Quantum HAL Specifications

Release dev

October 26, 2021

List of Tables

Chapter 1

Scope and Purpose

This document sets out a Hardware Abstraction Layer (HAL) for quantum computers based on four leading qubit technologies: superconducting qubits, trapped-ion qubits, photonic systems and silicon-based qubits. The aim is to define a multi-level HAL that makes software portable across platforms but not at the cost of performance. The HAL allows high-level quantum computer users, such as application developers, platform and system software engineers, cross-platform software architects, to abstract away the hardware implementation details while keeping the performance.

This document defines the HAL levels, categorised by the types of applications that they enable. The definition includes the general HAL architecture, HAL features (e.g. which commands need to be implemented) and the HAL specification format. The document does not define the HAL implementation or how to compile/transpile between the different levels. This document is a part of the NISQ.OS ISCF project as a collaborative effort of ARM, Duality Quantum Photonics, Hitachi Europe Limited, the National Physical Laboratory, Oxford Ionics, Oxford Quantum Circuits, Riverlane, Seeqc, and Universal Quantum.

This joint project's commitment is to implement applications that require the fastest classical/quantum interaction, such as measurement and control-based applications and error correction. Deltaflow.OS, the operating system for quantum computers which will be developed within the ISCF NISQ.OS project, builds on this open HAL specification.

The HAL is to be an open standard on which other parties can also build. One aim of the ISCF project is to engage in international standardisation efforts with this HAL.

Table 1.1. Contributors

Company/Entity
ARM
Duality Quantum Photonics (DQP)
Hitachi Europe Ltd (HEU)
National Physical Laboratory (NPL)
Oxford Ionics (OI)
Oxford Quantum Circuits (OQC)
Riverlane (RL)
Seeqc
Universal Quantum (UQ)

This specification must be considered a work in progress. The document is currently used to guide discovery, initiate discussion and enable future improvements. Even though all the parties involved are putting their best efforts on verifying the validity and correctness of what is stated, extensive reviews are still to be conducted. This disclaimer will be removed once the document reaches sufficient maturity.

Chapter 2

Glossary and Abbreviations

Table 2.1. Glossary

Term	Definition/Description
SHALL	This word, or the terms "REQUIRED" or "MUST", mean that the definition is an absolute requirement of the specification [1].
SHOULD	This word, or the adjective "RECOMMENDED", mean that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course [1].
MAY	This word, or the adjective "OPTIONAL", mean that an item is truly optional. One vendor may choose to include the item because a particular marketplace requires it or because the vendor feels that it enhances the product while another vendor may omit the same item. An implementation which does not include a particular option MUST be prepared to interoperate with another implementation which does include the option, though perhaps with reduced functionality. In the same vein an implementation which does include a particular option MUST be prepared to interoperate with another implementation which does not include the option (except, of course, for the feature the option provides) [1].
Northbound interface	In Computer Networking and Computer Architecture, a northbound interface of a component is an interface that allows the component to communicate to a higher-level component, using the latter component's southbound interface.
Transpiler	Transpiling is a specific term for taking source code written in one language and transforming into another language that has a similar level of abstraction.
Quantum machine	A human-made device whose collective operation follows the laws of quantum mechanics.
Adjacency matrix	A square matrix normally used to represent a finite graph by defining adjacency of vertices as well as self-loops.

Table 2.2. Abbreviations

Term	Definition/Description
AMBA	Advanced microcontroller bus architecture
API	Application programming interface
ASIC	Application specific integrated circuit
CNOT	Controlled-not
CPU	Central processing unit

1 RFC 2119: Key words for use in RFCs to Indicate Requirement Levels, S. Bradner, Harvard University, March 1997; https://www.ietf.org/rfc/rfc2119.txt

FPGA	Field programmable gate array
HAL	Hardware abstraction layer
ISCF	Industrial strategy challenge fund
NISQ	Noisy intermediate-scale quantum
PCI	Peripheral component interconnect
QFT	Quantum Fourier transform
QNN	Quantum neural network
QoS	Quality of service
QPU	Quantum processing unit
SPI	Serial peripheral interface
VQA	Variational quantum algorithm
VQE	Variational quantum eigensolver

Chapter 3

Introduction

The main purpose of the HAL is to establish a unified northbound API-based framework across different QPU technologies. The challenges and architectural issues we endeavour to resolve in developing the HAL are:

1. Defining the position of Multi-level HAL within the system stack (see Figure 3.1 below) [2]
2. Maximising portability with minimal loss of performance
3. Maximising the range of common features, while keeping the optional, hardware dependent features at a minimum
4. Supporting for advanced features such as compiler optimisations, measurement-based control, and error correction

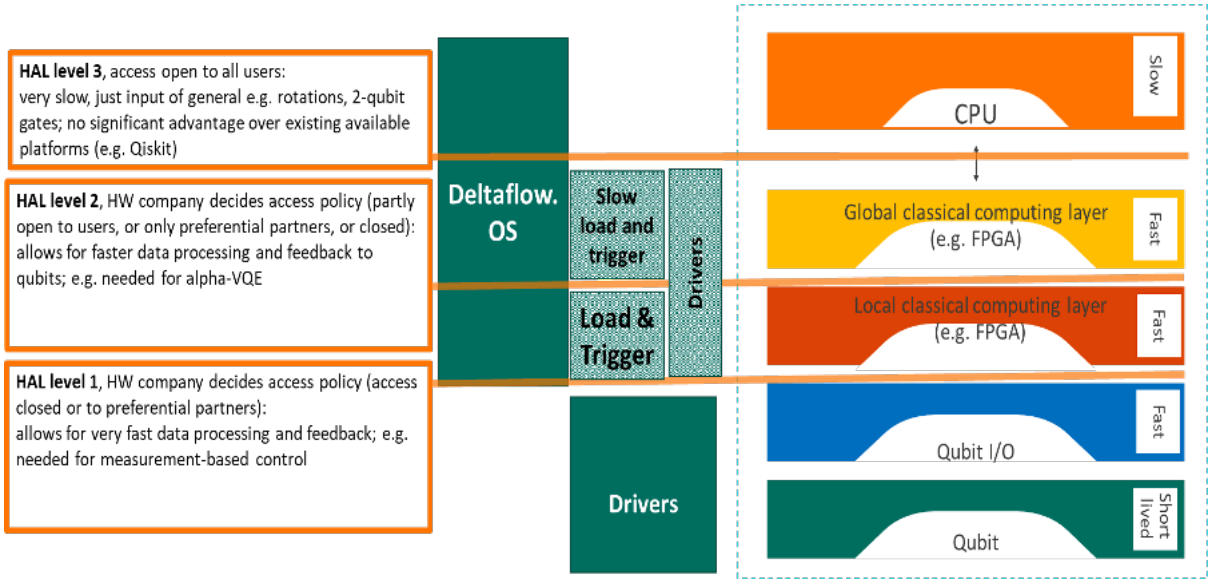

Figure 3.1. Positions of Multi-level HAL layers within the QPU system stack

2 Metadata is part of the HAL, and in some cases, it seems that data may reflect details of the lowest levels of the QPU stack including some details of the qubit implementation to help guide cross-implementation translation. The CPU could be involved in translation too, especially if it changes across implementations as should be anticipated (example: a constant representing the CPU architecture). If this is correct, then The HAL may cover as much as the whole stack, not just the middle. [Does it need clarifications?]

