CEN/CLC/JTC 22/WG 3 "Quantum Computing and Simulation"

Convenor: **PAUL Alexandra MME**

HAL_requirements_Draft02

Document type	Related content	Document date	Expected action
Project / Draft	Meeting: VIRTUAL 18 Mar 2026	2026-02-17	**COMMENT/REPLY** by 2026-03-18

Description

Dear members,

Please find attached the lastest draft of HAL WI.

Best,

Simon Del Nin

CEN/TC XXX

Date: 20YY-XX

prEN XXXXX:20YY

Secretariat: XXX

JTC 22 WG3 Quantum Computing

Hardware Abstraction Layer

Functional requirements

Draft 02, 2026-02-17

CCMC will prepare and attach the official title page.

Contents

Page

European foreword

This document (prEN XXXX:20YY) has been prepared by Technical Committee CEN/TC JTC22/WG3 "Quantum Computing and simulation", 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 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).

Introduction

A hardware abstraction layer (HAL) is a software layer within a quantum computing software stack to enable higher-level software and compilers to operate independently of hardware-specific details. This TS defines functional descriptions and functional requirements for the HAL, while specific implementation details and values remain out of scope. The HAL resides between the assembly layer and the control software layer, as described in TR 18202:202 on “Layer Models for Quantum Computing”. Its purpose is to provide a standardised interface that abstracts hardware-specific details away and to expose essential capabilities to higher layers. This capability enables portability and interoperability across diverse quantum architectures.

The HAL is designed to support a wide range of quantum hardware platforms, each with unique characteristics such as qubit topology, native gate sets, and error correction schemes. Its functionality is plug-in extendable via libraries to accommodate future developments. Rather than prescribing strict implementation details, this TS focuses on defining the functional roles of the HAL, including instruction translation, resource management, fault tolerant quantum computing, mapping and routing, virtualisation for multi-user environments, and reporting of hardware capabilities. These functionalities allow compilers and programming frameworks to optimise execution without requiring direct interaction with proprietary hardware interfaces.

By establishing these functional requirements, the HAL becomes a cornerstone for modular quantum computing systems, ensuring flexibility, scalability, and vendor-neutral integration.

1 Scope

This TS describes the functionalities of the hardware abstraction layer (HAL) for use in the software stack of quantum computers, as described in TR 18202:202 on “Layer Models for Quantum Computing”. The present TS is focussed on a high-level functional description of the HAL while additional details on implementation and interfacing are reserved for other future CEN/TSs.

The word “functional” means within this context that a precise definition of implementation, interfaces, information flows (via instructions, commands, signals, etc.) and values is out of scope. This TS concentrates therefore on general descriptions of what the HAL supports and which properties or quantities are to be considered for future specification. It offers mainly an enumeration of characteristics that are considered as relevant as well as a motivation why they are relevant.

The aim of the Hardware Abstraction Layer is to inform higher layers with capabilities and limitations supported by the underlying hardware. It hides many implementation-specific details, and provides a more unified interface to higher layers. This includes instruction translation, resource management, fault tolerant quantum computing, mapping and routing, virtualisation for multi-user environments, and reporting of hardware capabilities.

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 behave different as well. The HAL can therefore provide information about the underlying architecture, such as for instance being “gate-based”, “annealing” or “simulation”. When applicable, the HAL can also select between multiple quantum architectures and even partition a single task into multiple queues to perform calculations in parallel on multiple architectures.

In order to speed-up progress, the first version of this TS may put a focus on gate-based quantum computer, but that does not exclude functional requirements for other architectures. If needed this first version may leave details of those architectures for future revisions of this TS.

2 Normative references

- CEN/CLC/TR 18202:2025, "*Layer model of Quantum Computing*", sept 2025

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

ISA (Instruction Set Architecture)

a lower-level method of defining operations on a quantum computer.

Note 1: Instead of defining specific gates, this method defines gates (or other instructions) as operations, using pulses pulsed for a certain time, on specific qubits.

3.2

Universal gate-based quantum computing

a quantum computer being capable of processing an arbitrary quantum circuit.

