Embedded Inference Client API

X-CUBE-AI Expansion Package

r5.1
AI PLATFORM r8.0.0 (Embedded Inference Client API 1.2.0)
Command Line Interface r1.7.0

Introduction

This article describes the embedded inference client API which must be used by a C-application layer (AI client) to use a deployed C-model. All model-specific definitions and implementations can be found in the generated C-files: `<name>.c`, `<name>.h` and `<name>_data.h` files (refer to [UM], *"Generated STM32 NN library"* section). For debug, advanced use-cases and profiling purposes, Platform observer API is available (refer to [OBS]).

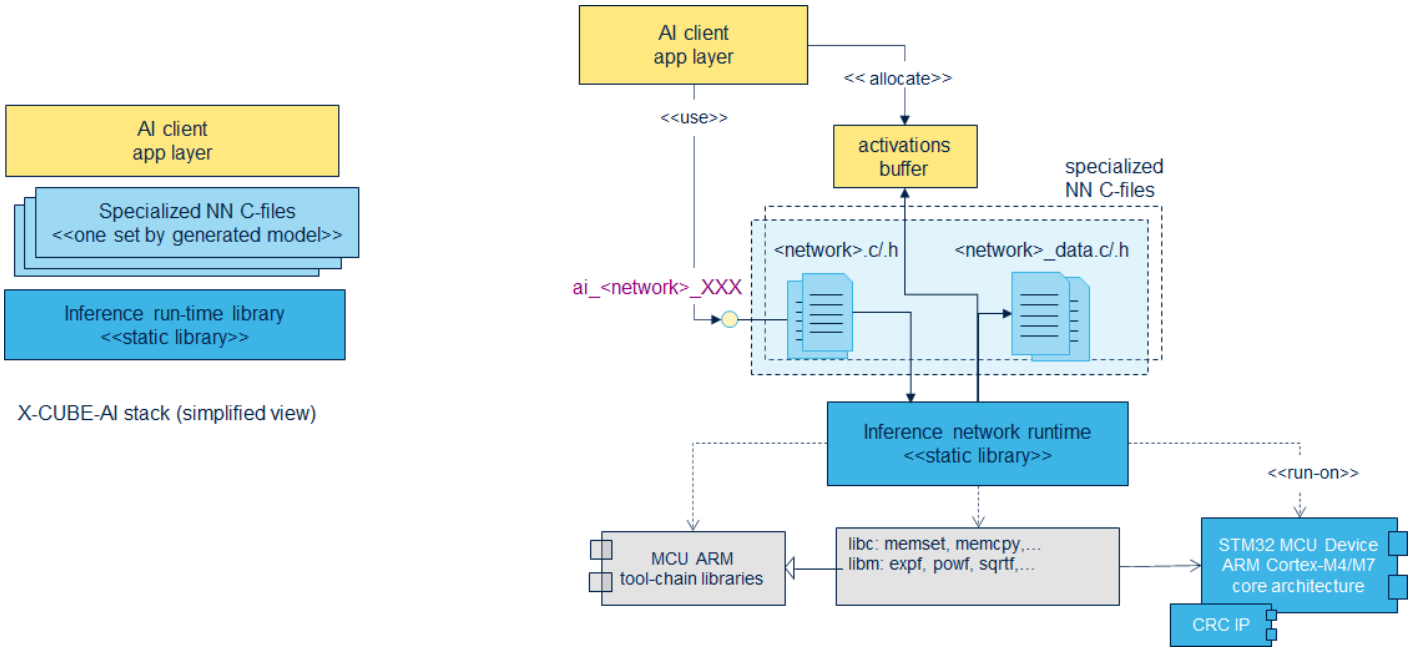

Figure 1: MCU integration model/view and dependencies

Above figure shows that the integration of the AI stack in an application is simple and straightforward. There is few or standard SW/HW dependencies with the run-time. Only STM32 CRC IP should be clocked to be able to use the inference runtime library. AI client uses the generated model through a set of well-defined `ai_<name>_XXX()` functions (also called *"Embedded inference client API"*). The X-CUBE-AI pack provides a compiled library (i.e. network runtime library) by STM32 series and by supported tool-chains.

Getting started - Minimal application

The following code snippet provides a typical and minimal example using the API for a 32b floating-point model. The pre-trained model is generated with the default options (i.e. input buffer is not allocated in the "activations" buffer and default `network` c-name is used). Note that all AI requested client resources (activations buffer and data buffers for the IO) are allocated at compile time thanks the generated macros: `AI_NETWORK_XXX_SIZE` allowing a minimalist, easier and quick integration.

```c
#include <stdio.h>

#include "network.h"
#include "network_data.h"

/* Global handle to reference the instantiated C-model */
static ai_handle network = AI_HANDLE_NULL;

/* Global c-array to handle the activations buffer */
AI_ALIGNED(32)
static ai_u8 activations[AI_NETWORK_DATA_ACTIVATIONS_SIZE];

/* Array to store the data of the input tensor */
AI_ALIGNED(32)
static ai_float in_data[AI_NETWORK_IN_1_SIZE];
/* or static ai_u8 in_data[AI_NETWORK_IN_1_SIZE_BYTES]; */

/* c-array to store the data of the output tensor */
AI_ALIGNED(32)
static ai_float out_data[AI_NETWORK_OUT_1_SIZE];
/* static ai_u8 out_data[AI_NETWORK_OUT_1_SIZE_BYTES]; */

/* Array of pointer to manage the model's input/output tensors */
static ai_buffer *ai_input;
static ai_buffer *ai_output;

/*
 * Bootstrap
 */
int aiInit(void) {
  ai_error err;

  /* Create and initialize the c-model */
  const ai_handle acts[] = { activations };
  err = ai_network_create_and_init(&network, acts, NULL);
  if (err.type != AI_ERROR_NONE) { ... };

  /* Reteive pointers to the model's input/output tensors */
  ai_input = ai_network_inputs_get(network, NULL);
  ai_output = ai_network_outputs_get(network, NULL);

  return 0;
}

/*
 * Run inference
 */
int aiRun(const void *in_data, void *out_data) {
  ai_i32 n_batch;
  ai_error err;

  /* 1 - Update IO handlers with the data payload */
  ai_input[0].data = AI_HANDLE_PTR(in_data);
  ai_output[0].data = AI_HANDLE_PTR(out_data);

  /* 2 - Perform the inference */
  n_batch = ai_network_run(network, &ai_input[0], &ai_output[0]);
  if (n_batch != 1) {
      err = ai_network_get_error(network);
      ...
  };

  return 0;
}

/*
 * Example of main loop function
 */
void main_loop() {
  /* The STM32 CRC IP clock should be enabled to use the network runtime library */
  __HAL_RCC_CRC_CLK_ENABLE();

  aiInit();

  while (1) {
    /* 1 - Acquire, pre-process and fill the input buffers */
```

```
    acquire_and_process_data(in_data);

    /* 2 - Call inference engine */
    aiRun(in_data, out_data);

    /* 3 - Post-process the predictions */
    post_process(out_data);
  }
}
```

Only the following `CFLAGS/LDFLAGS` extensions (Embedded GCC-based for ARM tool-chain) are requested to compile the specialized c-files and to add the inference runtime library in a STM32 Cortex-m4 based project.

```
CFLAGS += -mcpu=cortex-m4 -mthumb -mfpu=fpv4-sp-d16  -mfloat-abi=hard

CFLAGS += -IMiddlewares/ST/AI/Lib/Inc
LDFLAGS += -LMiddlewares/ST/AI/Lib/Lib -l:NetworkRuntime510_CM4_GCC.a
```

> **Warning** — Be aware that all provided inference runtime libraries for the different STM32 series (excluding STM32WL series) are compiled with the FPU enabled and the `hard` float EABI option for performance reasons.

CRC IP usage

To use the network-runtime library, the STM32 CRC IP should be enabled (or clocked) else the application can hang. The STM32 CRC IP is used to check that the library is effectively used on a STM32 device at each call of an `ai_<name>_xx()'` function. If this IP is used in parallel, the application code must make sure to save/restore its on-going context. Similarly, if the power consumption is privileged, it can be also interesting to enable and to disable the CRC IP between two calls thanks to support of an advanced feature, refer to "STM32 CRC IP as shared resource" article to have more fine control granularity.