The HAL APIs considered in this document MAY be divided into the following groups: [3]

- General APIs These are most common APIs across different interfaces and platforms.
  - Register/De-register APIs
  - Discover APIs
  - HAL/APIs authentication/authorisation
  - HAL versioning
- Technical area specific (QPU System related)
  - Metadata of the system capabilities/properties
  - Required HAL/QPU commands
  - Optional HAL/QPU hardware specific commands
- Technical area specific (QPU System advanced features related)
  - HAL supported level authentication and authorisation
  - HAL Advanced/Optional features

3 Are groups and levels disjoint and distinct? Are all groups (or levels) expected to be implemented by all vendors even if they are not all exposed to all customers? Or may vendors choose to build bespoke versions of some in ways that do not coordinate with other components? This could prove relevant if the OS implementation and HAL components are "certified" or "conformant" for use and might even be commercially traded among parties to Deltaflow.OS, since the HAL itself appears to have the option to be closed source software even if Deltaflow.OS itself remains open source. If they are not required to be implemented should some be explicitly denoted as optional, and if they are, then should that be stated here at the start? [Added Multi-Level HAL extra specifications]

Chapter 4

HAL Architecture

4.1 Introduction

The HAL allows algorithm developers to abstract away the details of a hardware implementation by providing a standard set of commands which can be implemented to some degree on most devices. This brings two benefits:

- the developer can focus on the algorithm as opposed to the implementation.
- the algorithm can be easily ported to other devices.

Basic quantum algorithms and software use a high-level HAL representing a circuit model, which means taking advantage in a controlled way of advanced hardware capabilities. There are now algorithms being developed that require functionalities which this circuit model does not support.

In general, it is because they require some functionality that cannot be implemented by a classical CPU model connected over some high latency link (e.g. the cloud) to a quantum device. These algorithms require much quicker communication between a classical controller and device to be efficient and/or access to some native functionalities of the device.

4.2 Multi-level HAL and associated algorithms

The HAL must be capable of supporting advanced algorithms with different degrees of quantum/-classical interaction. Current algorithms can be classified into three main groups according to requirements on classical to quantum latency. We associate these groups to three levels of HAL as follows.

The highest level is 3, which supports the ability to run large batches of static circuits. This is implementable in a setting with high latency, typically much larger than the decoherence time, and is equipotent to commercial quantum devices available over the cloud.

In Level 2, there is no change to the quantum device's abilities, but the latency of the classical control is now on the order of qubit decoherence time. The controlling hardware can now make circuit updates based upon the results of a single circuit, without significant qubit "dead time".

In Level 1, the ability to make mid-circuit measurements, and control of the QPU based on the measurement outcome, is included. This requires the controlling device to make changes or store results on the quantum device, on the order of gate time and hence well below the decoherence time, so communication must also be of very low latency. The following table summarises the HAL three levels, and the timescales and corresponding algorithms considered in the first version of the specification. A general aim is to define a multi-level HAL flexible enough to cater to future developments

and additions. [4]

Table 4.1. HAL Levels

HAL Level	Timescale	Ability
3	Slow, communication between server and QPU (timescale much longer than coherence time)	Able to run large batches of circuits (e.g., may contain thousands of shots). Equipotent to what is available via IBM cloud, AWS, etc. Much slower than the coherence time. Supported algorithms: Gradient-free VQE.
2	Faster, communication between QPU and controller (timescale on the order of coherence time)	Actions can be taken based on the results of a single circuit and small batches of circuits (e.g., may contain tenths of shots). This usually cannot be done in Level 1 due to the bandwidth or latency issues encountered when making decisions on small numbers of circuits. Operates within coherence time.
1	Fastest, within decoherence time of qubits (timescale much shorter than coherence time)	Results of qubit measurement can be acted upon within a single circuit. This requires the HAL to be implemented via fast local control elements (e.g. FPGAs, application-specific CPUs). Supported algorithms: QNN dropout, holo-VQE, quantum autoencoder, simple error correction.

4.3 Multi-level HAL extra considerations

It is important to raise awareness of the following considerations:

- Hardware companies can expose one or more of the HAL levels
- Companies may want to expose a high HAL level publicly, but only expose a lower HAL level to selected partners or customers. In this case, care should be taken to implement public level(s) as per specifications. A potential benefit of this approach: A hardware company can outsource the development of applications to preferred developers, to whom privileged low-level HAL access is given.
- Metadata should allow the conversion of a sequence of HAL commands across architectures and layers. Each conversion must come with an associated set of acceptance checks that the user/hardware company can execute. In order of complexity, we envision:
    - Metadata checks. The conversion can be checked for feasibility by simply examining the metadata with no compilation.
        - Example of conversion: from a Level 3 HAL representation targeting different hardware.
        - Example of check: the number of qubits and circuit depth required must be available on the new target architecture.
    - Compilation checks. A conversion that needs to be remapped to a new gate set and analysed to understand if they meet the hardware constraints.
        - Example of conversion: from a Level 2 representation, Hardware A to a Level 2, Hardware B
        - Example of check: verify all original gates can be transpiled into Hardware B native gateset
    - Performance checks. In the case of guaranteed QoS (for example, on error rates), conver-

[4] Again, are we going to encourage vendors to follow the level structure for their internal use, even if they don't expose them to any customers? Is Level 3 mandatory? Is Level 2 encouraged? Is Level 1 truly optional? Is there an implication that some or all levels may be licensed? Is it anticipated that some vendors may choose to open source their implementations? It is likely that there will be a need to validate the authenticity of any level for supply chain and security-related reasons. [Tentative response in Multi-Level HAL additional considerations]

sions need to analyse the final solution's performance.

- Example of conversion: from a Level 2 representation, Hardware A to a Level 2, Hardware B with user expecting final fidelity > X.
- Example of check: on top of the compilation checks, verify that the transpiled version of the circuit can meet the QoS requirement by using single- and two-qubit fidelities of Hardware B.

Chapter 5

Metadata Format Specification

5.1 General

The primary purpose of Metadata is to:

- Allow the hardware companies to defend their trade secrets
- Allow the users to identify the hardware platform most suitable for their problems and utilise it to its best
- Discourage independent efforts to extract/infer undisclosed information. This prevents hardware companies from being falsely accused of suboptimal service and/or overcharging consumers.

To reach these goals, we believe metadata should be different at the different layers of the HAL. Table entries marked as required are described in more details at the bottom of this section. We will use the definition *valid* to indicate that the circuit, shot, or gate does not infringe the information provided by the metadata (e.g. a five-qubit circuit on a four-qubit system).

Tables should be seen as extensions of the higher levels. For example, Level 2 MUST contain all the fields of Level 3. Fields of an higher level HAL MAY be converted from OPTIONAL to REQUIRED but not vice-versa.

5.2 Level 3 HAL – application level

"Able to run large batches of circuits".

At this level, the final stage compiler (executed by the hardware lab) takes care of converting an abstract representation made with universal gatesets, into a native one.

Users are entitled to:

- Fair billing. Note that the cost per time on the quantum machine will likely be different from that on the supporting classical infrastructure.

Users won't appreciate:

- If they send a valid circuit and it gets refused/does not complete in time.

Hardware companies won't appreciate:

- Unfair accusations on performance/correctness/costing that can't be easily disproved and might lead to legal actions.

Table 5.1. Level 3 Metadata

Metadata	Description	Required	Notes
NUM_QBITS	Number of qubits available	Yes	It can be smaller than the actual number of qubits in the quantum machine.
MAX_DEPTH (as universal gates)	Maximum depth of the circuit to execute	Yes	If **NATIVE_GATES** are not provided, this needs to be a conservative value. The conversion from a universal to a native gate set causes not deterministic (but bound) overhead.
NATIVE_GATES	List of Native Gates	No	The **MAX_DEPTH** could be improved significantly by providing the definition of native gates here. Effect: Users will benefit from longer circuits.
GATE_TIMES	The duration of the gates in **NATIVE_GATES**	No	With this information, users will be able to optimise their running costs. Note that advanced users are able to infer this information regardless of whether it is provided by the HAL.

- **NUM_QUBITS**:
  - Type: unsigned int
  - Example: 5
  - Forbidden Values: [0]
- **MAX_DEPTH**:
  - Type: unsigned int
  - Example: 200
  - Forbidden Values: [0]

5.3 Level 2 HAL – shot level

"The results of a single circuit and small batches of circuits can be acted upon."

At this level, the final stage compiler (executed by the hardware lab) takes care of converting and mapping a native representation of a circuit and executing it. Conversion is performed "on the fly".

