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

Convenor: **PAUL Alexandra MME**

HAL_requirements_Draft03

Document type	Related content	Document date	Expected action
Project / Draft	Meeting: Delft (Netherlands) 29 Apr 2026	2026-04-23	

Description

Dear members,

Please find attached the latest draft of the WI "HAL".

Kind regards,

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 03, 2026-04-23

CCMC will prepare and attach the official title page.

Contents

Page

European foreword ... 3
Introduction ... 4
1 Scope ... 5
2 Normative references ... 5
3 Terms and definitions ... 6
4 Abbreviations ... 6
5 Overview ... 7
5.1 Queries about qubits ... 8
5.2 Queries about supported gates ... 8
5.3 Queries about quantum states via measurements ... 11
5.4 Interfacing considerations ... 11
6 Quantum instructions and gates ... 12
6.1 Instructions ... 12
6.2 Quantum gates ... 12
7 Libraries ... 19
7.1 Library with Optimised Circuits ... 20
7.2 Library for Fault Tolerant Quantum Computing (FTQC) ... 20
7.3 Library for Mapping and Routing ... 21
8 Resource management ... 22
8.1 The concept of partition sequences ... 22
8.2 The concept of checkpointing ... 23
Bibliography ... 24

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 Standardisation 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 5-1. 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.

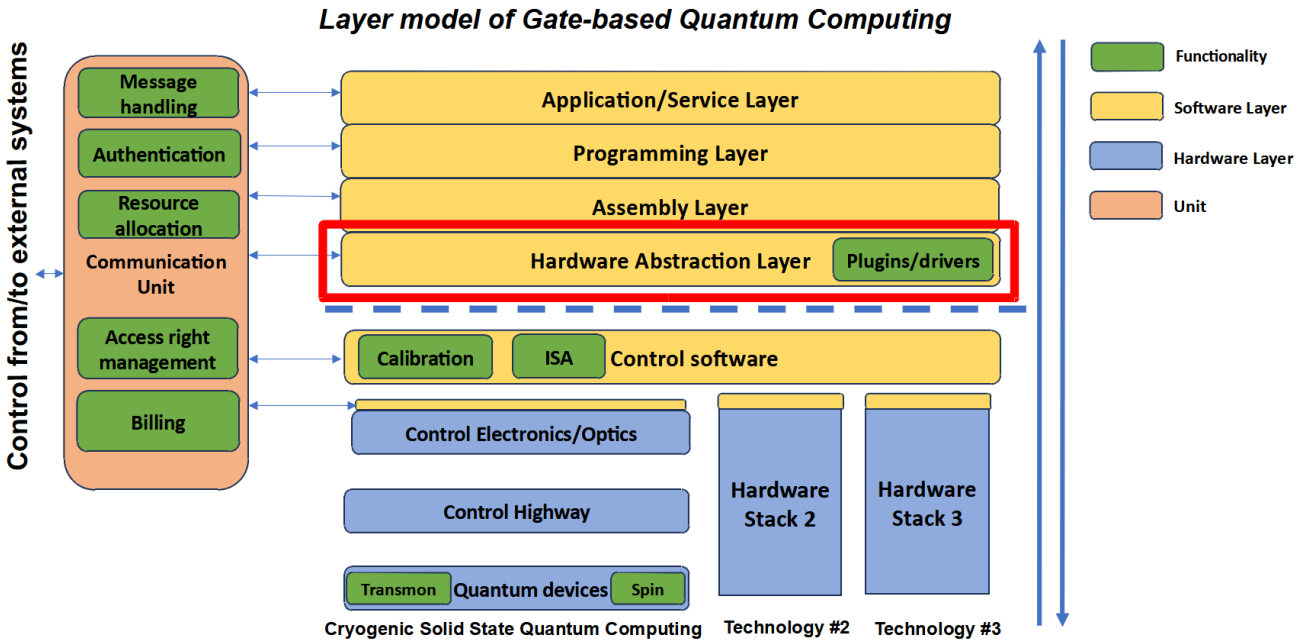


Figure 5-1 *- Overview of the layer model of quantum computing.*
Only the layer within the red box is within the scope of this document.

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 5-2 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.

Table 5-2 *- 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.

6 Quantum instructions and gates

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..

