QPlayer User's Manual

Version 1.0-Leopard, July '23

ETRI 한국전자통신연구원
Electronics and Telecommunications Research Institute

Lightweight, Scalable, and Fast Quantum Simulator

CONTENTS

CHAPTER 1 Introduction to QPlayer

In this chapter...

- What is QPlayer?
- User Interfaces
- Version History
- Terms
- License
- Contact Information

QPlayer is a quantum simulation SW developed by the Electronics and Telecommunications Research Institute (ETRI). QPlayer provides a new method for simulating quantum circuits using classical computers. QPlayer enables faster simulations of quantum circuits of more qubits with fewer memory resources through unique quantum state management that differs from existing quantum simulators.

This chapter introduces the key concepts and interfaces of QPlayer, development history by version, and licensing policies.

What is QPlayer?

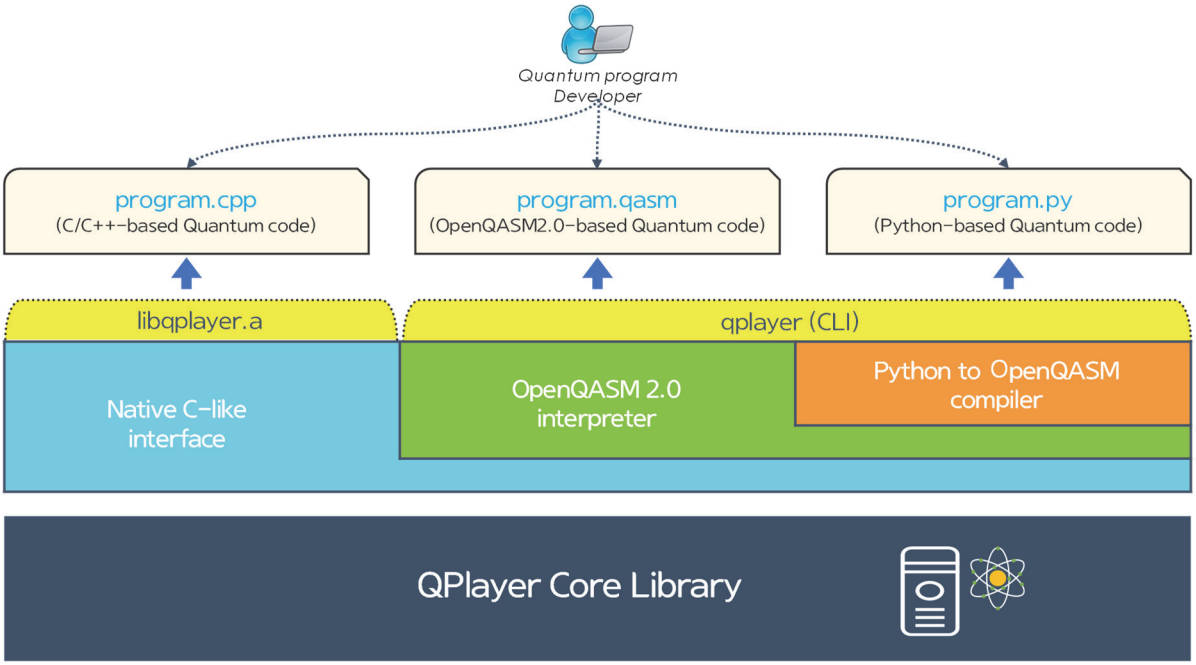
With the rapid development of quantum computing, classical quantum simulation is considered essential for quantum algorithm verification, quantum error analysis, and novel quantum applications. However, the exponential increase in memory overhead and quantum computation time remains a critical problem that has not been solved for many years. QPlayer provides a novel approach that provides more qubits and faster quantum operations with smaller memory than before. Instead of storing the entire quantum state in memory, QPlayer uses a reduced quantum state representation scheme to selectively manage quantum states with physical reality. This approach has the advantage of ensuring fast quantum computation with little memory space without compromising the global quantum state. In addition, experimental results using QPlayer showed that various quantum algorithms are overcoming the limitations of existing quantum simulators. The Grover algorithm supports simulations of up to 55 qubits, and we experimentally verify that the Surface Code algorithm can be simulated with up to 85 qubits in 512GB memory of a single computing node.

