CEN/CLC/JTC 22/WG 3 "Quantum Computing and Simulation"
WG Secretariat: **AFNOR**
Convenor: **Lefebvre Catherine Mrs**

LayerModel_Draft03

Document type	Related content	Document date	Expected action
Meeting / Document for discussion	Meeting: Torino (Italy) 20 Nov 2023	2023-11-06	

CEN/TC XXX

Date: 20YY-XX

prEN XXXXX:20YY

Secretariat: XXX

JTC 22 WG3 Quantum Computing Layer Model

Draft 03, 2023-11-01

CCMC will prepare and attach the official title page.

Contents

Page

[NOTE to the drafter: To update the Table of Contents please select it and press "F9". To recreate the Table of Contents, select *Custom Table of Contents – Options* and choose the appropriate headings/titles to display. For further instructions, see the *CEN Simple Template Quick Start Guide*.]

European foreword

This document (prEN XXXX:20YY) has been prepared by Technical Committee CEN/TC XXX "Title", the secretariat of which is held by XXX.

This document is currently submitted to the CEN Enquiry/Formal Vote/Vote on TS/Vote on TR.

This document will supersede EN XXXX:YYYY.

EN XXXX:YYYY includes the following significant technical changes with respect to EN XXXX:YYYY:

This document has been prepared under a Standardization Request given to CEN by the European Commission and the European Free Trade Association, and supports essential requirements of EU Directive(s) / Regulation(s).

For relationship with EU Directive(s) / Regulation(s), see informative Annex ZA, ZB, ZC or ZD, which is an integral part of this document.

[NOTE to the drafter: Add information about related documents or other parts in a series as necessary. A list of all parts in a series can be found on the CEN website: www.cencenelec.eu.]

Introduction

A layer model is an abstract description of a (computing) system via a common stack of layers. The layer model for quantum computing slices down the overall complexity of quantum computing into two main layer models of addressing the whole system. The lower main layer model addresses mainly hardware, and it is dependent of the physical platform. The upper main layer model addresses mostly software at a higher level of abstraction.

Each of these two main layer models comprises a stack of inner layers. The lower (hardware) main layer model comprises multiple stacks, one for each identified architecture family.

The higher up in the stack the more hardware-agnostic the inner layers of the upper (software) main layer model will gradually be. By agnostic we mean that the same system works for different quantum computing hardware platforms such as solid state quantum computing, ion traps, neutral atoms, optical quantum computing and topological quantum computing.

This structure decouples the software design from the hardware design to some extent, which has clear advantages, such as the reusability of algorithms for different hardware. At the same time the structure does not impose a fully hardware-agnostic upper main layer model to encompass the design of quantum hardware and software in a co-design approach, that is, adapt software to make optimal use of the hardware used and the vice versa. This approach is inevitable for current and near-future quantum computer development, just as it turned out to be vital for classical computers in early stage and current classical computing disciplines, e.g., in microcontroller design.

The first purpose of this document is to define a common language that will be used to describe the features and functional requirements for each layer of the stack of a quantum computer. Another purpose is to analyse and describe the interaction between the layers by means of well-defined interfaces. These are essential steps towards interworking between modules from different origins. The functional description of each layer should offer sufficient guidance on where a desired functionality should be described, and what kind of exchange is needed with other modules through the interfaces. The boundaries between the layers are natural locations for such interfaces. Correctly defining such boundaries requires careful analysis of the interaction between the layers.

EDITORIAL NOTE: The question has been raised if a strict description of the concept “layer” should be included here (or in another chapter), as well as an explanation of the purpose of layers. Functionalities like Quantum Error Corrections are typically distributed over several layers, including low-level hardware solutions, the concept of logical qubits, error-correcting algorithms, etc. Such distributed functionalities should not be confused with a single layer within the context of this document

---

[NOTE to the drafter: If patent rights have been identified during the preparation of the document, the following text shall be included:

“The European Committee for Standardization (CEN) draws attention to the fact that it is claimed that compliance with this document may involve the use of a patent concerning (…subject matter…) given in (…subclause…) and which is claimed to be relevant for the following clause(s) of this document:

Clause(s)…

CEN takes no position concerning the evidence, validity and scope of this patent right. The holder of this patent right has assured CEN that they are willing to negotiate licences under reasonable and non-

discriminatory terms and conditions with applicants throughout the world. In this respect, the statement of the holder of this patent right is registered with CEN. Information may be obtained from:

Name of holder of patent right …

Address ...

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights other than those identified above. CEN shall not be held responsible for identifying any or all such patent rights."]

