Intel® Quantum SDK API Documentation

Release v1.1

Intel® Corporation

Feb 15, 2024

CONTENTS:

CHAPTER

ONE

BACKENDS

1.1 quantum_backend.h

namespace **iqsdk**

struct **DeviceConfig**

#include <quantum_backend.h> Configurations for a quantum device. This is a struct that users of the SDK will make an instance of when creating a quantum device.

Sets default settings for classes constructed with *DeviceConfig*

Subclassed by *iqsdk::CliffordSimulatorConfig*, *iqsdk::IqsConfig*, *iqsdk::TensorNetworkConfig*

Public Functions

virtual bool **isValid**()

DeviceConfig()

DeviceConfig(*std*::string backend, bool verbose = false, bool synchronous = true)

DeviceConfig(const *DeviceConfig*&) = default

DeviceConfig &**operator=**(const *DeviceConfig*&) = default

DeviceConfig(*DeviceConfig*&&) = default

DeviceConfig &**operator=**(*DeviceConfig*&&) = default

virtual **~DeviceConfig**() = default

Public Members

std::string **backend**

Backend to use.

Valid backends include “IQS”, “QD_SIM”, “Tensor_Network”, “Clifford”

bool **verbose**

bool **synchronous**

class **QuantumDevice**

#include <quantum_backend.h>

Public Functions

QuantumDevice() = default

virtual **~QuantumDevice**() = default

QuantumDevice(const *QuantumDevice*&) = delete

QuantumDevice &**operator=**(const *QuantumDevice*&) = delete

QuantumDevice(*QuantumDevice*&&) = default

QuantumDevice &**operator=**(*QuantumDevice*&&) = default

class **Device**

#include <quantum_backend.h> Subclassed by *iqsdk::CliffordSimulator*, *iqsdk::CustomSimulator*, *iqsdk::SimulatorDevice*

Public Functions

Device()

Device(const *Device*&) = delete

Device &**operator=**(const *Device*&) = delete

Device(*Device*&&) = delete

Device &**operator=**(*Device*&&) = delete

virtual **~Device**()

bool **isValid**()

QRT_ERROR_T **initialize**(*DeviceConfig* &device_config)

Initialize the simulator instance with the given settings.

QRT_ERROR_T **ready**()

Specify that the next `quantum_kernel` called will be run on the device.

QRT_ERROR_T **printVerbose**(bool printVerbose)

Specify whether the device will or will not be put in verbose mode.

QRT_ERROR_T **wait**()

Wait for the device to stop running before continuing.

Useful in asynchronous mode, to ensure all `quantum_kernel` functions that are queued have finished running, and the appropriate `cbit` variables have been set.

Note: APIs that retrieve data from the device are synchronous. APIs that set device properties are asynchronous, so users may not always need to use this method.

Protected Functions

virtual int **getDeviceType**() = 0

Protected Attributes

bool **valid_device_**

std::shared_ptr<*QuantumDevice*> **device**

class **SimulatorDevice** : public *iqsdk*::*Device*

#include <quantum_backend.h> Subclassed by *iqsdk::FullStateSimulator*, *iqsdk::TensorNetworkSimulator*

Public Functions

SimulatorDevice() = default

SimulatorDevice(const *SimulatorDevice*&) = delete

SimulatorDevice &**operator=**(const *SimulatorDevice*&) = delete

SimulatorDevice(*SimulatorDevice*&&) = delete

SimulatorDevice &**operator=**(*SimulatorDevice*&&) = delete

virtual **~SimulatorDevice**() = default

std::vector<double> **getProbabilities**(*std*::vector<*std*::reference_wrapper<*qbit*>> &qids)

Return the conditional probabilities of the qubits given in `qids`.

Useful for returning the conditional probabilities of certain qubits. For example, in a 4-qubit system $\{q_0, q_1, q_2, q_3\}$, the probability associated with `QssIndex("|11>")` with `qids={ref(q0); ref(q1)}` is the sum of the probabilities of being in state $|1100\rangle$, $|1101\rangle$, $|1110\rangle$, or $|1111\rangle$.

Parameters

qids – A non-repeating list of qubit reference wrappers. Not all qubits need be included in `qids` vector, in which case *getProbabilities()* will return conditional probabilities.

QssMap<double> **getProbabilities**(*std*::vector<*std*::reference_wrapper<*qbit*>> &qids, *std*::vector<*QssIndex*> bases, double threshold = -1)

Return a `QssMap` of *QssIndex* to `double`.

The same conditions on qids apply as above.

Parameters

- **bases** – Specifies a subset of the states whose probability you are interested in. If nonempty, *getProbabilities()* will only return the probabilities associated with the given indices. If empty, it will return the probability associated with each quantum state.

- **threshold** – Omits {state,double} pairs where the probability falls below the argument value. If specified,*getProbabilities()* will only return results whose probabilities are greater than or equal to the threshold. Consider retrieving 2^N probabilities for large N. It can become a cumbersome operation. Specifying a subset of states in the bases argument will cause speed up in the execution time of *getProbabilities()* by reducing the total number of operations. Specifying a threshold argument will increase the computational overhead due to performing a comparison of the probability and threshold.

double **getProbability**(*std*::vector<*std*::reference_wrapper<*qbit*>> &qids, *QssIndex* basis)

Return a single probability corresponding to the basis element given.

std::vector<*std*::vector<bool>> **getSamples**(unsigned int num_shots,
std::vector<*std*::reference_wrapper<*qbit*>> &qids)

Return a vector with `num_shots` different samples of measurement results for the given `qubits_ids`.

Generates samples from the full state vector at the end of the last run quantum basic block. Each sample is a vector of length `qids.size()`, whose value at index `i` corresponds to the measurement result of the qubit `qids[i]`. Each `qbit` produces `false` for the measurement result of the `0` state, and `true` for the measurement result of the `1` state.

Efficiently sample from the full-state data in the simulator without having to repeat the gates to prepare that state `num_shots` number of times.

std::vector<double> **getSingleQubitProbs**(*std*::vector<*std*::reference_wrapper<*qbit*>> &qids)

Return a vector of single qubit probabilities of length `qids.size()`.

The order of the `single_qubit_probs` is given by the order of the qubits in the `qids` vector.

Useful if you want only the probabilities of each qubit being in state $|0\rangle$ or $|1\rangle$, ignoring entanglement with other qubits.

std::vector<*std*::complex<double>> **getAmplitudes**(*std*::vector<*std*::reference_wrapper<*qbit*>>
&qids)

Return the complex amplitudes of the state space with respect to the order given by `qids`.