SEE ALSO:

"QPlayer: Lightweight, scalable, and fast quantum simulator", ETRI Journal, 2022

"Multilayered logical qubits and synthesized quantum bits", Quantum Science and Technology, 2023

User Interfaces



QPlayer is a software that runs on classical computer with Linux operating systems (CentOS 7.x and later,

Ubuntu 20.x and later). The QPlayer consists of a core library for quantum simulation and interfaces for users to develop their quantum applications. The types of programming interfaces and their characteristics are shown in the table below.

type	Native C-like	OpenQASM 2.0	Python
target user	IT developer	none - IT developer	IT developer
required programming skills	high	low	medium
compiling quantum code	required	N/A	N/A
execution type	Run Compiled Binary	Using QPlayer CLI	Using QPlayer CLI
ease of quantum code development	High (support c/c++ rules)	Low (simple function and limited if statement)	High (support python rules)
version	v0.2-Cat or later	v1.0-Leopard or later	under development

Version History

v0.2-Cat

- release: June 24, 2021
- features

  - Schrödinger style state vector simulator

  - Fast operation in the reduced Hilbert space

  - User-friendly interfaces: native-C

  - Support for 21 quantum gate operations

v0.5-Jaguar

- release: March 30, 2023
- features

  - optimize matrix calculation logic

  - fix memory overflow in log-run testing

  - enhance gate operation performance

  - OpenQASM v2.0 /w a simple grammar check

  - Support for 32 quantum gate operations

v1.0-Leopard

- release: June 23, 2023
- features

  - re-organize whole directory trees & sources

  - OpenQASM v2.0 /w a complete grammar check

  - add state memory pool for handling dynamical state memory

  - provide simulation statistics through the json file

Terms

The terms used in this document are shown in the table below.

qubit	The basic unit of quantum information processing that utilizes quantum mechanical phenomena of superposition and entanglement.
superposition	The fundamental principle of quantum mechanics in which one qubit is represented by a stochastic linear combination of eigenstates
entanglement	The phenomenon that occurs when a group of particles are generated, interact, or share spatial proximity in a way such that the quantum state of each particle of the group cannot be described independently of the state of the others, including when the particles are separated by a large distance.
measurement	A quantum operation in which the superposition state of a qubit collapses into one particular eigenstate that makes up the qubit.
quantum simulator	A quantum simulator is a software tool for simulating and simulating the behavior of quantum computers and the properties of quantum systems. Quantum simulators simulate the behavior of quantum circuits and quantum gates with classical computers, and track the evolution of quantum states by early quantum states and quantum computations
quantum gate	It refers to a quantum operation applied to the state of the qubit, and there are 1-qubit gate, 2-qubit gate, etc. depending on the number of qubits to which the gate is applied.
quantum circuit	A computational model consisting of a series of quantum gate operations designed for quantum computation in quantum computers and quantum systems

License

This software adheres to the GNU General Public License v3 licensing policy. It is important to familiarize yourself with the terms and conditions of this license, as outlined in the accompanying manual. By using this software, you are granted certain rights and are obligated to comply with the license requirements, including ensuring that any modifications or derivative works are also distributed under the GPL v3 license. Failure to adhere to these conditions may result in legal consequences. Please carefully read the license description (https://github.com/eQuantumOS/QPlayer/blob/master/LICENSE) for further information and instructions on how to appropriately utilize and distribute this software.

Contact Information

The QPlayer development team is always committed to making its best efforts to accommodate various requirements from quantum researchers. If you have any questions regarding functional extension, bug reports, collaborative research, or any other issues, please feel free to contact us through the following channels:

- Github: https://github.com/eQuantumOS/QPlayer.git
- Email: ksjin@etri.re.kr or gicha@etri.re.kr
- Phone: +82-042-860-1535 (1259)

CHAPTER 2 Software Installation

In this chapter...

- Running Environment
- Software Installation
- Verification

QPlayer is a SW that runs in a Linux environment. Anyone is free to download, install, and operate the source code to the extent that it complies with the GPL v3 policy, a public software license.

This chapter describes the source code structure, installation method, and post-installation verification method of the QPlayer.

Running Environment

Operating System

QPlayer runs on CentOS 7.x and later and Ubuntu 20.x and later.