6.1 Instructions

Editor's Note:
This section may use significant portions of what has been written about the ISA in the control software layer. See the project on "Cryogenic Solid State Quantum Computers"

6.2 Quantum gates

Quantum gates are operators that operate one or more qubits to change the overall quantum state. These operations preserve the total probability of quantum states (the norm of the state vector remains 1) and can be described via matrix operations. These matrices **Q** are always unitary, and have the property that the modulus of their determinant $|det(\mathbf{Q})|=1$.

Quantum gates acting on n qubits require a multiplication with 2n× 2n unitary matrices. Both (classical) matrix multiplications (division) and Kronecker products of unitary matrices result in unitary matrices, so the norm of the state vector is preserved.
A convenient way to define the unitary matrix **Q** associated with a gate in a consistent manner is by means of matrix functions like *expm*(**Q**), *logm*(**Q**) and *sqrtm*(**Q**). and other matrix operations.

Via standard eigenvalue decompositions, these Q-matrices can always be decomposed as $\mathbf{Q}=\mathbf{V}\times diag(\mathbf{d})/\mathbf{V}$, where **Q** and **V** are square matrices and **d** a column vector. This property makes the evaluation of matrix functions like *expm*(**Q**), *logm*(**Q**) and *sqrtm*(**Q**) relatively simple. The used functions and symbols are defined as:

- matrix exponent: $expm(\mathbf{Q}) = expm(\mathbf{V}\times diag(\mathbf{d})/\mathbf{V}) = \mathbf{V}\times diag(exp(\mathbf{d}))/\mathbf{V}$
- matrix logarithm: $logm(\mathbf{Q}) = logm(\mathbf{V}\times diag(\mathbf{d})/\mathbf{V}) = \mathbf{V}\times diag(log(\mathbf{d}))/\mathbf{V}$
- matrix square root: $sqrtm(\mathbf{Q}) = sqrtm(\mathbf{V}\times diag(\mathbf{d})/\mathbf{V}) = \mathbf{V}\times diag(sqrt(\mathbf{d}))/\mathbf{V}$
- Kronecker product: $\mathbf{X} \otimes \mathbf{Y}$
- matrix product: $\mathbf{X} \times \mathbf{Y}$
- matrix division: $\mathbf{X}/\mathbf{Y}$ (*right hand division*), $\mathbf{X}\backslash\mathbf{Y}$ (*left hand division*)
- imaginary unit: $j = \sqrt{-1}$

Note that the symbols *i* and *j* refer to the same imaginary unit but *i* and 1 are sometimes too look-alike in expressions that it can be confusing. Therefore *j* is used within this document.

6.2.1 Naming conventions for gates within this document

The HAL supports a collection of predefined gates, which can be invoked via a unique name in a gate instruction. Some of these gates are directly provided by the control software layer, while others are

emulated within the HAL in order to harmonize gate-sets among different quantum computer implementations.

Some de-facto consensus has been achieved on names and definitions of commonly used gates. This holds for well known gates like X, Y, Z, cNOT, cZ, SWAP, etc, but not for all gates being defined within different implementations. Sometimes the same name is used for different gates, and in other cases different names are used for identical gates. It may be obvious that this can be confusing and error-prone.

In order to harmonize naming and definitions among different HAL implementations, names and definitions of several pre-defined gates are provided in succeeding sections. There is no need that the HAL should support all these gates, nor should it be restricted to these gates. Therefore a HAL implementation should support a mechanism to let higher layers query the names of gates that are pre-defined and how they are defined.

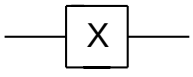
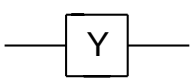
The acceptance of strings with names of predefined gates is case-insensitive. This means that the HAL treats strings like 'cNOT' and 'CNOT' and 'cnot' allways as names for the same predefined gate. The mixed use of lower and upper case in these name is only a matter of style.
The following style convention has been used in this document:

- The base-line name is the fragment of a full name that starts with uppercase characters, and may be followed by a suffix with lowercase characters. For instance base-line names like X, NOT, and SWAP. A suffix with lowercase characters is used when the same expression is used for defining different gates, but differ only on the arguments they apply on. So Rx, Rzx, Rz are examples of such base-line names. Sdg and Tdg is an exception due to historical reasons.
- The full name of an identifier is a base-line name that can be prefixed by lowercase characters. Such a prefix is only used when the gate is derived from another gate by means of a generic matrix function that can operate on any unitary matrix. So "rQ" for *sqrtm*(**Q**) and "cQ" for *controlled*(**Q**). As such, the prefix characters 'r' and 'c' are acting like a modifier from a base-line gate.
- Sometimes a gate has a similarity with another gate but cannot be derived from it via a generic matrix function. In that case it is not a modifier and is written with uppercase characters. For instance SWAP and ISWAP.

