
# Axon Driver User Guide

*06/23/2022*
*Revision 1.0*

*Atlazo Inc*
*4250 Executive Square, Suite 675*
*La Jolla, CA 92037*

**Abstract**

This document describes the usage and integration of the Axon software device driver onto a host processor SoC.

## Revision History

| Version | Date | Description |
|---|---|---|
| 1.0 | *06/23/2022* | Initial document |

# PREFACE

*About This Document*

This document provides information on how the Axon driver can be integrated into a host processor software stack, and the usage of the driver functionality.

*Related Detailed Design Documents*

# REFERENCES

- Axon TensorFlow Pre-compiler User Guide
- TLSR9x Axon Software Development Kit User Guide

# ACRONYMS AND DEFINITIONS

*Glossary*

Axon – Atlazo's proprietary machine learning accelerator core.

FFT – Fast Fourier Transform

FIR – Finite Impulse Response filter

MAR – Multiply/Accumulate Recursive

RTOS – Real Time Operating System

WFI – Wait For Interrupt machine processor instruction

# TABLE OF CONTENTS

 

 

# 1 AXON DRIVER OVERVIEW

The Axon driver is delivered as “bare-metal”, precompiled, library code. It has no compilation dependencies on the host processor and operating system other than the processor architecture and compilation tool chain.

## 1.1 HOST INTEGRATION

The driver provides link-time and run-time provisions for integrating with the host SoC. These provisions include:

- **Driver Provisioning**
  - The host provisions one driver instance per Axon core instance in the SoC.
  - The host provisions memory buffers for each instance; the driver has no internal variables.
- **RTOS/OS Integration**
  - The host registers each Axon instance with its interrupt vector.
  - The host provides mutex get/release services.
  - The host provides notification functions for signalling that an interrupt needs to be serviced.

    **NOTE: The Axon driver supports synchronous and asynchronous operation. RTOS/OS integration is only required for asynchronous operation.**
- **Debug Logging**
  - The host provides a function for logging messages from the driver.

## 1.2 USER FUNCTIONALITY

The driver provides a variety of vector based mathematical operations. These operations can be performed discretely or be pre-defined and repeatedly invoked in a batch.

Additionally, operations can be executed synchronously (ie, the operations are completed when the function returns) or asynchronously (a notification will occur when the operations have completed).

# 2 AXON DRIVER INTEGRATION GUIDE

Integrating axon onto a host system requires the integrator to choose an execution model, implement the host integration functions, and allocate memory to the driver. Of these three tasks, choosing the execution model is the most complex.

## 2.1 CHOOSING AN EXECUTION MODEL

An execution model is a combination of the following choices (though not all combinations are valid)

- Discrete vs Batch vs Queued Batches

- Synchronous vs Asynchronous execution
- Execution Context, “RTOS” vs “Bare Metal” (for asynchronous execution)
- Internal WFI vs External (or no) WFI (for synchronous execution)

### 2.1.1 Discrete vs Batch Mode vs Queued Batch Mode

In *discrete* mode, users specify all aspects of the operation each and every time the function is invoked, and only one operation will execute within the function. The driver performs parameter checking. Examples of discrete APIs include `AxonApiFft` and `AxonApiFir`.

In *batch* mode, the user pre-defines one or more operations then invokes `AxonApiExecuteOps` with a list of operations to perform. The driver performs parameter checking in the operation define APIs, but not as part of execution.

After submitting a batch, no other API calls can be made until the driver has completed the batch.

**NOTE: the user must guarantee that the underlying parameters for each operation (specifically the input and output buffer locations) must not be altered for a defined operation. The contents of the input buffer are the only thing that can be changed between operation invocation.**

Examples of operation define APIs include `AxonApiDefineOpFft` and `AxonApiDefineOpFir`.

*Queued batch* mode extends batch mode by allowing the user to queue multiple batches and receive a separate callback when as each batch is completed. Batches can be added to the queue at any time; Axon does not need to be idle to add a batch to the queue. This makes it easier to manage multiple independent users of Axon. `AxonApiQueueOpsList` supports queued batches.

Batch and Queued Batch modes are the preferred operational modes to maximise axon driver performance.

### 2.1.2 Synchronous vs Asynchronous Mode

Both discrete and batch functions can execute synchronously or asynchronously. Driver APIs with the `async_mode` parameter support asynchronous operation.

*Synchronous* functions do not return until all operations are complete. This mode can be simpler to implement, but at the potential cost of power and CPU cycles, as the driver will poll axon hardware until operations are complete. Synchronous operations are preferred for smaller operations (ones with less input data and/or operations).

*Asynchronous* functions will initiate the axon operations then return without waiting for completion. An interrupt is asserted when the operations complete, and the driver will invoke `AxonHostInterruptNotification` if there is a result ready. The user must then call `AxonApiGetAsyncResult` to retrieve the result.

Asynchronous operation gives the host CPU an opportunity to sleep or service lower priority threads, but at the expense of context switching overhead.

Synchronous with WFI has the calling simplicity of synchronous mode, but with the potential power savings of asynchronous mode. To support this mode, the host must implement interrupt support for

 

Axon, AxonHostDisableInterrupts(), and AxonHostWfi(), and AxonHostEnableInterrupts(). This mode will enter WFI state when waiting for Axon operations to complete, and use the Axon interrupts to detect completion.

### 2.1.3 Execution Context "RTOS" vs "Bare Metal" (for Asynchronous)

Asynchronous mode relies on callback functions invoked from the Axon interrupt context.