HW requirements

QPlayer was developed to operate in a single server system, and the hardware restrictions for operating it are shown in the table below.

type	requirement	description
CPU	no restrictions	Faster CPUs improve simulation performance
GPGPU	do not support	Performance acceleration using GPGPU will be supported from v2.0-Puma and later versions
Memory	1 GB or more	QPlayer has the characteristic of dynamically expanding or shrinking memory according to quantum state evolution. The larger the memory, the more qubits can be simulated. We recommend at least 1GB of memory for quantum simulations within 30 qubits.
Storage	no restrictions	
Network	no restrictions	

Software Installation

Step1: install boost library

QPlayer uses boost library, an open SW, to represent more than 64 qubits. You can also download and install the boost library source code directly, but it is recommended to install it automatically through a standard SW repository as follows.

- (CentOS) yum install boost-devel
- (Ubuntu) apt-get install libboost-all-dev

Step2: download QPlayer source code

Download the latest source code for QPlayer from the github repository as shown below.

```
≫ git clone https://github.com/eQuantumOS/QPlayer.git
≫ ls -l QPlayer
  drwxr-xr-x 4 root root  4096 7월  3 10:41 core/
  drwxr-xr-x 2 root root  4096 7월  3 10:39 docs/
  drwxr-xr-x 5 root root  4096 7월  5 13:31 qasm/
  drwxr-xr-x 5 root root  4096 6월 23 13:55 test/
  -rw-r--r-- 1 root root   452 6월 20 09:22 AUTHOR
  -rw-r--r-- 1 root root   795 6월 20 09:22 CONTRIBUTION
  -rw-r--r-- 1 root root 35149 6월 20 09:22 LICENSE
  -rw-r--r-- 1 root root   911 6월 23 13:55 Makefile
  -rw-r--r-- 1 root root  2996 7월  5 13:31 README.md
```

Step3: Compile QPlayer source code

Compile QPlayer source code as follows.

```
≫ cd QPlayer
≫ make
≫ ls -l
  drwxr-xr-x 4 root root  4096 7월  3 10:41 core/
  drwxr-xr-x 2 root root  4096 7월  3 10:39 docs/
  drwxr-xr-x 5 root root  4096 7월  5 13:31 qasm/
  drwxr-xr-x 5 root root  4096 6월 23 13:55 test/
  drwxr-xr-x 5 root root  4096 6월 23 13:55 release/
  -rw-r--r-- 1 root root   452 6월 20 09:22 AUTHOR
  -rw-r--r-- 1 root root   795 6월 20 09:22 CONTRIBUTION
  -rw-r--r-- 1 root root 35149 6월 20 09:22 LICENSE
  -rw-r--r-- 1 root root   911 6월 23 13:55 Makefile
  -rw-r--r-- 1 root root  2996 7월  5 13:31 README.md
≫ ls -l release
  drwxr-xr-x 5 root root  4096 7월  5 13:31 bin/
  drwxr-xr-x 5 root root  4096 7월  5 13:31 include/
  drwxr-xr-x 5 root root  4096 7월  5 13:31 lib/
```

When the compilation is complete, a new 'release' directory is created, including the bin, include, and lib directories. Among the subdirectories, the release/bin directory stores CLI files for executing OpenQASM 2.0 or python code, and the release/include, release/lib stores libraries and header files that can perform quantum programming using native-C.

Verification

Run the test below to verify that all installation processes have completed successfully. When the following result message is displayed on the console, it means that the basic functions operates normally.

```
≫ make TEST
≫ cd test/general
≫ ./install_test

[P=0.250000] [+0.500000, +0.000000] |0000000000100000>
[P=0.250000] [+0.500000, +0.000000] |0000000000111110>
[P=0.250000] [-0.500000, +0.000000] |0000001111100000>
[P=0.250000] [-0.500000, +0.000000] |0000001111111110>
======== dump quantum states(4) ========
```

CHAPTER 3 CLI for OpenQASM (and Python)

In this chapter...

- CLI Specification
- Usage Scenarios
- Exceptions

QPlayer provides a 'QPlayer' CLI command, a dedicated execution binary, to execute quantum programs written on OpenQASM 2.0 or Python. This command can be executed directly in the Linux console or used to link with a web-based GUI tool.