The tables in following sections define gates and their associated names, definitions, matrices and symbols.

6.2.2 Single qubit gates

Table 6-1 defines a list with names and definition of various pre-defined gates that operate on a single qubit.

Name	Definition	Matrix	Symbol	Properties
I	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$			$= -\mathbf{X} \times \mathbf{Y} \times \mathbf{Z}$
X	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$			$= j \cdot \mathbf{Rx}(\pi)$
Y	$\begin{bmatrix} 0 & -j \\ j & 0 \end{bmatrix}$			$= j \cdot \mathbf{Ry}(\pi)$

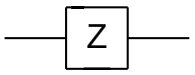
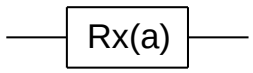
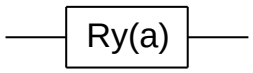
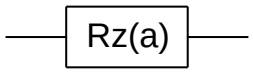
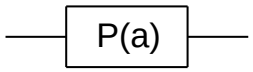
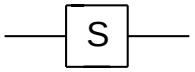
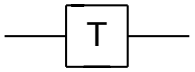
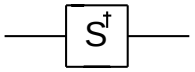
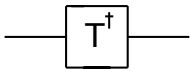
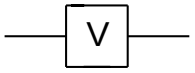
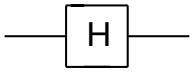
Z	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$			$= j \cdot \mathbf{Rz}(\pi)$
Rx(a)	$= expm(-j \cdot a/2 \cdot \mathbf{X})$	$\begin{bmatrix} cos(a/2) & -j \cdot sin(a/2) \\ -j \cdot sin(a/2) & cos(a/2) \end{bmatrix}$		
Ry(a)	$= expm(-j \cdot a/2 \cdot \mathbf{Y})$	$\begin{bmatrix} cos(a/2) & -sin(a/2) \\ sin(a/2) & cos(a/2) \end{bmatrix}$		
Rz(a)	$= expm(-j \cdot a/2 \cdot \mathbf{Z})$	$\begin{bmatrix} exp(-j \cdot a/2) & 0 \\ 0 & exp(j \cdot a/2) \end{bmatrix}$		
P(a)	$= expm(-j \cdot a/2 \cdot (\mathbf{Z} - \mathbf{I}))$	$\begin{bmatrix} 1 & 0 \\ 0 & exp(j \cdot a) \end{bmatrix}$		
S	$= expm(-j\pi/4 \cdot (\mathbf{Z} - \mathbf{I}))$	$\begin{bmatrix} 1 & 0 \\ 0 & j \end{bmatrix}$		$= \mathbf{P}(\pi/2)$ $= sqrtm(\mathbf{Z})$
T	$= expm(-j\pi/8 \cdot (\mathbf{Z} - \mathbf{I}))$	$\begin{bmatrix} 1 & 0 \\ 0 & (1+j)/\sqrt{2} \end{bmatrix}$		$= \mathbf{P}(\pi/4)$ $= sqrtm(\mathbf{S})$
Sdg	$= expm(+j\pi/4 \cdot (\mathbf{Z} - \mathbf{I}))$	$\begin{bmatrix} 1 & 0 \\ 0 & -j \end{bmatrix}$		$= \mathbf{P}(-\pi/2)$
Tdg	$= expm(+j\pi/8 \cdot (\mathbf{Z} - \mathbf{I}))$	$\begin{bmatrix} 1 & 0 \\ 0 & (1-j)/\sqrt{2} \end{bmatrix}$		$= \mathbf{P}(-\pi/4)$
V	$= expm(-j\pi/4 \cdot (\mathbf{X} - \mathbf{I}))$	$\begin{bmatrix} 1+j & 1-j \\ 1-j & 1+j \end{bmatrix}/2$		$= sqrt(j) \cdot \mathbf{Rx}(\pi/2)$ $= sqrtm(\mathbf{X})$
H	$= j \cdot \mathbf{Rx}(\pi) \times \mathbf{Ry}(\pi/2)$	$\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}/\sqrt{2}$		$= \mathbf{X} \times sqrtm(-j\mathbf{Y})$ $= \mathbf{Z} \times sqrtm(+j\mathbf{Y})$ $= sqrtm(-j\mathbf{Y}) \times \mathbf{Z}$ $= sqrtm(+j\mathbf{Y}) \times \mathbf{X}$
NOT	same as **X**			$= \mathbf{X}$
rNOT	same as **V**			$= sqrtm(\mathbf{NOT})$ $= sqrtm(\mathbf{X})$