The implementer has the choice of either using the callback to signal a user context thread, or to process the callback entirely in the interrupt context. The former is typically done when an RTOS is available, and the latter is for a bare-metal system.

### 2.1.4 Internal WFI vs External (or no) WFI (for Synchronous)

WFI (wait for interrupt) is an instruction that allows the processor to enter a low power state (ie clock gating) while allowing peripherals to continue to run.

In very simple, bare-metal, synchronous systems, Axon driver can invoke WFI while waiting for Axon to complete its workload.

### 2.1.5 Execution Models

#### *2.1.5.1 Discrete, Synchronous Bare Metal Execution with External (or no) WFI*

This is the simplest model. The system consists of a single, master loop, and invokes discrete Axon functions in synchronous mode in response to system events.

- No need for an Axon interrupt handler and signalling events.
- No need for Axon host WFI and Interrupt Disable/Restore.
- No operations are predefined (AxonInternalBuffers can be sized to 1)
- User can optionally invoke WFI in the master loop when no more processing is pending).

This model is most appropriate for demos and systems where Axon's workload is small and infrequent. The power savings of entering WFI will be trivial as Axon is performing a single operation at a time.

#### *2.1.5.2 Batch, Synchronous Bare Metal Execution with Internal WFI*

This adds some complexity to the Discrete/External WFI model, but is still a synchronous model (ie, fairly simple to implement). The benefit is that larger workloads can be performed with better power savings.

- Needs an Axon interrupt handler (but no signalling).
- Needs Axon host WFI and Interrupt Disable/Restore.
- AxonInternalBuffers must be sized to the number of operations to pre-define.

This model is most appropriate for demos and systems where Axon's workload is larger but very deterministic, and the CPU is not needed for other tasks while Axon is busy. The power savings of entering WFI for longer periods outweighs the simplicity of discrete/no WFI mode.

#### *2.1.5.3 Queued Batches, Asynchronous Bare Metal Execution with External WFI*

Asynchronous operation is inherently more complex than synchronous, but also more powerful and efficient. In a complex system with many different axon work-loads executing on different schedules,

 

Queued Batches simplifies state machine management by allowing each work load to execute independently of others and with dedicated call backs.

Queued batches and the axon driver simplify state machines by removing the need to wait for Axon to be idle before starting a batch. Instead, Axon will link the batch into its queue, and execute it when everything before it is completed.

In bare metal/asynchronous mode, the axon ISR context is used to queue addition work

This adds some complexity to the Discrete/External WFI model, but is still a synchronous model (ie, fairly simple to implement). The benefit is that larger workloads can be performed with better power savings.

- Needs an Axon interrupt handler.
- `AxonHostInterruptNotification` must invoke `AxonApiGetAsyncResult`.
- Needs Axon host WFI and Interrupt Disable/Restore.
- Each operation batch requires a callback function.
- AxonInternalBuffers must be sized to the number of operations to pre-define.

This model is most appropriate for bare metal dynamic systems where Axon's workload is large but not deterministic. Algorithms that use axon execute on different schedules making it impossible to define a single operations list that covers all scenarios.

#### *2.1.5.4 Queued Batches, Asynchronous RTOS Execution with External WFI*

This model differs from the bare metal version in that the Axon driver ISR is used to signal a thread that follows up with the processing of the event outside of the interrupt context.

This conforms to a traditional RTOS paradigm of ISRs being short and sweet to reduce to allow the thread priorities determine the order of execution.

#### *2.1.5.5 Batch, Asynchronous Bare Metal Execution*

This is a legacy mode of operation. It is almost identical to Queued Batches in complexity, but without the flexibility to start batches at any time.

## 2.2 Implementing Host Integration Functions

Host integration functions are declared in axon_dep.h. Functions beginning with "AxonHost" are implemented by the host.

Some of the functions are common to all execution models, and others have requirements specific to the execution model.

### 2.2.1 Functions common to all execution models

#### *2.2.1.1 void AxonHostLog(AxonInstanceStruct *axon, char *msg);*

This function writes a debug message out of a serial port. This function is not strictly required by the axon driver itself, but most of the sample code provided in the SDK depends on it. The AxonInstanceStruct argument can be used to access the serial log buffer allocated to it.

#### *2.2.1.2 uint32_t AxonHostTransformAddress(uint32_t from_addr);*

Certain systems will have a different address map for the Axon peripheral vs the CPU. This function is required to map addresses from CPU address space to Axon's address space.

#### *2.2.1.3 uint32_t AxonHostGetTime();*

Like AxonHostLog(), this function is not required by the driver itself, but is used by much of the sample code.

#### *2.2.1.4 void AxonHostAxonEnable(uint8_t power_on_reset);*

Performs all low-level initialization to enable Axon (enable clocks, head switches, etc). Implementation should call this at start-up before initializing the driver. Implementation should call AxonReInitInstance for all invocations except the 1st (when power_on_reset==1). A call AxonHostAxonDisable() disables/turns-off Axon, and AxonHostAxonEnable() must be called subsequently before using Axon again.

Like AxonHostLog(), this function is not required by the driver itself, but is used by much of the sample code.

#### *2.2.1.5 void AxonHostAxonDisable();*

Powers off Axon. After being called, Axon cannot be used until AxonHostAxonEnable() is called.

Like AxonHostLog(), this function is not required by the driver itself, but is used by much of the sample code.