1 Scope

This document describes a layer model that covers the entire stack of a quantum computer. The group of lower-level (hardware) layers are organized in different hardware stacks tailored to different hardware architectures, while the group of higher-level (software) layers are built on top of these and expected to be common for all quantum computing systems. The higher-up in the stack, the more agnostic it will be from underlying layers. Reducing the dependencies between higher and lower layers is a crucial point for optimized quantum computations. A co-requisite point is to allow for a free but well-defined flow of information up and down the higher and lower layers to allow for co-designing hardware and software.

This document is limited to a high-level (functional) description of the layers involved. Additional details of the individual layers will be described in other, future, CEN/TRs.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

[NOTE to the drafter: The Normative references clause is compulsory. If there are no normative references, add the following text below the clause title: "There are no normative references in this document."]

EN XXXX, *Title of document*

EN XXXX-1:20YY, *General title of series — Part X: Title of part*

EN XXXXX (all parts), *General title of series*

[NOTE to the drafter: If a dated reference is impacted by a standalone amendment or corrigendum, list the main standard and include a footnote as follows:

EN XXXX:20YY[1], *General title*]

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply:

ISO and IEC maintain terminology databases for use in standardization at the following addresses:

- ISO Online browsing platform: available at https://www.iso.org/obp/
- IEC Electropedia: available at https://www.electropedia.org/

3.1 Terminology

Codesign - A design approach where (software) modules query lower layers for identifying the (hardware) capabilities and limitations of a system and subsequently tailor their behaviour to these capabilities and limitation. This approach allows for hardware-specific optimizations and adaptations to optimize quantum computations.

[1] As impacted by EN XXXX:20YY/A1:20YY.

3.2 Abbreviations

API - Application Programming Interface

PCB – Printed Circuit Board

SDK – Software Development Kits

QEC - Quantum Error Correction

4 Overview

Quantum computing is an area covering many different implementations. A convenient way of specifying its requirements is via a stack of layers, as shown in Figure 4.1. This layer approach is inspired on the OSI model that computer systems use to communicate over a network. The layers are chosen in such a manner that the functionality of each layer can be described in an independent manner. This causes that the interworking between these layers can be described through well-defined interfaces at the boundaries of these layers. Note that such an interface can be virtual (hidden internally within the implementation of the same origin) or real (between implementations of different origin).

The stack covers both hardware and software layers, while some layers are a mix of both. The software layers are drawn in Figure 4.1 above the hardware layers with another colour. Each layer aims to be more agnostic to the exact implementation of lower layers. The functional description of these layers are described in further detail in succeeding chapters.