Table 6-1 *- List with pre-defined gates operating on a single qubit*

6.2.3 Double qubit gates

Table 6-2 to 6-4 define lists with names and definition of various pre-defined gates that operate on two qubits.

Name	Definition	Matrix	Properties
Rxx(a)	$= expm(-j \cdot a/2 \cdot \mathbf{X} \otimes \mathbf{X})$	$\begin{bmatrix} cos(a/2) & 0 & 0 & -j \cdot sin(a/2) \\ 0 & cos(a/2) & -j \cdot sin(a/2) & 0 \\ 0 & -j \cdot sin(a/2) & cos(a/2) & 0 \\ -j \cdot sin(a/2) & 0 & 0 & cos(a/2) \end{bmatrix}$	$\mathbf{X} \otimes \mathbf{X} = j \cdot \mathbf{Rxx}(\pi)$
Ryy(a)	$= expm(-j \cdot a/2 \cdot \mathbf{Y} \otimes \mathbf{Y})$	$\begin{bmatrix} cos(a/2) & 0 & 0 & j \cdot sin(a/2) \\ 0 & cos(a/2) & -j \cdot sin(a/2) & 0 \\ 0 & -j \cdot sin(a/2) & cos(a/2) & 0 \\ j \cdot sin(a/2) & 0 & 0 & cos(a/2) \end{bmatrix}$	$\mathbf{Y} \otimes \mathbf{Y} = j \cdot \mathbf{Ryy}(\pi)$
Rzz(a)	$= expm(-j \cdot a/2 \cdot \mathbf{Z} \otimes \mathbf{Z})$	$\begin{bmatrix} exp(-j \cdot a/2) & 0 & 0 & 0 \\ 0 & exp(j \cdot a/2) & 0 & 0 \\ 0 & 0 & exp(j \cdot a/2) & 0 \\ 0 & 0 & 0 & exp(-j \cdot a/2) \end{bmatrix}$	$\mathbf{Z} \otimes \mathbf{Z} = j \cdot \mathbf{Rzz}(\pi)$
Rxy(a)	$= expm(-j \cdot a/2 \cdot \mathbf{X} \otimes \mathbf{Y})$	$\begin{bmatrix} cos(a/2) & 0 & 0 & -sin(a/2) \\ 0 & cos(a/2) & sin(a/2) & 0 \\ 0 & -sin(a/2) & cos(a/2) & 0 \\ sin(a/2) & 0 & 0 & cos(a/2) \end{bmatrix}$	$\mathbf{X} \otimes \mathbf{Y} = j \cdot \mathbf{Rxy}(\pi)$
Ryx(a)	$= expm(-j \cdot a/2 \cdot \mathbf{Y} \otimes \mathbf{X})$	$\begin{bmatrix} cos(a/2) & 0 & 0 & -sin(a/2) \\ 0 & cos(a/2) & -sin(a/2) & 0 \\ 0 & sin(a/2) & cos(a/2) & 0 \\ sin(a/2) & 0 & 0 & cos(a/2) \end{bmatrix}$	$\mathbf{Y} \otimes \mathbf{X} = j \cdot \mathbf{Ryx}(\pi)$
Rzx(a)	$= expm(-j \cdot a/2 \cdot \mathbf{Z} \otimes \mathbf{X})$	$\begin{bmatrix} cos(a/2) & -j \cdot sin(a/2) & 0 & 0 \\ -j \cdot sin(a/2) & cos(a/2) & 0 & 0 \\ 0 & 0 & cos(a/2) & j \cdot sin(a/2) \\ 0 & 0 & j \cdot sin(a/2) & cos(a/2) \end{bmatrix}$	$\mathbf{Z} \otimes \mathbf{X} = j \cdot \mathbf{Rzx}(\pi)$
Rzy(a)	$= expm(-j \cdot a/2 \cdot \mathbf{Z} \otimes \mathbf{Y})$	$\begin{bmatrix} cos(a/2) & -sin(a/2) & 0 & 0 \\ sin(a/2) & cos(a/2) & 0 & 0 \\ 0 & 0 & cos(a/2) & sin(a/2) \\ 0 & 0 & -sin(a/2) & cos(a/2) \end{bmatrix}$	$\mathbf{Z} \otimes \mathbf{Y} = j \cdot \mathbf{Rzy}(\pi)$
Rxz(a)	$= expm(-j \cdot a/2 \cdot \mathbf{X} \otimes \mathbf{Z})$	$\begin{bmatrix} cos(a/2) & 0 & -j \cdot sin(a/2) & 0 \\ 0 & cos(a/2) & 0 & j \cdot sin(a/2) \\ -j \cdot sin(a/2) & 0 & cos(a/2) & 0 \\ 0 & j \cdot sin(a/2) & 0 & cos(a/2) \end{bmatrix}$	$\mathbf{X} \otimes \mathbf{Z} = j \cdot \mathbf{Rxz}(\pi)$
Ryz(a)	$= expm(-j \cdot a/2 \cdot \mathbf{Y} \otimes \mathbf{Z})$	$\begin{bmatrix} cos(a/2) & 0 & -sin(a/2) & 0 \\ 0 & cos(a/2) & 0 & sin(a/2) \\ sin(a/2) & 0 & cos(a/2) & 0 \\ 0 & -sin(a/2) & 0 & cos(a/2) \end{bmatrix}$	$\mathbf{Y} \otimes \mathbf{Z} = j \cdot \mathbf{Ryz}(\pi)$