Note that only the `ai_<network>_create()` function returns a specific `ai_error ( .type=AI_ERROR_CREATE_FAILED , .code=AI_ERROR_CODE_LOCK )` if the CRC IP is not enabled else the other `ai_<name>_XX()` functions hang.

AI buffers and privileged placement

Application/integration point of view, only three memory-related objects are considered as dimensioning for the system. They are a fixed-size, there is no support for the dynamic tensors meaning that all the sizes and shapes of the tensors are defined/fixed at generation time. Generated c-model can be considered as a static inference model. The system heap is not requested to use the inference C run-time engine.

- "activations" buffer is a simple contiguous memory-mapped buffer, placed into a read-write memory segment. It is owned and allocated by the AI client. It is passed to the network instance (see `ai_<name>_init()` function) and used as **private heap** (or working buffer) during the execution of the inference to store the intermediate results. Between two *runs*, the associated memory segment can be used by the application. Its size, `AI_<NAME>_DATA_ACTIVATIONS_SIZE` is defined during the code generation and corresponds to the reported `RAM` metric.
- "weights" buffer is a simple contiguous memory-mapped buffer (or multiple memory-mapped buffers with the `--split-weights` option). It is generally placed into a non-volatile and read-only memory device. The total size, `AI_<NAME>_DATA_WEIGHTS_SIZE` is defined during the code generation and corresponds to the reported `ROM` metric.
- "output" and "input" buffers must be also placed in the read-write memory-mapped buffers. By default, they are owned and provided by the AI client. Their sizes are model dependent and known as generation time (`AI_<NAME>_IN/OUT_SIZE_BYTES`). They can be also located in the "activations" buffer.

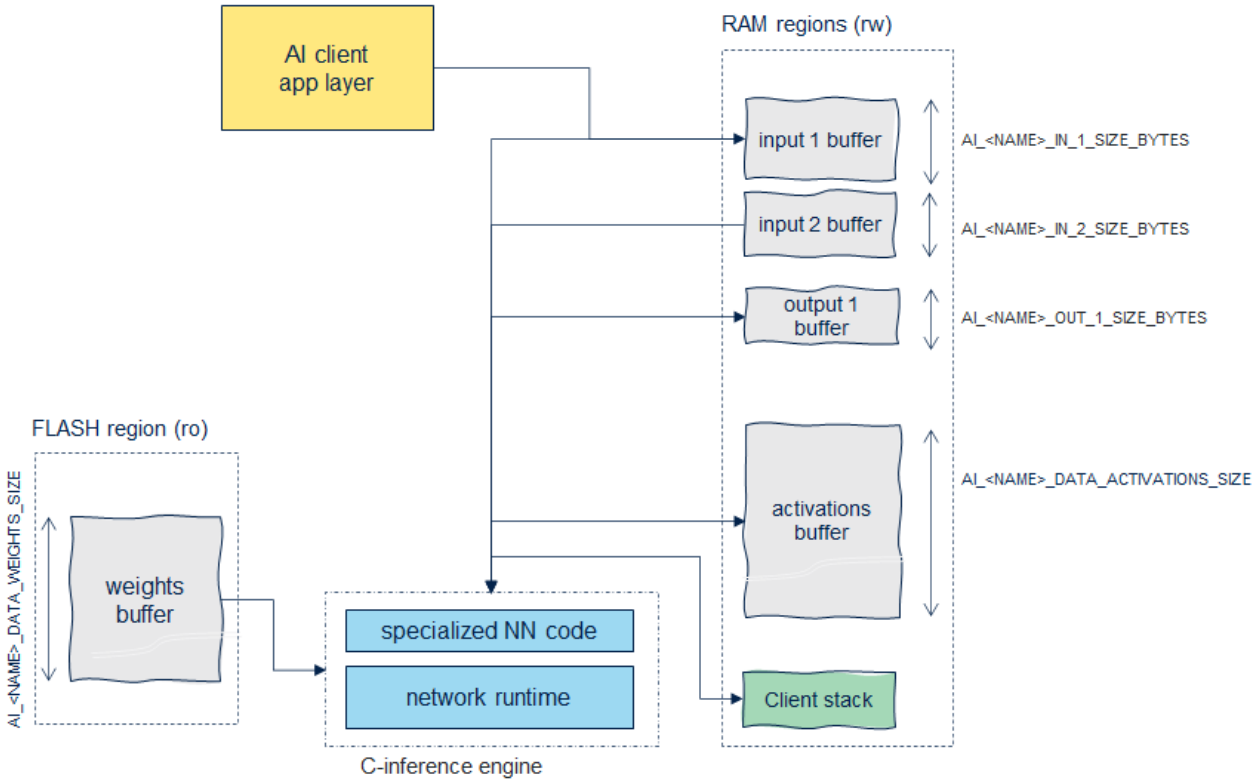

Figure 2: Default data memory layout

The kernels (inference runtime library) is executed in the context of the caller, the minimal requested **stack** size can be evaluated at run-time by the *aiSystemPerformance* application (refer to [UM], “AI system performance application” section)

> **Note** — Placement of these objects are application linker or/and runtime dependent. Additional ROM and RAM for the network runtime library itself and network c-files (txt/rodata/bss and data sections) can be also considered but they are generally not significant to dimension the system in comparison of the requested size the “weighs” and “activations” buffers. However, for the small models, as detailed in [], the `--relocatable` option allows to refine and to know the requested memory including the kernels w/o parsing and analyzing of the generated firmware map.

Following table indicates the privileged placement choices to minimize the inference time. According the model, the most constrained memory object is the “activations” buffer.

memory object type	**preferably placed in**
client stack	a low latency & high bandwidth device. STM32 embedded SRAM or data-TCM when available (zero wait-state memory).
activations, inputs/outputs	a low/medium latency & high bandwidth device. STM32 embedded SRAM first or external RAM. Trade-off is mainly driven by the size and if the STM32 MCU has a data cache (Cortex-M7 family). If input buffers are not allocated in the “activations” buffer, the “activations” buffer should be privileged.
weights	a medium latency & medium bandwidth device. STM32 embedded FLASH memory or external FLASH. Trade-off is driven by the STM32 MCU data cache availability (Cortex-M7 family), the weights can be split between different memory devices.

I/O buffers inside the “activations” buffer

The `--allocate-inputs` (respectively `--allocate-outputs`) option permits to use the “activations” buffer to allocate the data of the input tensors (respectively the output tensors). At generation time, the minimal size of the “activations” buffer is adapted accordingly. Be aware that the base addresses of the respective memory sub-regions are dependent of the model, they are not necessarily aligned with the base address of the “activations” buffer and are pre-defined/pre-calculated at generation time (see the snippet code to find them).