This chapter describes the detailed specifications, usage scenarios, and exception handling rules of the QPlayer command.

CLI Specification

A dedicated CLI, 'QPlayer', enables the execution of quantum code written in OpenQASM 2.0. This CLI can be found in the ~release/bin directory after QPlayer's compilation is completed, and the specifications for use are shown in the table below.

option	argument	description	type
-i	path of input QASM file	Quantum code file written in OpenQASM 2.0	mandatory
-o	path of output file	Simulation result output file	mandatory
-j	path of json file	The execution information is generated as a json file.	optional
-s	number	the number of shots (default: 1)	optional
-h	N/A	help message	optional
--version	N/A	QPlayer version message	optional
--verbose	N/A	display execution information on the console	optional

※ We do not support the python interface in the v2.0-Leopard version. Some of the options may be changed for the python interface in a future development version.

Usage Scenarios

UseCase1: check QPlayer version

The version information of the distributed QPlayer can be checked as follows.

```
≫ cd ~/release/bin
≫ ./QPlayer --version

QPlayer v-1.0-Leopard
```

UseCase2: run OpenQASM 2.0 code

The following is an example of running a quantum circuit written in OpenQASM 2.0. After compiling the QPlayer, you can try to run it in the same way as the below in the release/bin directory generated.

```
≫ cd ~/release/bin
≫ ./QPlayer -i examples/test.qasm -o log/simulation.res
≫ cat log/simulation.res

Total Shots: 1
Measured States: 1

100%, 1/1, |1011>
```

UseCase3: run OpenQASM 2.0 code, repeatedly

UseCase #2 is to execute a given code only once. If you want to execute the quantum code repeatedly, you can use it as follows. In this case, the measurement result may vary for each individual execution, and the probability distribution may be checked through the result log file.

```
≫ ./QPlayer -i examples/test.qasm -o ./log/simulation.res -s 10
≫ cat log/simulation.res

Total Shots: 10
Measured States: 8

10%, 1/10, |0010>
10%, 1/10, |0011>
20%, 2/10, |0110>
10%, 1/10, |0111>
10%, 1/10, |1010>
10%, 1/10, |1011>
20%, 2/10, |1110>
10%, 1/10, |1111>
```

UseCase4: generate JSON file which contains simulation execution status

QPlayer provides the execution information in a separate JSON file. The categories of output information consist of 1. execution quantum circuit information, 2. execution time, 3. execution work, 4. system information, etc.

```
≫ ./QPlayer -i examples/test.qasm -o ./log/simulation.res -j ./log/task.json
```

SEE ALSO: JSON Specification

```
{
        "circuit" : {
                "used qubits" : Number,
                "used gates" : Number,
                "gate calls" : {
                        "Gate" : Number,
```

```
                        "Gate" : Number,
                        "Gate" : Number,
                    }
        },
        "runtime" : {
                "total simulation time" : Number,
                "individual gate time" : {
                        "Gate" : [total(Number), max(Number), min(Number), avg(Number)],
                        "Gate" : [total(Number), max(Number), min(Number), avg(Number)],
                        "Gate" : [total(Number), max(Number), min(Number), avg(Number)]
                }
        },
        "simulation jobs" : {
                "max states" : Number,
                "final states" : Number,
                "used memory" : String
        },
        "system" : {
                "OS" : {
                        "name" : String,
                        "version" : String
                },
                "CPU" : {
                        "model" : String,
                        "cores" : Number,
                        "herz" : String
                },
                "Memory" : {
                        "total" : String,
                        "avail" : String
                }
        }
}
```

※ In runtime, time is in microsecond units

※ In runtime, total, max, min, and avg mean the total time, maximum time, minimum time, and average time for which the gate was executed, respectively.

UseCase5: display simulation execution status on the console

The JSON file generated in UseCase #4 can be checked through a separate viewer. However, that file lacks readability to be viewed by an editor such as vi from the console. Therefore, we provide the function of displaying the same information to the console in a refined form. The following screen is an example of a screen captured by the --verbose option.