***Table 6-2**- List with pre-defined two-qubit rotation gates*

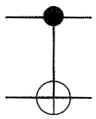
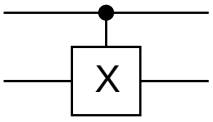
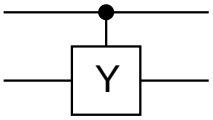
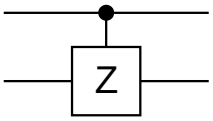
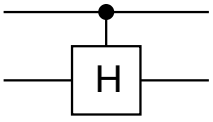
name	definition	matrix	symbols
		$\mathbf{C} \equiv \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$	**C** is not a gate but an auxiliary matrix
cNOT	same as **cX**	same as **cX**	
cX	$= expm(\mathbf{C} \otimes logm(\mathbf{X}))$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$	
cY	$= expm(\mathbf{C} \otimes logm(\mathbf{Y}))$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -j \\ 0 & 0 & j & 0 \end{bmatrix}$	
cZ	$= expm(\mathbf{C} \otimes logm(\mathbf{Z}))$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$	
cS	$= expm(\mathbf{C} \otimes logm(\mathbf{S}))$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & j \end{bmatrix}$	
cSdg	$= expm(\mathbf{C} \otimes logm(\mathbf{Sdg}))$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -j \end{bmatrix}$	
cT	$= expm(\mathbf{C} \otimes logm(\mathbf{T}))$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & (1+j)/\sqrt{2} \end{bmatrix}$	
cTdg	$= expm(\mathbf{C} \otimes logm(\mathbf{Tdg}))$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & (1-j)/\sqrt{2} \end{bmatrix}$	
cH	$= expm(\mathbf{C} \otimes logm(\mathbf{H}))$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \sqrt{1/2} & \sqrt{1/2} \\ 0 & 0 & \sqrt{1/2} & -\sqrt{1/2} \end{bmatrix}$	
cP	$= expm(\mathbf{C} \otimes logm(\mathbf{P}))$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & exp(j \cdot a) \end{bmatrix}$	

