMITIAN` is a QG8 specific packing for Hermitian matrices which indicates that only half of the matrix elements (upper-diagonal, lower-diagonal or a mix of either) will be

stored, and that a Hermitian matrix will be constructed by adding the conjugate transpose upon import or use.

Other packing formats may be defined by the implementation and assigned other values up to 255.

2.2 Data structures

2.2.1 `qg8_graph`

The `qg8_graph` structure points to one or more `qg8_chunk` structs which can represent tensor nodes, op nodes and an optional adjacency tensor which specifies the (directed) edges between nodes.

2.2.2 `qg8_file`

The `qg8_file` structure is used to point to files for read and write operations. Files may be opened in either read mode or write mode.

2.2.3 `qg8_iter`

The `qg8_iter` structure is used as an iterator for `qg8_file` which allows `qg8_chunk`s to be loaded sequentially from the file and into memory. Chunks loaded by this method should be freed from memory in implementations that do not have automatic garbage collection via the `qg8_chunk_destroy` function.

2.2.4 `qg8_chunk`

A `qg8_chunk` structure references chunk header data and an optional `qg8_tensor`.

2.2.5 `qg8_tensor`

The `qg8_tensor` structure references tensor header information and data as arrays of indices, real values and imaginary values. The structure must reference all the data specified in sections 1.2.3.1 and 1.2.3.2. Tensor formats provided by external libraries should be converted to a `qg8_tensor` for read and write operations. Any other content is implementation specific and up to the implementer.

2.3 File I/O operations

File operations are those that act directly upon a `qg8_file`. These are distinct from the data functions which are used to buffer data for preparation for a file write (via `qg8_file_flush`).

2.3.1 `qg8_file_open`

Signature : `qg8_file *qg8_file_open(const char *filename, int mode)`

Create a new reference to a file denoted by the `filename` parameter. The file can be opened in either read or write mode, but not both at the same time. The mode may be selected by setting the `mode` parameter to `QG8_MODE_READ` or `QG8_MODE_WRITE`.

Returns : A new instance of `qg8_file` that points to the open file.

Errors : Raise an exception or terminate with an appropriate error message for any of the following conditions:

- `mode` is not equal to `QG8_MODE_READ` or `QG8_MODE_WRITE`
- `mode` is `QG8_MODE_READ` and `filename` is not a valid QG8 file
- any other error occurs during execution

2.3.2 `qg8_file_flush`

Signature : `int qg8_file_flush(qg8_file *file)`

Flushes all queued write data to the specified file. The `file` parameter must be of the type `qg8_file`. This function must be called after all write data has been queued using the `qg8_file_write_chunk` function. Indices of tensors provided in a type other than indicated by the `dynamic itype_id` will be automatically converted upon flush.

Returns : 1 if the file flushed successfully.

Errors : Raise an exception or terminate with an appropriate error message for any of the following conditions:

- `file` is `NULL` or not a `qg8_file`
- `file` is not open in write mode
- any other error occurs during execution

2.3.3 `qg8_file_close`

Signature : `int qg8_file_close(qg8_file *file)`

Closes a file that has previously been opened via. `qg8_file_open`. The `file` parameter must be of the type `qg8_file`.

Returns : 1 if the file closes successfully.

Errors : Raise an exception or terminate with an appropriate error message for any of the following conditions:

- `file` is `NULL` or not a `qg8_file`
- any other error occurs during execution

2.3.4 `qg8_file_write_chunk`

Signature : `int qg8_file_write_chunk(qg8_file *file, qg8_chunk *chunk)`

Prepares a chunk for writing to a file. If the chunk and/or its tensor data is modified after calling this function, it should also be reflected in the data designated for writing before the function `qg8_file_flush` is called. That is to say, this function must store a reference to the chunk upon calling the function. If this function is called multiple times with the same input, a copy of the chunk will be written once for each call.

Returns : 1 if the chunk was successfully written.