### 2.2.2 Functions specific to execution models

#### *2.2.2.1 void AxonHostInterruptNotification(AxonInstanceStruct *axon);*

This function must be provided by the host in order for the axon driver to successfully link into an application. Its implementation differs depending on the execution model:

***Synchronous (with or without internal WFI):***

This function can be empty; it does not get called.

***Queued Batch, Asynchronous Bare Metal:***

The implementation must call AxonApiGetAsyncResult().

***Queued Batch, Asynchronous RTOS:***

 

The implementation must signal a user thread, which in turn must call AxonApiGetAsyncResult().

#### *2.2.2.2 uint32_t AxonHostDisableInterrupts();*

This function disables interrupts and returns the state of interrupts prior to being disabled. The result is passed directly to AxonHostRestoreInterrupts().

This function is used by the driver for two conditions only:

1) ***Synchronous with internal WFI***, immediately before executing AxonHostWfi()
2) ***Queued Batch mode***, for the time it takes to append a batch to the queue.

#### *2.2.2.3 void AxonHostWfi();*

This function executes the WFI instruction. It will be invoked immediately after invoking AxonHostDisableInterrupts(). It is used internally by the driver only for Synchronous with internal WFI, but is also used by sample code for external WFI examples.

The expected processor behavior is:

AxonHostDisableInterrupts() prevents interrupts from being processed, but not from being asserted.

An interrupt that asserts during AxonHostWfi() will wake the processor and will execute the instruction immediately following WFI.

AxonHostRestoreInterrupts() will cause interrupts to be processed, and the highest priority pending interrupt will be vectored to immediately.

#### *2.2.2.4 uint32_t AxonHostRestoreInterrupts();*

This function restores interrupts to the state they were prior to being disabled by AxonHostDisableInterrupts(). The result from AxonHostDisableInterrupts() is passed directly to AxonHostRestoreInterrupts().

This function is used by the driver for two conditions only:

1) ***Synchronous with internal WFI***, immediately after executing AxonHostWfi()
2) ***Queued Batch*** mode, after the batch has been appended to the queue.

## 2.3 PROVISIONING MEMORY FOR THE DRIVER

The user provisions memory for the driver by declaring AxonInstanceStruct instance in retained, non-stack memory, and populating the AxonInstanceHostProvidedStruct fields within it.

The caller must provide the address of the Axon instance in the base_address field, as well buffers and buffer sizes as described below, before calling AxonInitInstance.

 

| Buffer field/ size field | Usage | Memory attributes | Required? |
|---|---|---|---|
| log_buffer/ log_buffer_size | Buffer for formatting log messages. | No retention/alignment requirements | Not used by the driver directly so can be NULL. Provided as a convenience for users. |
| internal_buffers/ internal_buffer_size | Buffers for storing user-defined operations. | Retained memory, 16 byte aligned. | At least one buffer is required. One additional buffer is required for each pre-defined operation that is to be stored. |
| acorr_buffer | Buffer used to support autocorrelation operation. | Unretained memory, 4 byte aligned. | Only required if ACORR operation is used. Can be NULL if system does not use ACORR. |
| matrix_mult_buffer/ matrix_mult_buffer_size | Internal operation buffer for matrix multiplication. | Unretained memory, 16 byte aligned. | Only required if matrix multiplication is used. Driver needs a minimum of 1 for matrix multiplication, but can use up to 16. Higher numbers result in lower power/faster execution of matrix multiplication. |
| mm_line_buffer/ mm_line_buffer_size | Buffer for storing matrix rows in matrix multiplication. | Unretained memory, 16 byte aligned. | Only required if matrix multiplication is used. Needs to sized to hold a minimum of 2 rows of the largest matrix row to be multiplied. Sizing needs to account for 2 rows back-to-back with padding between to align on a 16byte boundary. |

# 3 AXON DRIVER USAGE GUIDE

Axon Driver APIs can be divided into two classes, *operation* APIs and *control* APIs.

Operation APIs define or perform a mathematical operation on one or two vectors of input data. All operation APIs take as input an AxonInputStruct argument (though not all fields are used for every API) and either define an operation for later use or execute it immediately.

Control APIs are used for controlling when Axon executes and retrieving status information.

## 3.1 OPERATIONS

The axon driver supports the following operations.

*Table 1 Axon Driver Functions*

| OP | Equation | Max Size | Condition |
|---|---|---|---|
| FFT | $O = FFT(X)$ | 512 | n = 32, 64, 128, 256, 512 |
| FIR | $O[i{:}f..n-1] = \sum_{j=0}^{f-1} X_{i-j} \cdot F_j$ | 1024 | n = mult of 4<br>f = mult of 4<br>n - f >= 4 |
| SQRT | O = SQRT.FX(X) | 512 | n = mult of 2 |
| LOGN | O = LOGN.FX(X) | 512 | n = mult of 2, Q11.12 |
| EXP | O = EXP.FX(X) | 512 | n = mult of 2 |
| MAR | $o = \sum_{i=0}^{n-1} X_i \cdot Y_i$ | 1024 | n = mult of 4 |
| ACORR | $O[t{:}0..d-1] = \sum_{i=t}^{n-1} X_{i-t} \cdot X_i$ | 512 | n = mult of 4<br>d < n |
| L2NORM | $o = \sum_{i=0}^{n-1} X_i^2$ | 1024 | n = mult of 4 |
| ACC | $o = \sum_{i=0}^{n-1} X_i$ | 1024 | n = mult of 4 |
| XPY/XMY | $O = X \pm Y$ | 512 | n = mult of 2 |
| XSPYS/XSMYS | $O = X^2 \pm Y^2$ | 512 | n = mult of 2 |
| AXPBY | $O = aX + bY$ | 512 | n = mult of 2 |
| AXPB | $O = aX + b$ | 512 | n = mult of 2 |
| XTY | $O = X \cdot Y$ | 512 | n = mult of 2 |
| XS | $O = X^2$ | 512 | n = mult of 2 |
| RELU | O = RELU(X) | 512 | n = mult of 2 |
| MatrixMult* | O[1,m] = X[1,n] • Y[n,m] | 512 | |
| MemCpy/ MemCpySafe* | O=X | None | |
| FullyConnected* | 8bit quantized fully-connected (dense) layer with optional normalization | | |