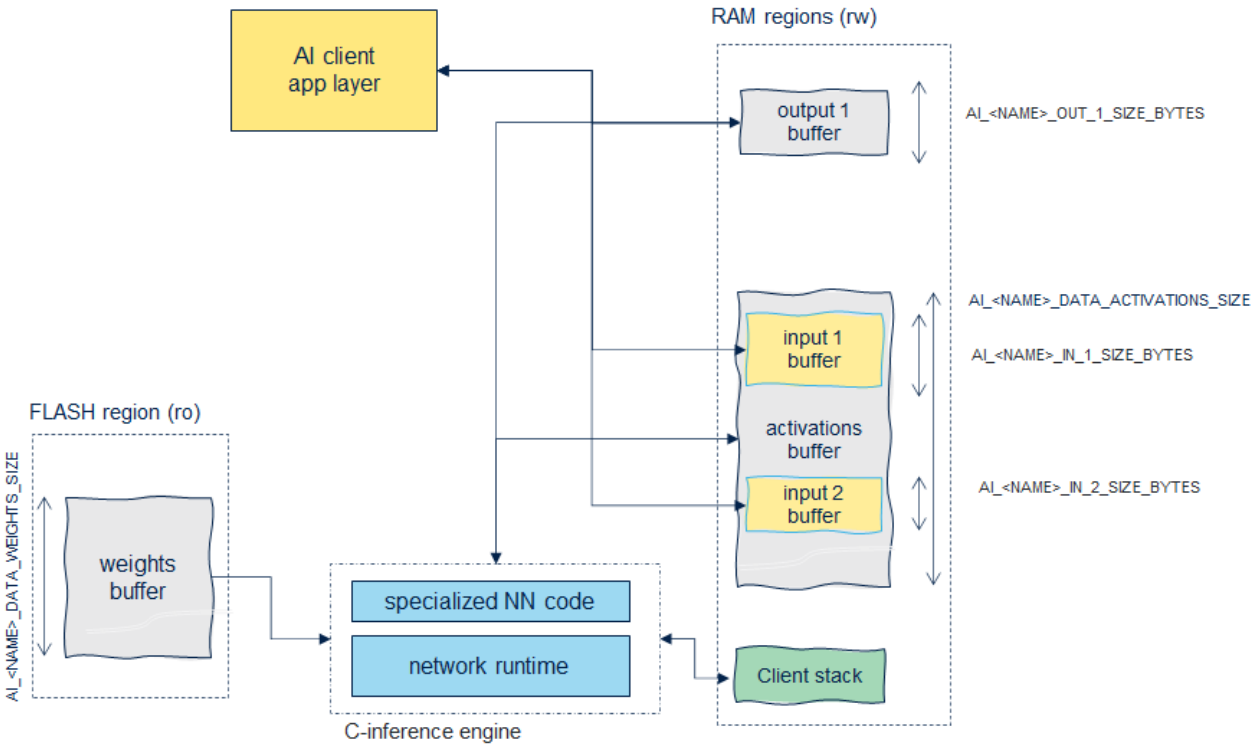

Figure 3: Data memory layout with `--allocate-inputs` option

- "external" input buffers (i.e. allocated outside the "activations" buffer) can be always used even if `--allocate-inputs` option is used.
- `--allocate-inputs` option reserves only the place for *one* buffer by input tensor. if a double buffer scheme should be implemented, `--allocate-inputs` flag should be not used.

Multiple heap support

To optimize usage of the RAM for performance reasons or because the available memory is fragmented between different memory devices (embedded in the device or externally), the activations buffer can be dispatched in different memory segments.

Thanks to the `--target` option, the user has the possibility to indicate the available memories (i.e. memory pools) which can be used to place the activation tensors (and/or scratch buffers). During the code generation, the allocator will privilege first the memory pool according to the description order in the JSON file. If a memory pool can be not used (insufficient size), the next will be used if available else an error is generated. The main assumption of the allocator is to consider the first memory pool as the privileged location to place the *critical* buffers to optimize the inference time. First memory pool should be placed in high throughput, low latency memory device.

Following figure illustrates the case, where the activations buffer is split in 3. The first is placed in the low latency/high throughput memory (like DTCM for STM32H7/F7 series), second is placed in a "normal" internal memory and the last in an external memory. The JSON file is requested to indicate the maximum size of memory segment (`"usable_size"` key) which could be used by the AI stack allowing to reserve a part of the critical memory resources for other SW objects.

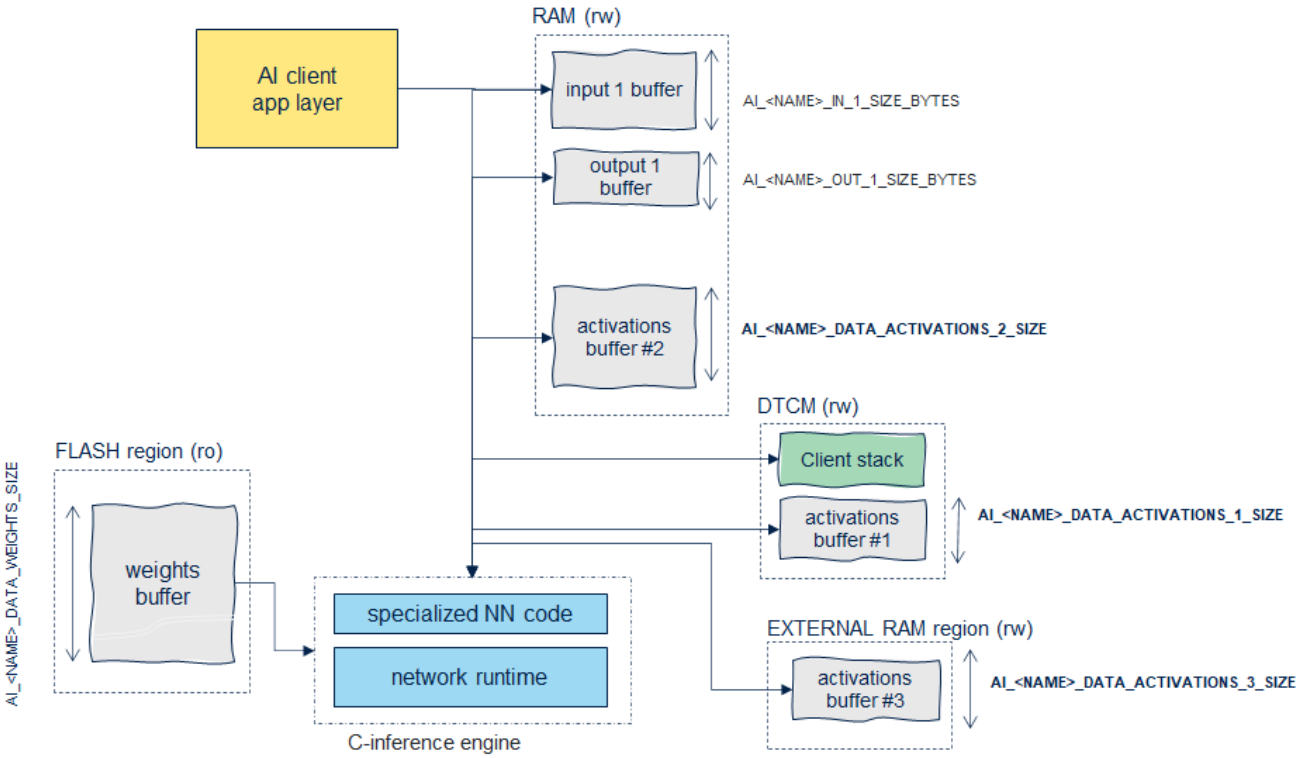

Figure 4: Data memory layout with multiple heaps

Following snippet code illustrates the initialization sequence. The `'activationsX'` objects will be placed in different memory devices thanks to specific linker directives by the end-user.

```
...
AI_ALIGNED(32)
static ai_u8 activations1[AI_NETWORK_DATA_ACTIVATIONS_1_SIZE];

AI_ALIGNED(32)
static ai_u8 activations2[AI_NETWORK_DATA_ACTIVATIONS_2_SIZE];

AI_ALIGNED(32)
static ai_u8 activations3[AI_NETWORK_DATA_ACTIVATIONS_3_SIZE];
...
int aiInit(void) {
  ai_error err;

  /* Create and initialize the c-model */
  const ai_handle acts[] = { activations1, activations2, activations3 };
  err = ai_network_create_and_init(&network, acts, NULL);
  if (err.type != AI_ERROR_NONE) { ... };
  ...
}
```

or without the helper function

```
...
int aiInit(void) {
    ai_network_params params;

    ai_network_create(&network, NULL);
    ai_network_data_params_get(&params);
    AI_BUFFER_ARRAY_ITEM_SET_ADDRESS(&params.map_activations, 0, activations1);
    AI_BUFFER_ARRAY_ITEM_SET_ADDRESS(&params.map_activations, 1, activations2);
    AI_BUFFER_ARRAY_ITEM_SET_ADDRESS(&params.map_activations, 2, activations3);
    ai_network_init(network, &params);
}
```

Split weights buffer

The `--split-weights` option is a convenience to be able to place statically tensor-by-tensor the weights in different STM32 memory segments (on or off-chip) thanks to specific linker directives for the end-user application.

- it relaxes the placing constraint of a large buffer into a constrained and non-homogenous memory sub-system.
- after profiling, it allows to improve the global inference time, by placing the critical weights into a low latency memory. Or in contrary can free the critical resource (i.e. internal flash) which can be used by the application.