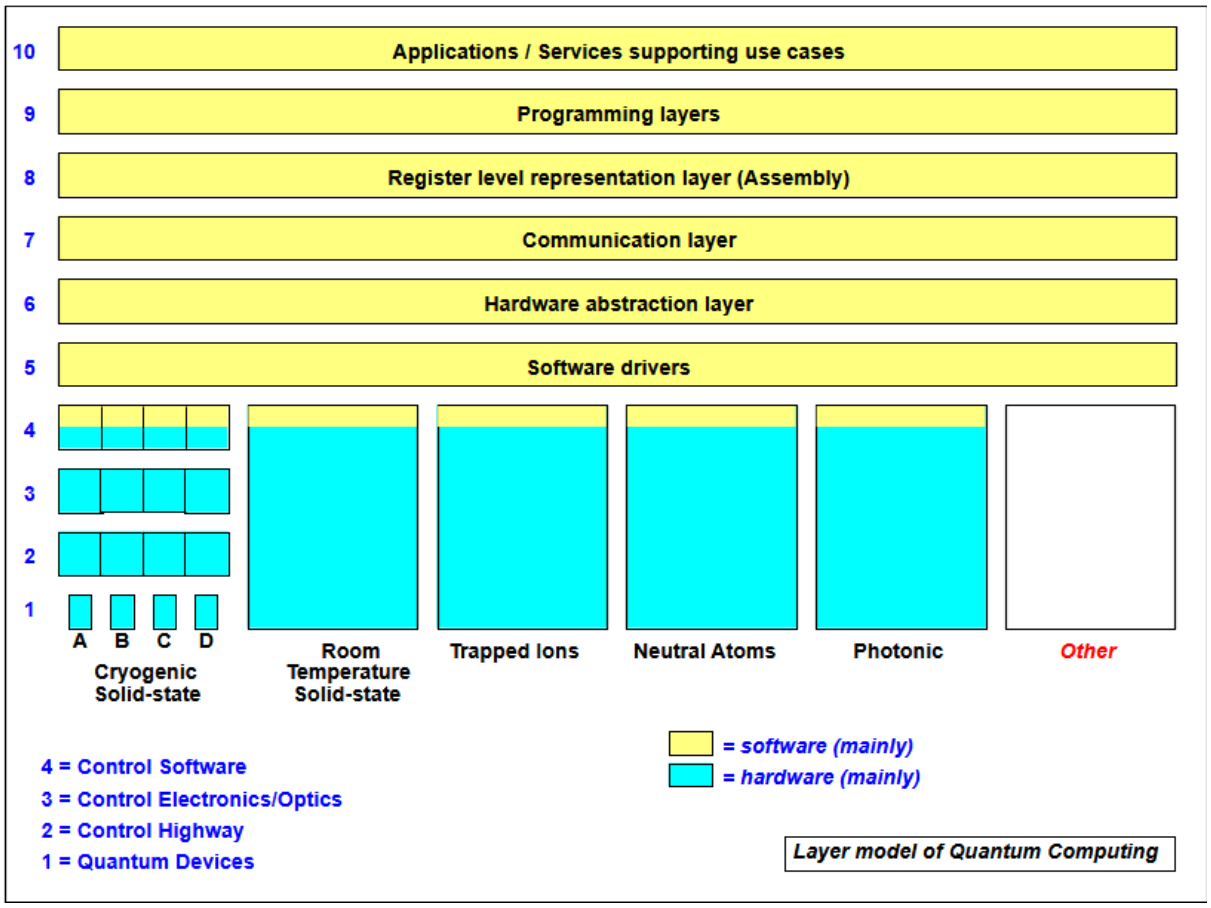

Figure 4.1 *- Overview of the layer model of quantum computing.*
It supports multiple hardware implementations in the hardware layers

The layered approach allows for using different hardware stacks for specifying the requirements of different architecture families. Each architecture family can have multiple members (A, B, C, ...) and the description of its hardware layers (1,2,3,4) may account for differences between these members. The diagram in Figure 4.1 has illustrated this symbolically by drawing different "boxes" in the layers for different members of the same family.

So far, the following quantum architecture families have been identified (in arbitrary order):

- Cryogenic solid-state based;
- Room temperature solid-state based;
- Trapped ions;
- Neutral atoms;
- Photonic quantum computing;
- Other architectures that may be identified in the future

These architectures are described in further detail in succeeding chapters.

A module is an implementation that may be constructed from (smaller) modules and components. It could offer the functionality of a single layer, of multiple layers, or just of a fragment of a layer. A module may also support different operating modes, such that it complies with the requirements of multiple members and/or multiple architecture families. As such, the functionality of a module may cover multiple layers and/or families and/or members.

4.1 Subclause title

4.2 Subclause title

4.2.1 Subclause title

4.2.1.1 Subclause title

4.2.1.1.1 Subclause title

4.2.1.1.1.1 Subclause title

Text of subclause.

5 Low level Hardware layers

5.1 Cryogenic Solid State

The members of this architecture family have in common that they all make use of a cryostat, where the quantum devices in a holder are controlled from outside the fridge by room-temperature electronics. Consequently, a huge amount of control channels is required to interconnect those two, especially when many qubits are to be controlled in a single fridge.

The following members have been identified within this architecture family:

- Transmons;
- Flux qubits;
- Semiconductor spin qubits;
- Topological qubits;
- Artificial atoms in solids.

Four hardware layers have been identified for this architecture family.

5.1.1 Layer 1 – Quantum Devices

The quantum devices in hardware layer 1 are modules with qubits that are typically operating at cryogenic temperatures and may be implemented as chip and/or on PCB. They may have though requirements on shielding, operating temperature, magnetic aspects, etc.

5.1.2 Layer 2 – Control Highway

Hardware layer 2 covers all infrastructure needed for transporting microwave, light wave, RF and DC signals (via electrical and/or optical means) between the control electronics at room temperature and the quantum devices at cryogenic temperatures. It is usually a mix of transmission lines, filtering, attenuation, amplification, (de)multiplexing, as well as means for proper thermalization. A huge number of control channels are required to control many qubits in a single fridge (which clarifies the name) and this can easily become very bulky. It could have tough requirements on aspects like heat-flow, thermal noise and vacuum properties.

5.1.3 Layer 3 – Control Electronics

Hardware layer 3 covers all electronics for generating, receiving, and processing microwave, RF and DC signals. Some implementations make use of routing/switching and/or multiplexing of control signals at room temperatures. It may have some firmware on board to guide the signal generation and signal processing.