Errors : Raise an exception or terminate with an appropriate error message for any of the following conditions:

- `file` is `NULL` or not a `qg8_file`
- `file` is not open in write mode
- `chunk` is `NULL` or not a `qg8_chunk`
- any other error occurs during execution

2.4 I/O iterator operations

The file iterator exists to load chunks individually from a file which allows for more efficient use of memory (rather than reading the entire file at once).

2.4.1 `qg8_file_iterator`

Signature : `qg8_iter qg8_file_iterator(qg8_file *file)`

Return a new iterator for a given file. This iterator may be used to skip or iteratively load each individual chunk from a file into memory.

Returns : A new iterator for a given file.

Errors : Raise an exception or terminate with an appropriate error message for any of the following conditions:

- `file` is not a `qg8_file` or is not open in read mode
- any other error occurs during execution

2.4.2 `qg8_file_has_next`

Signature : `int qg8_file_has_next(qg8_iter *iter)`

Checks if there is another chunk at the next position in the file, otherwise returns 0 to indicate end of file.

This function may be called before extracting a chunk from a file so that it may be used in a for loop as so:

```
qg8_file *f;
qg8_iter *i;
qg8_chunk *c;
for (i = qg8_file_iterator(f); qg8_file_has_next(i) == 1; )
{
    c = qg8_file_extract(i);
    ...
}
```

Returns : 1 if the next chunk may be read from an iterator, 0 if at the end of the file.

Errors : Raise an exception or terminate with an appropriate error message for any of the following conditions:

- `iter` is `NULL` or not a `qg8_file`
- `iter` is associated with a file that is not open in read mode
- any other error occurs during execution

2.4.3 `qg8_file_next`

Signature : `int qg8_file_next(qg8_iter *iter)`

Advances a file iterator to the next chunk if possible.

This can be used in place of `qg8_file_extract` to skip to the next chunk without reading the data.

Returns : 1 if a chunk is advanced successfully, 0 if it cannot, e.g. the iterator is at the end of the file.

Errors : Raise an exception or terminate with an appropriate error message for any of the following conditions:

- `iter` is `NULL` or not a `qg8_file`
- `iter` is associated with a file that is not open in read mode
- any other error occurs during execution

2.4.4 `qg8_file_extract`

Signature : `qg8_chunk *qg8_file_extract(qg8_iter *iter)`

Reads a chunk from a file given an iterator.

Returns : A chunk loaded from a file.

Errors : Raise an exception or terminate with an appropriate error message for any of the following conditions:

- `iter` is `NULL` or not a `qg8_iter`
- `iter` is associated to a `qg8_file` that is not open in read mode
- any other error occurs during execution

2.5 Chunk operations

Chunk operations apply to `qg8_chunk` data types.

2.5.1 `qg8_chunk_create`

Signature :`qg8_chunk *qg8_chunk_create(uint16_t type, uint8_t flags, uint8_t *string_id, qg8_tensor *tensor)`

Creates a new chunk of type `type`, that optionally holds a tensor. The string id of the chunk may also be optionally specified and be up to 16 characters long. `flags` specifies optional settings for the chunk. The least significant bit of the flag byte will be automatically set to 1 if a `string_id` is provided other than `NULL`.

Returns : A new instance of a chunk that can be used in graphs or for file output.

Errors : Raise an exception or terminate with an appropriate error message if:

- any error occurs during execution

2.5.2 `qg8_chunk_destroy`

Signature : `int qg8_chunk_destroy(qg8_chunk *chunk)`

Destroy a chunk and free it from memory. If the chunk contains a tensor, the tensor is freed as well. Note that for implementations in programming languages with automatic garbage collectors, this function may do nothing.

Returns : 1 if the chunk is destroyed.