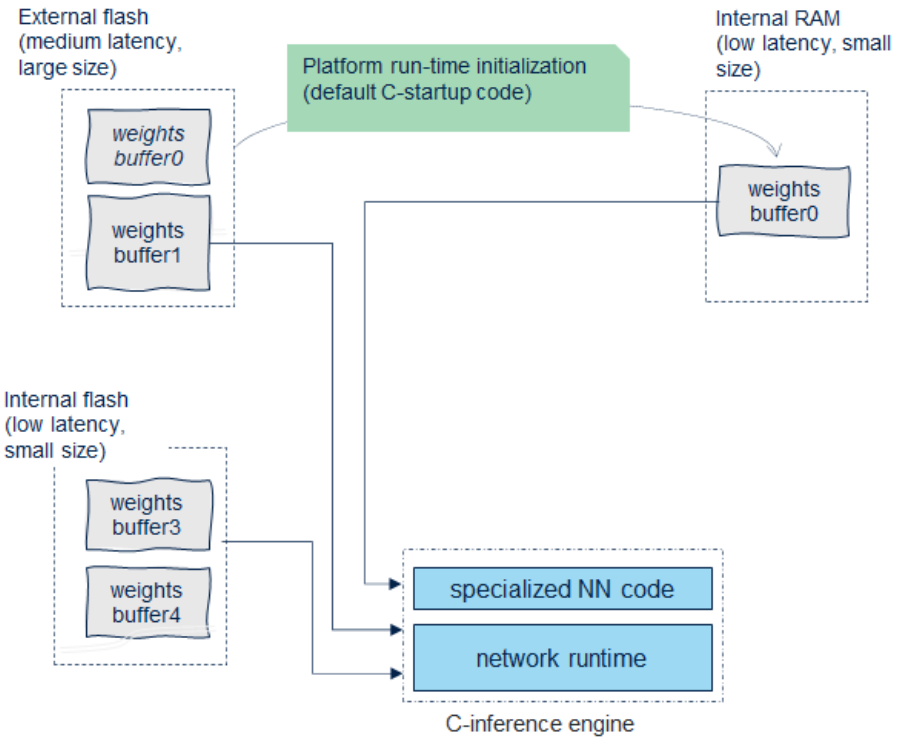

Figure 5: Split weights buffer (static placement)

The `--split-weights` option prevents the generation of a unique c-array for the whole data of the weights/bias tensors (`<name>_data.c` file) as following:

```
ai_handle ai_network_data_weights_get(void)
{
  AI_ALIGNED(4)
  static const ai_u8 s_network_weights[ 794136 ] = {
    0xcf, 0xae, 0x9d, 0x3d, 0x1b, 0x0c, 0xd1, 0xbd, 0x63, 0x99,
    0x36, 0xbd, 0xdb, 0x67, 0x46, 0xbe, 0x3b, 0xe7, 0x0d, 0x3e,
    ...
    0x41, 0xbf, 0xc6, 0x7d, 0x69, 0x3e, 0x18, 0x87, 0x37,
    0xbe, 0x83, 0x63, 0x0f, 0x3f, 0x51, 0xa1, 0xdd, 0xbe
  };
  return AI_HANDLE_PTR(s_network_weights);
}
```

A `s_<network>_<layer_name>_[bias|weights|*]_array_weights[]`) c-array is created to store the data of each weight/bias tensors. A global map table is also built. It is used by the run-time to retrieve the addresses of the different c-arrays.

```
...
/* conv2d_1_weights_array - FLOAT|CONST */
AI_ALIGNED(4)
const ai_u8 s_network_conv2d_1_weights_array_weights[ 2048 ] = {
  0xcf, 0xae, 0x9d, 0x3d, 0x1b, 0x0c, 0xd1, 0xbd, 0x63, 0x99,
...
}
...
/* dense_3_bias_array - FLOAT|CONST */
AI_ALIGNED(4)
const ai_u8 s_network_dense_3_bias_array_weights[ 24 ] = {
  0xa2, 0x72, 0x82, 0x3e, 0x5a, 0x88, 0x41, 0xbf, 0xc6, 0x7d,
  0x69, 0x3e, 0x18, 0x87, 0x37, 0xbe, 0x83, 0x63, 0x0f, 0x3f,
  0x51, 0xa1, 0xdd, 0xbe
};

/* Entry point to retrieve the address of the c-arrays */
ai_handle ai_network_data_weights_get(void) {
  static const ai_u8* const s_network_params_map_table[] = {
    &s_conv2d_1_weights_array_weights[0],
...
    &s_dense_3_bias_array_weights[0],
  };
  return AI_HANDLE_PTR(s_network_params_map_table);
};
```

- without particular linker directives, the placement of these multiple c-arrays are always placed in a `.rodata` section as for the unique c-array.
- client API is not changed. the `ai_network_data_weights_get()` function is used to pass the entry point of the weights buffer to the `ai_<name>_init()` function.
- as illustrated in the previous figure, `const` C-attribute can be manually commented to use the default C-startup behavior to copy the data in an initialized RAM data section.

Re-entrance and thread safety considerations

No internal synchronization mechanism is implemented to protect the entry points against concurrent accesses. If the API is used in a multi-threaded context, the protection of the instantiated NN(s) must be guaranteed by the application layer itself. To minimize the usage of the RAM, a same activation memory chunk (`SizeSHARED`) can be used to support multiple networks. In this case, the user must guarantee that an on-going inference execution cannot be preempted by the execution of another network.

```
SizeSHARED = MAX(AI_<name>_DATA_ACTIVATIONS_SIZE) for name = “net1” … “net2”
```

Tip — If the preemption is expected for real-time constraint or latency reasons, each network instance must have its own and private activations buffer.

Debug support

The network runtime library must be considered as an optimized black box object in binary format (sources files are not delivered). There is no run-time services allowing to dump internal states. Mapping and port of the model is guaranteed by the X-CUBE-AI generator. Some integration issues can be highlighted by the `ai_<name>_get_error()` function or by the usage of the Platform observer API to inspect the intermediate results.

Versioning

A dedicated `<network>_config.h` is generated with the C-defines which allows to know the version of the tool used to generated the specialized NN C-files and the versions of the associated run-time API.

Warning — Backward or/and forward compatibility is never considered, if a new version of the tool is used to generate the new specialized NN c-files, it is **highly recommended** to update also the associated header files and network run-time library.

```
/* <network>_config.h file */
#define AI_TOOLS_VERSION_MAJOR 7
#define AI_TOOLS_VERSION_MINOR 2
#define AI_TOOLS_VERSION_MICRO 0

#define AI_PLATFORM_API_MAJOR  1
#define AI_PLATFORM_API_MINOR  1
#define AI_PLATFORM_API_MICRO  0

#define AI_TOOLS_API_VERSION_MAJOR 1
#define AI_TOOLS_API_VERSION_MINOR 5
#define AI_TOOLS_API_VERSION_MICRO 0
```

type	description
AI_TOOLS_VERSION_XX	global version to indicate the version of the tool package
AI_PLATFORM_API_XX	indicates the version of the generated API or embedded inference client API. Can be used by the application code to check if there is a API break (source level).
AI_TOOLS_API_VERSION_XX	indicates the version of the API which is used by the generated NN c-files to call the network runtime library.

These C-defines can be used by the application code to check at compile time that the version of the used network run-time library is aligned with the generated NN c-files. At run-time, the network runtime library checks strictly only the `AI_TOOLS_API` versions. The following `ai_error` will be returned by the `ai_<network>_create()` function is the version is not respected.

```
.code = AI_ERROR_CODE_NETWORK
.type = AI_ERROR_TOOL_PLATFORM_API_MISMATCH
```

At run-time, `ai_<network>_get_report()` function allows to retrieve the different versions through the `ai_network_report` C-structure (see `ai_platform.h` file).

```
.tool_version           /* return compiled version of the tool - AI_TOOLS_VERSION_XX */
.tool_api_version       /* return compiled version of the tool API - AI_TOOLS_API_VERSION_XX */
.api_version            /* return compiled version of the embedded client API - AI_PLATFORM_API_XX */
```

Embedded inference client API

AI_<NAME>_XXX C-defines

Different C-defines are generated in the `<name>.h` and `<name>_data.h` files. They can be used by the application code to allocate at compile time or dynamically the requested buffers, or for debug purpose. At run-time, `ai_<network>_get_report()` can be used to retrieve the requested sizes.

