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

Convenor: **PAUL Alexandra MME**

Nxx_HAL_Overview

Document type	Related content	Document date	Expected action
Project / Draft	Meeting: VIRTUAL 11 Feb 2026	2026-02-06	**INFO**

Description

Dear members,

Please find attached an overview of the HAL foreseen WI.

Best,

Simon

An overview of several HAL functionalities

Draft 01

Date of submission:	2026-02-xx
Submitted by:	Rob F.M. van den Brink (Rob.vandenBrink@Delft-Circuits.com) Michele Amoretti Juan Boschero
Expected action:	Vote
Expected date:	2026-02-11 (JTC22/WG3 meeting)
WG3-Project:	Hardware Abstraction Layer

---

1. Abstract

Document N201 gives a first Draft for a Technical Specification (TS) about the hardware abstraction layer. This first draft is currently a preliminary table of contents only.

The present contribution proposes literal text for inclusion into chapter 5 of that first draft, which is dedicated to a rough overview of functionalities supported by a HAL. The text is inspired on what has been written about it in the Technical Report (TR) on the "Layer Model".

2. Literal text proposal

Start of literal text proposal

5. Overview

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. Optimizing 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, organization 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 organized 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 organized. 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 organized 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 **C** has a size of *N* x *N*. 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*).

In addition, the HAL also supports instructions to operate on such registers for initializing, 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 standardized list with names and definitions, between providing a list containing all these names and definitions or a mix of these two.

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.

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 optimizing 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*

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 optimized 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 allways 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.

End of literal text proposal