Parameters

qids – a non-repeating list of all qubit ids in scope.

QssMap<*std*::complex<double>> **getAmplitudes**(*std*::vector<*std*::reference_wrapper<*qbit*>> &qids,
std::vector<*QssIndex*> bases, double threshold = -1)

Same functionality as *getAmplitudes()*

The optional arguments `bases` and `threshold` are useful for querying subsets of a state space. If a user only wants to query the amplitudes of a subset of basis elements, specifying those `bases` will be significantly more efficient in time and space. If a user only wants amplitudes above a threshold, specifying a `threshold` will significantly reduce the space required.

Parameters

- **bases** – If nonempty, only return the amplitudes associated with the given indices.
- **threshold** – If given, only return results $a + bi$ such that $a^2 + b^2 >=$ threshold.

std::complex<double> **getAmplitude**(*std*::vector<*std*::reference_wrapper<*qbit*>> &qids, *QssIndex*
basis)

Return a single amplitude corresponding to the basis element given.

Public Static Functions

static *QssMap*<double> **amplitudesToProbabilities**(*QssMap*<*std*::complex<double>> &amplitudes)

Calling `amplitudesToProbabilities(getAmplitudes(...))` is equivalent to *getProbabilities()* but if a user wants to obtain both probabilities and amplitudes, this helper function can be useful.

static void **displayProbabilities**(*std*::vector<double> &probability, *std*::vector<*std*::reference_wrapper<*qbit*>> &qids)

Display to standard output the probability of observing the states composed by the qubits listed in `qids`.

Intended to be passed the input and output pair from a past call to *getProbabilities()*.

static void **displayProbabilities**(*QssMap*<double> &probability)

Display to standard output the probability value of observing each state represented by a *QssIndex* key in `probability`.

static void **displaySamples**(*std*::vector<*std*::vector<bool>> &samples)

static *std*::vector<int> **sampleToEigenvalues**(*std*::vector<bool> &sample)

static *std*::vector<bool> **eigenvaluesToSample**(*std*::vector<int> &eigenvalues)

static *QssMap*<unsigned int> **samplesToHistogram**(*std*::vector<*std*::vector<bool>> &samples)

static *QssMap*<double> **samplesToProbabilities**(*std*::vector<*std*::vector<bool>> &samples)

static void **displayAmplitudes**(*std*::vector<*std*::complex<double>> &amplitudes, *std*::vector<*std*::reference_wrapper<*qbit*>> &qids)

Display to standard output the complex amplitude for the states composed by the qubits listed in `qids`.

Intended to be passed the input and output pair from a past call to *getAmplitudes()*.

static void **displayAmplitudes**(*QssMap*<*std*::complex<double>> &amplitudes)

Display to standard output a summary of the complex amplitude value for each state represented by a *QssIndex* key in `amplitudes`.

namespace **qrt**

Functions

void ***returnRuntimePtr**()

void **setRuntimePtr**(void*)

1.2 quantum_clifford_simulator_backend.h

namespace **iqsdk**

struct **ErrSpec1Q**

#include <quantum_clifford_simulator_backend.h>

Public Functions

ErrSpec1Q() = default

ErrSpec1Q(double x_rate, double y_rate, double z_rate)

ErrSpec1Q(double mag, *std*::pair<double, double> angles)

double **getMagnitude**() const

double **getXRate**() const

double **getYRate**() const

double **getZRate**() const

Private Members

double **magnitude_** = 0.

double **x_** = 0.

double **y_** = 0.

double **z_** = 0.

struct **ErrSpec2Q**

#include <quantum_clifford_simulator_backend.h>

Public Functions

ErrSpec2Q() = default

ErrSpec2Q(double e, double phi, double delta)

double **getZRate**() const

double **getXXRate**() const

double **getXYRate**() const

double **getZZRate**() const

Public Members

double **tot_** = 0

double **zi_** = 0

double **xx_** = 0

double **xy_** = 0

double **zz_** = 0

struct **ErrSpecIdle**

#include <quantum_clifford_simulator_backend.h>

Public Functions

ErrSpecIdle()

ErrSpecIdle(double T1, double T2)

double **getIT1Rate**() const

double **getIT2Rate**() const

Public Members

double **it1_** = 0.

double **it2_** = 0.

struct **ErrorRates**

#include <quantum_clifford_simulator_backend.h>

Public Functions

ErrorRates() = default

ErrorRates(*ErrSpec1Q* NU, *ErrSpec1Q* U1, *ErrSpec2Q* U2, *ErrSpecIdle* I = *ErrSpecIdle*())

Public Members

ErrSpec1Q **prep**

ErrSpec1Q **meas**

ErrSpec1Q **xyrot**

ErrSpec1Q **zrot**

ErrSpecIdle **idle**

ErrSpec2Q **cz**

ErrSpec2Q **swap**

struct **GateTimes**

#include <quantum_clifford_simulator_backend.h>

Public Functions

GateTimes() = default

GateTimes(double NU, double U1, double U2)

Public Members

double **prep**

double **cz**

double **xyrot**

double **zrot**

double **meas**

double **swap**

struct **CliffordSimulatorConfig** : public *iqsdk*::*DeviceConfig*

#include <quantum_clifford_simulator_backend.h>

Public Functions

CliffordSimulatorConfig(unsigned int seed, bool use_errors = false, *ErrorRates* error_rates = {}, *GateTimes* gate_times = {}, bool verbose = false, bool synchronous = true)

Constructor for Clifford Simulator Configuration.

