



## **CNStream-Developer-Guide-EN**

*Release 6.1.0*

October 19, 2021



# Table of Contents

<b>Table of Contents</b>	<b>i</b>
<b>1 Copyright</b>	<b>1</b>
<b>2 Datatypes Reference</b>	<b>3</b>
2.1 enum EventType . . . . .	3
2.2 enum EventHandleFlag . . . . .	4
2.3 enum CNFrameFlag . . . . .	4
2.4 enum CNDataFormat . . . . .	5
2.5 enum StreamMsgType . . . . .	6
2.6 struct CNConfigBase . . . . .	8
2.7 struct ProfilerConfig . . . . .	9
2.8 struct CNModuleConfig . . . . .	10
2.9 struct CNSubgraphConfig . . . . .	11
2.10 struct CNGraphConfig . . . . .	12
2.11 struct Event . . . . .	14
2.12 struct DevContext . . . . .	15
2.13 struct CNInferBoundingBox . . . . .	15
2.14 struct CNInferAttr . . . . .	16
2.15 struct CNInferObjs . . . . .	17
2.16 struct InferData . . . . .	17
2.17 struct CNInferData . . . . .	18
2.18 struct StreamMsg . . . . .	19
2.19 struct ESPacket . . . . .	20
2.20 struct MaximumVideoResolution . . . . .	20
2.21 struct StreamProfile . . . . .	21
2.22 struct ProcessProfile . . . . .	23
2.23 struct ModuleProfile . . . . .	25
2.24 struct PipelineProfile . . . . .	26
2.25 struct TraceElem . . . . .	28
2.26 struct PipelineTrace . . . . .	30
2.27 typedef ModuleParamSet . . . . .	31

2.28	typedef BusWatcher . . . . .	31
2.29	typedef CNFrameInfoPtr . . . . .	32
2.30	typedef CNInferFeature . . . . .	32
2.31	typedef CNInferFeatures . . . . .	32
2.32	typedef StringPairs . . . . .	32
2.33	typedef CNInferObjectPtr . . . . .	32
2.34	typedef CNDataFramePtr . . . . .	33
2.35	typedef CNInferObjsPtr . . . . .	33
2.36	typedef CNObjsVec . . . . .	33
2.37	typedef CNInferDataPtr . . . . .	33
2.38	typedef Clock . . . . .	33
2.39	typedef Duration . . . . .	34
2.40	typedef Time . . . . .	34
2.41	typedef RecordKey . . . . .	34
2.42	typedef ProcessTrace . . . . .	34
2.43	typedef ModuleTrace . . . . .	34
<b>3</b>	<b>Class . . . . .</b>	<b>35</b>
3.1	Framework . . . . .	35
3.1.1	Class Collection . . . . .	35
3.1.2	Class ParametersChecker . . . . .	38
3.1.3	Class ParamRegister . . . . .	39
3.1.4	Class EventBus . . . . .	40
3.1.5	Class CNFrameInfo . . . . .	42
3.1.6	Class IModuleObserver . . . . .	45
3.1.7	Class Module . . . . .	45
3.1.8	Class ModuleEx . . . . .	49
3.1.9	Class Pipeline . . . . .	50
3.1.10	Class StreamMsgObserver . . . . .	55
3.1.11	Class SourceHandler . . . . .	56
3.1.12	Class SourceModule . . . . .	58
3.1.13	Class CNSyncedMemory . . . . .	60
3.2	Profiler . . . . .	65
3.2.1	Class ModuleProfiler . . . . .	65
3.2.2	Class PipelineProfiler . . . . .	67
3.2.3	Class PipelineTracer . . . . .	71
3.2.4	Class ProcessProfiler . . . . .	73
3.2.5	Class StreamProfiler . . . . .	75
3.2.6	Class TraceSerializeHelper . . . . .	77