Note 1: A universal gate-based quantum computer ought to have a gate set which is universal. A gate set is said to be universal if any unitary operation may be approximated to arbitrary accuracy by a quantum circuit involving only those gates [2]. The definition also comprises non-fault-tolerant universal quantum computers, which can process an arbitrary quantum circuit reliably only up to a certain length, size or gate count.

4 Abbreviations

ISA – Instruction Set Architecture

HAL – Hardware Abstraction Layer

FTQC - Fault Tolerant Quantum Computing

QEC – Quantum Error Correction

5 Overview

The description of the Hardware Abstraction Layer comprises the HAL software layer of the CEN/CENELEC Layer model for gate-based quantum computers [TR 18202:2025], as shown within the box in figure 1. It involves all transformations of instructions from higher layers that is needed for interworking with the control software layer(s) of one or more hardware stacks.

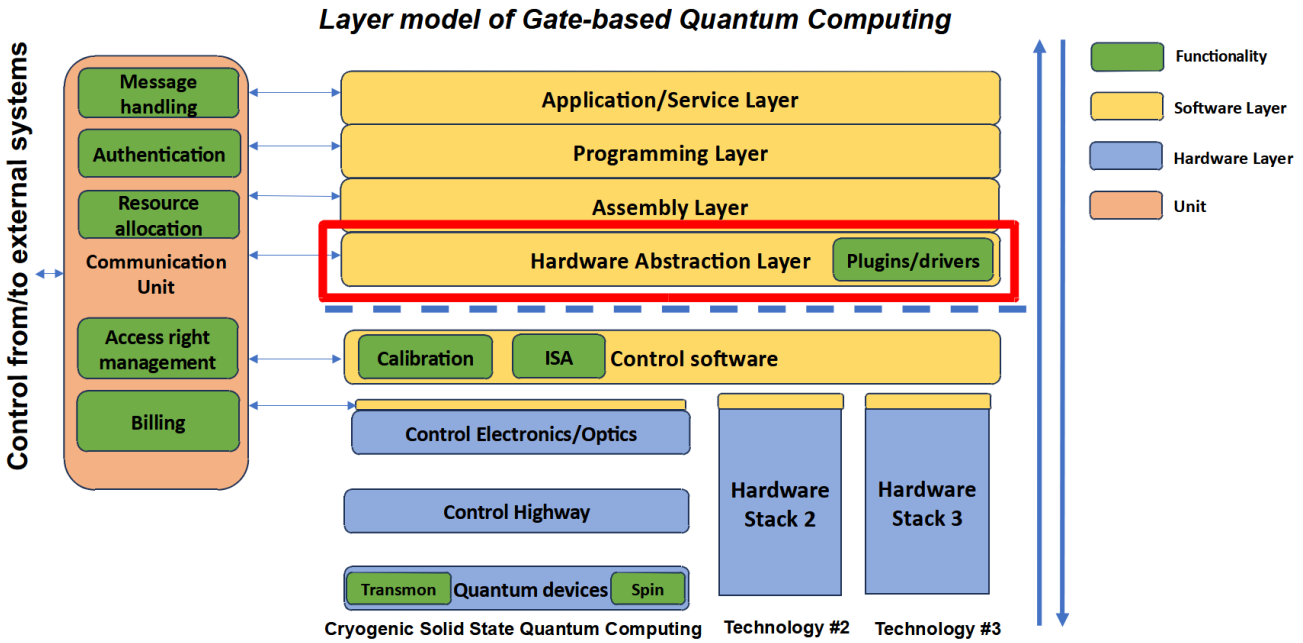

Figure 1 *- Overview of the layer model of quantum computing.*
Only the layer within the red box is within the scope of this document.

The primary aim of the Hardware Abstraction Layer is to hide many implementation-specific details of lower layers and to offer a more uniform interface to higher layers for instructing lower layers. This does not mean that higher layers are to be fully hardware agnostic of the underlying hardware and software. Optimising compilers, for instance, need dedicated knowledge on the hardware implementation to generate optimal code. In such cases, they can query the HAL about the capabilities and limitations of the underlying hardware and software. Once compiled, the HAL can be used to handle the entire exchange of queries, instructions and replies through a uniform interface.