Users are entitled to:

- Fair billing. Note that the cost per time on the quantum machine will likely be different from that on the supporting classical infrastructure.
- Guaranteed execution. If they send a valid circuit, it shouldn't get refused as it might be part of a long sequence.

Users won't appreciate:

- Unknown QoS – mainly if the error rates are unknown

Hardware companies won't appreciate:

- Unfair accusations on performance/costing that can't be easily disproved and might lead to legal actions.

Table 5.2. Level 2 Metadata

Metadata	Description	Required	Notes
NUM_QBITS	Number of Qubits available	Yes	It can be smaller than the actual number of qubits in the quantum machine.
MAX_DEPTH (as native gates)	Maximum depth of the circuit to execute	Yes	Total number of gates that can be executed. Without the **GATE_TIMES** information the depth will be conservative to allow for additional margin within the coherence time.
NATIVE_GATES	List of Native Gates	Yes	It can be a subset of all the available gates
CONNECTIVITY	The connectivity matrix of the Qubits	Yes	It is required to support correct compilation of circuits. The hardware company can return a subgraph of the connectivity as they deem appropriate (e.g. when only a subset of the qubits is exposed they won't need to expose the full connectivity). Connectivity MUST be maintained within two Metadata updates.
GATE_TIMES	The duration of the gates in **NATIVE_GATES**	No	With this information, users will be able to optimise their running costs. Note that advanced users are able to infer this information regardless of whether it is provided by the HAL.

ERROR_RATE	The average error rate for one- and two-qubit operations in **NATIVE_GATES**	No	Without this information the users will have to personally evaluate the performance of the hardware before committing to run intensive applications. Users at this level have all the information required to run randomised benchmarking or similar techniques to extract the metrics.

- **NATIVE_GATES**:
  - Type: List of parametrisable matrices
  - Example:

```
     [0 1]                    [1 0 0       0       ]
X =  [1 0]       CR(theta) = [0 1 0       0       ]
                             [0 0 1       0       ]
                             [0 0 0 exp(i*theta)]
```

  - Forbidden Values:
    - Any non-canonical form representation
    - Null matrix

- **CONNECTIVITY**:
  - An adjacency matrix (symmetric) of size N x N (where N is the number of qubits) that represents with a 1 an edge that connects two qubits and with a 0 a not-connected edge
  - Example (refer to Figure 5.1):

```
                [0 1 0 1 0 0 0 0]
                [1 0 1 0 1 0 0 0]
                [0 1 0 0 0 1 0 0]
                [1 0 0 0 1 0 0 0]
CONNECTIVITY =  [0 1 0 1 0 0 0 0]
                [0 0 1 0 0 0 0 1]
                [0 0 0 0 0 0 0 1]
                [0 0 0 0 0 1 1 0]
```

  - Forbidden Values: Empty matrices

- **ERROR RATE**:
  - Error rate is defined as the probability for a quantum operation to introduce an error. A matrix of size N x N (where N is the number of qubits that contains: on the diagonal an average error rate for 1 qubit gate(s); off-diagonal the average error rate of 2 qubits gate(s). To clarify **ERROR_RATE** (1,1) describes the average error rate when executing single qubit gates on qubit0; **ERROR_RATE** (1,2) indicates the average error rate when executing gates two qubit gates on qubit0 and qubit1 with (where applicable) 1 being the control qubit and 2 the target one. Multiple matrices can be returned to define the behaviour of different gates. Optionally the values can be provided as intervals.

- Example:

```
                [0.014 0.02  0     0     0     0     0     0    ]
                [0.02  0.014 1     0     0     0     0     0    ]
                [0     0.021 0.013 0     0     0     0     0    ]
                [0     0     0     0.015 1     0     0     0    ]
ERROR_RATE =    [0     0     0     0     0.012 0     0     0    ]
                [0     0     0     0     0     0.016 0     0    ]
                [0     0     0     0     0     0     0.011 0    ]
                [0     0     0     0     0     0     0.02  0.012]
```

- Forbidden Values: Empty matrices and matrices that violate connectivity. Entries outside the range [0,1].

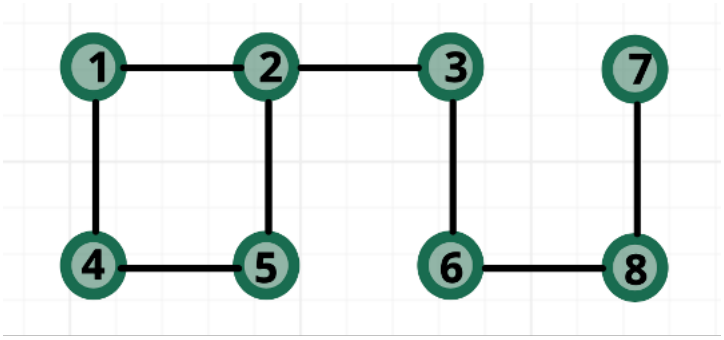

Figure 5.1. Topology used in the example

5.4 Level 1 HAL – gate level

"Results of qubit measurement can be acted upon within a single circuit."

At this level, the final stage compiler (executed by the hardware lab) takes care of converting and mapping a single gate and executing it.

Table 5.3. Level 1 Metadata

Metadata	Description	Required	Notes
NUM_QBITS	Number of Qubits available	Yes	It can be lower than the actual number of available qubits.
MAX_DEPTH (as native gates)	Maximum depth of the circuit to execute	Yes	Total number of gates that can be executed.
NATIVE_GATES	List of Native Gates	Yes	It can be a subset of all the available gates
CONNECTIVITY	The connectivity matrix of the Qubits	Yes	It is required to support correct compilation of circuits.
GATE_TIMES	The duration of the gates in **NATIVE_GATES**	Yes	Shuttling time should be considered as an atomic command of which time execution will be required. This to prevent performance inconsistencies

ERROR_RATE	The average error rate for one- and two-qubit operations in **NATIVE_GATES**	No	Without this information the users will have to personally evaluate the performance of the hardware before committing to run intensive applications. Users at this level have all the information required to run randomised benchmarking or similar techniques to extract the metrics.

MAX_DEPTH:

- Type: unsigned int [unit ps]
- Example: 32000000 ps[32 us]
- Forbidden Values: [0]

GATE_TIMES:

- Type: List of unsigned int [unit ps]
- Example: X: 16000, Y: 16000, CNOT: 28000
- Forbidden Values: [0]

ERROR_RATE: [optional]

- Type: List of decimal numbers (between 0 and 1) defining probability of quantum gate to introduce an error.
- Example: X: 0.05 , Y: 0.04
- Forbidden Values: Any usage of NaN (not a number)

5.5 Metadata Encoding

Here we outline the format in which metadata requests and results are encoded into 64 bit HAL commands. Depending on the metadata requested the result will either be returned as a single 64 bit integer or a stream of 64 bit integers to be collected and decoded into the appropriate data format.