5.1.4 Layer 4 – Control Software

Hardware layer 4 covers a mix of hardware and low-level driver software for instructing the control electronics and means for performing calibration. It has a software interface to higher layers for receiving sequences of instructions about when, where and what pulses are to be generated, and how to process and read-out the response.

Placed on top of quantum hardware, control software delivers high-performing qubit operations to higher level of abstraction in the quantum stack with minimal user intervention. It may include calibration means, low-level code to translate instructions from higher software layers into commands for guiding the control electronics/optics, and comprises the techniques used to define error-robust physical operations and associated supporting protocols designed to tune-up and stabilize the hardware.

Control software for quantum hardware is typically stored on digital computers, i.e., there is a very strict separation between the place where the control software is stored and the quantum registers. In the long term, control software may work in concert with Quantum Error Correction (QEC) imposed ny higher software layers, to provide more resilience on various error types. More specifically, control software could improve the efficiency of QEC, i.e., reduce resource overheads required for encoding, by homogenizing error rates and reducing error correlations.

5.2 Room Temperature Solid State

The members of this architecture family have in common that solid-state qubits are all operating at room temperatures. Examples of members in this architecture family are:

- Artificial atoms in solids, such as NV centres;
- Optical quantum dots.

5.2.1 Layer 1 – Quantum Devices

5.2.2 Layer 2 – ...

5.2.3 Layer 3 – ...

5.2.4 Layer 4 – ...

5.3 Trapped Ions

The members of this architecture family can operate either at room temperature or at cryogenic temperatures (e.g. 4K). Quantum devices are controlled by electronics operating either at room temperature or under cryogenic conditions. For a larger number of qubits, the required amount of routing signals becomes bulky, and efficient thermal management, low-noise electrical and magnetic components are required.

Room temperature architectures that are identified are

- Optical qubits;
- Raman qubits;
- Spin (microwave) qubits;

Cryogenic (4K) architectures that are identified are

- Optical qubits;
- Raman qubits;
- Spin (microwave) qubits

5.3.1 Layer 1 – Quantum Devices

5.3.2 Layer 2 – ...

5.3.3 Layer 3 – ...

5.3.4 Layer 4 – ...

5.4 Neutral Atoms

Systems of individually-controlled neutral atoms, interacting with each other when excited to Rydberg states, have emerged as a possible platform for quantum information processing. The two main examples are ensembles of individual atoms trapped in optical lattices or in arrays of microscopic dipole traps separated by a few micrometres. In these platforms, the atoms are almost fully controllable by optical addressing techniques.

5.4.1 Layer 1 – Quantum Devices

5.4.2 Layer 2 – ...

5.4.3 Layer 3 – ...

5.4.4 Layer 4 – ...

5.5 Photonics

These architectures have in common that the quantum information during computing is encoded into photonic properties. We can divide different families of photonic quantum computers in two categories, universal and non-universal quantum computers. Non-universal quantum computers cannot perform every task but can at least perform one task.

Non-universal photonic quantum computing families that are identified are:

- Boson sampling;
- Gaussian boson sampling.

Universal families that are identified are:

- Knill-Laflamme-Milburn scheme (This was a theoretical proof-of-principle, but not practically feasible);
- Measurement based quantum computing using cluster states;
- Continuous variable quantum computing.

5.5.1 Layer 1 – Quantum Devices

5.5.2 Layer 2 – ...

5.5.3 Layer 3 – ...

5.5.4 Layer 4 – ...

5.6 *Other…*

Other architectures may be identified in future, and will then be added to this list.

6 Software Drivers

In the layered view illustrated in Figure 12, software drivers are components that are plugged into the operating system and allow hardware-abstraction programs to call the control software of the underlying quantum hardware. If the hardware changes, the software drivers must change as well.

7 Hardware abstraction Layer (HAL)

The aim of the Hardware Abstraction Layer is to inform higher layers with capabilities and limitations supported by the underlying hardware. Layers above the HAL can use this information to hide many implementation-specific details to higher layers by offering a more unified interface. Layers above may also use this information to provide higher-level commands to programmers or programs allowing for implementing hardware-specific optimizations and adaptations.

Not all quantum computers make use of the same paradigm. Annealing quantum computers behave differently from gate-based quantum computers, and therefore their HALs might be different as well. The HAL can therefore provide information about the underlying architecture, such as for instance being “gate-based”, “annealing” or “simulation”.

7.1 HAL for gate-based quantum computers

A gate-based quantum computer processes a sequence of instructions to change the state of a quantum register with many qubits before the resulting state is queried by measurements. A convenient graphic representation of such a sequence has the appearance of a circuit where the elements seem to operate on one or more qubits simultaneously. Due to this convenient graphic representation, these instructions are called gates.