C-defines	description
AI_<NAME>_MODEL_NAME	C-string with the C-name of the model
AI_<NAME>_ORIGIN_MODEL_NAME	C-string with the original name of the model
AI_<NAME>_IN/OUT_NUM	indicates the total number of input/output tensors
AI_<NAME>_IN/OUT_SIZE	C-table (integer type) to indicate the number of item by input/output tensors (= H x W x C) (see "IO tensor description" section)
AI_<NAME>_IN/OUT_SIZE_BYTES	C-table (integer type) to indicate the size in bytes by input/output tensors
AI_<NAME>_IN/OUT_x_SIZE	indicates the total number of item for the x-th input/output tensor
AI_<NAME>_IN/OUT_x_SIZE_BYTES	indicates the size in bytes for the x-th input/output tensor (see `ai_<name>_run()` function)
AI_<NAME>_IN/OUT_x_HEIGHT	indicates the expected "height" dimension value for the x-th input/output tensor
AI_<NAME>_IN/OUT_x_WIDTH	indicates the expected "width" dimension value for the x-th input/output tensor
AI_<NAME>_IN/OUT_x_DEPTH	indicates the expected "depth" dimension value for the x-th input/output tensor
AI_<NAME>_IN/OUT_x_CHANNEL	indicates the expected "channel" dimension value for the x-th input/output tensor
AI_<NAME>_DATA_ACTIVATIONS_SIZE	indicates the minimal size in bytes which must provided by a client application layer as activations buffer (see `ai_<name>_init()` function)
AI_<NAME>_DATA_ACTIVATIONS_COUNT	indicates the number of activations buffers
AI_<NAME>_DATA_ACTIVATIONS_x_SIZE	indicates the size in bytes of the x-th activations buffer
AI_<NAME>_DATA_WEIGHTS_SIZE	indicates the size in bytes of the generated weights/bias buffer
AI_<NAME>_INPUTS_IN_ACTIVATIONS	indicates that the input buffers can be used from the activations buffer. It is *only* defined if the `--allocate-inputs` option is used.
AI_<NAME>_OUTPUTS_IN_ACTIVATIONS	indicates that the outputs buffers can be used from the activations buffer. It is *only* defined if the `--allocate-outputs` option is used.

ai_<name>_create()

```
ai_error ai_<name>_create(ai_handle* network, const ai_buffer* network_config);
ai_handle ai_<name>_destroy(ai_handle network);
```

This **mandatory** function is the *early* function which must be called by the application to create an instance of the c-model. Provided `ai_handle` object is updated and it references a context (opaque object) which should be passed to the other functions.

- `network_config` parameter is a specific network configuration buffer (opaque structure) coded as a `ai_buffer` . It is generated by the code generator and should be *not modified* by the application. Currently, this object is always empty and `NULL` can be passed but it is preferable to pass `AI_NETWORK_DATA_CONFIG` (see `<name>_data.h` file).

When the instance is no more used by the application, `ai_<name>_destroy()` function should be called to release the possible allocated resources.

Note — The STM32 CRC IP should be enabled before to call the `ai_<network>_create()` function else the following `ai_error` is returned: `(.type =AI_ERROR_CREATE_FAILED, .code=AI_ERROR_CODE_LOCK)` .

Warning — Only one instance by a c-model is supported. Returned `ai_handle` references always the same object. Consequently a same C-model can be not used in a pre-emptive runtime environment w/o additional user synchronization mechanism.

ai_<name>_init()

```
ai_bool ai_<name>_init(ai_handle network, const ai_network_params* params);
```

This **mandatory** function is used by the application to initialize the internal run-time data structures and to set the activations buffer and weights buffer.

- `params` parameter is a structure (`ai_network_params` type) which allows to pass the references of the weights and the activations buffers (array format)
- `network` handle should be a valid handle, see `ai_<name>_create()` function

The `ai_network_data_params_get()` function must be used to retreive an initialized object (see `network_data.c` file). `'AI_BUFFER_ARRAY_ITEM_SET_ADDRESS()'` C-macro can be used to set the address of the underlying memory segment. Usage of the `ai_network_create_and_init()` helper function is privileged for the default UCs.

```
#include "network.h"
#include "network_data.h"

AI_ALIGNED(32)
static ai_u8 activations[AI_NETWORK_DATA_ACTIVATIONS_SIZE];
...
ai_network_params params;

ai_network_data_params_get(&params);
AI_BUFFER_ARRAY_ITEM_SET_ADDRESS(&params.map_activations, 0, activations);

ai_network_init(network, &params);
```

ai_<name>_create_and_init()

```
ai_error ai_<name>_create_and_init(ai_handle* network,
      const ai_handle activations[], const ai_handle weights[]);
```

This helper function aggregates the call of the create and init functions (see code in the generated `'network.c'` file). It allows to pass a simple array of pointers (`'ai_handle'`) to pass the addresses of the activations buffers and optionally the addresses of the weights buffers.

```
#include "network.h"
#include "network_data.h"

AI_ALIGNED(32)
static ai_u8 activations[AI_NETWORK_DATA_ACTIVATIONS_SIZE];
...
const ai_handle acts[] = { activations };

ai_network_create_and_init(&network, acts, NULL);
```

ai_<name>_get_report()

```
ai_bool ai_<name>_get_info(ai_handle network, ai_network_report* report);
```

This function allows to retrieve the run-time data attributes of an instantiated model. Refer to `ai_platform.h` file to show the details of the returned `ai_network_report` C-structure.

> **Warning** — If it is called before the `ai_<name>_init()` (or `ai_<name>_create_and_init()`), the reported information for the `activations` , `params` , `* inputs` and `* outputs` fields will be uncompleted. In particular the effective addresses of the buffers (the instance is not yet fully initialized), but all static information are available.

Typical usage

```
#include "network.h"
...
ai_network_report report;
ai_bool res;
...
  res = ai_network_get_report(network, &report)
  if (res) { /* display the report */ }
```

Example of possible reported informations

```
Network information
-------------------------------------------------
 model_name            : network
 model_signature       : e38d1b6095099638ae20a42e53398dd7
 model_datetime        : Mon Nov 29 19:40:02 2021
 compile_datetime      : Nov 30 2021 06:52:31
 tool_version          : 7.1.0
 tool_api_version      : 1.5.0
 api_version           : 1.2.0
 n_macc                : 336088
 map_activations       : 1
  shape=(1, 1, 1, 5712) size=5712 type=2 (bits=8 fbits=0 sign=+)
 map_weights           : 1
  shape=(1, 1, 1, 16688) size=16688 type=2 (bits=8 fbits=0 sign=+)
 n_inputs              : 1
  shape=(1, 1, 1, 1960) size=1960 type=2 (bits=8 fbits=0 sign=-)
   quantized scale=0.101716 zp=-128
 n_outputs             : 1
  shape=(1, 1, 1, 4) size=4 type=2 (bits=8 fbits=0 sign=-)
   quantized scale=0.003906 zp=-128
```

ai_<name>_data_params_get()

```
ai_bool ai_<name>_data_params_get(ai_network_params* params);
```

This function allows to retreive the network param configuration data structure. It should be used to update it and to pass it to the init function.

ai_<name>_[in|out]puts_get()

```
ai_buffer* ai_<name>_inputs_get(ai_handle* network, ai_u16 *n_buffer);
ai_buffer* ai_<name>_outputs_get(ai_handle* network, ai_u16 *n_buffer);
```

These functions return inputs/outputs array descriptors as a ai_buffer array pointer. If provided `n_buffer` allows to return the number of inputs/outputs.

ai_<name>_run()

```
ai_i32 ai_<name>_run(ai_handle network, const ai_buffer* input, ai_buffer* output);
```

This function is called to feed the neural network. The input and output buffer parameters (`ai_buffer` type) allow to provide the input tensors and to store the predicted output tensors respectively (see “IO tensor description” section).

- Returned value is the number of the input tensors processed when n_batches >= 1. If <=0 , `ai_network_get_error()` function should be used to know the error

Tip — Two separate array of inputs and outputs `ai_buffer` should be passed. This permits to support a neural network model with multiple inputs or/and outputs. `AI_NETWORK_IN_NUM` and respectively `AI_NETWORK_OUT_NUM` helper macro can be used to know at compile-time the number of inputs and outputs. These values are also returned by the `struct ai_network_report` (see `ai_<name>_get_report()` function).