Below is the list of individual metadata items that may be requested and the structure of their corresponding HAL command request:

NUM_QUBITS:

- Request

Table 5.4. HAL command for NUM_QUBITS metadata

Opcode [12]	Argument [16]	Padding [36]
000000001000	0000000000000001	000000000000000000000000000000000000

- Response (single)

Table 5.5. HAL response for NUM_QUBITS metadata

Metadata Index [3] ***(NUM_QUBITS = 1)***	**Number of qubits [61]** ***(e.g. 4 qubits)***
001	0000000000000000000000000000000000000000000000000000000000000100

MAX_DEPTH:

- Request

Table 5.6. HAL command for MAX_DEPTH metadata

Opcode [12]	**Argument [16]**	**Padding [36]**
000000001000	0000000000000010	000000000000000000000000000000000000

- Response (single)

Table 5.7. HAL response for MAX_DEPTH metadata

Metadata Index [3] ***(MAX_DEPTH = 1)***	**Number of qubits [61]** ***(e.g. 200)***
010	0000000000000000000000000000000000000000000000000000000011001000

NATIVE_GATES/GATE_TIMES:

- Request

Table 5.8. HAL command for NATIVE_GATES/GATE_TIMES metadata

Opcode [12]	**Argument [16]** ***(3)***	**Padding [36]**
000000001000	0000000000000011	000000000000000000000000000000000000

- Response (one per native gate)

Table 5.9. HAL response for NATIVE_GATES/GATE_TIMES metadata

Metadata Index [3] ***(3)***	**Final [1]** ***(e.g. True)***	**Gate index [4]** ***(e.g. 0)***	**Opcode [12]** ***(e.g. 10 = RX gate)***	**Parameter [16]** ***(e.g. 32768 = pi/2)***	**Gate Time [30]** ***(e.g. 16000 ps)***
011	1	0000	000000001010	1000000000000000	000000000000000011111010000000

- Notes:
    - **Final**: flag used to specify last stream packet
    - **Parameter**: used to specify the parameter for gates that have a generic matrix definition parameterised by some rotation angle
    - **Gate Time**: 30-bit unsigned integer for gate time, specified in picoseconds

CONNECTIVITY:

- Request

Table 5.10. HAL command for CONNECTIVITY metadata

Opcode [12]	**Argument [16]** ***(4)***	**Single row [1]** ***(e.g. False)***	**Row index [35]** ***(e.g. 0)***
000000001000	0000000000000100	0	00000000000000000000000000000000000

- Response (one per N/3 groupings of non-zero off-diagonal matrix elements)

Table 5.11. HAL response for CONNECTIVITY metadata

Metadata Index [3] ***(4)***	**Final [1]** ***(e.g. False)***	**Row idx 1 [10]** ***(e.g. 0)***	**Col idx 1 [10]** ***(e.g. 1)***	**Row idx 2 [10]** ***(e.g. 1)***	**Col idx 2 [10]** ***(e.g. 2)***	**Row idx 3 [10]** ***(e.g. 2)***	**Col idx 3 [10]** ***(e.g. 3)***
100	0	0000000000	0000000001	0000000001	0000000010	0000000010	0000000011

- Notes:
    - We make use of the 36-bit padding in the HAL request to specify if we want the whole matrix back or just a single row
    - **Final**: Non-zero off-diagonal row/column pairs are returned in groups of 3 in row-order, where the final respone packet is marked by this flag
    - Connectivity matrix is symmetric, so only off-diagonal upper half of matrix is returned

ERROR_RATE:

- Request

Table 5.12. HAL command for ERROR_RATE metadata

Opcode [12]	**Argument [16]** ***(5)***	**Gate index [3]** ***(e.g. 2)***	**Single row [1]** ***(e.g. False)***	**Row index [32]** ***(e.g. 0)***
000000001000	0000000000000101	010	0	00000000000000000000000000000000

- Response (one per N/4 groupings of non-zero matrix elements)

Table 5.13. HAL response for ERROR_RATE metadata (first 8 bits)

Metadata Index [3] ***(5)***	**Final [1]** ***(e.g. False)***	**Diagonal [1]** ***(e.g. True)***	**Gate index [3]** ***(e.g. 2)***
101	0	1	010

Table 5.14. HAL response for ERROR_RATE metadata (final 56 bits)

Mantissa 1 [10] ***(e.g. 2)***	**Exponent 1 [4]** ***(e.g. 1)***	**Mantissa 2 [10]** ***(e.g. 3)***	**Exponent 2 [4]** ***(e.g. 1)***	**Mantissa 3 [10]** ***(e.g. 4)***	**Exponent 3 [4]** ***(e.g. 1)***	**Mantissa 3 [10]** ***(e.g. 3)***	**Exponent 3 [4]** ***(e.g. 1)***
0000000010	0001	0000000011	0001	0000000100	0001	0000000011	0001

- Notes:
    - We make use of the 36-bit padding in the HAL request to specify which native gate we want data for (obtained from the order of NATIVE_GATES metadata responses), and if we want the whole matrix back or just a single row
    - **Final**: Non-zero error rate values are returned in groups of 4 in top-left to bottom-right order for diagonal (1-qubit gate) data, and in the same order of row/column indexes returned from CONNECTIVITY metadata request for off-diagonal (2-qubit gate) data. The final respone packet for a given gate is marked by this flag
    - Error rate data (value between 0 and 1) is stored in a pair of integers with a 10-bit mantissa and 4-bit exponent (distance of mantissa from decimal point). This allows us to store mantissas up to three 9s, up to 15 places after the decimal point. For example, the number

0.01 is expressed by 0000000001|0001, and the number 0.00245 is expressed by 0011110101|0010

- Error rate matrix is **not** symmetric, so off-diagonal upper and lower halves of matrix returned. Upper half is returned in the same order of row/column indexes returned from CONNECTIVITY metadata request (row-wise), lower half returned with equivalent row/-column indexes flipped (column-wise)
- **Must** have knowledge of CONNECTIVITY metadata in order to map the error rate values to appropriate qubits

Chapter 6

HAL Commands Minimal Requirements

To allow for the best usage of resources while preserving the desired user intents, we should allow each HAL layer to have a different set of commands. This allows a tuning of the commands to fit the associated level best. In this Section we have used the following considerations to drive our proposal:

- A Level 3 or Level 2 HAL needs to perform one or more of compilation, transpilation and timing allocation of instructions. At Level 1, commands that are already usable by the hardware should be presented.
- Users will want to execute on emulators as well as on real quantum resources. Hardware vendors might want to expose a single HAL interface and internally route the circuits to an emulator or the real system. To ease this process and consequently allocation, billing and scheduling, Level 2 and Level 3 HAL should have implemented this concept.
- We don't believe the user needs a complete set of traditional sequence modifiers (FOR, WHILE, DO, etc) but just the bare minimum to express repetition and branching. We are suggesting FOR and IF statements to achieve that.

6.1 Level 3 HAL

"Able to run large batches of circuits"

Table 6.1. Level 3 HAL commands

Command	Motivations	Implications	Notes
Gates from a universal gateset	Define the circuits to execute	Compilers and transpilers are needed to convert it to a usable representation	None
Section commands	Confines the code that belong to one user and associate it to hardware or emulation facilities	The compilation flow should support both targets	The user can transmit circuits back-to-back as a binary sequence. Section commands are used to delimit these sequences (as a START and STOP equivalent) allowing optimisations and compilations on the received circuits.

At Level 3, the classical logic is in charge of acting upon measurements and selecting the next sequence of circuits to execute. Circuits can be fully precompiled and buffered. Acceptance criteria may be applied to the provided code to verify that it is within the capabilities of the compiler and the hardware.

Requires:

- Validation of the user-provided algorithm to access its feasibility

- Highly parallelised compilation flow to avoid execution underflowing and suboptimal utilisation of the quantum hardware.

6.2 Level 2 HAL

"The results of a single circuit and small batches of circuits can be acted upon."

Table 6.2. Level 2 HAL commands

Command	Motivations	Implications	Notes
Gates from the native gateset	Define the circuits to execute	Transpilers are needed to convert it to a hardware representation (e.g. sequence of pulses)	It can contain optional commands (e.g. CPHASE, CCX, ACTIVE RESET) that the hardware supports

At Level 2, the classical logic is in charge of acting upon measurements and select the next circuit(s) to execute.

Requires:

- Parallel compilation of circuits to handle branching statements
- Low overhead repetition/reloading of the same circuit

6.3 Level 1 HAL

"Results of qubit measurement can be acted upon within a single circuit."

Table 6.3. Level 1 HAL commands

Command	Motivations	Implications	Notes
Gates from the native gateset	Define the circuits to execute	None	If the hardware supports them the user should be allowed to use arbitrary controlled gates (e.g. CPHASE) and (b) multi-qubit (>2) gates

At Level 1, the classical logic is in charge of acting upon measurements and select the next gate(s) to execute.