Public Members

unsigned int **seed**

bool **use_errors**

ErrorRates **error_rates**

GateTimes **gate_times**

class **CliffordSimulator** : public *iqsdk*::*Device*

#include <quantum_clifford_simulator_backend.h>

Public Functions

CliffordSimulator()

CliffordSimulator(*iqsdk*::*DeviceConfig* &device_config)

CliffordSimulator(const *CliffordSimulator*&) = delete

CliffordSimulator &**operator=**(const *CliffordSimulator*&) = delete

CliffordSimulator(*CliffordSimulator*&&) = delete

CliffordSimulator &**operator=**(*CliffordSimulator*&&) = delete

~CliffordSimulator() = default

double **getExpectationValue**(*std*::vector<*std*::reference_wrapper<*qbit*>> &qids, *std*::string pauli_string)

Private Functions

virtual int **getDeviceType**()

1.3 quantum_custom_backend.h

namespace **iqsdk**

class **CustomInterface**

#include <quantum_custom_backend.h>

Public Functions

virtual void **RXY**(*qbit* q, double theta, double phi) = 0

Rotation in the XY plane, ideally described by the 2x2 matrix:

$$R_{XY}(\theta,\phi) = \begin{bmatrix} \cos(\theta/2) & -i\exp(-i\phi)\sin(\theta/2) \\ -i\exp(i\phi)\sin(\theta/2) & \cos(\theta/2) \end{bmatrix}$$

virtual void **RZ**(*qbit* q, double angle) = 0

Rotation around the Z axis, ideally described by the 2x2 matrix:

$$R_Z(\theta) = \begin{bmatrix} \cos(\theta/2) - i\sin(\theta/2) & 0 \\ 0 & \cos(\theta/2) + i\sin(\theta/2) \end{bmatrix}$$

virtual void **CPhase**(*qbit* ctrl, *qbit* target, double angle) = 0

CPHASE gate ideally corresponding to the 4x4 matrix:

$$CPHASE(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \exp(-i\theta) \end{bmatrix}$$

virtual void **SwapA**(*qbit* q1, *qbit* q2, double angle) = 0

SWAP-type gate, with arbitrary angle α. Ideally it corresponds to the 4x4 matrix:

$$SWAP(\alpha) = \frac{1}{2}\begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 1+\exp(i\alpha) & 1-\exp(i\alpha) & 0 \\ 0 & 1-\exp(i\alpha) & 1+\exp(i\alpha) & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix}$$

Special cases are the common SWAP gate associated with $\alpha = 0$:

$$SWAP = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

And its square root for $\alpha = \pi/2$:

$$\sqrt{SWAP} = SWAP\left(\frac{\pi}{2}\right) = \frac{1}{2}\begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 1+i & 1-i & 0 \\ 0 & 1-i & 1+i & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix}$$

virtual void **PrepZ**(*qbit* q) = 0

One-qubit state preparation in the Z basis (state $|0\rangle$)

virtual *cbit* **MeasZ**(*qbit* q) = 0

One-qubit measurement in the Z basis.

CustomInterface() = default

CustomInterface(const *CustomInterface*&) = default

CustomInterface &**operator=**(const *CustomInterface*&) = default

CustomInterface(*CustomInterface*&&) = default

CustomInterface &**operator=**(*CustomInterface*&&) = default

virtual **~CustomInterface**() = default

class **CustomSimulator** : public *iqsdk*::*Device*

#include <quantum_custom_backend.h>

Public Functions

CustomSimulator()

CustomSimulator(*iqsdk*::*DeviceConfig* &device_config)

CustomSimulator(const *CustomSimulator*&) = delete

CustomSimulator &**operator=**(const *CustomSimulator*&) = delete

CustomSimulator(*CustomSimulator*&&) = delete

CustomSimulator &**operator=**(*CustomSimulator*&&) = delete

~CustomSimulator() = default

iqsdk::*CustomInterface* ***getCustomBackend**()

Public Static Functions

template<typename **T**, typename ...**Ts**>
static inline *CustomSimulator* ***createSimulator**(*std*::string device_id, *Ts*... args)

template<typename **T**, typename ...**Ts**>
static inline *QRT_ERROR_T* **registerCustomInterface**(*std*::string device_id, *Ts*... args)

Private Functions

virtual int **getDeviceType**()

Private Static Functions

static *QRT_ERROR_T* **registerCustomInterfaceWrapper**(*std*::function<*CustomInterface**()>, *std*::string device_id)

1.4 quantum_full_state_simulator_backend.h

namespace **iqsdk**

Functions

std::vector<*std*::complex<double>> **ParseChiMatrixFromCsvFiles**(unsigned num_qubits, *std*::string fname_real, *std*::string fname_imag)

Utility function to load a process matrix from a pair of CSV files.

The process matrix is a square matrix of size $4^n x 4^n$, with n being the `num_qubits` value. The entries are complex and this function expects the real and imaginary part to be stored in separate files. Each files corresponds to comma separated values, with one row stored in each line.

Variables

static const struct *IqsCustomOp* **k_iqs_ideal_op** = {0, 0, 0, 0, { }, "ideal", 0, 0, 0, 0}

Constant *iqsdk::IqsCustomOp* object corresponding to an ideal operation.

struct **IqsCustomOp**

#include <quantum_full_state_simulator_backend.h> Simplified description of custom IQS operation.

Public Members

double **pre_dephasing** = 0

double **pre_depolarizing** = 0

double **pre_amplitude_damping** = 0

double **pre_bitflip** = 0

std::vector<*std*::complex<double>> **process_matrix** = { }

std::string **label** = ""

double **post_dephasing** = 0

double **post_depolarizing** = 0

double **post_amplitude_damping** = 0

double **post_bitflip** = 0

struct **IqsConfig** : public *iqsdk*::*DeviceConfig*

#include <quantum_full_state_simulator_backend.h> Configuration struct for IQS backend.

Users of the SDK can use an instance of this struct to create an IqsDevice. As part of the device, one can specify its noise model by explicitly defining the backend behavior corresponding to the various quantum macroinstructions (i.e. gates and SPAM operations).

Public Functions

virtual bool **isValid**()

Whether given config instance is valid.

IqsConfig(int num_qubits = 1, *std*::string simulation_type = "noiseless", bool verbose = false, *std*::size_t seed = time(NULL), bool synchronous = true, double depolarizing_rate = 0.01)

Constructor for IQS.

Public Members

int **num_qubits**

Number of qubits in simulation.

std::string **simulation_type**

Type of simulation to be run.