Typical usages

Default UC is illustrated by the "Getting starting" code snippet. Following code is an example with a C-model which has one input and two output tensors. Note that the data payload of the input buffers are also used in the "activations" buffer.

```
#include <stdio.h>
#include "network.h"
...
/* C-table to store the @ of the input buffer */
static ai_float *in_data[AI_NETWORK_IN_NUM];

/* AI input/output handlers */
static ai_buffer *ai_inputs;
static ai_buffer *ai_outputs;

/* data buffer for the output buffers */
static ai_float out_1_data[AI_NETWORK_OUT_1_SIZE];
static ai_float out_2_data[AI_NETWORK_OUT_2_SIZE];

/* C-table to store the @ of the output buffers */
static ai_float* out_data[AI_NETWORK_OUT_NUM] = {
  &out_1_data[0],
  &out_2_data[0]
  }

...
int aiInit(void) {
  ...
  /* 1 - Create and initialize network */
  ...

  /* 2 - Retrieve IO network infos */
  ai_input = ai_network_inputs_get(network);
  ai_ouput = ai_network_outputs_get(network);

  /* 3 - Retreive the effective @ of the input buffers  */
  for (int i=0; i < AI_NETWORK_IN_NUM; i++) {
    in_data[i] = (ai_u8 *)(ai_inputs[i].data);
  }

  /* 4- Update the AI output handlers */
  for (int i=0; i < AI_NETWORK_OUT_NUM; i++) {
    ai_outputs[i].data = AI_HANDLE_PTR(&out_data[i]);
  }
  ...
}

void main_loop()
{
  while (1) {
    /* 1 - Acquire, pre-process and fill the input buffers */
    acquire_and_process_data(in_data);

    /* 2 - Call inference engine */
    ai_network_run(network, &ai_inputs[0], &ai_outputs[0]);

    /* 3 - Post-process the predictions */
    post_process(out_data);
  }
}
```

ai_<name>_get_error()

```
ai_error ai_<name>_get_error(ai_handle network);
```

This function can be used by the client application to retrieve the 1st error reported during the execution of a `ai_<name>_xxx()` function.

- See `ai_platform.h` file to have the list of the returned error type (`ai_error_type`) and associated code (`ai_error_code`).

Typical AI error function handler (debug/log purpose)

```
#include "network.h"
...
void aiLogErr(const ai_error err)
{
  printf("E: AI error - type=%d code=%d\r\n", err.type, err.code);
}
```

IO tensor description

N-dimensional tensors are supported with a fixed format: *BHWC* format or *channel last* format representation. They are defined by a `'struct ai_buffer'` C-structure object. The referenced memory buffer (`'data'` field) is physically stored and referenced in memory as a standard C-array type. Scattered memory buffers are not supported.

- `ai_buffer_shape` - indicates the dimensions of the tensor
- `format` - indicates the format of the data
- `meta_info` - extra field to reference the additional data-dependent parameters which can be requested to handle a buffer

> **Note** — If the dimension order in the original toolbox is different that HWC (e.g. ONNX: CHW) it's up to the application to properly re-arrange the element order.

ai_buffer C-structure

```
/* @file: ai_platform.h */

typedef struct ai_buffer_ {
  ai_buffer_format        format;     /*!< buffer format */
  ai_handle               data;       /*!< pointer to buffer data */
  ai_buffer_meta_info*    meta_info;  /*!< pointer to buffer metadata info */
  ai_flags                flags;      /*!< shape optional flags */
  ai_size                 size;       /*!< number of elements of the buffer
                                           (including optional padding) */
  ai_buffer_shape         shape;      /*!< n-dimensional shape info */
} ai_buffer;
```

Note that the C-tensors are **always** defined with minimum 4 dimensions, following table shows the generated mapping of the 1d, 2d and 3d-array tensor:

tensor shape	mapped on (B, H, W, C)
1d-array	(-, 1, 1, c)
2d-array	(-, h, 1, c)
3d-array	(-, h, w, c)

For a 5d or 6d tensor (respectively 4d-array and 5d-array), following representation is respected:

tensor shape	mapped on
4d-array	(-, h, w, d, c)
5d-array	(-, h, w, d, e, c)

Retrieve tensor information

Following code snippets show how to retrieve the tensor information from a buffer descriptor. `'format'` and `'meta_info'` fields are described in the next section.

```
#include "network.h"

{
  /* Use the generated macro to set the buffer input descriptors */
  const ai_buffer *ai_input = ai_network_inputs_get(network, NULL);

  /* Extract format of the first input tensor (index 0) */
  const ai_buffer *ai_input_1 = &ai_input[0];

  /* Extract format of the tensor */
  const ai_buffer_format fmt_1 = AI_BUFFER_FORMAT(ai_input_1);

  /* Extract height, width and channels of the first input tensor */
  const ai_u16 height_1 = AI_BUFFER_SHAPE_ELEM(ai_input_1, AI_SHAPE_HEIGHT);
  const ai_u16 width_1 = AI_BUFFER_SHAPE_ELEM(ai_input_1, AI_SHAPE_WIDTH);
  const ai_u16 ch_1 = AI_BUFFER_SHAPE_ELEM(ai_input_1, AI_SHAPE_CHANNEL);
  const ai_u16 size_1 = AI_BUFFER_SIZE(ai_input_1);  /* number of item*/
  const ai_u32 size_in_bytes_1 = AI_BUFFER_BYTE_SIZE(size_1, fmt_1);
  ...
}
```

or with the `ai_network_info` structure

```
#include "network.h"

{
  /* Fetch run-time network descriptor */
  ai_network_report report;
  ai_network_get_report(network, &report);

  /* Set the descriptor of the first input tensor (index 0) */
  const ai_buffer *ai_input_1 = &report.inputs[0]

  /* Extract format of the tensor */
  const ai_buffer_format fmt_1 = AI_BUFFER_FORMAT(ai_input_1);
  ...
}
```

Tensor dimension

Follwing macros should be used to retreive the associated dimensions:

macros	description
`AI_BUFFER_SHAPE_ELEM(buff, AI_SHAPE_WIDTH)`	return the `width` dimension
`AI_BUFFER_SHAPE_ELEM(buff, AI_SHAPE_HEIGHT)`	return the `height` dimension
`AI_BUFFER_SHAPE_ELEM(buff, AI_SHAPE_CHANNEL)`	return the `channel` dimension
`AI_BUFFER_SHAPE_ELEM(buff, AI_SHAPE_BATCH)`	return the `batch` dimension
`AI_BUFFER_SHAPE_ELEM(buff, AI_SHAPE_DEPTH)`	return the `depth` dimension if available else 0
`AI_BUFFER_SHAPE_ELEM(buff, AI_SHAPE_EXTENSION)`	return the `extension` dimension if available else 0

Tensor format

The format of the data is mainly defined by the field `format` , a 32b word (`ai_buffer_format` type). 3 main types are supported.

```
const ai_buffer_format fmt = AI_BUFFER_FORMAT(@ai_buffer_object);
```

type	description
`AI_BUFFER_FMT_TYPE_FLOAT`	indicates that the data container handles the **floating-point data**. mapped on a 32b float C-type (`ai_float` or `float`).
`AI_BUFFER_FMT_TYPE_Q`	indicates that the data container handles the **quantized data**, mapped on 8b signed or unsigned integer C-type. See to [QUANT], "Quantized tensors" section, to detail the used integer arithmetic.
`AI_BUFFER_FMT_TYPE_BOOL`	indicates that the data container handles the **boolean**, mapped on 8b unsigned integer C-type

Helper C-macros

Following set of C-macros can be used with the `format` field from the `struct ai_buffer` C-structure object to extract the information.

macros	description
`AI_BUFFER_FMT_GET_TYPE(fmt)`	returns `AI_BUFFER_FMT_TYPE_FLOAT` / `AI_BUFFER_FMT_TYPE_Q` or `AI_BUFFER_FMT_TYPE_BOOL` buffer type
`AI_BUFFER_FMT_GET_FLOAT(fmt)`	returns `1` if the data is a float type else `0`
`AI_BUFFER_FMT_GET_SIGN(fmt)`	returns `1` if the data is signed else `0` .
`AI_BUFFER_FMT_GET_BITS(fmt)`	returns the total number of bit which is used to encode the data. This is M+N+sign for `AI_BUFFER_FMT_TYPE_Q` type. Available values: `32` or `8`
`AI_BUFFER_FMT_GET_FBITS(fmt)`	returns the total number of bit which is used to encode the fractional part for the 8b quantized data type.