1. Circuit Information

1. used qubits	4
2. used gates	9
3. total gate calls	27
4. indivisual gate calls	
H	4 (14 %)
S	4 (14 %)
SDT	4 (14 %)
T	4 (14 %)
TDG	4 (14 %)
RX	1 (3 %)
RY	1 (3 %)
RZ	1 (3 %)
MEASURE	4 (14 %)

2. Runtime (micro seconds)

1. total simulation time	658			
2. each gate execution time	total	max	min	avg
H	257	190	18	64
S	75	19	18	19
SDT	75	19	18	19
T	74	19	18	19
TDG	74	19	18	18
RX	21	21	21	21
RY	19	19	19	19
RZ	20	20	20	20
MEASURE	44	15	9	11

3. Simulation Jobs

1. max number of quantum states	16
2. final number of quantum states	1
3. used memory	4.8 MB

4. System Information

OS	name	Ubuntu
	version	20.04.3 LTS (Focal Fossa)
CPU	model	11th Gen Intel(R) Core(TM) i7-11700KF @ 3.60GHz
	cores	2
	herz	3600.002
MEM	total	3.8 GB
	avail	1.8 GB

Exceptions

Memory Overflow

Most classical quantum simulators have the problem of exponentially increasing the memory capacity required according to the number of qubits. On the other hand, since QPlayer selectively stores only quantum states with physical reality in memory, the used memory space fluctuates dynamically as quantum circuits evolve. Therefore, to prevent the system from falling into an abnormal state due to memory overuse, the process is forcibly terminated with a memory shortage warning message when memory space is predicted to be insufficient by comparing the number of quantum states with remained memory.

OpenQASM grammar error

If a grammar error occurs in the QASM Code, we terminate the process by printing a message as shown in the table below.

number	error message
1	ERROR: undefined OPENQASM 2.0
2	ERROR: undefined include
3	ERROR: expected □□ in line-○
4	ERROR: argument is not a qreg in line-○
5	ERROR: argument is not a creg in line-○
6	ERROR: index of qreg is out of bound in line-○
7	ERROR: index of creg is out of bound in line-○
8	ERROR: unexpected numerical argument in line-○
9	ERROR: undefined gate or code string in line-○
10	ERROR: invalid gate in line-○

CHAPTER 4 Native C-like Interface

In this chapter...

- Classes
- Gate Functions
- Measurement
- Utility Functions
- Programming Examples

Researchers familiar with IT development can easily write complex quantum programs using the C/C++ library provided by QPlayer. Unlike OpenQASM 2.0, which allows only simple functions or limited if conditional statements, quantum programs can be written in conjunction with various native-C like style programming techniques.

This chapter describes the main classes provided by QPlayer, gate functions, and examples of programming using them.

Classes

QRegister Class

QRegister class manages the repository that stores quantum states. In order for a user to apply a quantum gate to a qubit, an instant of a QRegister class must be created in advance.

Class specification:

Member functions	descriptions
QRegister (int qubits)	Class constructor - creating quantum register with the number of qubits
~QRegister ()	Class destructor - free a quantum register instance
void reset ()	Initialize all state stored in the quantum register
int getNumQubits ()	Returns the number of qubits defined in the quantum register
qsize_t getNumStates ()	Returns the number of quantum states stored in the quantum register

Examples:

```
void activate_qregister(int qubits) {
        QRegister *QReg = new QRegister(qubits)

        // execute quantum gates...

        delete QReg;
}
```

QTimer Class

The QTimer class is a kind of utility class used to measure the simulation time of a quantum circuit.

Class specification:

Member functions	descriptions
QTimer ()	Class constructor
~QTimer ()	Class destructor

void start ()	start timer
void end ()	end timer
double getElapsedSec ()	return the elapsed time (in seconds)
double getElapsedMSec ()	return the elapsed time (in milli seconds)
double getElapsedUSec ()	return the elapsed time (in micro seconds)
double getElapsedNSec ()	return the elapsed time (in nano seconds)
const char *getTime ()	return the elapsed time string (in refined format)

Examples:

```
void check_runtime(void) {
	QTimer timer;

	timer.start();

	// run your tasks...

	timer.end();

	printf("Elapsed Time: %.f nano seconds\n", timer.getElapsedNSec());
	printf("Elapsed Time: %s\n", timer.getTime());
}
```

Gate Functions

`initZ(QRegister *QReg, int qubit)`

initialize the state of the given qubit to |0⟩.