Note the following conventions/requirements:

- "n" is the length of the input vector(s)
- Q11.12 format is a 24bit, fixed point number format. Bits[11:0] are the fractional portion, bits[22:12] are the whole number portion, and bit[23] is the sign bit.
- Lower case letters represent scalars, upper case represent vectors.
- All Axon operation APIs accept an `AxonInputStruct` structure that contains the operation parameters. Not all fields of this structure are used for all operations; consult the specific API declaration for the precise use of the fields for that operation.

- All pointers provided to APIs must persist for the life of the API or the life of the operation handle.
- MatrixMult* operation is in implemented as a series of MARs by the driver. It copies y_in from Flash into RAM and performs padding/alignment internally.
- MemCpy/MemCpySafe* is implemented as software within the driver. It allows parameters to be copied from Flash to RAM in an optimized manner; just-in-time, and while Axon hardware is busy computing. MemCpySafe differs from MemCpy in that MemCpy can occur while previous operations are executing in hardware, MemCpySafe waits for Axon hardware to be idle before performing the copy.
- FullyConnected returns a series of operations that implement a complete fully-connected(dense) layer using quantized 8bit weights.

### 3.1.1 Common Operation Parameters

The behavior of each operation is specified through an AxonInputStruct parameter. Behaviours that are controlled through it include input/output data location, data width, data packing, data stride

#### *3.1.1.1 Data Width*

Axon supports data widths of 8, 12, 16, and 24bits. All vector operations except matrix multiply require the input and output data width must match (the exception is matrix multiply, where the 8bit Y matrix values can be converted from 8 to 16 bit).

Scalar operations (L2Norm, ACC, MAR) produce a 32bit output regardless of input data width.

Vector operations FFT, FIR, Square Root, Natural log, and Exp only support 24bit data.

Note that 12 bit values are sign extended to 16 bits, and 24bit values are sign extended to 32 bits.

#### *3.1.1.2 Data Packing*

Data packing refers to how data is stored in memory.

“Unpacked” data is aligned at 32bits regardless of data width.

“Packed” data is aligned to the next 8bit boundary for that data width as follows:

- 8bit data width => 8bit data alignment
- 12bit data width => 16bit data alignment
- 16bit data width => 16bit data alignment
- 24bit data width => 32bit data alignment

#### *3.1.1.3 Data Stride*

Each of the three vectors (X, Y, and Q) support an independent stride value of 1 or 2.

Striding is useful when channel data is interleaved but only one channel is to be processed (ie, left and right audio). The stride value is in units of data width if data packing is enabled, and 32bits if data packing is disabled.

Note that striding does not impact the length parameter; if there are 512 elements in a vector then this is true regardless of the stride value.

#### *3.1.1.4 Output Rounding*

Vector operation output values (except FFT) can be rounded by up to 31 bits. Scalar operation (MAR, L2NORM, ACC) output values cannot be rounded.

User specifies number of bits to round to ROUND_BITS. Each output value is divided by $2^{ROUND\_BITS}$, then rounded up if

1) the remainder is greater than .5 (ie, the MSB and at least 1 other bit are set in the remainder) or

2) The remainder equals .5 (ie, only the MSB is set in the remainder) and the quotient is an odd value (LSB is set).

#### *3.1.1.5 Output processing (ie, Activation) functions*

All vector output functions except SQRT, LOGN, and ExP support output processing functions *RELU*, Sigmoid, Tanh, amd QuantSigmoid.

The output processing function occurs after rounding.

RELU:

$$O_i = max(0,O_i)$$

Sigmoid:

("S" shaped curve that asymptotically approaches 0 in the negative direction and 1 in the positive direction)

$$O_i = \frac{e^{Oi}}{e^{Oi}+1}$$

Tanh:

(Similar to sigmoid, but 2x the slope and ranging between –1 and + 1)

$$O_i = \frac{e^{Oi} - e^{Oi}}{e^{Oi}+e^{Oi}}$$

QuantSigmoid:

(Add 1, divide by 2, then clamp between 0 and 1)

$$O_i = MAX(0,MIN(1,(Oi+1)/2 + 0.5))$$

#### *3.1.1.6 Data Buffers*

Axon operations accept up to two input vectors (X and Y), up to two input scalars (a and b), and one output vector (Q).

Axon operations can execute “in place” (output buffer is the input buffer).

For operations AXPB and AXPBY, the A and B scalar values can be passed by value if they or constant, or by address if the scalar values are themselves the output of an operation.