7.1.1 Organization of qubits

Quantum register: A quantum register is a system comprising multiple qubits. The HAL supports instructions to operate on such a register, for initializing, changing, and querying its state.

Width: The HAL can specify the number of available qubits and how they are organized in these registers. It can also specify if all qubits are part of a single quantum register or if are they are allocated to multiple (smaller) registers. The use of multiple registers may occur when using modular hardware architectures.

Depth: The HAL can specify the maximum depth for circuits of gates that can be executed before the calculated result becomes unreliable. This value is related to coherence time of the implementation and other imperfections of underlying hardware.

Connections: The HAL can also provide an "adjacency matrix" for each quantum register, to indicate which qubits are edge-connected. For instance, when a register has N qubits, then this adjacency matrix C has size NxN. The default of each element in this matrix is false, but if qx and qy are the indices of two adjacent qubits then C(qx,qy)=C(qy,qx)=true. Matrix C is therefore a symmetric matrix, since C(k,r)=C(r,k).

The HAL can provide additional information about the underlying architecture.

7.1.2 The concept of native gates

The HAL can specify a list of "native gates" supported by the underlying hardware. The name "native gate" refers to an operation for changing the quantum state of a register by means of a "single" physical action on one or more qubits simultaneously. An example is a single pulse composition that cannot be broken down into two or more shorter pulse compositions. In other words, if a gate can be divided into two or more shorter independent sequential physical actions, it is not native.

As a result, a native gate can be executed in the minimum amount of execution time. Knowledge about which gates are native is relevant information for compilers that try to optimize a circuit with respect to execution time.

Gates that can only be implemented by a sequence of two or more native gates are called "compound" gates.

The boxed example in figure 7.1 illustrates for a specific case that the single qubit gates X, Y, Rx(a), Ry(b) are all native for that implementation, while the gates Z and Rz(c) are compound gates. A similar example can be elaborated with dual qubit gates. For a specific implementation, a gate like CNOT may turn out to be compound as well when it cannot be implemented with one native dual qubit gate.

Example

The concept of native gates can be explained by the following example. Assume that a specific hardware implementation supports a mechanism to rotate a qubit via a "single" pulse composition that can be controlled with two real parameters "a" and "b". Assume that the definition of this rotation function equals:

```
RN(a,b) = [ cos(a/2),              -j*exp(-j*b)*sin(a/2)]
          [-j*exp(j*b)*sin(a/2),               cos(a/2)]
```

Then some of the well known gates can be implemented via:

```
Rx(a) = [   cos(a/2), -j*sin(a/2)] = RN(a,0)
        [-j*sin(a/2),    cos(a/2)]
Ry(b) = [   cos(b/2),   -sin(b/2)] = RN(a,pi/2)
        [   sin(b/2),    cos(b/2)]
Rz(c) = [exp(-j*c/2),       0    ] = RN(pi,0) * RN(pi,-c/2)       * exp(j*pi)
        [          0,  exp(j*c/2)]


X     = [ 0,  1] = Rx(pi) * exp(j*pi/2) = RN(pi,0)                * exp(j*pi/2)
        [ 1,  0]
Y     = [ 0, -j] = Ry(pi) * exp(j*pi/2) = RN(pi,pi/2)             * exp(j*pi/2)
        [+j,  0]
Z     = [ 1,  0] = Rz(pi) * exp(j*pi/2) = RN(pi,pi) * RN(pi,pi/2) * exp(-j*pi/2)
        [ 0, -1]
```

In this hardware implementation, Rx(a), Ry(b), X, Y can be considered as native gates. The gates Rz(c) and Z are to be combined from two sequential native gates, so they are compound. Knowledge about which gates are native is relevant for quantum algorithms that try to find an optimal circuit representation in terms of execution time.

Figure 7.1 *- Example of a specific hardware implementation, where Rx, Ry, X, Y are native gates and Rz, Z are compound gates*

7.1.3 Concept of primitive gates

A compiler or interpreter does not always know how to convert well-known gates into a smart combination of native gates for any possible set of native gates. In those cases, a fall-back situation should be supported by the HAL in terms of predefined solutions for well-known gates like Rx(a), Ry(b), Rz(c), X, Y, Z, H, S, T, CNOT, etc.

Therefore, the HAL can specify a list of "primitive gates" that it can emulate by a sequence of one or more native gates.