`initX(QRegister *QReg, int qubit)`

initialize the state of the given qubit to |1⟩.

`X(QRegister *QReg, int qubit)`

The X gate flips the bit of the quantum state.

$|0\rangle \rightarrow |1\rangle$

$|1\rangle \rightarrow |0\rangle$

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Z(QRegister *QReg, int qubit)

The Z gate flips the phase of the quantum state. $\|0\rangle \rightarrow \|0\rangle$ $\|1\rangle \rightarrow -\|1\rangle$	$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$

Y(QRegister *QReg, int qubit)

The Y gate flips the bit and phase of the quantum state. $\|0\rangle \rightarrow i\|1\rangle$ $i\|1\rangle \rightarrow -i\|0\rangle$	$Y = \begin{pmatrix} 0 & i \\ -i & 0 \end{pmatrix}$

H(QRegister *QReg, int qubit)

The H gate rotates the quantum state by π radians with respect to the X+Z axis.	$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$

S(QRegister *QReg, int qubit)

The S gate rotates the quantum state by an angle of π/2 about the Z axis.	$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$

T(QRegister *QReg, int qubit)

The T gate rotates the quantum state by an angle of π/4 about the Z axis.	$T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}$

SDG(QRegister *QReg, int qubit)

The SDG gate rotates the quantum state by -π/2 degrees about the Z axis.	$SDG = \begin{pmatrix} 1 & 0 \\ 0 & -i \end{pmatrix}$

TDG(QRegister *QReg, int qubit)

The TDG gate rotates the quantum state by -π/4 degrees about the Z axis.

$$TDG = \begin{pmatrix} 1 & 0 \\ 0 & -e^{i\pi/4} \end{pmatrix}$$

P(QRegister *QReg, int qubit, double angle)

The P gate rotates the quantum state by angle about the Z axis.

$$P(\theta) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{pmatrix}$$

U1(QRegister *QReg, int qubit, double lambda)

The U1 gate rotates the quantum state by lambda with respect to the Z axis.

$$U1(\lambda) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{pmatrix}$$

U2(QRegister *QReg, int qubit, double phi, double lambda)

The U2 gate rotates the quantum state by phi about the X axis and by theta about the Z axis.

$$U2(\phi,\lambda) = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -e^{i\lambda} \\ e^{i\phi} & e^{i(\phi+\lambda)} \end{pmatrix}$$

U3(QRegister *QReg, int qubit, double theta, double phi, double lambda)

The U3 gate rotates the quantum state for three Euler angles.

$$U3(\theta,\phi,\lambda) = \frac{1}{\sqrt{2}} \begin{pmatrix} \cos\left(\frac{\theta}{2}\right) & -e^{i\lambda}\sin\left(\frac{\theta}{2}\right) \\ e^{i\phi}\sin\left(\frac{\theta}{2}\right) & e^{i(\phi+\lambda)}\cos\left(\frac{\theta}{2}\right) \end{pmatrix}$$

RX(QRegister *QReg, int qubit, double angle)

The RX gate rotates the quantum state by an angle about the X axis.

$$RX(\theta) = \begin{pmatrix} \cos\left(\frac{\theta}{2}\right) & -i\sin\left(\frac{\theta}{2}\right) \\ -i\sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{pmatrix}$$

RY(QRegister *QReg, int qubit, double angle)

The RY gate rotates the quantum state by an angle about the Y axis.	$RY(\theta) = \begin{pmatrix} \cos\left(\frac{\theta}{2}\right) & -\sin\left(\frac{\theta}{2}\right) \\ \sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{pmatrix}$

RZ(QRegister *QReg, int qubit, double angle)

The RZ gate rotates the quantum state by an angle about the Z axis.	$RZ(\theta) = \begin{pmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{-i\theta/2} \end{pmatrix}$

CU1(QRegister *QReg, int control, int target, double lambda)

CU1 applies U1 operation to target when control bit is 1.

CU2(QRegister *QReg, int control, int target, double phi, double lambda)

CU2 applies U2 operation to target when control bit is 1.

CU3(QRegister *QReg, int control, int target, double theta, phi, lambda)

CU3 applies U3 operation to target when control bit is 1.