For supported binary format (1b signed), `AI_BUFFER_FMT_TYPE_Q` is returned and the total number of bit is 1 (`AI_BUFFER_FMT_GET_BITS(fmt)`).

Additional macros are defined for the meta parameters:

```
const ai_buffer_meta_info * meta_info = AI_BUFFER_META_INFO(@ai_buffer_object);
```

macros	description
`AI_BUFFER_META_INFO_INTQ(meta_info)`	indicates if scale/zero-point meta info are available. If true, a reference of a `ai_intq_info` object is returned else `NULL` .
`AI_BUFFER_META_INFO_INTQ_GET_SCALE(meta_info, pos)`	generic macro to returns the scale value at the pos-th position is available else `0` is returned. `ai_float` type. For the IO tensor only the position 0 is available.
`AI_BUFFER_META_INFO_INTQ_GET_ZEROPOINT(meta_info, pos)`	generic macro to returns the zero-point value at the pos-th position is available else `0` is returned. `ai_i8` or `ai_u8` type. Type can be deduced from the output of the `AI_BUFFER_FMT_GET_SIGN()` and `AI_BUFFER_FMT_GET_BITS()` macros.

> **Warning** — Be aware that the `meta_info` field is only available through the returned `ai_network_report()` structure. Otherwise the value defined by the generated `AI_<NAME>_IN/OUT` C-define are `NULL` .

Following code snippet illustrates a typical code to extract the `scale` and `zero_point` values:

```c
#include "network.h"

static ai_handle network;

{
  /* Get the descriptor of the first input tensor (index 0) */
  const ai_buffer *ai_input = ai_network_inputs_get(network, NULL);
  const ai_buffer *ai_input_1 = &ai_input[0];

  /* Extract format of the tensor */
  const ai_buffer_format fmt_1 = AI_BUFFER_FORMAT(ai_input_1);

  /* Extract the data type */
  const uint32_t type = AI_BUFFER_FMT_GET_TYPE(fmt_1); /* -> AI_BUFFER_FMT_TYPE_Q */

  /* Extract sign and number of bits */
  const ai_size sign = AI_BUFFER_FMT_GET_SIGN(fmt_1);  /* -> 1 or 0*/
  const ai_size bits = AI_BUFFER_FMT_GET_BITS(fmt_1);  /* -> 8 */

  /* Extract scale/zero_point values (only pos=0 is currently supported, per-tensor) */
  const ai_float scale = AI_BUFFER_META_INFO_INTQ_GET_SCALE(ai_input_1->meta_info, 0);
  const int zero_point = AI_BUFFER_META_INFO_INTQ_GET_ZEROPOINT(ai_input_1->meta_info, 0);
  ...
}
```

Floating-point case.

```c
#include "network.h"

{
  /* Get the descriptor of the first input tensor (index 0) */
  const ai_buffer *ai_input = ai_network_inputs_get(network, NULL);
  const ai_buffer *ai_input_1 = &ai_input[0];

  /* Retrieve format of the first input tensor (index 0) */
  const ai_buffer_format fmt_1 = AI_BUFFER_FORMAT(ai_input_1);

  /* Retrieve the data type */
  const uint32_t type = AI_BUFFER_FMT_GET_TYPE(fmt_1); /* -> AI_BUFFER_FMT_TYPE_FLOAT */

  /* Retrieve sign/size values */
  const ai_size sign = AI_BUFFER_FMT_GET_SIGN(fmt_1);   /* -> 1 */
  const ai_size bits = AI_BUFFER_FMT_GET_BITS(fmt_1);   /* -> 32 */
  const ai_size N = AI_BUFFER_FMT_GET_FBITS(fmt_1);     /* -> 0 */
  ...
}
```

Life-cycle of the IO tensors

When the input buffers and output buffers are passed to the `ai_<name>_run()` function, the caller should wait the end of the inference to re-use the associated memory segments. There is no default mechanism to notify the application that the input tensors are released or no more used by the c-inference engine. This is particular true when the buffers are allocated in the activations buffer. However, in the case where an input buffer is allocated in the user space the Platform Observer API can be used to be notified when the operator has finished (see “End-of-process input buffer notification” section).

Base address of the IO buffers

Following code snippet illustrates the minimum requested instructions to retrieve the effective address of the buffer from the *activations* buffer. If the `--allocate-inputs` (or `--allocate-outputs`) option is not used, `NULL` is returned. Note that the instance should be previously fully initialized, because the returned address is dependent to the base address of the *activations* buffer.

```
#include "network.h"

static ai_handle network;

#if defined(AI_NETWORK_INPUTS_IN_ACTIVATIONS)
static ai_u8 *in_data_1;
#else
  /* Buffer should be allocated by the application
     in this case: ai_input_1->data == NULL */
static ai_u8 in_data_1[AI_NETWORK_IN_1_SIZE_BYTES];
#endif

{
  const ai_buffer *ai_input = ai_network_inputs_get(network, NULL);
  const ai_buffer *ai_input_1 = &ai_input[0];

#if defined(AI_NETWORK_INPUTS_IN_ACTIVATIONS)
  /* Set the descriptor of the first input tensor (index 0) */
  /* Retrieve the @ of the input buffer */
  in_data_1 = (ai_u8 *)ai_input_1->data;
#endif
...
}
```

float32 to 8b data type conversion

Following code snippet illustrates the float (ai_float) to integer (ai_i8/ai_u8) format conversion. Input buffer is used as destination buffer.

```
#include <network.h>

#define _MIN(x_, y_) \
    ( ((x_)<(y_)) ? (x_) : (y_) )

#define _MAX(x_, y_) \
    ( ((x_)>(y_)) ? (x_) : (y_) )

#define _CLAMP(x_, min_, max_, type_) \
    (type_) (_MIN(_MAX(x_, min_), max_))

#define _ROUND(v_, type_) \
    (type_) ( ((v_)<0) ? ((v_)-0.5f) : ((v_)+0.5f) )

const ai_buffer *get_input_desc(idx)
{
  const ai_buffer *ai_input = ai_network_inputs_get(network, NULL);
  return &ai_input[idx];
}

ai_float input_f[AI_<NAME>_IN_1_SIZE];
ai_i8 input_q[AI_<NAME>_IN_1_SIZE]; /* or ai_u8 */

{
  const ai_buffer *input = get_input_desc(0);
  ai_float scale  = AI_BUFFER_META_INFO_INTQ_GET_SCALE(input->meta_info, 0);
  const ai_i32 zp = AI_BUFFER_META_INFO_INTQ_GET_ZEROPOINT(input->meta_info, 0);

  scale = 1.0f / scale;

  /* Loop */
  for (int i=0; i < AI_<NAME>_IN_1_SIZE; i++)
  {
    const ai_i32 tmp_ = zp + _ROUND(input_f[i] * scale, ai_i32);
    /* for ai_u8 */
    input_q[i] = _CLAMP(tmp_, 0, 255, ai_u8);
    /* for ai_i8 */
    input_q[i] = _CLAMP(tmp_, -128, 127, ai_i8);
  }
  ...
}
```

8b to float32 data type conversion

Following code snippet illustrates the integer (ai_i8/ai_u8) to float (ai_float) format conversion. The output buffer is used as source buffer.

```
#include <network.h>

ai_i8 output_q[AI_<NAME>_OUT_1_SIZE]; /* or ai_u8 */
ai_float output_f[AI_<NAME>_OUT_1_SIZE];

const ai_buffer *get_output_desc(idx)
{
  const ai_buffer *ai_input = ai_network_outputs_get(network, NULL);
  return &ai_input[idx];
}

{
  const ai_buffer *output = get_output_desc(0);
  ai_float scale  = AI_BUFFER_META_INFO_INTQ_GET_SCALE(output->meta_info, 0);
  const ai_i32 zp = AI_BUFFER_META_INFO_INTQ_GET_ZEROPOINT(output->meta_info, 0);

  /* Loop */
  for (int i=0; i<AI_<NAME>_OUT_1_SIZE; i++)
  {
    output_f[i] = scale * ((ai_float)(output_q[i]) - zp);
  }
  ...
}
```

C-memory layouts

To store the elements of the differnet tensor, standard C-array-of-array struct are used. For the `float` and `integer` data-type, the associated c-type is used. For the `bool` type, value is stored on the `uint8_t` and for binary tensor, to optimize the size, values are packed on the 32b words (see “c-layout of the s1 type” section for more details).