Table 6-3 - *List with pre-defined controlled gates operating on two qubits*

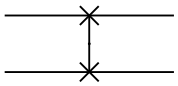
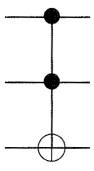
Name	definition + matrix	Properties + Symbol
SWAP	$= expm(-j\pi/4 \cdot (\mathbf{X} \otimes \mathbf{X} + \mathbf{Y} \otimes \mathbf{Y} + \mathbf{Z} \otimes \mathbf{Z} - \mathbf{I} \otimes \mathbf{I}))$ $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	
rSWAP	$= expm(-j\pi/8 \cdot (\mathbf{X} \otimes \mathbf{X} + \mathbf{Y} \otimes \mathbf{Y} + \mathbf{Z} \otimes \mathbf{Z} - \mathbf{I} \otimes \mathbf{I}))$ $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & (1+j)/2 & (1-j)/2 & 0 \\ 0 & (1-j)/2 & (1+j)/2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$= sqrtm(\mathbf{SWAP})$
ISWAP	$= expm(-j\pi/4 \cdot (\mathbf{X} \otimes \mathbf{X} + \mathbf{Y} \otimes \mathbf{Y}))$ $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & j & 0 \\ 0 & j & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	
rISWAP	$= expm(-j\pi/8 \cdot (\mathbf{X} \otimes \mathbf{X} + \mathbf{Y} \otimes \mathbf{Y}))$ $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1/\sqrt{2} & j/\sqrt{2} & 0 \\ 0 & j/\sqrt{2} & 1/\sqrt{2} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$= sqrtm(\mathbf{ISWAP})$

Table 6-4 *- List with various other pre-defined gates operating on two qubits*

6.2.4 Multiple qubit gates

Table 6-5 defines a list with names and definition of various pre-defined gates that operate on more than two qubits. The systematic used within the expressions used for defining them illustrate how to extend this list in case a gate is needed with multiple controls.

Name	definition + matrix	Properties + Symbol
	$\mathbf{C} \equiv \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$	**C** is not a gate but an auxiliary matrix
ccNOT	same as **ccX**	
ccX	$= expm(\mathbf{C} \otimes \mathbf{C} \otimes logm(\mathbf{X}))$ (*also known as Toffoli gate*)	

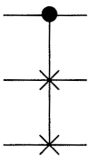
	$\begin{bmatrix} 1&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 0&0&0&1&0&0&0&0\\ 0&0&0&0&1&0&0&0\\ 0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&0&1\\ 0&0&0&0&0&0&1&0 \end{bmatrix}$	
cSWAP	$= expm(\mathbf{C} \otimes logm(\mathbf{SWAP}))$, (*also known as the Fredkin gate*) $\begin{bmatrix} 1&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 0&0&0&1&0&0&0&0\\ 0&0&0&0&1&0&0&0\\ 0&0&0&0&0&0&1&0\\ 0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&0&1 \end{bmatrix}$	
crSWAP **rcSWAP**	$= expm(\mathbf{C} \otimes logm(\mathbf{rSWAP}))$ $= expm(\mathbf{C} \otimes logm(\mathbf{SWAP})/2)$ $\begin{bmatrix} 1&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 0&0&0&1&0&0&0&0\\ 0&0&0&0&1&0&0&0\\ 0&0&0&0&0&(1+j)/2&(1-j)/2&0\\ 0&0&0&0&0&(1-j)/2&(1+j)/2&0\\ 0&0&0&0&0&0&0&1 \end{bmatrix}$	crSWAP and rcSWAP give the same results $= sqrtm(\mathbf{cSWAP})$
cISWAP	$= expm(\mathbf{C} \otimes logm(\mathbf{iSWAP}))$ $\begin{bmatrix} 1&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 0&0&0&1&0&0&0&0\\ 0&0&0&0&1&0&0&0\\ 0&0&0&0&0&0&j&0\\ 0&0&0&0&0&j&0&0\\ 0&0&0&0&0&0&0&1 \end{bmatrix}$	
crISWAP **rcISWAP**	$= expm(\mathbf{C} \otimes logm(\mathbf{riSWAP}))$ $= expm(\mathbf{C} \otimes logm(\mathbf{iSWAP})/2)$ $\begin{bmatrix} 1&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 0&0&0&1&0&0&0&0\\ 0&0&0&0&1&0&0&0\\ 0&0&0&0&0&1/\sqrt{2}&j/\sqrt{2}&0\\ 0&0&0&0&0&j/\sqrt{2}&1/\sqrt{2}&0\\ 0&0&0&0&0&0&0&1 \end{bmatrix}$	crISWAP and rcISWAP give the same results $= sqrtm(\mathbf{cISWAP})$

Table 6-5 *- List with various other pre-defined gates operating on three qubits*

7 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.