Valid simulation types are: “noiseless”, “depolarizing”, “custom”

double **depolarizing_rate**

Depolarizing rate for noisy simulation.

bool **use_custom_seed**

Whether custom seed for RNG is used.

std::size_t **seed**

Custom seed for RNG.

std::function<*IqsCustomOp*(unsigned)> **MeasZ** = `[](unsigned) {return k_iqs_ideal_op;}`

One-qubit measurement in the Z basis.

std::function<*IqsCustomOp*(unsigned)> **PrepZ** = `[](unsigned) {return k_iqs_ideal_op;}`

One-qubit state preparation in the Z basis (state $|0\rangle$)

std::function<*IqsCustomOp*(unsigned, double, double)> **RotationXY** = `[](unsigned, double, double) {return k_iqs_ideal_op;}`

Rotation in the XY plane, ideally described by the 2x2 matrix:

$$R_{XY}(\theta,\phi) = \begin{bmatrix} \cos(\theta/2) & -i\exp(-i\phi)\sin(\theta/2) \\ -i\exp(i\phi)\sin(\theta/2) & \cos(\theta/2) \end{bmatrix}$$

std::function<*IqsCustomOp*(unsigned, double)> **RotationZ** = `[](unsigned, double ) {return k_iqs_ideal_op;}`

Rotation around the Z axis, ideally described by the 2x2 matrix:

$$R_Z(\theta) = \begin{bmatrix} \cos(\theta/2) - i\sin(\theta/2) & 0 \\ 0 & \cos(\theta/2) + i\sin(\theta/2) \end{bmatrix}$$

std::function<*IqsCustomOp*(unsigned, unsigned, double)> **ISwapRotation** = `[](unsigned, unsigned, double) {return k_iqs_ideal_op;}`

SWAP-type gate, with arbitrary angle. Ideally it corresponds to the 4x4 matrix:

$$SWAP(\alpha) = \frac{1}{2}\begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 1+\exp(i\alpha) & 1-\exp(i\alpha) & 0 \\ 0 & 1-\exp(i\alpha) & 1+\exp(i\alpha) & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix}$$

Special cases are the common SWAP gate associated with $\alpha = 0$:

$$SWAP = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

And its square root for $\alpha = \pi/2$:

$$\sqrt{SWAP} = SWAP\left(\frac{\pi}{2}\right) = \frac{1}{2}\begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 1+i & 1-i & 0 \\ 0 & 1-i & 1+i & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix}$$

std::function<*IqsCustomOp*(unsigned, unsigned, double)> **CPhaseRotation** = `[](unsigned, unsigned, double) {return k_iqs_ideal_op;}`

CPHASE gate ideally corresponding to the 4x4 matrix:

$$CPHASE(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \exp(-i\theta) \end{bmatrix}$$

class **FullStateSimulator** : public *iqsdk*::*SimulatorDevice*

#include <quantum_full_state_simulator_backend.h> Class with API calls to both set up a quantum simulator device and access the underlying quantum state during simulation.

Note: re *FullStateSimulator::getSamples()*: Measurement with MeasX, MeasY, or MeasZ collapses the quantum state, so if a qubit that has recently been measured is included in `qids`, this will always return the same value that previously resulted from measurement

---

Note: Collecting data with IQS expects any qids provided to be from a previously-run quantum kernel

---

Public Functions

FullStateSimulator()

FullStateSimulator(*DeviceConfig* &device_config)

FullStateSimulator(const *FullStateSimulator*&) = delete

FullStateSimulator &**operator=**(const *FullStateSimulator*&) = delete

FullStateSimulator(*FullStateSimulator*&&) = delete

FullStateSimulator &**operator=**(*FullStateSimulator*&&) = delete

virtual **~FullStateSimulator**() = default

Private Functions

virtual int **getDeviceType**()

1.5 quantum_tensor_network_backend.h

namespace **iqsdk**

struct **TensorNetworkConfig** : public *iqsdk*::*DeviceConfig*

#include <quantum_tensor_network_backend.h>

Public Functions

TensorNetworkConfig(bool verbose = false, bool synchronous = true)
Constructor for Tensor Network Configuration.

class **TensorNetworkSimulator** : public *iqsdk*::*SimulatorDevice*

#include <quantum_tensor_network_backend.h>

Public Functions

TensorNetworkSimulator()

TensorNetworkSimulator(*iqsdk*::*DeviceConfig* &device_config)

TensorNetworkSimulator(const *TensorNetworkSimulator*&) = delete

TensorNetworkSimulator &**operator=**(const *TensorNetworkSimulator*&) = delete

TensorNetworkSimulator(*TensorNetworkSimulator*&&) = delete

TensorNetworkSimulator &**operator=**(*TensorNetworkSimulator*&&) = delete

~TensorNetworkSimulator() = default

void **draw**()

double **getExpectationValue**(*std*::vector<*std*::reference_wrapper<*qbit*>> &qids, *std*::string pauli_string)

void **setContractionPathOptimizer**(*std*::string optimizer_method)

Private Functions

virtual int **getDeviceType**()

CHAPTER

TWO

FUNCTIONAL LANGUAGE EXTENSION FOR QUANTUM (FLEQ)

2.1 datalist.h

DataList definition

Defines

LLVM_CLANG_QUANTUM_DATALIST_H

STRINGIFYDATA(D)

define_quantum_data(NAME, DATA)

import_with_name_begin(NAME)

import_with_name_end(NAME)

namespace **qlist**

Functions

template<class Type> Type *WRAP_ATTR IQC_alloca (const DataList &name, const unsigned long size)

struct **DataList**

#include <datalist.h>

Public Functions

inline WRAP_ATTR **DataList**(const char *d)

Create a *DataList* from a string.

Parameters

d –

Returns

DataList(const *DataList* &dl) = default

inline WRAP_ATTR **DataList**(int val)

inline char WRAP_ATTR operator[] (unsigned long i) const

Index into a *DataList*.

Parameters

i –

Returns

inline char WRAP_ATTR operator* () const

inline bool WRAP_ATTR empty () const

inline unsigned long WRAP_ATTR size () const

Return the size of the *DataList*.

Returns

inline const DataList WRAP_ATTR operator() (unsigned long a, unsigned long b) const

Produces a slice of a *DataList*, starting from index a and ending at index b-1.

Example: If d = "uvxyz" then d(2,4) = "xy".

Parameters

- **a** –
- **b** –

Returns

inline const DataList WRAP_ATTR operator() (unsigned long from, const DataList &to) const

inline const DataList WRAP_ATTR operator() (const DataList &from, unsigned long to) const

inline const DataList WRAP_ATTR operator() (const DataList &from, const DataList &to) const

inline const DataList WRAP_ATTR operator++ () const

inline const DataList WRAP_ATTR operator<< (const unsigned i) const

Shifts the *DataList* left by i. Example: "abc" << 1 = "ab".

Parameters

i –

Returns

inline const DataList WRAP_ATTR operator>> (const unsigned i) const

Shifts the *DataList* right by i. Example: "abc" >> 1 = "bc".