1d-array tensor

For a 1-D tensor, standard C-array type with the following memory layout is expected to handle the input and output tensors.

```
#include "network.h"

#define xx_SIZE  VAL  /* = H * W * C = C */

ai_float xx_data[xx_SIZE];     /* n_batch = 1, height = 1,
                                  width = 1, channels = C */
ai_float xx_data[B * xx_SIZE]; /* n_batch = B, height = 1,
                                  width = 1, channels = C */
ai_float xx_data[B][xx_SIZE];
```

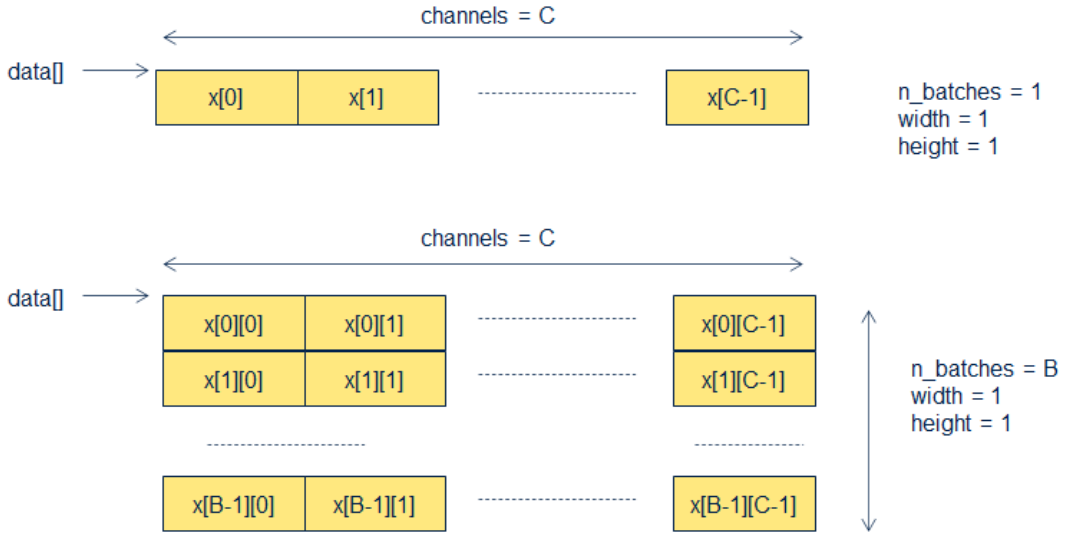

Figure 6: 1-D Tensor data layout

2d-array tensor

For a 2-D tensor, standard C-array-of-array memory arrangement is used to handle the input and output tensors. 2-dim are mapped to the first two dimensions of the tensor in the original toolbox representation: e.g. H and C in Keras / Tensorflow.

```
#include "network.h"

#define xx_SIZE  VAL  /* = H * W * C = H * C */

ai_float xx_data[xx_SIZE];  /* n_batch = 1, height = H,
                               width = 1, channels = C */
ai_float xx_data[H][C];
ai_float xx_data[B * xx_SIZE]; /* n_batch = B, height = H,
                                  width = 1, channels = C */
ai_float xx_data[B][H][C];
```

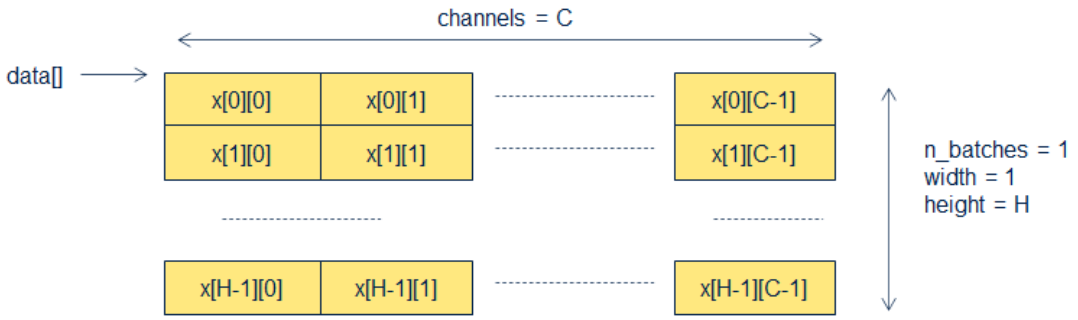

Figure 7: 2-D Tensor data layout

3d-array tensor

For a 3D-tensor, standard C-array-of-array-of-array memory arrangement is used to handle the input and output tensors.

```
#include "network.h"

#define xx_SIZE  VAL  /* = H * W * C */

ai_float xx_data[xx_SIZE];  /* n_batch = 1, height = H,
                               width = W, channels = C */
ai_float xx_data[H][W][C];
ai_float xx_data[B * xx_SIZE]; /* n_batch = B, height = H,
                                  width = W, channels = C */
ai_float xx_data[B][H][W][C];
```

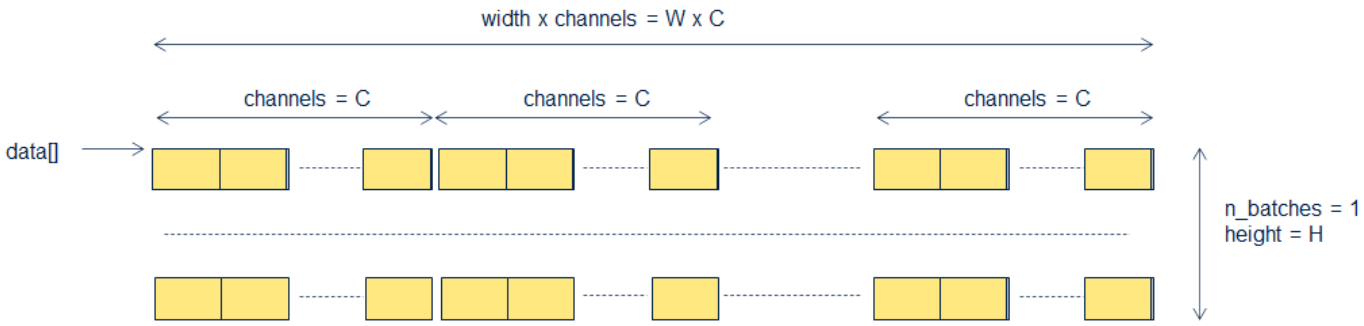

Figure 8: 3-D Tensor data layout

References

ref	description
[DS]	X-CUBE-AI - AI expansion pack for STM32CubeMX https://www.st.com/en/embedded-software/x-cube-ai.html
[UM]	User manual - Getting started with X-CUBE-AI Expansion Package for Artificial Intelligence (AI) (pdf)
[CLI]	stm32ai - Command Line Interface (link)
[API]	Embedded inference client API (link)
[METRIC]	Evaluation report and metrics (link)
[TFL]	TensorFlow Lite toolbox (link)
[KERAS]	Keras toolbox (link)
[ONNX]	ONNX toolbox (link)
[ONNX-ML]	Machine Learning support (ONNX-ML operators) (link)
[DQNN]	Deep Quantized Neural Network support (link)
[FAQS]	FAQ generic, validation, quantization
[QUANT]	Quantization (link)
[QUANT_CMD]	Quantization Command (link)
[RELOC]	Relocatable binary network support (link)
[CUST]	Support of the Keras Lambda/custom layers (link)
[TFLM]	TensorFlow Lite for Microcontroller support (link)
[INST]	Setting the environment (link)
[OBS]	Platform Observer API (link)
[C-RUN]	Executing locally a generated c-model (link)
[BREAK]	API Breaking changes (link)
[CRC]	STM32 CRC IP as shared resource (link)

Embedded Documentation - **Embedded Inference Client API**

X-CUBE-AI Expansion Package

r5.1 - AI PLATFORM r8.0.0 (Embedded Inference Client API 1.2.0) - Command Line Interface r1.7.0