CH(QRegister *QReg, int control, int target)

CH applies H operation to target when control bit is 1.

CX(QRegister *QReg, int control, int target)

CX applies X operation to target when control bit is 1.

CZ(QRegister *QReg, int control, int target)

CZ applies Z operation to target when control bit is 1.

CY(QRegister *QReg, int control, int target)

CY applies Y operation to target when control bit is 1.

CRZ(QRegister *QReg, int control, int target, double angle)

CRZ applies RZ operation to target when control bit is 1.

CCX(QRegister *QReg, int control1, int control2, int target)

CCX applies X operation to target when both control1 and control2 bits are 1.

SWAP(QRegister *QReg, int qubit1, int qubit2)

The SWAP gate swaps the states of the two input qubits.

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

CSWAP(QRegister *QReg, int control, int qubit1, int qubit2)

CSWAP applies a SWAP operation to two qubits (qubit1, qubit2) when the control bit is 1.

SX(QRegister *QReg, int qubit)

The SX gate applies the Sqrt(X) operation to the quantum state of a qubit.

$$SX = \sqrt{X} = \begin{pmatrix} 1+i & 1-i \\ 1-i & 1+i \end{pmatrix}$$

iSWAP(QRegister *QReg, int qubit1, int qubit2)

The iSWAP gate changes the state |01〉, |10〉 by adding two qubit imaginary numbers (i).

$$|00\rangle \rightarrow |00\rangle$$
$$|01\rangle \rightarrow i|01\rangle$$
$$|10\rangle \rightarrow i\,|10\rangle$$
$$|11\rangle \rightarrow |11\rangle$$

$$iSWAP = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & i & 0 \\ 0 & i & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Measurement Functions

int M(QRegister *QReg, int qubit)

Measures a qubit based on its amplitude probability and returns the result. The integer value returned according to the measured quantum state is as follows.

$$|0\rangle \rightarrow 0$$
$$|1\rangle \rightarrow 1$$

int MF(QRegister *QReg, int qubit, int state)

This function forces the qubit to collapse to a given state regardless of the amplitude probability and returns the result.
※ This operation can be used for purposes such as debugging of quantum circuits or algorithm verification.

Utility Functions

void dump(QRegister *QReg)

Display whole quantum states stored in a given quantum register on the screen.

void getMemory(uint64_t *memTotal, uint64_t *memAvail, uint64_t *memUsed)

Returns memory information for system and process

- memTotal: Maximum amount of memory installed in the system (bytes)
- memAvail: Amount of memory available on the system (bytes)
- memUsed: Amount of memory in use by the simulation process (bytes)

Programming Examples

Quantum hello world !!

```
#include "QPlayer.h"

int main(int argc, char **argv)
{
        QRegister *QReg = new QRegister(3)
                                                    // create quantum register with 3 qubits
        H(QReg, 0);                                 // apply Hadamard to qubit-0
        CX(QReg, 0, 1);                             // apply CNOT with qubit-0 → qubit-1
        Z(QReg, 2);                                 // apply Pauli-Z to qubit-2

        dump(QReg);                                 // display quantum states in quantum register

        delete QReg;
}
```

QFT algorithm

```
#include "QPlayer.h"

using namespace std;

void QFT(QRegister *QReg, int qubits) {
	X(QReg, qubits-1);

	for(int i=0; i<qubits; i++) {
		double angle = M_PI;

		H(QReg, i);
		for(int j=i+1; j<qubits; j++) {
			angle /= 2;
			CRZ(QReg, j, i, angle);
		}
	}

	for(int i=0; i<qubits/2; i++) {
		SWAP(QReg, i, qubits-i-1);
	}

	dump(QReg);
}

int main(int argc, char **argv)
{
	int qubits = 0;
	int c;

	while ((c = getopt_long(argc, argv, "q:", NULL, NULL)) != -1) {
		switch(c) {
		case 'q':
		    qubits = atoi(optarg);
		    break;
		default:
		    break;
		}
	}

	if(qubits == 0) {
		printf("<USAGE> : %s -q <qubits>\n", argv[0]);
		exit(0);
	}

	QRegister *QReg = new QRegister(qubits);
	qft(QReg, qubits);
}
```