Parameters

i –

Returns

```
inline unsigned long WRAP_ATTR find (const DataList &d) const

inline unsigned long WRAP_ATTR find_last (const DataList &d) const

inline unsigned long WRAP_ATTR find_any (const DataList &d) const

inline unsigned long WRAP_ATTR find_any_last (const DataList &d) const

inline unsigned long WRAP_ATTR find_not (const DataList &d) const

inline unsigned long WRAP_ATTR find_not_last (const DataList &d) const

inline const DataList WRAP_ATTR next (const DataList &d) const

template<typename...
Args> inline const DataList WRAP_ATTR next (const DataList &d, Args... ds) const

inline const DataList WRAP_ATTR after_next (const DataList &d) const

template<typename...
Args> inline const DataList WRAP_ATTR after_next (const DataList &d, Args...
ds) const

inline const DataList WRAP_ATTR next_not (const DataList &d) const

inline const DataList WRAP_ATTR last (const DataList &d) const

template<typename...
Args> inline const DataList WRAP_ATTR last (const DataList &d, Args... ds) const

inline const DataList WRAP_ATTR after_last (const DataList &d) const

template<typename...
Args> inline const DataList WRAP_ATTR after_last (const DataList &d, Args...
ds) const

inline const DataList WRAP_ATTR last_not (const DataList &d) const

inline bool WRAP_ATTR contains (const DataList &d) const

inline unsigned long WRAP_ATTR count (const DataList &d) const

template<typename...
Args> inline unsigned long WRAP_ATTR count (const DataList &d, Args... ds) const
```

Public Static Functions

static inline const DataList WRAP_ATTR empty_list ()

Private Functions

inline WRAP_ATTR **operator const char***() const

Private Members

const char *const **_datalist**

Friends

inline friend int WRAP_ATTR _i (const DataList &a)

inline friend double WRAP_ATTR _d (const DataList &a)

inline friend bool WRAP_ATTR _b (const DataList &a)

inline friend unsigned long WRAP_ATTR size (const DataList &a)

inline friend const DataList WRAP_ATTR operator+ (const DataList &a, const DataList &b)

Concatenates two DataLists together. Example: “abc” + “xyz” = “abcxyz”.

Parameters

- **a** –
- **b** –

Returns

inline friend const friend bool WRAP_ATTR operator== (const DataList &a, const DataList &b)

inline friend std::string WRAP_ATTR to_string (const DataList &a)

inline friend const char *WRAP_ATTR to_char_array (const DataList &a)

template<class Type> inline friend Type *WRAP_ATTR IQC_alloca (const DataList &name, const unsigned long size)

friend QExpr **exitAtCompile**(*DataList* d)

friend QExpr **exitAtRuntime**(*DataList* d)

friend QExpr **printDataList**(*DataList* d, QExpr e)

namespace **qexpr**

2.2 qexpr.h

Basic set of functions which return QExpr's

Defines

LLVM_CLANG_QEXPR_H

this_as_expr

qexpr wrappers around builtins

namespace **qexpr**

Functions

QExpr **exitAtCompile**(*qlist*::*DataList* d = *qlist*::*DataList*::empty_list())

Raise a compile-time error.

Parameters

d –

Returns

QExpr **exitAtRuntime**(*qlist*::*DataList* d = *qlist*::*DataList*::empty_list())

Raise a runtime error.

Parameters

d –

Returns

QExpr **printDataList**(*qlist*::*DataList* d, QExpr e)

Print the DataList at compile-time and return e.

Parameters

- **d** –
- **e** –

Returns

QExpr WRAP_ATTR identity ()

QExpr WRAP_ATTR global_phase (double angle)

template<auto *F, class ...Args>
QExpr **convert**(*Args*&&... args)

```
void WRAP_ATTR eval_hold (QExpr e)
```

QExpr evaluation methods.

```
void WRAP_ATTR eval_release (QExpr e)

QExpr WRAP_ATTR _H (qbit &q)

QExpr WRAP_ATTR _X (qbit &q)

QExpr WRAP_ATTR _Y (qbit &q)

QExpr WRAP_ATTR _Z (qbit &q)

QExpr WRAP_ATTR _S (qbit &q)

QExpr WRAP_ATTR _Sdag (qbit &q)

QExpr WRAP_ATTR _T (qbit &q)

QExpr WRAP_ATTR _Tdag (qbit &q)

QExpr WRAP_ATTR _RX (qbit &q, double angle)

QExpr WRAP_ATTR _RY (qbit &q, double angle)

QExpr WRAP_ATTR _RZ (qbit &q, double angle)

QExpr WRAP_ATTR _CZ (qbit &ctrl, qbit &tgt)

QExpr WRAP_ATTR _CNOT (qbit &ctrl, qbit &tgt)

QExpr WRAP_ATTR _SWAP (qbit &q1, qbit &q2)

QExpr WRAP_ATTR _Toffoli (qbit &q1, qbit &q2, qbit &q3)

QExpr WRAP_ATTR _PrepX (qbit &q)

QExpr WRAP_ATTR _PrepY (qbit &q)

QExpr WRAP_ATTR _PrepZ (qbit &q)

QExpr WRAP_ATTR _MeasX (qbit &q, cbit &c)
```

`QExpr WRAP_ATTR _MeasY (qbit &q, cbit &c)`

`QExpr WRAP_ATTR _MeasZ (qbit &q, cbit &c)`

`QExpr WRAP_ATTR _CPhase (qbit &ctrl, qbit &tgt, double angle)`

`QExpr WRAP_ATTR _RXY (qbit &q, double theta, double phi)`

`QExpr WRAP_ATTR _SwapA (qbit &q1, qbit &q2, double angle)`

`QExpr WRAP_ATTR join (QExpr e1, QExpr e2)`

QExpr coherent-logic ///.

Concatenate two QExprs in composition order. Example: join(_H(q), _PrepZ(q)) is equivalent to the circuit |q> — PrepZ — H —

Parameters

- `e1` –
- `e2` –

Returns

`QExpr WRAP_ATTR bind (QExpr e1, QExpr e2)`

Concatenate two QExprs in composition order, with a circuit barrier. Measurement outcomes in e2 will be available to e1. Will not optimize across the barrier.

Parameters

- `e1` –
- `e2` –

Returns

`QExpr WRAP_ATTR invert (QExpr e)`

Implement the inverse of the QExpr e.

Parameters
`e` – A unitary QExpr.

Returns

`QExpr WRAP_ATTR power (unsigned int n, QExpr e)`

Concatenate e with itself n times. Assumes n is a constant resolvable at compile-time. Example: power(3,e) = join(e, join(e, e))

Parameters

- `n` –
- `e` –

Returns

`QExpr WRAP_ATTR control (qbit &ctrl, QExpr e)`

Control e by the qubit ctrl. Assumes e is unitary. Example: control(ctrl, _H(q)) implements.

|ctrl> — o — | |q> — H —

Parameters

- **ctrl** – A qubit that does not appear in e.
- **e** – A Unitary QExpr.