When using packed data, vector buffers have the following alignment requirements:

- Data width = 8bits => 16byte alignment
- Data width = 12/16bits => 8byte alignment
- Data width = 24bits => 4byte alignment

**TLSR9x NOTE: Axon cannot access flash on this device; so all buffers must be located in RAM.**

**For regular batch operations, the operation MemCpy will perform a memory copy from Flash to RAM. This operation can be inserted into the operation list to copy from Flash to RAM on an as-needed basis, ideally while Axon hardware is occupied with other operations.**

**MatrixMultiply is the exception. It will copy the y_in vector (matrix) from Flash to RAM without additional operations being defined.**

The descriptions below list the driver API function for each function in table 1, as well as the fields in the input struct that map to the mathematical function inputs and outputs.

The driver release package includes examples on the usage of each function.

### 3.1.2 FFT

$$O = FFT(X)$$

**Driver APIs**

`AxonApiFft` => discrete version

`AxonApiDefineOpFft` => batch version

**Input**

`AxonInputStruct::x_in[AxonInputStruct::length*2]` => array of long int (24bit), complex number pairs, with real components on even indexes (0, 2, 4…) and imaginary components on odd indexes (1, 3, 5…).

`AxonInputStruct::length` => 32, 64, 128, 256, 512 complex number pairs

**Output**

`AxonInputStruct::q_out[AxonInputStruct::length*2]` Same size and format as x_in.

**Packing Support** => *No*

### 3.1.3 FIR

$$O[i{:}\, f..n-1] = \sum_{j=0}^{f-1} X_{i-j} \cdot F_j$$

**Driver APIs**

`AxonApiFir` => discrete version

`AxonApiDefineOpFir` => batch version

**Input**

`AxonInputStruct::x_in[AxonInputStruct::length]` => array of long int (24bit) signal values $X_i$.

`AxonInputStruct::length` => Number of entries in `x_in` $n$. Must be a multiple of 4, <= 512, and at least 4 greater than filter coefficients $f$. Last coefficient must be 0. For filters fewer than 12 coefficients, pad with 0's to 12.

`AxonInputStruct::y_in[AxonInputStruct::y_length]` => array of long int (24bit) filter coefficient values $F_j$.

`AxonInputStruct::y_length` => Number of entries in `y_in` $f$. Must be a multiple of 4, >= 12, <= 508, and at least 4 less than filter coefficients $n$. For filters with fewer than 12 coefficients the buffer must be 0 padded.

**Output**

`AxonInputStruct::q_out[AxonInputStruct::length-AxonInputStruct::y_length]` Same format as `x_in`.

**Packing Support** => *No*

### 3.1.4 Square Root

$$O = \sqrt{X}$$

**Driver APIs**

`AxonApiSqrt` => discrete version

`AxonApiDefineOpSqrt` => batch version

**Input**

`AxonInputStruct::x_in[AxonInputStruct::length]` => array of long int (24bit) signal values $X_i$.

`AxonInputStruct::length` => Number of entries in `x_in` $n$. Must be a multiple of 2 and <= 512.

**Output**

`AxonInputStruct::q_out[AxonInputStruct::length]` Square root for each corresponding entry in `x_in`. `q_out[i]=sqrt(x_in[i])`. Same format and length as `x_in`.

**Packing Support** => *No*

### 3.1.5 Natural Log

$$O = ln(X)$$

**Driver APIs**

`AxonApiLogn` => discrete version

`AxonApiDefineOpLogn` => batch version

**Input**

`AxonInputStruct::x_in[AxonInputStruct::length]` => array of long int (24bit Q11.12) signal values $X_i$.

`AxonInputStruct::length` => Number of entries in `x_in` $n$. Must be a multiple of 2 and <= 512.

**Output**

`AxonInputStruct::q_out[AxonInputStruct::length]` Square root for each corresponding entry in `x_in`. `q_out[i]=ln(x_in[i])`. Same format and length as `x_in`.

**Packing Support** => *No*

### 3.1.6 Exp ($e^x$)

$$O = e^{(X)}$$

**Driver APIs**

`AxonApiExp` => discrete version

`AxonApiDefineOpExp` => batch version

**Input**

`AxonInputStruct::x_in[AxonInputStruct::length]` => array of long int (24bit Q11.12) signal values $X_i$. All values must be positive.

`AxonInputStruct::length` => Number of entries in `x_in` $n$. Must be a multiple of 2 and <= 512.

**Output**

`AxonInputStruct::q_out[AxonInputStruct::length]` Square root for each corresponding entry in `x_in`. `q_out[i]=e^(x_in[i])`. Same format and length as `x_in`.

**Packing Support** => *No*

### 3.1.7 ACC

$$o = \sum_{i=0}^{n-1} X_i$$

**Driver APIs**

`AxonApiAcc` => discrete version

`AxonApiDefineOpAcc` => batch version

**Input**

`AxonInputStruct::x_in[AxonInputStruct::length]` => array of integers to accumulate $X_i$, in packed or unpacked form.

`AxonInputStruct::length` => Number of entries in `x_in` $n$. Must be a multiple of 4 and <= 1024.

**Output**

`AxonInputStruct::q_out[1]` => Sum of all the entries in `x_in[]` $o$. 32 bit integer.

**Packing Support** => *Yes*

### 3.1.8 MAR

$$o = \sum_{i=0}^{n-1} X_i \cdot Y_i$$

**Driver APIs**

`AxonApiMar` => discrete version

`AxonApiDefineOpMar` => batch version

**Input**

`AxonInputStruct::x_in[AxonInputStruct::length]` => array of integers to multiply/accumulate $X_i$, in packed or unpacked form.