Errors : Raise an exception or terminate with an appropriate error message for any of the following conditions:

- `chunk` is `NULL` or not a `qg8_chunk`
- any other error occurs during execution

2.5.3 `qg8_chunk_get_tensor`

Signature : `qg8_tensor *qg8_chunk_get_tensor(qg8_chunk *chunk)`

Return the tensor that belongs to a chunk. If there is no tensor in the chunk, `NULL` is returned instead.

Returns : The tensor portion of a chunk if it exists, or `NULL` otherwise.

Errors : Raise an exception or terminate with an appropriate error message for any of the following conditions:

- `chunk` is `NULL` or not a `qg8_chunk`
- any other error occurs during execution

2.5.4 `qg8_chunk_get_string_id`

Signature : `uint8_t *qg8_chunk_get_string_id(qg8_chunk *chunk)`

Return the string id of a given chunk, an array of up to 16 characters (of which the first ASCII value 0 indicates the end / null terminator of the string), or `NULL` if the chunk doesn't have a `string_id`.

Returns : The string id of a chunk as an array of `uint8_t` characters, or `NULL` if the chunk is equal to `NULL` or does not contain a string id.

Errors : Raise an exception or terminate with an appropriate error message for any of the following conditions:

- `chunk` is `NULL` or not a `qg8_chunk`
- any error occurs during execution

2.5.5 `qg8_chunk_get_type`

Signature : `uint16_t qg8_chunk_get_type(qg8_chunk *chunk)`

Return the type of a chunk.

Returns : The type of a given chunk.

Errors : Raise an exception or terminate with an appropriate error message for any of the following conditions:

- `chunk` is `NULL` or not a `qg8_chunk`
- any error occurs during execution

2.5.6 `qg8_chunk_get_flags`

Signature : `uint8_t qg8_chunk_get_flags(qg8_chunk *chunk)`

Return the flags of a chunk.

Returns : The flags of a given chunk.

Errors : Raise an exception or terminate with an appropriate error message for any of the following conditions:

- `chunk` is `NULL` or not a `qg8_chunk`
- any error occurs during execution

2.6 Tensor operations

Tensor operations are those that apply to tensors of the `qg8_tensor` structure.

2.6.1 `qg8_tensor_create_float`

Signature : `qg8_tensor *qg8_tensor_create_float(uint64_t **indices, float *re, float *im, uint64_t num_elements, uint64_t *dims, uint8_t rank, uint8_t packing)`

Create a QG8 representation of a tensor using data stored as single-precision floating point numbers. Details on the input parameters are documented in sections 1.2.3.1 and 1.2.3.2.

This function accepts only single-precision floating point numbers for both the real and imaginary parts of the tensor. For double precision data use `qg8_tensor_create_double`. For real tensors, the `im` parameter may be set to `NULL`.

Returns : A QG8 tensor.

Errors : Raise an exception or terminate with an appropriate error message for any of the following conditions:

- `rank` is less than 1 or greater than or equal to 2^{16}
- Any elements of `dims` are less than or equal to 0 or greater than or equal to 2^{64}

- `indices`, `re`, or `dims` is `NULL`
- `re` (or `im`) is not an array of type float (single precision)
- any other error occurs during execution

2.6.2 `qg8_tensor_create_double`

Signature : `qg8_tensor *qg8_tensor_create_double(uint64_t **indices, double *re, double *im, uint64_t num_elements, uint64_t *dims, uint8_t rank, uint8_t packing)`

Create a QG8 representation of a tensor using data stored as double-precision floating point numbers. Details on the input parameters are documented in sections 1.2.3.1 and 1.2.3.2.

This function accepts only double-precision floating point numbers for both the real and imaginary parts of the tensor. For single precision data use `qg8_tensor_create_float`. For real tensors, the `im` parameter may be set to `NULL`.

Returns : A QG8 tensor.