Returns

`QExpr WRAP_ATTR control (qbit &ctrl, bool control_if, QExpr e)`

Control the unitary e by ctrl = |control_if>. Example: control(ctrl, false, _H(q)) implements.

|ctrl> — X — o — X — | |q> ——— H ———

Parameters

- **ctrl** – A qubit that does not appear in e.
- **control_if** –
- **e** – A unitary QExpr.

Returns

`QExpr WRAP_ATTR control (qbit ctrls[], QExpr e)`

Control e by the entire qubit array ctrls. Example: if ctrls = {a,b} then control(ctrls, _X(q)) implements.

|a> — o — | |b> — o — | |q> — X —

Parameters

- **ctrls** – A qubit array whose elements do not appear in e.
- **e** – A unitary QExpr.

Returns

`QExpr WRAP_ATTR control (qbit ctrls[], unsigned int control_on, QExpr e)`

Control the unitary e by ctrls = |control_on>, where control_on is an index corresponding to a basis state. (See QssIndex for details of the index encoding.) Example: If ctrls = {a,b} and control_on = QssIndex("|01>").getIndex() then control(ctrls, control_on, _X(q)) implements.

|a> — X — o — X — | |b> ——— o ——— | |q> ——— X ———

Parameters

- **ctrls** – A qubit array that whose elements do not appear in e
- **control_on** –
- **e** – A unitary QExpr.

Returns

`QExpr WRAP_ATTR control (const qlist::QList &ctrls, QExpr e)`

`QExpr WRAP_ATTR control (const qlist::QList &ctrls, unsigned int control_on, QExpr e)`

`QExpr WRAP_ATTR qIf (qbit &ctrl, QExpr eT, QExpr eF)`

Control eT by ctrl=true and eF by ctrl=false.

Parameters

- `ctrl` –
- `eT` – A unitary QExpr.
- `eF` – A unitary QExpr.

Returns

`QExpr WRAP_ATTR cIf (bool b, QExpr eTrue, QExpr eFalse)`

QExpr classical control ///.

If b=true, execute eTrue, otherwise execute eFalse.

Parameters

- `b` – A runtime boolean value.
- `eTrue` –
- `eFalse` –

Returns

`QExpr WRAP_ATTR cIfTrue (bool b, QExpr eTrue)`

If b=true, execute eTrue, otherwise do nothing.

Parameters

- `b` – A runtime boolean value.
- `eTrue` –

Returns

`QExpr WRAP_ATTR cIfFalse (bool b, QExpr eFalse)`

If b=false, execute eFalse, otherwise do nothing.

Parameters

- `b` – A runtime boolean value.
- `eTrue` –

Returns

`void WRAP_ATTR let (const char key[], QExpr e)`

QExpr build control utilities ///.

Assign the QExpr e to a string key.

Parameters

- `key` –

- **e** –

Returns

`QExpr WRAP_ATTR get (const char key[])`

Return the QExpr previously assigned to the key.

Parameters
key –

Returns

`QExpr WRAP_ATTR fence (QExpr e)`

Implement a barrier/fence around the QExpr e.

Parameters
e –

Returns

`QExpr WRAP_ATTR printQuantumLogic (QExpr e)`

Compile-time printing ///.

Print out the PCOAST graph(s) generated by e at compile-time, and return e.

Parameters
e –

Returns

template<auto *F>

struct **Converter**

#include <qexpr.h>

template<class ...Args>

struct **Converter**<F>

#include <qexpr.h>

Public Functions

`inline PROTECT QExpr convert (Args... args) const`

2.3 qexpr_utils.h

Utilities for working with quantum kernel expressions.

namespace **qexpr**

Functions

template<typename QExprFun, typename... Args> PROTECT QExpr map (QExprFun f, qlist::QList qs, Args... args) noexcept

Map a QExpr function over each element in a QList.

The input function f should take in a qubit and some additional arguments, and returns a QExpr. In the simplest case, where no additional arguments are given, map(f,qs) will expand to f(qs[0]) + f(qs[1]) + ... + f(qs[size(qs)-1])

The additional arguments to map are either array arguments or scalar arguments.

Array arguments are either QList or pointer types T*; map will map the function f over each element of the array type. map expects the size of all array arguments to be at least as large as the initial QList argument qs; any additional elements of the array arguments will be ignored.

All other types of arguments are treated as scalar arguments, and are passed to each invocation of f.

As an example, suppose we have (1) a QList qs; (2) an int x; (3) a QList rs; and (4) a boolean array bs. In a call to map, x will be treated as a scalar argument and rs and bs as array arguments. Let f be a function with signature QExpr f(qbit q, int x, qbit r, bool b); Then map(f,qs,x,rs, bs) will expand to f(qs[0], x, rs[0], bs[0]) + ... + f(qs[n-1], x, rs[n-1], bs[n-1]) where n = qs.size().

Template Parameters

- **QExprFun** – The type of f
- **...Args** – The type of the argument list

Parameters

- **f** – A function of type QExpr f(qbit q, ...)
- **qs** – A QList
- **...args** – A collection of array and scalar arguments.

Returns

The QExpr obtained by mapping f over all qubits in qs

template<typename QExprFun, typename... Args> PROTECT QExpr map1 (QExprFun f, qlist::QList qs, Args... args) noexcept

mapN(f,qs,...) maps a QExpr function f over arrays arguments, the first of which must be a QList. Additional arguments are treated as scalar arguments, and passed to f directly.

Example (map1): Let f be a function with signature QExpr f(qubit& q, int x); If qs = {a,b,c} is a QList and n is an int, then map1(f, qs, n) implements f(a,n) + f(b,n) + f(c,n)

Example (map2): Let f be a function with signature QExpr f(qbit& q, qbit& r, qbit& s); Let qs = {q1, q2, q3} and rs = {r1, r2, r3} be QLists, and let s be a single qubit. Then map2(f, qs, rs, s) will expand to f(q1, r1, s) + f(q2, r2, s) + f(q3, r3, s). We see that map2(f, qs, rs, s) treats s as a scalar argument, since it is the third argument after f; map2 only maps the function over the first two arguments.

template<typename QExprFun, typename ArrayArg, typename... Args> PROTECT QExpr map2 (QExprFun f, qlist::QList qs, ArrayArg xs, Args... args) noexcept

template<typename QExprFun, typename ArrayArg1, typename ArrayArg2, typename... Args> PROTECT QExpr map3 (QExprFun f, qlist::QList qs, ArrayArg1 xs, ArrayArg2 ys, Args... args) noexcept

template<typename QExprFun, typename ArrayArg1, typename ArrayArg2, typename ArrayArg3, typename... Args> PROTECT QExpr map4 (QExprFun f, qlist::QList qs, ArrayArg1 xs, ArrayArg2 ys, ArrayArg3 zs, Args... args) noexcept

template<typename QExprFun, typename... Args> PROTECT QExpr mapWithIndex (QExprFun f, qlist::QList qs, Args... args)

Map over a `QList` of qubits with a function that takes both a qubit and the index at which it appears in the overall `QList`.