7.1.4 Concept of measurement

The HAL supports instructions to query the state of one or more qubits in a quantum register by means of a measurement. The answer will be returned as a binary string stored in a dedicated register. Note that the state will be collapsed after such a query.

The HAL also supports instructions to read out the bits in this register and/or to use these bits for instructing controlled gates.

If the hardware supports it, the HAL can also offer instructions to specify the basis for these measurements.

7.1.5 Interfacing considerations

A preferred way of communicating with the HAL is by means of binary instructions, preferably common for all quantum computing implementations. Therefore, it requires a list of binary commands for letting the HAL report capabilities and limitations of the underlying hardware, and for executing all aforementioned instructions.

Such an interface may also offer a convenient format for instructing a simulator that emulates a quantum computer with a limited set of qubits.

7.2 HAL for annealing quantum computers

EDITORIAL NOTE: Contributions are invited for adequate text that should fit here.

7.3 HAL for simulation of quantum physics models

EDITORIAL NOTE: Contributions are invited for adequate text that should fit here.

8 Communication Layer

EDITORIAL NOTE: The question has been raised if this topic should be considered as a "layer" (in the full sense of all other layers of the stack) or as something else beside the stack.

A quantum computer must be provided with a local operating system (OS), which is a resource manager for the underlying quantum hardware, provided with built-in networking functions allowing multiple clients to use the resources. Programs can make use of facilities only as offered by the OS. For example, the OS provides communication primitives (for instance based on the POSIX standard for the sockets interface [ref1[2]]) and only by means of these primitives it should be possible to pass messages between programs.

The communication layer can handle these messages to send and receive instructions between client applications outside the quantum stack and layers inside the quantum stack. This message exchange between inside and outside the quantum stack can be *internal* as well as *external*. Internal refers to processes running on the same OS, and external refers to processes running on a nearby computer or on a remote server somewhere in the cloud. Exchange means both receiving and sending of messages.

The communication layer can handle all messages that are needed for starting a quantum computing session (for instance handshaking, authentication, resource allocation, billing, rights-management, etc.). A quantum computing session offers an application the experience as if it has its own resources and as if it is fully protected from other applications.

Once a session is initiated, the communication layer can start handling incoming messages for instructing layers higher up in the stack. For instance, to load and run a quantum assembly task. Results can be passed back to the communication layer, which in turn can send messages with those results to the client application outside the quantum stack (see Figure 8.1).

The communication layer can also communicate directly with lower layers in the quantum stack, if the user is allowed to according to allocated usage rights. For instance, to send low-level commands directly to the control electronics for firing a specific pulse to a qubit. And again, detected results from the control electronics can also be passed back to the communication layer, which in turn can send messages with those results to the client application outside the quantum stack.

Figure 8.1 illustrates the instruction flow through the layers in a graphic way. In this example the communication layer receives a message to instruct the register-level representation layer to run a quantum assembly task. When completed, the register-level representation layer informs the communication layer about the result, which in turn sends a message to the sender of the initial message.

In other words:

- Step 1 is receiving a message from a client outside the stack.
- Step 2 is forwarding the request to for instance a register-level interpreter.
- Step 3 is an instruction sequence to the quantum devices.
- Step 4 is the register-level processing of the quantum computation result.
- Step 5 is forwarding the result back to the communication layer.
- Step 6 is sending the results back to the requesting client.

[2] IEEE/Open Group 1003.1-2017, Standard for Information Technology - Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7.

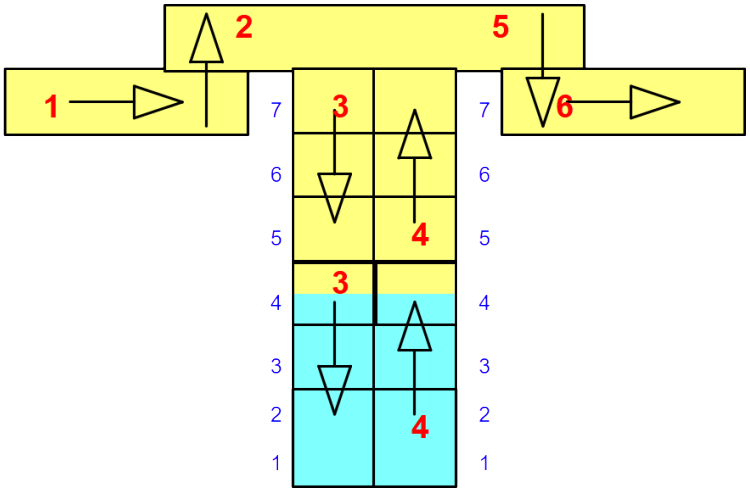

Figure 8.1 *- Instruction flow through the layers within the quantum stack*

9 Register level representation layer

This layer concerns quantum assembly languages (QASM) that describe quantum computations according to one specific model (e.g., circuit model, measurement-based model, quantum annealing model), with a per-architecture instruction set.