Requires:

- Fast conversion of native gateset representation to hardware controls
- Fast loading of these sequences
- Fast path from measurement to user logic

HAL Features

For purposes of this document, the HAL features are presented as a set of commands and a set of metadata. The commands are categorised as either core, fundamental qubit commands which are common across all quantum technologies, or advanced or optional as defined in this document, which is specific to the vendor implementation.

7.1 Required HAL commands

The following is a non-exhaustive list of core HAL commands that MAY be extended in the future. Core HAL commands are mandatory and SHOULD be implemented for every system following the HAL specification. HAL command support will be conveyed through HAL metadata. Core commands MAY be extended in future with the introduction of new universal commands.

7.2 Control commands

The following table lists control commands that are required to enable advanced functionalities (e.g. multi-users, large addressing).

Table 7.1. Control Commands

Command	Parameters	Description	HAL Level
Start of Session	Type of Section	Defines the type of session, emulator, hardware, simulator. It is used to route the commands to the right destinations.	3-2 [*]
End of Session	None	Closes a session.	3-2 [*]
Set Page Qubit0	Offset for the qubit index (0)	Modifies the offset used in the qubit index computation. The register associated with the offset must be reset by a new Start of Session Command.	All
Set Page Qubit1	Offset for the qubit index (1)	Modifies the offset used in the qubit index computation. The register associated with the offset must be reset by a new Start of Session Command.	All

7.3 Single-qubit HAL commands

The following table lists the basic single qubit HAL commands.

Table 7.2. Single-qubit HAL commands

Command	Parameters	Description	HAL Level
NOP	None	Performs no operation	All
State Prepare	|0>or |1>, qubit address	Prepare specific qubit to a known state	All
State Prepare all	|0>or |1>, qubit address	Prepares all the qubits to a known state	All
Qubit measure	None	Return the measured state of a qubit	All
Arbitrary rotate x	Angle	Perform qubit rotation [5]	All
Arbitrary rotate y	Angle	Perform qubit rotation [5]	All
Arbitrary rotate z	Angle	Perform qubit rotation [5]	All
Pauli-X	None	None	All
Pauli-Y	None	None	All
Pauli-Z	None	None	All
Hadamard	None	None	All
Phase	None	None	All
T	None	None	All

7.4 Two-qubit HAL commands

The implementation of 2 qubit gates commands across the HAL is for further consideration, and it might even be outside the scope of this document. [6]

Table 7.3. Two-qubit HAL commands

Command	Parameters	Description	HAL Level
CNOT	Qubit addresses	Performs a CNOT operation	3

However, implementing core native 2-qubit gate sets will, in most cases, be necessary. Each vendor should define via optional commands the Level 2 and Level 1 implementation of the CNOT command.

7.5 Native two-qubit gates

Since native two-qubit gates are necessary to operate at a Level 1 HAL, hardware vendors SHOULD specify their native gates in the Optional HAL section.

7.6 Optional HAL commands

Commands specific to qubit implementations that are not relevant to others or contain potentially confidential information of a specific hardware platform are optional. The disclosure of a specific native hardware gate or the hardware topology is optional: disclosure to the user will improve performance, but some vendors might prefer not to disclose such information. [7]

Additionally, native 2-qubit gates are optional. For example, the RZZ 2-qubit gate or the CPHASE gate.

5 This is still open for debate and will depend on hardware provider as well as qubit tech. Likely, something to include in metadata rather than specify.

6 If a vendor conforms to the structure of the HAL for their internal features then they could benefit from examples and some standardisation for their group properties APIs even if not for their implementation.

7 Consequently, do we want to explicitly state that members of this category may not translate across implementations, resulting in defaulting back to core commands and speeds? [Tentative response in Multi-Level HAL additional considerations]

Table 7.4. Optional HAL commands.

Command	Parameters	Description	HAL Level
32 QBit Measure	Starting index of the qubit to read	Returns 32 measurements in parallel.	All [8]
For/If/While	To be defined.	Conditional execution. Hardware specific in terms of format and limits	All [8]
Opt1	None	Optional commands for hardware-specific instructions.	Specific.
Opt2	None	Optional commands for hardware-specific instructions.	Specific.

7.7 Required HAL responses

Users should at least be informed when:

- The circuit completes successfully. Only required at Level 3 and Level 2 and define as completion ACKNOWLEDGE.
- The commands they sent are INVALID. An example would be CNOT(0,0), a CNOT with both inputs being qubit 0;
- An error has occurred in the quantum computer and the computation is INCORRECT.

Hardware labs can specify additional error codes to handle specific scenarios.

The format of the response:

Table 7.5. Response format

Response (4 bits)	CIRCUIT ID (12 bits)
Defines the type of error as per Table 7.6	Unique ID that identifies user and circuit. Needed in case of multi-user/multi-circuit execution

And the codes for the responses:

Table 7.6. Response codes

Response	VALUE	Description
ACKNOWLEDGE	0	The circuit execution was succesful
INCORRECT	1	The execution encountered an error. Returned measurements should be discarded
INVALID	2	One or more of the commands sent are incorrect. Nothing has been executed.

Level 1 access types are not required to return responses as the latency to acknowledge them would impact significantly performance and quantum up time.

[8] For optional commands, the hardware provider has to specify the HAL level(s) to which they apply.

HAL Commands Format Specification

8.1 Introduction

The HAL commands should have a format that provides the best generality and expressivity whilst keeping the decoding logic (and consequently its latency) to the minimum. Before describing the format, the considerations that have motivated the choice of this format are outlined. We will touch on the two main sets of implications that relate to the command format.

- Commands must be transmitted to the QPU
- Commands must be parsed and decoded. If errors occur in any of the two processes they should be returned.

8.2 Considerations on transmission

- The transmission of the HAL commands occurs via physical wires (electrical or optical) and/or via internal on-chip traces (e.g. FPGA routing resources)
- We will assume error-free (or classically error-corrected) transmission of the commands in this version of this proposal.
- We will assume that different quantum hardware will have different requirements in terms of connectivity, required bandwidth (of commands), and link-latencies. For this reason, we have tentatively listed in Table 7.1 some metrics related to standard (public) interfaces.
- It is important to point out that:
    1. Most transmission protocols listed in the physical layer could be adapted to handle arbitrary packet sizes. It is worth pointing out that this will require a custom implementation of the link-layer logic for both transmitter and receiver.
    2. Ad-hoc protocols might use generic parallel-busses of any width (i.e., not standard interfaces). We think that addressing these specific scenarios might lead to a loss of generality for this set of specifications.

Table 8.1. Transport Protocols - illustration

Protocol	Minimum Packet Size/Increments	Technology Supported	Notes
Ethernet (raw)	46 bytes/1 byte (up to 1500 bytes)	CPU, FPGA, ASIC	Large transmission overhead

PCIe3.0/4.0	128/256/512 bytes (Root complex dependant)	CPU, FPGA, ASIC	Short reach
USB3.x	1 byte/1 byte (up to 1024 bytes)	CPU, FPGA[*], ASIC	Transmission overhead (protocol defined transmission timeslots)
Intra-chip communication (AMBA-AXI)	4 bytes/4 bytes (up to 128 bytes[**])	FPGA, ASIC, CPU[***]	Ultra-short reach
Serial Peripheral Interface (SPI)	1 byte/1 byte (up to 1024 bytes[****])	FPGA, ASIC, CPU	Low bandwidth

[*] USB3.2 2x2 might require special cards to implement the initial speed negotiation (10 Gbps mode) that might not be commercially available. [**] The AMBA protocol does not set an upper limit on the size of the bus but the physical routing of the logic normally limits this value to be 1024 bits threshold. [***] Hard/Soft-CPU only. Only CPU that are integrated into the same die as the ASIC/FPGA (either permanently or in a reconfigurable fashion). [****] Generally, controller limited. Some controllers support up to 65535 bytes.