Errors : Raise an exception or terminate with an appropriate error message for any of the following conditions:

- `rank` is less than 1 or greater than or equal to 2^{16}
- Any elements of `dims` are less than or equal to 0 or greater than or equal to 2^{64}
- `indices`, `re`, or `dims` is `NULL`
- `re` (or `im`) is not an array of type double (double precision)
- any other error occurs during execution

2.6.3 `qg8_tensor_create_uint8`

Signature : `qg8_tensor *qg8_tensor_create_uint8(uint64_t **indices, uint8_t *re, uint64_t num_elements, uint64_t *dims, uint8_t rank, uint8_t packing)`

Create a QG8 representation of a tensor using data stored as 8 bit unsigned integers. Integer formats can only be used to represent real valued tensors.

Returns : A QG8 tensor.

Errors : Raise an exception or terminate with an appropriate error message for any of the following conditions:

- `rank` is less than 1 or greater than or equal to 2^{16}
- Any elements of `dims` are less than or equal to 0 or greater than or equal to 2^{64}
- `indices`, `re`, or `dims` is `NULL`
- `re` is not an array of the correct integer type
- any other error occurs during execution

2.6.4 qg8_tensor_create_uint16

Signature : `qg8_tensor *qg8_tensor_create_uint16(uint64_t **indices, uint16_t *re, uint64_t num_elements, uint64_t *dims, uint8_t rank, uint8_t packing)`

Create a QG8 representation of a tensor using data stored as 16 bit unsigned integers. For other implementation details refer to section 2.6.3

2.6.5 qg8_tensor_create_uint32

Signature : `qg8_tensor *qg8_tensor_create_uint32(uint64_t **indices, uint32_t *re, uint64_t num_elements, uint64_t *dims, uint8_t rank, uint8_t packing)`

Create a QG8 representation of a tensor using data stored as 32 bit unsigned integers. For other implementation details refer to section 2.6.3

2.6.6 qg8_tensor_create_uint64

Signature : `qg8_tensor *qg8_tensor_create_uint64(uint64_t **indices, uint64_t *re, uint64_t num_elements, uint64_t *dims, uint8_t rank, uint8_t packing)`

Create a QG8 representation of a tensor using data stored as 64 bit unsigned integers. For other implementation details refer to section 2.6.3

2.6.7 qg8_tensor_create_int8

Signature : `qg8_tensor *qg8_tensor_create_int8(uint64_t **indices, int8_t *re, uint64_t num_elements, uint64_t *dims, uint8_t rank, uint8_t packing)`

Create a QG8 representation of a tensor using data stored as 8 bit signed integers. For other implementation details refer to section 2.6.3

2.6.8 qg8_tensor_create_int16

Signature : `qg8_tensor *qg8_tensor_create_int16(uint64_t **indices, int16_t *re, uint64_t num_elements, uint64_t *dims, uint8_t rank, uint8_t packing)`

Create a QG8 representation of a tensor using data stored as 16 bit signed integers. For other implementation details refer to section 2.6.3

2.6.9 qg8_tensor_create_int32

Signature : `qg8_tensor *qg8_tensor_create_int32(uint64_t **indices, int32_t *re, uint64_t num_elements, uint64_t *dims, uint8_t rank, uint8_t packing)`

Create a QG8 representation of a tensor using data stored as 32 bit signed integers. For other implementation details refer to section 2.6.3

2.6.10 qg8_tensor_create_int64

Signature : `qg8_tensor *qg8_tensor_create_int64(uint64_t **indices, int64_t *re, uint64_t num_elements, uint64_t *dims, uint8_t rank, uint8_t packing)`

Create a QG8 representation of a tensor using data stored as 64 bit signed integers. For other implementation details refer to section 2.6.3

2.6.11 `qg8_tensor_destroy`

Signature : `int qg8_tensor_destroy(qg8_tensor *tensor)`

Free a QG8 tensor from memory. In systems where garbage collection is automatic, the behaviour of this function is implementation defined. Otherwise, this function should clear memory. Accessing the tensor after this function is called is undefined behaviour. The `tensor` parameter must be of the type `qg8_tensor`.

Returns : 1 if the tensor is destroyed.