By uniform we mean that implementations of higher layers can works for different quantum computing hardware platforms such as solid state quantum computing, ion traps, neutral atoms, optical quantum computing, topological quantum computing and so on. By queries we mean requests for information about number, organisation and performance of qubits, about the set of instructions understood by lower layers, about optional capabilities added to HAL or lower layers, and so on.

Where possible, the HAL will simply relay requests and instructions to lower layers. It depends on lower layers if it can relay directly or if it needs a simple format conversion into something understood by lower layers. But the HAL should also offer capabilities that are far more complicated and are to be implemented within the HAL itself. Examples are:

- Emulating multi-qubit gates, such as a Quantum Fourier Transform, as if they are single compound gates within the hardware.

- Scheduling multiple tasks submitted concurrently by multiple users via a queue of instructions. This can offer the perception to each user as if he is the only user of the quantum computer without being disturbed by other users.
- Emulating fault tolerant (software) qubits, by using the redundancy of many (hardware) qubits for quantum error correction mechanisms.
- Selecting between multiple quantum architectures, or even partition a single task into multiple queues, to perform calculations in parallel on multiple architectures.

Since these are just examples, the aim of the HAL is also to offer a mechanism for extending its functionality via "plug-in" libraries. This enables the support of all kinds of proprietary solutions from different vendors that are accessible via a uniform interface.

The text below gives a short overview of the kind of capabilities that should be supported by a HAL.

5.1 Queries about Qubits

The HAL should support the handling of several queries about how qubits are organised and how they perform. Most of these queries can be forwarded directly to lower layers, such as an ISA (Instruction Set Architecture) in the control software layer of a hardware platform. A conversion between a uniform and proprietary interfaces might be the only functionality that a HAL should add to offer this functionality.

Qubits are usually grouped into one or more quantum registers, so the HAL should report how they are organised. A quantum register is a system comprising multiple qubits, each with its own index. All available qubits can be fixed members of a single register, can be spread out over multiple (smaller) registers, or can be spread out over a user-definable grouping into registers. The HAL should at least support queries about the following characteristics:

- *Width*: The HAL can report the number of available qubits for each quantum register and how they are organised within these registers. It can also specify whether all qubits are part of a single quantum register or if they are allocated to multiple (smaller) registers. The use of multiple registers may occur when using modular hardware architectures.
- *Depth*: The HAL can report on qubit performance by means of the maximum depth for circuits of gates that can be executed before the calculated result becomes unreliable. This value is related to the coherence time of the implementation and other imperfections of the underlying hardware.
- *Connections*: The HAL can report 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 $\mathbf{C}$ has a size of $N \times N$. The default of each element in this matrix is false, but if qx and qy are the indices of two adjacent qubits, then $\mathbf{C}(qx,qy)=\mathbf{C}(qy,qx)=true$. Matrix $\mathbf{C}$ is therefore a symmetric matrix since $\mathbf{C}(k,r)=\mathbf{C}(r,k)$.

In addition, the HAL also supports instructions to operate on such registers for initialising, changing, and querying the state of the qubits.

5.2 Queries about supported gates

The HAL should be able to report information about all gates it supports. This can vary between providing a simple identifier that refers to a standardised list with names and definitions, between providing a list containing all these names and definitions or a mix of these two.

5.2.1 Reporting the definition of gates

The definition of a gate is simply a matrix describing the associated operation. A way to report such matrices is by means of returning a character string for each supported gate name. For instance:

```
X      = '[0,1;1,0]'
Y      = '[0,-j;+j,0]'
Rx(a)  = '[cos(a/2), -j*sin(a/2); -j*sin(a/2), cos(a/2)]'
```

This notation defines the matrix row by row, where each comma (,) separates the elements in each row and each semicolon (;) separates the rows. Identifier *j* or *i* refers to the imaginary unit.

Several gate names are commonly used, such as X, Y, Z, CNOT, and their definition might we well-known as well. But other gate names might be spelled differently, are less known or are only available on dedicated hardware platforms. This holds especially for gates where entanglement between two or more qubits is involved. Therefore it is highly recommended that a HAL can report all names and definitions of supported gates such that a compiler can account on which gates are supported and which not.