`AxonInputStruct::y_in[AxonInputStruct::length]` => array of integers to multiply/accumulate $Y_i$, in packed or unpacked form.

`AxonInputStruct::length` => Number of entries in `x_in[]` and `y_in[]` $n$. Must be a multiple of 4 and <= 1024.

**Output**

`AxonInputStruct::q_out[1]` => Sum of the product of all entries in `x_in[i]` * `y_in[i]` $o$. 32 bit integer.

**Packing Support** => *Yes*

### 3.1.9 ACORR

$$O[t: 0..d-1] = \sum_{i=t}^{n-1} X_{i-t} \cdot X_i$$

**Driver APIs**

`AxonApiAc` => discrete version

`AxonApiDefineOpAcc` => batch version

**Input**

`AxonInputStruct::x_in[AxonInputStruct::length]` => array of integers to autocorrelate $X_i$, in packed or unpacked form.

`AxonInputStruct::length` => Number of entries in `x_in[]` $n$. Must be a multiple of 4 and <= 1024.

`AxonInputStruct::a_in` => Value of the delay $d$.

**Output**

`AxonInputStruct::q_out[AxonInputStruct::a_in]` => Vector of autocorrelation calculated for each value of d from 0..d-1. Same format as input.

 

**Packing Support** => *Yes*

### 3.1.10 L2NORM

$$o = \sum_{i=0}^{n-1} X_i^2$$

**Driver APIs**

`AxonApiL2norm` => discrete version

`AxonApiDefineOpL2norm` => batch version

**Input**

`AxonInputStruct::x_in[AxonInputStruct::length]` => array of integers to square and accumulate $X_i$, in packed or unpacked form.

`AxonInputStruct::length` => Number of entries in `x_in[]` *n*. Must be a multiple of 4 and <= 1024.

**Output**

`AxonInputStruct::q_out[1]` => Sum of the square of all entries in `x_in[]` *o*. 32 bit integer.

**Packing Support** => *Yes*

### 3.1.11 XPY / XMY

*O = X + Y* *O = X - Y*

**Driver APIs**

`AxonApiXpy, AxonApiXmy` => discrete versions

`AxonApiDefineOpXpy, AxonApiDefineOpXmy` => batch versions

**Input**

`AxonInputStruct::x_in[AxonInputStruct::length]` => array of integers to add/subtract $X_i$, in packed or unpacked form.

`AxonInputStruct::y_in[AxonInputStruct::length]` => array of integers to add/subtract $Y_i$, in packed or unpacked form.

 

`AxonInputStruct::length` => Number of entries in `x_in[]` and `y_in[]` *n*. Must be a multiple of 2 (24,16, and 12bit packing) or 4 (8bit packing) and <= 512.

**Output**

`AxonInputStruct::q_out[AxonInputStruct::length]` => Output vector *O:*

`q_out[i]` = `x_in[i]` ± `y_in[i]`. Format matches input.

**Packing Support** => *Yes*

### 3.1.12 XSPYS / XSMYS

$$O = X^2 + Y^2 \qquad O = X^2 - Y^2$$

**Driver APIs**

`AxonApiXspys, AxonApiXsmys` => discrete versions

`AxonApiDefineOpXspys, AxonApiDefineOpXsmys` => batch versions

**Input**

`AxonInputStruct::x_in[AxonInputStruct::length]` => array of integers to square and add/subtract $X_i$, in packed or unpacked form.

`AxonInputStruct::y_in[AxonInputStruct::length]` => array of integers to square add/subtract $Y_i$, in packed or unpacked form.

`AxonInputStruct::length` => Number of entries in `x_in[]` and `y_in[]` *n*. Must be a multiple of 2 (24,16, and 12bit packing) or 4 (8bit packing) and <= 512.

**Output**

`AxonInputStruct::q_out[AxonInputStruct::length]` => Output vector *O:*

`q_out[i]` = `x_in[i]`$^2$ ± `y_in[i]`$^2$. Format matches input.

**Packing Support** => *Yes*

### 3.1.13 AXPBY

$$O = aX + bY$$

**Driver APIs**

`AxonApiAxpby, AxonApiAxpbyPointer` => discrete versions

`AxonApiDefineAxpby, AxonApiDefineAxpby` => batch versions

**Input**

`AxonInputStruct::x_in[AxonInputStruct::length]` => array of integers to multiply by *a* and add to *bY* $X_i$, in packed or unpacked form.

`AxonInputStruct::y_in[AxonInputStruct::length]` => array of integers to multiply by *b* and add to *aX* $Y_i$, in packed or unpacked form.

`AxonInputStruct::a_in` => scalar value to multiply *X a (Axpby), pointer to* scalar value to multiply *X* &*a (AxpbyPointer)*

`AxonInputStruct::b_in` => scalar value to multiply *Y b(Axpby), pointer to* scalar value to multiply *Y* &b *(AxpbyPointer).*

`AxonInputStruct::length` => Number of entries in `x_in[]` and `y_in[]` *n*. Must be a multiple of 2 and <= 512.

**Output**

`AxonInputStruct::q_out[AxonInputStruct::length]` => Output vector *O:*

`q_out[i]= a_in*x_in[i] + b_in*y_in[i].` Format matches input.