QASM is not a single assembly language and the syntax may also differ among various implementations. Languages for gate-based quantum computing have in common that they can describe universal circuits with single qubit gates, and entangled gates such as CNOT. Due to the huge diversity of quantum computing architectures, it is not likely that a unique, widely accepted QASM would emerge and later become a standard.

10 Programming layer

The specification of quantum algorithms using register-level representation languages is not easy for programmers. Indeed, quantum assembly programs are usually generated by a software library, from a piece of code written in a common programming language, such as Python.

In general, the Programming Layer includes all the languages, libraries, and software development facilities for coding quantum algorithms or high-level applications that use predefined quantum algorithms as subroutines.

10.1 Programming Languages and Libraries

In the quantum computing domain, Python is the most used high-level programming language. It is a general-purpose imperative language, as it allows developers to write code that specifies the steps the computer must take to accomplish the goal. Other imperativelanguages have been designed on purpose for quantum computing, such as Q# and Silq.

Alternative to imperative programming is functional programming, where programs are constructed by applying and composing functions. In the quantum computing domain, there are a few functional programming languages, such as QPL and Quipper.

Writing a program in a high-level language implies using software development kits (SDKs) that include application programming interfaces (APIs) for coding quantum algorithms from scratch, but also collections of ready-to-use quantum algorithms. The APIs may be very different, depending on the quantum computational model (quantum circuit model, quantum annealing, measurement-based quantum computation, etc.) and specific application domain (quantum optimisation, quantum machine learning, etc.).

For Python programmers, there are several advanced SDKs. Some of them are bound to proprietary hardware platforms. Other SDKs are general-purpose and support device architectures from multiple providers.

10.2 Quantum Compilation

EDITORIAL NOTE: This section seems to be dedicated to gate-based quantum computing only, and may not apply to annealing quantum computing or other solutions. Shouldn't we create different sections for each of them, such as:

- Quantum Compilation for gate-based QC
- Quantum Compilation for annealing DC
- other?

Being high-level programs hardware-agnostic, quantum compilers are necessary to translate abstract quantum algorithms into the most efficient equivalents of themselves, considering the constraints and features exposed by the Register-level representation layer.

For simplicity, we refer to the quantum circuit model of computation. In this context, the input to the quantum compiler is a quantum circuit including single or multi-qubit gates. Usually, the input circuit is the simplest (and most elegant) representation of a quantum algorithm (e.g., the Quantum Fourier Transform). Such a representation does not consider the constraints that may characterise the target quantum computer, such as the available gate set and the connectivity constraints between which qubits a two-qubit gate is natively allowed.

The quantum compiler leverages information provided by the Register-level representation layer to translate the input circuit into an equivalent circuit that fits the target device.

An example is provided in figure 10.1, in which a quantum circuit is compiled into another quantum circuit by considering the connectivity constraints of the target quantum computer.

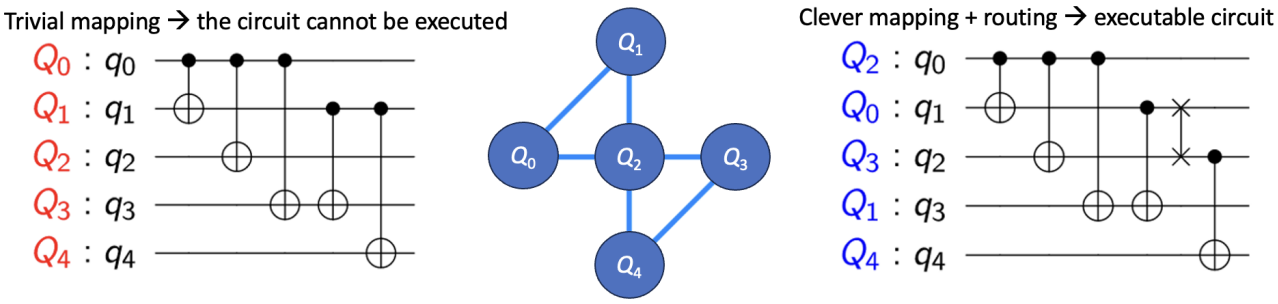

Figure 10.1 *– The circuit on the left does not fit the connectivity constraints of the target device, which are described by the graph in the middle of the figure. The circuit on the right is the compiled version of the circuit on the left, i.e., functionally equivalent but fitting the target device. To produce the output circuit, the compiler chose a different mapping for the input circuit's qubits to the device qubits and inserted a SWAP gate before the last CNOT gate.*

The description format of the output circuit may be different from the description format of the input circuit. If the input and output circuits have the same description format, the compiler is often denoted as "transpiler".