7.1 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.

7.2 Library for Fault Tolerant Quantum Computing (FTQC)

A fault-tolerant quantum computation (FTQC) library of the HAL is responsible for abstracting fault-tolerant quantum computation (FTQC) functionalities, including quantum error correction (QEC) functionalities, so that higher layers can operate on the logical level without requiring hardware-specific and FTQC-architecture specific knowledge.

The HAL may support multiple FTQC architectures with multiple QEC codes, such as surface codes, concatenated codes, or other vendor-defined approaches. The FTQC library may specify and organise the placement of magic state factories, buses and data qubits of an FTQC architecture in a static or dynamic manner. Alternatively, the FTQC library may specify a code switching strategy for implementing different fault-tolerant gates. The FTQC library may provide a standardised interface for logical gate sets and configuration parameters.

- Logical gate sets comprise logical gates which can be implemented with a chosen FTQC architecture and QEC code. A typical logical gate set may comprise Clifford gates and T gates, when the library obtained the information that magic T-state distillation is feasible on the hardware platform, for instance.
- Configuration parameters may comprise FTQC parameters such as an architecture choice, the current state of a dynamic architecture, magic state fidelities, and the like. Furthermore, configuration parameters may comprise QEC parameters such as error rates, thresholds, and syndrome extraction capabilities.

The logical gate set and the configuration parameters enable compilers and schedulers in higher layers to make informed decisions about circuit optimisation and resource allocation. The HAL should allow dynamic FTQC and QEC configuration based on real-time hardware feedback, such as changes in qubit fidelity or calibration status, ensuring that the employed strategies remain adaptive and efficient. By centralising FTQC organisation and reporting, the HAL ensures consistent execution across diverse hardware platforms, while maintaining transparency for higher layers that need to optimise performance without managing low-level fault-tolerant gate and measurement implementation, error handling and control.

FTQC and QEC functionality introduces logical qubits, which are constructed from multiple physical qubits to provide greater robustness against noise and operational errors.

- *Physical qubits* represent the raw hardware-level quantum states, which are highly susceptible to decoherence and gate errors.
- *Logical qubits* combine many physical qubits into a single error-protected unit, or logical qubit, allowing the use of the qubit for robust computation.

To support FTQC and QEC more efficiently, the control software layer may be able to handle queries pre-defined by the ISA, such as pre-defined operations for syndrome extraction and logical gate application.

7.3 Library for Mapping and Routing

8 Resource management

The HAL incorporates resource management dynamically by allocating qubits, memory buffers, and timing channels based on user requirements and system constraints. It also implements scheduling policies that optimise for different parameters, throughput, and latency, allowing concurrent execution of multiple workloads without compromising performance. Virtualisation ensures that each virtual instance behaves as an independent quantum system. By abstracting these complexities, the HAL provides a seamless experience for higher layers, enabling multi-user operation, cloud-based quantum services and hybrid computing environments to scale securely and efficiently.

A job refers to an instruction flow that contain the complete execution of an application of a single user. This could be for instance a single fixed circuit that has to run many times or a hybrid quantum application involving sequential or interleaved quantum program calls. This enables execution time to be shared across multiple users, as jobs can be temporarily paused to allow execution of another user's job. Therefore the resource manager has the following means to cut in jobs

- ***Pausing*** a job to hand-over execution time to another user. Such an interruption will not destroy the quantum calculation.
- ***Breaking*** a job when it takes too long before a convenient moment is found to pause the job. Such an interruption will destroy parts of a quantum calculation. and the resource manager should raise a fault to a higher layer so that it can redo the broken calculation at a later moment in time.
- ***Killing*** a job when it exceeds agreed allocated time or resource quota, or a higher layer has explicitly asked for it.

To facilitate the pausing of a job, the resource manager must have a robust mechanism to identify when it can pause a job, without destroying ongoing quantum calculations.

8.1 The concept of partition sequences

To enable fair and efficient multi-user operation, the resource manager within the HAL supports the concept of *partition sequences.* These are logical blocks of instructions treated as atomic units after which a quantum calculation can be paused without harm and without compromising correctness or isolation between jobs from different users.