**Packing Support** => *Yes*

### 3.1.14 AXPB

$$O = aX + b$$

**Driver APIs**

`AxonApiAxpby, AxonApiAxpbyPointer` => discrete versions

`AxonApiDefineAxpby, AxonApiDefineAxpbyPointer` => batch versions

**Input**

`AxonInputStruct::x_in[AxonInputStruct::length]` => array of integers to multiply by *a* and add to *b* $X_i$, in packed or unpacked form.

`AxonInputStruct::a_in` => scalar value to multiply *X a (Axpb), pointer to* scalar value to multiply *X* &*a (AxpbPointer).*

`AxonInputStruct::b_in` => scalar value to add *b (Axpby), pointer to* scalar value to add &b *(AxpbPointer).*

`AxonInputStruct::length` => Number of entries in `x_in[]` and `y_in[]` *n*. Must be a multiple of 2 and <= 512.

**Output**

`AxonInputStruct::q_out[AxonInputStruct::length]` => Output vector *O:*

`q_out[i]= a_in*x_in[i] + b_in`. Format matches input.

**Packing Support** => *Yes*

### 3.1.15 XTY

$$O = X \cdot Y$$

**Driver APIs**

`AxonApiXty` => discrete version

`AxonApiDefineXty` => batch version

**Input**

`AxonInputStruct::x_in[AxonInputStruct::length]` => array of integers to multiply $X_i$, in packed or unpacked form.

`AxonInputStruct::y_in[AxonInputStruct::length]` => array of integers to multiply $Y_i$, in packed or unpacked form.

`AxonInputStruct::length` => Number of entries in `x_in[]` and `y_in[]` *n*. Must be a multiple of 2 (24,16, and 12bit packing) or 4 (8bit packing) and <= 512.

**Output**

`AxonInputStruct::q_out[AxonInputStruct::length]` => Output vector *O:*

`q_out[i]= x_in[i] * y_in[i]`. Format matches input.

**Packing Support** => *Yes*

### 3.1.16 XS

$$O = X^2$$

**Driver APIs**

`AxonApiXs` => discrete version

`AxonApiDefineXs` => batch version

**Input**

`AxonInputStruct::x_in[AxonInputStruct::length]` => array of integers to square $X_i$, in packed or unpacked form.

`AxonInputStruct::length` => Number of entries in `x_in[]`$n$. Must be a multiple of 2 (24,16, and 12bit packing) or 4 (8bit packing) and <= 512.

**Output**

`AxonInputStruct::q_out[AxonInputStruct::length]` => Output vector *O:*

`q_out[i]= x_in[i] * x_in[i].` Format matches input.

**Packing Support** => *Yes*

### 3.1.17 RELU

$$O = RELU(X) = \max(0, X)$$

**Driver APIs**

`AxonApiRelu` => discrete version

`AxonApiDefineOpRelu` => batch version

**Input**

`AxonInputStruct::x_in[AxonInputStruct::length]` => array of integers to relu $X_i$, in packed or unpacked form.

`AxonInputStruct::length` => Number of entries in `x_in[]`$n$. Must be a multiple of 2 and <= 512.

**Output**

`AxonInputStruct::q_out[AxonInputStruct::length]` => Output vector *O:*

`q_out[i]= max(0,x_in[i]).` Format matches input.

**Packing Support** => *Yes*

 

### 3.1.18 Matrix Multiply

$$O[1,m] = X[1,n] \cdot Y[n,m]$$

**Driver APIs**

`AxonApiMatrixMult`=> discrete version, output format (data width) matches input.

`AxonApiDefineOpMatrixMult`=> batch version, output format (data width) matches input.

`AxonApiDefineOpMatrixMult32BitOutput=>` batch version, output format (data width) is 32bit regardless of input format.

**Input**

`AxonInputStruct::x_in[AxonInputStruct::length]` => array of integers to multiply/accumulate $X_i$, in packed or unpacked form.

`AxonInputStruct::y_in[AxonInputStruct::length][AxonInputStruct::y_length]` => two dimensional array of integers to multiply/accumulate $Y_i$, in packed or unpacked form.

`AxonInputStruct::length` => Number of entries in `x_in[]` and the width of `y_in[]` $n$. Must be <= 1024. Must follow alignment restrictions based on datawidth.

`AxonInputStruct::y_length` => Height of `y_in[]` and number of entries in `q_out[]` $m$.

**Output**

`AxonInputStruct::q_out[0..m-1]` => Sum of the product of all entries in `x_in[i]` * `y_in[j][i]` $o$.

**Packing Support** => *Yes*

**Notes:** Matrix multiply is not natively supported by Axon hardware. Instead, the driver combines a series of MAR operations in an optimized manner.

y_in[] vector can be placed in FLASH without any alignment restrictions.

### 3.1.19 MemCpy/MemCpySafe

$$O[1,n] = X[1,n], O[n+1, n+m] = 0$$

**Driver APIs**

`AxonApiDefineOpMemCpy`=> batch version, output format (data width) matches input.

**Input**

`AxonInputStruct::x_in[AxonInputStruct::length]` => array of element to copy $X_i$, in packed form.

`AxonInputStruct::length` => Number of entries in `x_in[]` $n$.

`AxonInputStruct::y_length` => Number of 0 pad elements to append $m$.

**Output**

`AxonInputStruct::q_out[0..n-1]` => `x_in[0..n-1]` * `y_in[j][i]` $o$.