5.2.2 Reporting if gates are native or primitive

Another gate property that should be reported by the HAL is the distinction between *native* and *primitive* gates.

- The term *native* refers to an operation that changes the quantum state of a register by means of a "single" physical action on one or more qubits simultaneously. For instance, a rotation Rx(a) or Ry(b) in x or y direction by firing a single pulse.
- The term *primitive* gate refers to a sequential combination of native gates that is emulated as a single operation. For instance, a Z-gate that is implemented as a sequence of an Rx and an Ry gate.

In other words, if an operation is implemented by firing one or more simultaneous pulses, then it is a native gate. If two or more sequential pulses are required to achieve the desired operation, it is a primitive gate. As a result, a native gate can be executed in the minimum execution time.

Knowledge about which gates are native is relevant for quantum algorithms and/or optimising compilers that try to find an optimal circuit representation in terms of execution time. The shortest circuit description with well-known predefined gates Rx(a), Ry(b), Rz(c), X, Y, Z, H, S, T, and CNOT, may work well but can be less efficient in execution time than a circuit representation with native gates only.

If a gate is native or not is implementation-dependent. The boxed example in Table 2 illustrates, for a specific case, that:

- The gates X, Y, Rx(a), and Ry(b) are all native within that implementation.
- The gates Z and Rz(c) are primitive gates within that implementation since they are to be combined from two sequential native gates.

A similar example can be elaborated with two-qubit gates. For a specific implementation, a gate like CNOT may also be considered primitive when it cannot be implemented with a single native two-qubit gate. But this cannot be stated in general.

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 2 *- Example of a specific hardware implementation*

5.2.3 Reporting emulated gates

Gates that are not implemented in the ISA (within the control software layer) can be emulated by the HAL to support commonly used gates.

- Desired gates for one or two-qubits that are missing are often simple enough to be hard-coded in the HAL. For instance if a SWAP gate is not supported by the ISA, the HAL can define it instead by combining supported gates.
- Desired gates for more qubits that are missing may be more complicated. In those cases it might be better to implement them within a "plug-in" library, as explained in a succeeding chapter. An example might be the emulation of a "single" QFT gate that emulates a Quantum Fourier Transform for many qubits in the most optimised way for the involved hardware.

Since the full list of emulated gates is implementation dependent, it may also be obvious that the HAL should offer a mechanism to report such lists. Otherwise a compiler cannot count on it.

Note that those emulated gates for more qubits may also be called compound gates. There is no fundamental difference between gates that are primitive or compound; both can be called via a single instruction and both are not native.

5.3 Queries about quantum states via measurements

The HAL should support 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 bit-register. Note that a state will always collapse after such a query. The HAL should also support instructions to read out the bits in this bit-register and/or to use these bits for instructing controlled gates. If the hardware supports it, a HAL may also provide instructions to specify or change the basis for these measurements.

5.4 Interfacing considerations

The interface between HAL and the lower "control software" layer can be proprietary and can even be based on function calls. But this approach is not applicable for the interface between higher layers and the HAL. That interface has to be be based on a sequence of individual instructions, not on function calls. The preferred format for these instructions is binary, but the use of human readable ASCII instructions is not excluded.

The reason for this is that the HAL should be able to schedule multiple tasks submitted concurrently by multiple users via different instruction queues. This requires that the HAL should be able to detect where a long stream of instructions from user "A" can be interrupted to allocate time for handling instructions from user "B". Such a detection might be based on finding the instructions for resetting the state of all quantum registers of by inserting dedicated "token" instructions.

1 Quantum gates and Instructions

A quantum gate is a quantum operation that transforms the quantum state of one ore more qubits into another state. It can be considered as an elementary calculation (native, primitive or predefined in a library) within a quantum computation sequence.

A quantum instruction is a request or order to perform an executable step within a quantum program or task. Instructions are encoded, such as in a human-friendly ascii format (as common in assembly languages) or in a computer-friendly binary format (as in byte-code representations). These instructions have to be executed in a specific order, as dictated by some quantum algorithm..