For example, if a user runs a job with 100,000 shots, or executions of a circuit, then a partition can be allocated to each shot. Assume that hardware calibration is required every 1,000 shots, then the resource manager can pause the job after 1000 partitions, allow a lower layer to perform a warm-start calibration, and then resume execution from the next partition. Something similar occurs when jobs from multiple users are running "simultaneously".

This mechanism is known as *snapshotting*. Snapshotting enables partitions from different users' partition sequences to be interleaved and scheduled according to priority, thereby ensuring fairness and maximising throughput under *fair-share scheduling.* When the resource manager decides to hand-over execution time to another job it waits a reasonable amount of time for detecting the end of a partition. If detected in time, it will pause that job, but if it takes too long the resource manager will break that job, as explained before. Pausing means that the execution context of a job is preserved, but they relinquish control of the hardware until the scheduler returns them to the top of the priority queue.

This decision making is controlled by priority, derived from the number of concurrent users waiting for execution time or from an interrupt request received from a lower layer. For instance when a lower layer needs to run a warm start calibration or even a system restart.

Figure 8-1 shows an example of how the resource manager within the HAL may process partition sequences from multiple users. It depends on the priority of other jobs how many partition sequences may run without being paused. Sometimes it can be many partitions, and sometimes pausing will occur directly after a first partition.

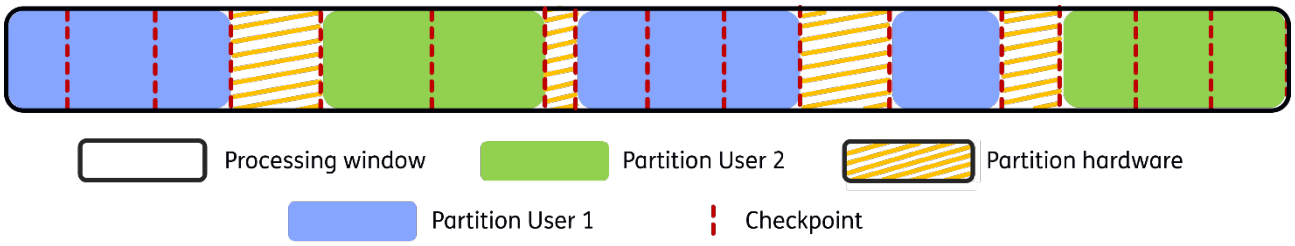


***Figure 8-1**: Representation of a processing window that can be subdivided into scheduled partitions with checkpoints for multiple users.*

This mechanism combines dynamic scheduling and resource management, allowing concurrent workloads to share hardware efficiently while maintaining isolation and performance guarantees. Although the HAL's current focus is on abstractions for quantum hardware, its architecture intentionally does not exclude classical high-performance computing components. The HAL is conceptualised as a collection of extensible libraries, and nothing prevents the inclusion of HPC-oriented modules, for example, routing to GPU clusters for simulation or pre-processing, or coordinating with distributed classical workflows. However, these capabilities remain outside the present scope of the HAL's core definition.

Future iterations may integrate HPC more deeply as hybrid quantum-classical pipelines become increasingly important, but for now, the HAL concentrates on providing robust, secure, and scalable interfaces to quantum hardware itself.

8.2 The concept of checkpointing

It may be obvious that the resource manager cannot always automatically detect when a user completes a subprocess or reaches a natural yield point marking the end of a partition. There are exceptions, such as when instructions are encountered that reset all quantum registers. These instructions are useful for indicating the start of a new partition. However, if they do not occur frequently enough, the resource manager must interrupt the job and defer further handling to higher layers.

The solution for preventing that is the use of *checkpointing* by insertion of so called *checkpoint instructions.* These are explicit no-operation instructions, with the only purpose to indicate where a job can be paused. In other words, to demarcate the end of a partition.

It is the responsibility of a higher layer, such as a compiler, to insert sufficient checkpoint instructions. Otherwise it must accept the risk that a job will suddenly be broken by the resource manager and that (parts of) the quantum calculation gets lost.

Natural places where a compiler can insert checkpoint instructions is at loop iterations, measurement batches, and synchronisation points.

Bibliography

[1] *A first reference*