`AxonInputStruct::q_out[n..n+m-1]` => `0`

**Packing Support** => *Data must be packed.*

**Notes:** MemCpy is implemented in software by the driver. Its intended usage is to copy arguments from Flash into RAM in a manner that performs the copy just before the data is needed and while Axon hardware is busy performing operations. The padding option allows parameters that do not meet the length requirements to be stored without padding in Flash.

MemCpySafe performs the same copy as MemCpy, except it waits for all previous axon operations to complete (in case the buffer being copied to is used by one or more of those operations).

### 3.1.20 FullyConnected

FullyConnected API differs from other Axon driver APIs in that it does not take an AxonInputStruct as a parameter, and it generates multiple operation handles instead of just one. This operation list can be appended to previous operations and be appended to by subsequent operations.

This API defines the operations needed to compute an 8 bit quantized, fully-connected neural net layer:

1) Saturate and pack input vector down to int8 (skipped if input is int8 natively).
2) Perform the input/weights dot product.
3) Add bias vector (bias is generated by the pre-compiler and includes the input zero-point term).
4) Perform the activation function.
5) Perform batch normalization.
6) Quantize the output to int8 stored in int32.

Note that the final output is not saturated to int8. It is possible for it overflow int8. In general this is not an issue because the non-saturated output can be fed directly to the next layer (which will saturate and pack it), and the final layer output is generally much smaller in length and so quantizing it is of little to no value.

Parameters to this API are generated by the Axon TensorFlow pre-compiler; the user generally only needs to provide buffers.

 

Please see the API declaration for more details on the parameters.

## 3.2 CONTROL APIS

The following is an overview of the control API functions. Please see axon_api.h for precise syntax.

### 3.2.1 AxonApiExecuteOps(void *axon_handle, uint32_t op_count, AxonOpHandle ops[], AxonAsyncModeEnum async_mode);

Used for batch mode only. Starts executing the passed batch of operations.

In asynchronous mode, the driver will invoke AxonHostInterruptNotification() one or more times depending on the size of the batch. Each time the user must invoke AxonApiGetAsyncResult() in response until it returns success or an error code, indicating that the batch is completed.

Batch mode can only be invoked when Axon is idle, and no other APIs can be called while it is busy.

### 3.2.2 AxonResultEnum AxonApiGetAsyncResult(void *axon_handle);

Function to invoke after the driver invokes AxonHostInterruptNotification().

Will indicate if all work is complete. In discrete and single batch modes, this means the submitted operation(s) is complete. In queued batches mode, this means all queued batches have completed (each queued batch has a callback function that indicates when it has completed).

### 3.2.3 AxonResultEnum AxonApiFreeOpHandles(void *axon_handle, uint32_t op_count, AxonOpHandle ops[]);

Frees one or more operation handles so they can be redefined for new operations.

### 3.2.4 AxonResultEnum AxonApiQueueOpsList(void *axon_handle, AxonMgrQueuedOpsStruct *ops_info);

Adds a batch of operations to the queue. If Axon is idle, the batch will be started immediately. If other batches are already queued, appends this batch to the end of the queue.

Must not be called if discrete or single batch mode operations are executing.

## 3.3 GUIDELINES

### 3.3.1 Threading/Re-entrancy

In general, the Axon driver is not thread-safe; the user is required to ensure that operations are completed and the driver is idle before accessing the driver.

The only exception is in queued batch mode, where it is safe to add batches to the queue while the driver is busy processing batches (provided AxonHostDisableInterrupts and AxonHostRestoreInterrupts are properly implemented).

### 3.3.2 MemCpy

On SOCs such as TLSR9x, Axon can only operate on buffers in RAM. Since it is not feasible to store large arrays in retained RAM, the memcpy operation can be used to transfer constant values from Flash to a RAM buffer without user software having to intervene. This facilitates longer operation batches with fewer state for user software to manage.

To use MemCpy operation efficiently and safely, it is important to understand how (or actually when) it works.

*Efficiently* means that minimal buffer space is used and mem copying occurs while Axon hardware is busy.

*Safely* means that buffers are not overwritten before they are used.

MemCpy operation works as follows:

1) Perform all MemCpy's at the beginning of the remaining operations in a batch.
2) Launch all operations up until the next MemCpy.
3) Perform the MemCpy.
4) Wait for the launched operations in step 2 to complete.
5) Loop back to step 1 until all operations in the batch have been processed.

The efficiency that is gained is that after step 4, the data has already been copied to RAM and is ready to go. The Axon hardware can be kept busy and does not have to wait for data to be copied from Flash to RAM.

The danger is that the MemCpy cannot operate on memory that is used by the block of operations between it and the previous MemCpy.

A simple, safe, strategy is to allocate to buffers (ping and pong), and make sure that "pong" is being filled while "ping" is being used. Consider the following example:

Op

| No. | Operation |
|---|---|
| 1 | MemCpy to "ping" buffer |
| 2 | Axon op using "ping" buffer |
| 3 | Axon op using ping |
| 4 | MemCpy to "pong" buffer |
| 5 | Axon op using "pong" buffer |

The driver will 1st fill the “ping” buffer, start operations 2-3, fill the “pong” buffer, then wait for operations 2-3 to complete.

Upon completion of operations 2-3, the driver launches operation 5.

The copy to “pong” buffer is essentially free; it occurs while hardware is processing operations 2-3 so operation 5 can be launched immediately.

By separating ping and pong, the ping buffer is not corrupted by operation 4 when it is still in use by operations 2-3.