5.5 Single qubit gates

Editor's Note:
This section may deal with naming and matrix definition of "standard" single qubit gates (like X, Y, Z, ..) that are supported by the ISA (in control software layer) or emulated within the HAL

5.6 Entangling gates

This section may deal with naming and matrix definition of "standard" gates that facilitates entangling between qubits (like CNOT, CZ, SWAP, ..) that are supported by the ISA (in control software layer) or emulated within the HAL

5.7 Instruction set

Editor's Note:
This section can have a lot of similarities with the ISA description in the project on "Cryogenic Solid State Quantum Computing"

6 Libraries

The HAL should support a software mechanism to let its functionality grow with future needs, and/or to improve existing functionalities. Rather than hard-coding vendor-specific logic in the HAL, the HAL should be able to load vendor-provided plug-ins (preferably in runtime) offering these new/improved functionalities.
This approach supports the integration of multiple vendor packages concurrently, and a convenient adaptation of the HAL to different hardware stacks. All without recompilation of higher layers. It also allows vendors to innovate independently while maintaining interoperability, and to provide users with a unified interface that abstracts hardware complexity.

The common software approach for offering this flexibility is the use of "plug-in" libraries. To make this happen the HAL should not only define the supported mechanism, but also a common way of internal interfacing to let the library interwork with other parts of the HAL. This means two aspects:

- A software mechanism to let libraries run, preferably by reusing existing mechanisms for dynamic linking of the underlying operating system. Examples are the use of the Dynamic

Linking Library (DLL) mechanism within Windows, or the use of Dynamic Shared Objects mechanism within Linux.

- A mechanism to tell the HAL that a library is operational, what its characteristics are (name, version, ..) and to intercept instructions that are to be handled by the inserted library.

An example of the latter is the interception of query instructions about capabilities so that new capabilities can be reported. Another example is that instructions for modifying the state of qubits become less error-prone after installing a new library with an improved Quantum Error Correction mechanism.

Each plug-in may encapsulate proprietary instruction translation, hardware control routines, and optimised algorithm libraries, allowing the HAL to dispatch calls dynamically based on the active hardware back-end. Plug-ins may be versioned and may be validated for integrity, ensuring compatibility and security.

Currently, most quantum software stacks rely on static linking in stead of dynamic linking. This means that vendor-specific logic and libraries are to be linked into the system at compile-time, and are to be implemented in the same programming language as the rest of the stack. This monolithic approach may simplify initial deployment but limits flexibility, especially as the ecosystem grows to include multiple hardware vendors and diverse programming environments.

The use of static linking is not excluded by the HAL, but in the future the use of dynamic linking will become increasingly important. By loading vendor plug-ins and libraries at run-time, the HAL can support heterogeneous hardware platforms without requiring recompilation of higher layers. This dynamic approach enables seamless integration with multiple programming languages, facilitates rapid updates, and allows vendors to innovate independently while maintaining interoperability.

The sub sections hereafter will discuss a few powerful libraries that are currently foreseen. These sub sections are examples only and may grow with future developments.

6.1 Library with Optimised Circuits

The library mechanism allows the HAL to offer vendor-specific optimised circuits to higher layers, which are fully tailored to the involved hardware in lower layers. For example, an optimised QFT (Quantum Fourier Transform) circuit provided by a hardware vendor is expected to be the best possible implementation of the QFT algorithm for the hardware of that vendor. Here, best means the one that can be executed with the minimum amount of execution time. Thus, it is expected that the optimised circuit is composed by native gates supported by the underlying hardware. We remind that 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, as explained in chapter 5.

Circuits that are frequently used as building blocks for applications, are:

- Quantum Fourier Transform - QFT
- Phase Estimation
- Grover Operator
- Logical operators (AND, OR, XOR, etc.)
- Bit-Flip Oracle Gate

- Diagonal Gate
- (circuit for preparing a) Graph State
- Phase Oracle

These circuits are suitable candidates for being provided in a library as vendor-specific optimised circuits.

6.2 Library for Fault Tolerant Quantum Computing (FTQC)

6.3 Library for Mapping and Routing

7 Virtualisation and resource management

7.1 Multi-User Task Scheduling and Cooperative Execution

7.2 Partitioned Sequences and Instruction-Flow Boundaries

Bibliography

[1] *A first reference*