8.3 Considerations on decoding

- To guarantee the portability of applications, we recommend for the HAL specification to define a consistent representation for all the commands in terms of the number of bits and their significance.
- Bit shifts and bit masking can be implemented with limited effort and low latency on CPU, FPGA and ASICs
- Command size should be limited to 64 bits to benefit from CPU ISAs and facilitate software development
- Commands to be executed in parallel can be sent to the Quantum Processing Unit in any order. This allows the usage of concepts like paging to index large number of qubits by decoupling it into two separate entities: BASE_OFFSET and a RELATIVE_OFFSET. The RELATIVE_OFFSET shall be embedded in all commands that require an index to operate while the BASE_OFFSET can be sent as a separate field to minimise overhead while keeping large addressability.
- The identifier of the command (OPCODE) can be of:

    1. Fixed-length (i.e. all OPCODES are implemented using the same number of bits)
    2. Variable-length (i.e. OPCODES can use a different number of bits)

    (1) provides the fastest decoding (e.g. look-up tables based) while (2) can increase the content of information transmitted via better usage of the available bits

- Qubit indexing can be implemented as:

    1. a combination of one-hot encoding (e.g. 1001 indicates that index 0 and 3 are active)
    2. or in a binary format (e.g. 1001 indicates that the index 9 is active).

    (1) enables the addressing of multiple qubits via a single command while (2) provides a much larger qubit addressing space (N vs 2^N)

- Commands that do not fit in a single word can be split and transmitted as a sequence of parts (multi-word commands). We envision three possible scenarios here:

    1. The list of commands that require more than one word (multi-word commands) is fixed and predefined. Their OPCODE is sufficient to inform the decoding logic that they are composed of multiple words.
    2. The list of multi-word commands is not known a priori. A field/flag can be set to indicate that the command is composed of extra words. This field/flag will require at

least one extra bit to be always dedicated to this specific purpose.

3. The list of multi-word commands is not known a priori. A special command needs to be issued to indicate that what follows is a sequence of multi-word commands. One possible implementation uses the first command argument to indicate the number of words composing the real multi-word command to execute.

(1) provides the simplest decoding logic (fixed-length commands with deterministic latency), (2) and (3) have slightly more complex logic with at least one extra conditional branch. If statistically, the likelihood of multi-word commands is low, (3) provides a lower bit requirement overhead than (2).

- Two-qubit commands (e.g. CNOT) require (a) the definition of two indexes as well as (b) the execution of two parallel sequences of control. While (a) is in line with previous considerations, (b) requires additional considerations. The decoder logic should effectively extract both the indexes (ideally in a single instruction) and inform the associated branches of the control logic (if independent). We identified the following options:

    1. Single-word command with halved addressing space. We preserve the format of the command but consider the lower half of the index field pertaining to qubit 0 and the upper part to qubit 1
    2. Longer command. We append a second indexing field to the end of the command to address the second index.
    3. Double-word command. We extend the command with the second index and padding.
    4. Two-words command. We split the command into two portions, and we send them as two separate tokens. e.g., we split a CNOT into in a "Control" and "Controlled" set of commands (CNOT_CTRL, CNOT_DATA).

    (1)-(4) require almost no changes to the architecture for 1 qubit commands in storage and decoding. (4) though does introduces a barrier on execution. Because now the two commands are independent, the transport layer can delay the transmission of the second one, requiring buffering of the command. (2) - (3) require an extra buffer/register to store the second portion of the command and potentially forces us to decouple the command width from the transport layer width, but they do enforce the command's atomicity.

8.4 Proposed command format

We would like to conclude this Section by proposing at least one possible format for the HAL commands. This has been investigated and tentatively validated on different integrations on both FPGA and CPUs for different quantum architectures. The table that follows contains three representations, respectively for "control commands", "single qubit commands" and "two qubits commands". All of them are encoded in 64 bits words. The goals of this format are (a) low complexity decoding logic (with buffering), (b) no significant performance penalty.

Table 8.2. Proposed command format

Command type	OPCODE (command to execute) bits	ARGUMENT (argument for the command) bits	RELATIVE_QUBIT_IDX (Relative index of the QUBIT) bits
CONTROL COMMANDS	[63-52]	[51-36]	[35-0]: BASE_QUBIT0/1_IDX
SINGLE QUBIT COMMANDS	[63-52]	[51-36] [35-20]: padding	[19-10]: padding [9-0]: RELATIVE_QUBIT0_IDX

DUAL QUBIT COMMANDS	[63-52]	[51-36]: qubit1 [35-20]: qubit0	[19-10]: RELATIVE_QUBIT1_IDX [9-0]: RELATIVE_QUBIT0_IDX

The following considerations have been made:

- By fixing the OPCODE length, the decoder logic can use lookup tables. We consider 4096 codes (12 bits) to be more than sufficient. Note: It might be possible to reduce them to 256 (8 bits) by intelligent usage of special commands that allow an exception to the format (MODIFIERS, two examples will follow).
- The RELATIVE_QUBIT_IDX is used in associate with the SET_PAGE_QUBIT0 and SET_-PAGE_QUBIT1 commands to allow for extremely large addressability (2^{46}). Two registers in the quantum backend keep track of the addresses by applying the formulas: (BASE_QUBIT0_IDX << 10) + RELATIVE_QUBIT0_IDX and (BASE_QUBIT1_IDX << 10) + RELATIVE_QUBIT1_IDX for qubit0 and qubit1 respectively.
- The BASE_QUBIT0_IDX and BASE_QUBIT1_IDX registers are preserved after being written. In other words, when a page is open it remains the same up to the next write to it. A START Session Command closes (resets to 0) both BASE_QUBIT0_IDX and BASE_QUBIT1_IDX values.
- The OPCODE requires shifting and masking (12 bits) but we believe that the benefits of having a more compact word outnumber the additional complexity. Further optimisations can be enabled by using an additional bit (bit 11 of 12) to indicate a long OPCODE (length > 8).
- No field has been allocated to support multi-word commands.
- The DUAL QUBIT COMMANDS can be clearly identified by the OPCODE (we suggest using the MSB bit to indicate whether it is a SINGLE or DUAL WORD command).

Chapter 9

Security

9.1 Threat model

Even at the end of the NISQ era, chances of having low-cost quantum machines are negligible. This implies that most of the quantum resources will be shared among a large pool of users and potentially exposed via cloud interfaces. From a security perspective, various aspects make quantum machines different from classical resources:

- Cost of the hardware, uptime concerns. Damage to some of the building blocks of a quantum machine might lead to extremely long times as well as high cost.
- Intellectual Property protection. Malicious attackers will try to obtain information on the quantum machine internals to make profits or increase their visibility.

These two problems have direct implications for quantum hardware providers and require countermeasures to be implemented at the classical interface level as discussed in this Section.

We would like to have the HAL to be future proof in terms of security with the caveat that additional work will be needed throughout the stack to guarantee the full security of the solution [9]. The threats that we currently consider out of the scope for this document are:

- Side channel attacks. Malicious users might try to infer what other users are running if multi-user access to the quantum resource is allowed. The number of practical experiments on the subject is not sufficient to identify the need for them and the mitigation strategies.
- Unsigned execution of code. Compilers might be tampered with, or unverified and malicious code may be crafted. Currently no attack has been identified that could damage the quantum machine and/or cause repercussion on the next user.

9.2 Implementation aspects

We would like to start the discussion on security of the quantum machine and of the quantum operations. As for most other form of modern technologies, security in quantum systems requires the introduction of various mechanisms at potentially all the operational levels. In this Section, will limit the scope to the measures that we think are implementable by the HAL or that have a direct effect on it. With the broad-brush term of security, we will refer in the following to:

1. Application security. We should define rules and guidelines to minimise the risk of the user:

    1. Having their application executed on a target that is not the expected one. A specific

[9] The one attack that could be most worrisome already is on the device firmware itself or on the management system—whether it be by alteration or replacement. Hence signatures and attestation should probably be assumed.

example: man-in-the-middle attacks.

2. Incurring extra costs caused by over-execution. A malicious attacker might able to introduce extra computation causing unforeseen costs to the user.