Errors : Raise an exception or terminate with an appropriate error message for any of the following conditions:

- `tensor` is `NULL` or not a `qg8_tensor`
- any other error occurs during execution

2.6.12 `qg8_tensor_get_rank`

Signature : `uint16_t qg8_tensor_get_rank(qg8_tensor *tensor)`

Return the rank of a given tensor.

Returns : The rank of a given tensor as an integer.

Errors : Raise an exception or terminate with an appropriate error message if:

- any error occurs during execution

2.6.13 `qg8_tensor_get_dims`

Signature : `void *qg8_tensor_get_dims(qg8_tensor *tensor)`

Return the dimensions of a given tensor. If the implementation language allows for duck typing or abstract types, `void` may be replaced with the dynamic type.

Returns : The dimensions of a given tensor as a list or array of integers.

Errors : Raise an exception or terminate with an appropriate error message if:

- any error occurs during execution

2.6.14 `qg8_tensor_get_num_elems`

Signature : `uint64_t qg8_tensor_get_num_elems(qg8_tensor *tensor)`

Return the number of defined elements of a given tensor. This should correspond to the size of the indices and values arrays given by `qg8_tensor_get_indices`, `qg8_tensor_get_re` and `qg8_tensor_get_im` respectively. If the tensor is empty or equal to `NULL`, 0 is returned.

Returns : The number of elements in the tensor as an integer.

Errors : Raise an exception or terminate with an appropriate error message if:

- any error occurs during execution

2.6.15 `qg8_tensor_get_indices`

Signature : `uint64_t **qg8_tensor_get_indices(qg8_tensor *tensor)`

Return the indices of a given tensor.

Returns : The indices of a given tensor as a list or array of arrays.

Errors : Raise an exception or terminate with an appropriate error message if:

- any error occurs during execution

2.6.16 `qg8_tensor_get_dtypeid`

Signature : `uint8_t qg8_tensor_get_dtypeid(qg8_tensor *tensor)`

Return the data typecode for tensor elements as a `QG8_DTYPE` constant

Returns : The data typecode used to store values of a given tensor as an integer.

Errors : Raise an exception or terminate with an appropriate error message if:

- any error occurs during execution

2.6.17 `qg8_tensor_get_itypeid`

Signature : `uint8_t qg8_tensor_get_itypeid(qg8_tensor *tensor)`

Return the type of the indices of the tensor that are stored in the file upon write operations as a `QG8_DTYPE` constant. This type always corresponds to an integer type large enough to hold indices up to the largest value specified by the tensor's shape `*dims`.

Returns : The data typecode used to store indices of a given tensor as an integer.

Errors : Raise an exception or terminate with an appropriate error message if:

- any error occurs during execution

2.6.18 `qg8_tensor_get_re`

Signature : `void *qg8_tensor_get_re(qg8_tensor *tensor)`

Return the real part of a tensor. This data must be cast to the correct type which is given by the function `qg8_tensor_get_dtypeid`. If the implementation language allows for duck typing or abstract types, `void` may be replaced with the dynamic type.

Returns : The real part of the given tensor as an array.

Errors : Raise an exception or terminate with an appropriate error message if:

- any error occurs during execution

2.6.19 `qg8_tensor_get_im`

Signature : `void *qg8_tensor_get_im(qg8_tensor *tensor)`

Return the imaginary part of a tensor or `NULL` if the tensor has no imaginary data. This data must be cast to the correct type which is given by the function `qg8_tensor_get_dtypeid`. If the implementation language allows for duck typing or abstract types, `void` may be replaced with the dynamic type.

Returns : The imaginary part of a given tensor as an array or `NULL` if `im` is not defined.

Errors : Raise an exception or terminate with an appropriate error message if:

- any error occurs during execution

2.7 Graph operations

Graph operations apply to `qg8_graph` data types which may in turn affect tensor data stored within graphs.