11 Applications / Services supporting use cases

This layer contains the user-side where a problem exists that requires solving. Quantum computers can help solving this problem and the user can then start programming algorithms to obtain the sought for answer. Depending on the used service, users may perform tasks locally on a quantum computer. An alternative is that tasks run mainly remotely on a classical computer and use a quantum cloud service to run specific tasks on a dedicated quantum computer.

Annex A
(informative)**111**

Title of Annex A, e.g. Example of a table, a figure and a formula

A.1 Clause title

A.1.1 Subclause title

A.1.1.1 Subclause title

A.1.1.1.1 Subclause title

A.1.1.1.1.1 Subclause title

Text of the annex.

A.2 Example of a table

Table A.1 — Table title

Table header[a]			
Table text	Text[b]		
NOTE Table note.			
[a] Table footnote. [b] Second table footnote.			

[NOTE to the drafter: For indented text, it is recommended to create new cells instead of using tabs. Similarly, when aligning text to the right or center, use Word alignment buttons rather than tabs.]

A.3 Example of a figure

Dimensions in millimetres

Insert and Link Figure

Key

X definition for X

Y definition for Y

NOTE Figure note.

Figure text.

Figure A.1 — Figure title

A.4 Examples of formulae

A + B = C (1)

where

A is … ;

B is … ;

C is … .

[NOTE to the drafter: For simple formulae, the keyboard can be used. For more complex formulae, it is recommended to use MathType, if available, or MS Word Equation Editor.]

$$D_1 = 5{,}77 \times 10^{-13} \frac{C_1 \rho_1}{4\pi} \sum \gamma_i \left(\frac{\mu_{en}}{\rho} \right) E_i \int B_i (1) \frac{e^{-\mu_i(1) s_1}}{\ell^2} dV \quad (2)$$

where

$B_i(1)$ is …

D_1 is…

…

Annex ZA
(informative)

Relationship between this European Standard and the [essential]/[interoperability]/[...] requirements of [Directive]/[Regulation]/[Decision]/[...][Reference numbers of the legal act] aimed to be covered

[NOTE to the drafter: This is the Generic Annex ZA template. For some Directives/Regulations, specific templates need to be used and these can be found on the CEN BOSS:
https://boss.cen.eu/reference-material/FormsTemplates/Pages/]

This European Standard has been prepared under a Commission's standardization request [Full reference to the request "M/xxx"/"C(2015) xxxx final"] to provide one voluntary means of conforming to [essential] / [interoperability] / [...] requirements of [Directive] / [Regulation] / [Decision] / [...] [Reference numbers of the legal act] [Full title].

Once this standard is cited in the Official Journal of the European Union under that [Directive] / [Regulation] / [Decision] / [...], compliance with the normative clauses of this standard given in Table [...] confers, within the limits of the scope of this standard, a presumption of conformity with the corresponding [essential] / [interoperability] / [...] requirements of that [Directive] / [Regulation] / [Decision] / [...], and associated EFTA regulations.

Table ZA.1 — Correspondence between this European Standard and [Annex ... of] / [Article(s) ... of] [Directive] / [Regulation] / [Decision] [Reference numbers of the legal Act]]

[Essential]/ [interoperability]/[...] Requirements of [Directive]/[Regulation]/[Decision] [...]	**Clause(s)/sub-clause(s) of this EN**	**Remarks/Notes**

[NOTE to the drafter, to be removed before publication:

This table can be used to accommodate all possible cases and independently how detailed correspondence is established or is possible to give:

— to declare the correspondence with a general statement 'all requirements are covered' by complying 'all (or indicated) clauses' (then the table would contain only one row);

— to declare more detailed correspondence (then the table would contain as many rows as needed).]

WARNING 1 — Presumption of conformity stays valid only as long as a reference to this European Standard is maintained in the list published in the Official Journal of the European Union. Users of this standard should consult frequently the latest list published in the Official Journal of the European Union.

WARNING 2 — Other Union legislation may be applicable to the product(s) / [service(s)] / [...] falling within the scope of this standard.

Bibliography

[1] IEEE/Open Group 1003.1-2017, Standard for Information Technology - Portable Operating *System Interface (POSIX(TM)) Base Specifications, Issue 7.*

[2]