Example: Let `f` be a function that takes a qubit and an integer, and returns a `QExpr`. Let `qs = {a,b,c}` be a `QList`. Then `mapWithIndex(qk, qs)` implements `qk(a,0) + qk(b,1) + qk(c,2)`

Parameters

- **f** – A `QExpr` function that takes a qubit `q`[i], the index `i`, and optionally additional arguments
- **qs** – The `QList` to map over

template<typename QExprFun, typename QExprReturn, typename... Args> PROTECT QExprReturn fold (QExprFun k, qlist::QList qs, QExprReturn base, Args... args)

Recursively fold a function over a list of qubits.

Recursively performs a right fold (in the functional programming sense) over a `QList` of qubits. Example: `fold(k, {q0, q1, q2}, b) = k(q0, k(q1, k(q2, b)))`

Template Parameters

- **QExprFun** – The type of `k`
- **QExprReturn** – The return type of the fold. Must either be `QExpr` or a type that contains a `QExpr`, such as a pair or tuple type where one component is `QExpr`

Parameters

- **k** – A function that takes a qubit and an argument of type `QExprReturn`, and produces a new `QExprReturn`
- **qs** – A `QList` of qubits
- **base** – A value of type `QExprReturn`, returned when `qs` is empty
- **qargs...** – A list of additional arguments to `k`

Returns

QExpr **conjugate**(QExpr e1, QExpr e2)

Conjugate `e2` by `e1`; i.e. return the `QExpr` `invert(e1) + e2 + e1`

Parameters

- **e1** – A unitary `QExpr`.
- **e2** – A `QExpr`. Need not be unitary.

template<typename QExprFun, typename... Args> PROTECT QExpr mapDataList (QExprFun f, datalist::DataList sep, datalist::DataList d, Args... args)

Map a QExpr function over a DataList with entries separated by sep.

For example, if sep is ";" and d is "d1; ... ; dn", will return the QExpr f(d1,args...) + ... + f(dn, args...).

Parameters

- **f** – A function that takes as an argument (1) a DataList; (2) some number of additional arguments args...
- **sep** – A single-character DataList that separates d into pieces to be passed to f.
- **d** – The DataList to be mapped over
- **...args** – Additional scalar arguments to be passed directly to every invocation of f.

Returns

template<typename QExprFun, typename... Args> PROTECT QExpr mapDataList (datalist::DataList startToken, datalist::DataList endToken, QExprFun f, datalist::DataList sep, datalist::DataList d, Args... args)

Map a QExpr function over a DataList representing a list with entries separated by the token sep, and optionally starting with the token startToken and ending with the token endToken.

For example, mapDataList("{", "}", f, ";", "{d1; d2; d3}", arg), will return the QExpr f(d1,arg) + f(d2,arg) + f(d3, arg).

If startToken and/or endToken are not found, will still parse.

const QExpr **qassert**(const bool b, const datalist::DataList error)

Raise a compile-time error if the boolean input is false.

Parameters

- **b** – A boolean that is compile-time resolvable
- **error** – Error message

2.4 qlist.h

QList definition

Defines

GET_LISTABLE(_1, _2, NAME, ...)

listable(...)

listable_array(NAME, NUM)

listable_scalar(NAME)

namespace **qlist**

Functions

const *QList* **join**(*qbit**, *qbit**)

const *QList* **slice**(*qbit**, unsigned long, unsigned long)

const *QList* **join**(const *QList*&, const *QList*&)

Parameters

- **reg1** –
- **reg2** –

Returns

const *QList* **slice**(const *QList*&, unsigned long, unsigned long)

Example: If qs = {a,b,c,d,e} then slice(qs,2,4) = {c,d}.

Parameters

- **reg** –
- **a** –
- **b** –

Returns

unsigned long WRAP_ATTR size (qbit *reg)

bool WRAP_ATTR qbits_equal (qbit &q1, qbit &q2)

struct **QList**

#include <qlist.h>

Public Functions

QList() = default

Empty *QList*.

inline WRAP_ATTR **QList**(*qbit* *qa)

Construct a *QList* from a qubit array.

Parameters

qa –

Returns

inline WRAP_ATTR **QList**(*qbit* &q)

Construct a *QList* consisting of a single qubit.

Parameters

q –

Returns

QList(const *QList* &ql) = default

inline qbit WRAP_ATTR & operator[] (unsigned long i) const

Index into the qlist at index i. Example: if qs={a,b,c} then qs[1] = b.

Parameters

i –

Returns

inline WRAP_ATTR qbit & operator* () const

inline unsigned long WRAP_ATTR size () const

inline qbit WRAP_ATTR & at (unsigned long i) const

inline const QList WRAP_ATTR operator() (unsigned long a, unsigned long b) const

Produces a slice of a *QList*, starting from index a and ending at index b-1.

Example: If qs = {u,v,x,y,z} then qs(2,4) = {x,y}.

Parameters

- **a** –
- **b** –

Returns

inline const QList WRAP_ATTR operator+ (unsigned i)

Increments the qlist by i. In other words, returns the slice qs(i,*qs.size()*)

Parameters

i –

Returns

inline const QList WRAP_ATTR operator++ () const

Increments the *QList* by 1.

Returns

Public Static Functions

static inline const QList WRAP_ATTR empty_list ()

Returns an empty *QList*.

Private Functions

inline explicit **operator** ***qbit******()** const

inline explicit **operator const** ***qbit******()** const

Private Members

qbit *_qlist

Friends

inline friend unsigned long WRAP_ATTR size (const QList &a)

inline friend const QList WRAP_ATTR operator+ (const QList &a, const QList &b)

Concatenate two QLists in sequence. For example, {a,b,c} + {x,y} = {a,b,c,x,y}.