2.7.1 `qg8_graph_load`

Signature : `qg8_graph *qg8_graph_load(const char *filename)`

This is a wrapper function which internally calls `qg8_file_open`, `qg8_file_extract`, and `qg8_file_close` functions to load a complete graph from a file.

Returns : A graph loaded from a file specified by `filename`.

Errors : Raise an exception or terminate with an appropriate error message for any of the following conditions:

- `filename` is not a valid filename or points to a file which cannot be opened for read.
- any other error occurs during execution

2.7.2 `qg8_graph_create`

Signature : `qg8_graph *qg8_graph_create(void)`

Return a new empty graph.

Returns : A new empty graph.

Errors : Raise an exception or terminate with an appropriate error message for any of the following conditions:

- any error occurs during execution

2.7.3 `qg8_graph_write`

Signature : `int qg8_graph_write(const char *filename, qg8_graph *graph)`

Wrapper function which prepares a collection of chunks (graph) and writes it to a `.qg8` file.

Returns : 1 if the graph was successfully written.

Errors : Raise an exception or terminate with an appropriate error message for any of the following conditions:

- `filename` cannot be opened in write mode
- `graph` is `NULL` or not a `qg8_graph`
- any other error occurs during execution

2.7.4 `qg8_graph_destroy`

Signature : `int qg8_graph_destroy(qg8_graph *graph)`

Destroy a graph and free it from memory. For implementations that use an automatic garbage collector, this function may not directly free the memory used by the graph and as such, this function is implementation defined.

Returns : 1 if the graph is successfully destroyed.

Errors : Raise an exception or terminate with an appropriate error message for any of the following conditions:

- `graph` is `NULL` or not a `qg8_graph`
- any other error occurs during execution

2.7.5 `qg8_graph_get_number_chunks`

Signature : `uint64_t qg8_graph_get_number_chunks(qg8_graph *graph)`

Returns the number of chunks that are currently stored within a graph.

Returns : The number of chunks stored within a graph. The output of this function will be limited to a maximum number of $2^{64} - 1$.

Errors : Raise an exception or terminate with an appropriate error message for any of the following conditions:

- `graph` is `NULL` or not a `qg8_graph`
- any other error occurs during execution

2.7.6 `qg8_graph_get_chunk`

Signature : `qg8_chunk *qg8_graph_get_chunk(qg8_graph *graph, uint64_t idx)`

Returns a chunk from any given graph at a specific index. The number of available chunks can be found using the `qg8_graph_get_number_chunks` function.

Returns : A chunk from a graph at a given index.

Errors : Raise an exception or terminate with an appropriate error message for any of the following conditions:

- `graph` is `NULL` or not a `qg8_graph`
- `idx` is larger than or equal to the number of chunks in the graph
- any other error occurs during execution

2.7.7 `qg8_graph_add_chunk`

Signature : `int qg8_graph_add_chunk(qg8_graph *graph, qg8_chunk *chunk)`

Add a chunk to the provided graph.

Returns : 1 if the chunk is added successfully.

Errors : Raise an exception or terminate with an appropriate error message for any of the following conditions:

- `graph` is `NULL` or not a `qg8_graph`
- `chunk` is `NULL` or not a `qg8_chunk`
- any other error occurs during execution

2.7.8 `qg8_graph_remove_chunk`

Signature : `int qg8_graph_remove_chunk(qg8_graph *graph, qg8_chunk *chunk)`

Removes a chunk from any given graph. This function does not take an index, but rather a pointer to the chunk itself. If the same chunk is referenced multiple times in the graph then all instances will be removed.

Returns : 1 if the chunk is successfully removed, or 0 if the chunk is not removed or not part of the graph.

Errors : Raise an exception or terminate with an appropriate error message for any of the following conditions:

- `graph` is `NULL` or or not a `qg8_graph`
- `graph` contains no chunks
- `chunk` is `NULL` or not a `qg8_chunk`
- any other error occurs during execution