2. Quantum machine security. We should define rules and guidelines to minimise the risk of a quantum machine:

    1. Being damaged or its QoS reduced by the user via low-level attacks. Example: Attacks that leverage patterns to cause extra noise in the circuit execution (in case of multi-users), attacks that cause excessive power dissipation on the fridge logic etc.
    2. Being brought in a condition of not being able to take extra requests from other users. We should expect malicious users to try denial-of-service attacks by injecting small requests (in terms of their data payload) that cause an intense computation or conversely increase the delay in the communication by saturating input channels.

3. Supply chain security. The quantum machine drivers can be compromised or modified by malicious attackers. This can cause identity theft and/or exposure of confidential information.

We suggest the following level of severity for these class of security considerations:

- **Extremely severe**: 2.a
- **Severe**: 1.a, 1.b, 3
- **Moderate** 2.b,

And the level of potential complexity required to implement mitigation strategies around them:

- **High complexity**: 2.a
- **Medium complexity**: 2.b
- **Low complexity**: 1.a, 1.b

We will propose in the next sub-sections few potential measures to address the set of above-listed threats. All the solutions that address at least one vulnerability of severity severe and extremely severe will be indicated as “Rules” while lower severity vulnerability as “Guidelines”.

9.3 Rule 1: parties’ authentication

We suggest that post-quantum cryptography or quantum-robust algorithms should be investigated in the near future as in the post NISQ era they will be relevant. As they don’t significantly impact the HAL, we will try here to define a generic approach that should allow us to move to quantum robust implementations when needed. We start by establishing a trusted channel between the user and hardware provider. By doing this we should be able to minimise the likelihood of 1.a, 1.b, and 3. For the latter we assume that components of the hardware control stack have a signature that can be verified by the users. Further enhancements to traditional protocols can be embedded into this HAL specification by introducing the following commands:

- Request public key
- Request authentication scheme
- Send authentication challenge
- Retrieve authentication response
- Send driver challenge
- Retrieve driver response

9.4 Rule 2: coarse-granularity machine statistics

All the commands that return data that can directly or indirectly be used to infer:

- Number of users currently sharing the Quantum machine
- Status of the hardware components

Should return values that have sufficiently coarse granularity to prevent any type of reverse-engineering of power models, components behaviour and number of users. This to reduce the likelihood of success of attacks of type 2.a. A set of selected users (e.g. system maintainer) could be granted a finer visibility to these data. Additionally, research groups could be granted access to historical data for which users have pre-approved for disclosure.

9.5 Guideline 1: prevention of denial of service

To prevent a malicious attacker from causing a denial of service to the quantum machine we recommend implementing a variable response time for all the query operations. This type of requests tends to have an asymmetric computational cost and could be used to generate a system load on the quantum machine interface with limited amount of data generated. As an example, consider Rule 1 and the challenge operation. If multiple requests of the same class of commands are to be issued to the quantum machine to perform a denial-of-service attack, the machine should respond with increasingly higher latency to this type of requests to invalidate the attack.

Optional HAL Packages/Modules

The following modules MAY be considered in the future releases.

10.1 Boson sampling HAL commands for photonic qubits

Boson Sampling is a catchall term for a set of NISQ devices in hardware based on today's photonic technologies.

10.2 HAL transpiler module support

CNOT gates are implemented on hardware using sets of native gates. Therefore, a transpile step is required to transform a CNOT gate into hardware compatible gate sequences. It is also possible to perform optimisation, converting native gate sequences into an equivalent but shorter circuit. The transpiler would generate circuits comprised of Level 1 commands.

Standards and Interfaces

Chapter 11

Chapter 12

Use Case Scenarios

To demonstrate the need for multiple HAL levels and the algorithms that can be run on each level, we provide example pseudocodes of the following algorithms:

1. Shor's Algorithm: Using Kitaev's Quantum Fourier Transform (QFT) approach, the qubit count of the quantum circuits run as part of Shor's Algorithm can be reduced. However, this reduction in qubit count needs to be compensated for by performing intermediate measurements on the QFT qubit and also applying rotation gates conditional on intermediate measurement results [10]. Hence, this algorithm will require Level 1 HAL access.
2. HoloVQE: Circuits run as part of the HoloVQE algorithm [11] require intermediate measurements on qubits and require intermediate qubit resets. For a user to implement active qubit reset themselves, they will require HAL Level 1 access due to the very low latency required. However, some hardware manufacturers may want to provide active qubit reset capabilities themselves as a HAL Level 2/Level 3 command. Hence, this is an example of an algorithm that will require HAL Level 2/3 depending on the available optional HAL command.

The pseudocodes are given in Appendix 2. Further use cases will be added in the future versions of this document, for example a minimal example of a conventional VQE code, and a minimal example of an alpha-VQE code.

10 Monz, Thomas, et al. "Realisation of a scalable Shor algorithm." Science 351.6277 (2016): 1068-1070.
11 Foss-Feig, Michael, et al. "Holographic quantum algorithms for simulating correlated spin systems." https://arxiv.org/abs/2005.03023 (2020).

Appendix 1

13.1 Notes and questions

1. How will metadata be implemented in other systems? Is this info stored in a config file? Assuming an application developer is writing an algorithm that then needs to be built to run on hardware, it would make sense that this data is store on the user's machine and used to compile and build?
2. Is metadata confined to static information that is especially useful at initialisation time? Does it include dynamic properties? Can it be extended by the vendor to include information that is specific to a specific machine, such as the calibration and quality of specific qubits hints that can be used by algorithm developers to select how to distribute computation across the qubits made available to them or even for the OS to allocate qubits based on quality or length of computation or even cost to the application customer for their use. These factors could make it important to authenticate such information. Therefore, should the metadata also be decomposed into "levels" of revelation? [Should be addressed by the Section Metadata Specification]
3. Should we have controlled rotation gates? [Added to the Notes on commands Layer 2 and 1]
4. Multi-user and session management – should this be part of the specification? [Addressed by HAL Commands Minimum Requirements]
5. Should qubit indices be handled as indices rather than bit fields? [Addressed in Command Format: Option I]
6. Should we include gates with more than two qubits? ? [Added to the Notes on commands Layer 2 and 1]
7. Add a command to the standard regarding selection of the backend - emulator or hardware. Deltaflow.OS can utilise this command to select a backend for HAL Software to run the algorithms on? [Addressed by HAL Commands Minimum Requirements]

Appendix 2: Use Cases

14.1 Use Case 1 – Shor's Algorithm

Here we provide two ways of implementing quantum circuits used in Shor's algorithm The first implementation uses a FOR loop to repeat sets of circuit operations whereas the second implementation avoids using a loop by repeating the code for the set of circuit operations an appropriate number of times. Another difference between the two implementation is that in the first implementation, consecutive controlled phase gates are combined using classical logic before the command is sent to the quantum device.

14.1.1 Implementation 1 pseudocode

```
/*
* Shor's algorithm circuit
* 1+4 qubits example with k=3, N = 15, a = 11
*/

/* 1 + number of bits used to represent N */
int n = 5;

/* number of QFT bits */
int k = 3;

/* declare qubit register with n qubits */
qubit q[n];

/* declare classical bit register with QFT bits ordered from most
* significant(c[0]) to least significant (c[k-1]) */
bits[k] c;

/* initialise qubit register */
reset q;

/* prepare qubits register |00001> */
x q[n-1];

/* define function that runs the appropriate pulse sequence */
def apply_controlled_unitary(int[k]: op_index) {
        // Apply appropriate controlled unitary
        //  = controlled-a^(2^(k-(1+op_index)))%N

        if (op_index == 0) {
                // Apply controlled (11^4)%15 = controlled 1%15
                // Identity operation => Nothing to do
```

```
        }

        if (op_index == 1) {
                // Apply controlled (11^2)%15 = controlled 1%15
                // Identity operation => Nothing to do
        }

        if (op_index == 2) {
                // Apply controlled (11^1)%15 = controlled 11%15

                // swap q[2] and q[4] conditioned on q[0]
                cswap q[0] q[2] q[4];
                cswap q[0] q[1] q[3];

                // apply X on q[1] conditioned on q[0]
                cx q[0] q[1];
                cx q[0] q[2];
                cx q[0] q[3];
                cx q[0] q[4];
        }
}

// Shor's algorithm loop
for i in [0: k - 1] {
        // Reset QFT qubit to |0> state
        reset q[0];

        // Apply Hadamard gate to create |+> state
        h q[0];

        apply_controlled_unitary(i)

        /* phase shift to apply depends on previous measurements;
        * sum up the phase rotation angles and then apply a
        * phase gate with the summed angle */
        float phase_shift = 0;

        if (c[0] == 1) {
            phase_shift += pi/2;
        }
        if (c[1] == 1) {
            phase_shift += pi/4;
        }
        if (c[2] == 1) {
            phase_shift += pi/8;
        }
        rz (phase_shift) q[0];

        // Apply Hadamard
        h q[0];

        /* Newest measurement outcome is associated with a pi/2 phase shift
        * in the next iteration, so shift all bits of c to the right */
        c >>= 1;

        /* Measure QFT qubit and save result to 0th index of
        * classical bit register */
        measure q[0] -> c[0];
}
```