Parameters

- a –
- b –

Returns

inline friend const QList WRAP_ATTR operator<< (const QList &a, const unsigned long i)

Returns the first *size()*-i elements of a. Examples: If qs = {a,b,c} then qs << 1 = {a,b}.

Parameters

- a –
- i –

Returns

inline friend const QList WRAP_ATTR operator>> (const QList &a, const unsigned long i)

Returns the last *size()*-i elements of a Example: If qs = {a,b,c} then qs >> q = {b,c}.

Parameters

- a –
- i –

Returns

inline friend const QList WRAP_ATTR slice (qbit *reg, const unsigned long a, const unsigned long b)

inline friend const QList WRAP_ATTR slice (qbit *reg, const int a, const int b)

inline friend const QList WRAP_ATTR join (qbit *reg1, qbit *reg2)

inline friend const QList WRAP_ATTR slice (const QList &reg, const unsigned long a, const unsigned long b)

Produces a slice of a *QList*, starting from index a and ending at index b-1.

Example: If qs = {a,b,c,d,e} then slice(qs,2,4) = {c,d}.

Parameters

- reg –
- a –
- b –

Returns

inline friend const QList WRAP_ATTR join (const QList &reg1, const QList &reg2)

Concatenate two QLists in sequence. For example, join({a,b,c}, {x,y}) = {a,b,c,x,y}.

Parameters

- **reg1** –
- **reg2** –

Returns

friend QExpr **control**(const *QList*&, unsigned int, QExpr)

friend QExpr **control**(const *QList*&, QExpr)

namespace **qexpr**

CHAPTER

THREE

QUANTUM RUNTIME UTILITIES

3.1 qrt_errors.hpp

Typedefs

using **qbit** = unsigned short int

using **cbit** = bool

namespace **iqsdk**

Enums

enum **QRT_ERROR_T**

Success/Failure return codes used in the library.

These are returned when interacting with quantum devices.

Values:

enumerator **QRT_ERROR_SUCCESS**

enumerator **QRT_ERROR_WARNING**

enumerator **QRT_ERROR_FAIL**

Functions

void **printIqsdkErrorMsg**(*std*::string msg)

Print error message for the Intel QSDK to screen.

The message is of the form: ERROR: Quantum SDK - <details of the error message>

void **printIqsdkWarningMsg**(*std*::string msg)

Print warning message for the Intel QSDK to screen.

The message is of the form: WARNING: Quantum SDK - <details of the warning message>

3.2 qrt_indexing.hpp

This file defines a class for abstracting away indices into a state space.

A collection of n qubits has 2^n basis states. We use length 2^n vectors to represent a variety of simulator state spaces and probability vectors. The Quantum State Space Index (QssIndex) class standardizes indexing into such a state vector, and provides features for easily working with indices.

namespace **iqsdk**

Typedefs

template<class **T**>

using **QssMap** = *std*::map<*QssIndex*, *T*>

Use this alias to get a map indexed by *QssIndex* with appropriate hashing.

A wrapper around a C++ map indexed by a *QssIndex*, to double or complex. Returned by `getProbabilities()` and `getAmplitudes()`. Useful for holding a state space or partial state space of probabilities (if double) or amplitudes (if complex).

Functions

template<class **T**>
std::vector<*T*> **qssMapToVector**(*QssMap*<*T*> &map, size_t num_qubits, *T* default_val)

Populate the vector with the elements in map, filling in the default value to any indices not in the map.

template<class **T**>
QssMap<*T*> **qssVectorToMap**(*std*::vector<*T*> &vec, size_t num_qubits)

Populate the map with elements of the vector.

class **QssIndex**

#include <qrt_indexing.hpp> Indexes into state spaces returned by *SimulatorDevice::getProbabilities()* and *SimulatorDevice::getAmplitudes()*

QssIndex stands for Quantum State Space Index. A state space can be defined with either a `std::vector` or a `QssMap`.

Public Functions

QssIndex()

QssIndex(size_t num_qubits, size_t idx)

Create a state from an integer representation idx.

Parameters

idx – Integer representation of the state, counting from the right (little-endian)

QssIndex(*std*::vector<bool> basis)

Create a state from a boolean vector representing a state.

```
QssIndex({true, false}).toStringWithoutKet() == "10"
```

QssIndex(*std*::string basis)

Useful for specifying a state when you want to request data from a large state-space.

Parameters

basis – The input string `basis` has the form `|b0...bn>` or `b0...bn` where each `b` is either 0 or 1 e.g. "|01101>" or "100".

size_t **getIndex**() const

Return the underlying index.

size_t **getNumQubits**() const

The number of qubits handled by this state space.

std::vector<bool> **getBasis**() const

The equivalent binary basis for this *QssIndex*.

std::string **toString**() const

Convert `idx` to a binary representation using `num_bits` bits, printed in reverse order.

This relates the index `idx` into a probability or amplitude vector to its corresponding basis element.

std::string **toStringWithoutKet**() const

inline **operator size_t**() const

bool **bitAt**(size_t i) const

Return the value b_i, where `b_i` corresponds to the basis element `|b0..bn>`.

void **setBitAt**(size_t i, bool b)

If `i` corresponds to the basis element `bi` in `|b0..bi..bn>`, changes `bi` to correspond to `|b0..b..bn>`

bool **operator==**(const *QssIndex* &idx) const

bool **operator!=**(const *QssIndex* &idx) const

bool **operator<**(const *QssIndex* &idx) const

Inequality operator is based on the ordering of the basis. A *QssIndex* is less than another if it has a fewer number of qubits, or if the binary representation of the basis is less than the other *QssIndex*.

Public Static Functions

static *std*::vector<*QssIndex*> **patternToIndices**(*std*::string pattern)

Given a pattern, produce a vector of *QssIndex* objects obtained by replacing `X`, `?`, or `Wildcard` in the pattern by every combination of `0` and `1`.

The patterns `|X1X>`, `?1?` and `{Wildcard, 1, Wildcard}` all represent the same set of indices: `{|010>, |110>, |011>, |111>}`.

static *std*::vector<*QssIndex*> **patternToIndices**(*std*::vector<int> pattern)

Given a pattern of a `std::vector` of `ints` returns a set of *QssIndex* consistent with the pattern. Similar to *patternToIndices(std::string)*, integers other than 0 or 1 will be interpreted as a wildcard.

Public Static Attributes

static const int **Wildcard**

Private Members

std::vector<bool> **basis**

Friends

inline friend *std*::ostream &**operator<<**(*std*::ostream &output, const *QssIndex* &idx)

namespace **std**

INDEX

L

Q

S

T