14.1.2 Implementation 2 pseudocode

```
/*
* Shor's algorithm circuit
* 1+4 qubits example with k=3, N = 15, a = 11
*/


/* 1 + number of bits used to represent N */
int n = 5;

/* number of QFT bits */
int k = 3;

/* declare qubit register with n qubits */
qubit q[n];

/* declare classical bit register with QFT bits ordered from most
* significant(c[0]) to least significant (c[k-1]) */
bits[k] c;

/* initialise qubit register */
reset q;

/* prepare qubits register |00001> */
x q[n-1];

/* Shor's algorithm loop
* ------------------------------------------------
* ------------------- k = 0 ----------------------
* ------------------------------------------------ */

// reset QFT qubit to |+> state
reset q[0];
h q[0];

// apply controlled (11^4)%15 = controlled 1%15
// Identity operation => Nothing to do

// phase shift to apply depends on previous measurements
if (c[0] == 1) {
        rz (pi/2) q[0];
}
if (c[1] == 1) {
        rz (pi/4) q[0];
}
if (c[2] == 1) {
        rz (pi/8) q[0];
}

h q[0];
/* newest measurement outcome is associated with a pi/2 phase shift
* in the next iteration, so shift all bits of c to the right */
c >>= 1;
measure q[0] -> c[0];

/* ------------------------------------------------
*  ------------------- k = 1 ----------------------
*  ------------------------------------------------ */

// reset QFT qubit to |+> state
reset q[0];
```

```
h q[0];

// apply controlled (11^2)%15 = controlled 1%15
// Identity operation => Nothing to do


if (c[0] == 1) {
        rz (pi/2) q[0];
}
if (c[1] == 1) {
        rz (pi/4) q[0];
}
if (c[2] == 1) {
        rz (pi/8) q[0];
}

h q[0];
/* newest measurement outcome is associated with a pi/2 phase shift
* in the next iteration, so shift all bits of c to the right */
c >>= 1;
measure q[0] -> c[0];

/* ------------------------------------------------
*  ------------------- k = 2 ----------------------
*  ------------------------------------------------ */

// reset QFT qubit to |+> state
reset q[0];
h q[0];

// apply controlled (11^1)%15 = controlled 11%15
cswap q[0] q[2] q[4];
cswap q[0] q[1] q[3];
cx q[0] q[1];
cx q[0] q[2];
cx q[0] q[3];
cx q[0] q[4];

if (c[0] == 1) {
        rz (pi/2) q[0];
}
if (c[1] == 1) {
        rz (pi/4) q[0];
}
if (c[2] == 1) {
        rz (pi/8) q[0];
}

h q[0];
/* newest measurement outcome is associated with a pi/2 phase shift
* in the next iteration, so shift all bits of c to the right */
c >>= 1;
measure q[0] -> c[0];

/* ------------------------------------------------
*  ------------------- DONE -----------------------
*  ------------------------------------------------ */
```

14.2 Use Case 2 – holoVQE

Below is an implementation for a single circuit run of the XXZ model energy calculation circuit in

[REF_5]. The circuit requires intermediate measurements and resets of qubits, but, it does not require modifying the circuit based on the measurement outcomes. Hence, assuming the hardware supports active qubit reset as a Level 2 (3) command, this is an example of a Level 2 (3) HAL algorithm.

Note that if active qubit reset is not available, the algorithm can be run using Level 1 HAL by replacing:

```
reset q[1];
```

with the following:

```
measure  q[1] -> c[0];
if (c[0] == 1) {
    x q[1];
}
```

```
/*
* holoVQE circuit for XXZ spin chain energy calculation
* 1 physical qubit, 1 bond qubit; 4 'burn in' lattice sites
*/

/* number of 'burn in' state preparation lattice  sites */
int lattice_sites = 4;

/* declare qubit register with 2 qubits (1 bond, 1 physical) */
qubit q[2];

/* declare classical bit register with 4 bits  (4 measurement results stored) */
bits[4] c;

/* parametrised angle */
float theta = 1.234;

/* initialise qubit register */
reset q;

// State preparation
for i in [0: lattice_sites - 1] {
    // Apply G_theta
    rx (pi/2) q[0];
    ry (pi/2) q[1];
    cz q[0] q[1];
    rx (-theta) q[0];
    ry (theta) q[1];
    cz q[0] q[1];
    rx (-pi/2) q[0];
    ry (-pi/2) q[1];

    // Reset physical qubit
    reset q[1];

    // Apply G_theta_tilda
    rx (pi/2) q[0];
    ry (pi/2) q[1];
    cz q[0] q[1];
    rx (-theta) q[0];
    ry (theta) q[1];
    cz q[0] q[1];
    rx (-pi/2) q[0];
    ry (-pi/2) q[1];
    x q[1];

    // Reset physical qubit
```

```
    reset q[1];
}
//Expectation value measurement

//Apply G_theta, measure in X basis, then reset physical qubit
rx (pi/2) q[0];
ry (pi/2) q[1];
cz q[0] q[1];
rx (-theta) q[0];
ry (theta) q[1];
cz q[0] q[1];
rx (-pi/2) q[0];
ry (-pi/2) q[1];

h q[1];
measure q[1] -> c[0];

reset q[1];

//Apply G_theta_tilda, measure in X basis, then reset physical qubit
rx (pi/2) q[0];
ry (pi/2) q[1];
cz q[0] q[1];
rx (-theta) q[0];
ry (theta) q[1];
cz q[0] q[1];
rx (-pi/2) q[0];
ry (-pi/2) q[1];
x q[1];

h q[1];
measure q[1] -> c[1];

reset q[1];

//Apply G_theta, measure in Z basis, then reset physical qubit
rx (pi/2) q[0];
ry (pi/2) q[1];
cz q[0] q[1];
rx (-theta) q[0];
ry (theta) q[1];
cz q[0] q[1];
rx (-pi/2) q[0];
ry (-pi/2) q[1];

measure q[1] -> c[2];

reset q[1];

//Apply G_theta_tilda, measure in Z basis, then reset physical qubit
rx (pi/2) q[0];
ry (pi/2) q[1];
cz q[0] q[1];
rx (-theta) q[0];
ry (theta) q[1];
cz q[0] q[1];
rx (-pi/2) q[0];
ry (-pi/2) q[1];
x q[1];

measure q[1] -> c[3];

reset q[1];
```

```
//-------------------------------------------------
//-------------------- DONE -----------------------
//-------------------------------------------------
```

Further reading

- [REF_1] Practical Quantum Computing: the value of local computation; James R. Cruise,Neil I Gillespie, Brendan Reid; Riverlane Ltd; Sep 2020; https://arxiv.org/abs/2009.08513
- [REF_2] J M Pino et al. Demonstration of the QCCD trapped-ion quantum computer architecture. 2020. https://arxiv.org/abs/2003.01293