3.2.7	Class TraceEvent . . . . .	81
3.3	Video Analysis . . . . .	85
3.3.1	Class CNDataFrame . . . . .	85
3.3.2	Class CNInferObject . . . . .	88
3.3.3	Class DataSource . . . . .	92
3.3.4	Class ESJpegMemHandler . . . . .	94
3.3.5	Class ESMemHandler . . . . .	95
3.3.6	Class FileHandler . . . . .	98
3.3.7	Class RawImgMemHandler . . . . .	100
3.3.8	Class RtspHandler . . . . .	102
3.3.9	Class ObjPostproc . . . . .	104
3.3.10	Class Postproc . . . . .	106
3.3.11	Class ObjPreproc . . . . .	107
3.3.12	Class Preproc . . . . .	108
3.3.13	Class VideoPostproc . . . . .	109
3.3.14	Class VideoPreproc . . . . .	111
<b>4</b>	<b>API Reference</b>	<b>113</b>
4.1	Framework Function . . . . .	113
4.1.1	GetMaxModuleNumber . . . . .	113
4.1.2	GetMaxStreamNumber . . . . .	113
4.1.3	GetPathRelativeToTheJSONFile . . . . .	113
4.1.4	VersionString . . . . .	114
4.1.5	MajorVersion . . . . .	114
4.1.6	MinorVersion . . . . .	114
4.1.7	PatchVersion . . . . .	114
4.2	Video Analysis Function . . . . .	114
4.2.1	CNGetPlanes . . . . .	114
4.2.2	GetCNDataFramePtr . . . . .	115
4.2.3	GetCNInferObjsPtr . . . . .	115
4.2.4	GetCNInferDataPtr . . . . .	115
<b>5</b>	<b>Release Notes</b>	<b>116</b>
5.1	CNStream Release Version 6.1.0 . . . . .	116
5.1.1	API Updates . . . . .	116
5.2	CNStream Release Version 6.0.0 . . . . .	116
5.2.1	API Updates . . . . .	116
5.3	CNStream Release 2021-01-25 (Version 5.3.0) . . . . .	117
5.3.1	API Updates . . . . .	117
5.4	CNStream Release 2020-09-18 (Version 5.2.0) . . . . .	119

5.4.1	API Updates	119
5.5	CNStream Release 2020-07-10 (Version 5.0.0)	120
5.5.1	API Updates	120
5.6	CNStream Release 2020-05-25 (Version 4.5.0)	123
5.6.1	API Updates	123
5.7	Release 2020-04-16 (Version 4.4.0)	124
5.7.1	API Updates	124
5.7.2	Doc Updates	124
5.8	Release 2020-02-24	125
5.8.1	API Updates	125
5.9	Release 2019-12-31	125
5.9.1	API Updates	125



# 1 Copyright

## **DISCLAIMER**

CAMBRICON MAKES NO REPRESENTATION, WARRANTY (EXPRESS, IMPLIED, OR STATUTORY) OR GUARANTEE REGARDING THE INFORMATION CONTAINED HEREIN, AND EXPRESSLY DISCLAIMS ANY AND ALL IMPLIED WARRANTIES OF MERCHANTABILITY, TITLE, NONINFRINGEMENT OF INTELLECTUAL PROPERTY OR FITNESS FOR A PARTICULAR PURPOSE, AND CAMBRICON DOES NOT ASSUME ANY LIABILITY ARISING OUT OF THE APPLICATION OR USE OF ANY PRODUCT OR SERVICES. CAMBRICON SHALL HAVE NO LIABILITY RELATED TO ANY DEFAULTS, DAMAGES, COSTS OR PROBLEMS WHICH MAY BE BASED ON OR ATTRIBUTABLE TO: (I) THE USE OF THE CAMBRICON PRODUCT IN ANY MANNER THAT IS CONTRARY TO THIS GUIDE, OR (II) CUSTOMER PRODUCT DESIGNS.

## **LIMITATION OF LIABILITY**

In no event shall Cambricon be liable for any damages whatsoever (including, without limitation, damages for loss of profits, business interruption and loss of information) arising out of the use of or inability to use this guide, even if Cambricon has been advised of the possibility of such damages. Notwithstanding any damages that customer might incur for any reason whatsoever, Cambricon's aggregate and cumulative liability towards customer for the product described in this guide shall be limited in accordance with the Cambricon terms and conditions of sale for the product.

## **ACCURACY OF INFORMATION**

Information provided in this document is proprietary to Cambricon, and Cambricon reserves the right to make any changes to the information in this document or to any products and services at any time without notice. The information contained in this guide and all other information contained in Cambricon documentation referenced in this guide is provided "AS IS." Cambricon does not warrant the accuracy or completeness of the information, text, graphics, links or other items contained within this guide. Cambricon may make changes to this guide, or to the products described therein, at any time without notice, but makes no commitment to update this guide.

Performance tests and ratings set forth in this guide are measured using specific chips or computer systems or components. The results shown in this guide reflect approximate performance of Cambricon products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. As set forth above, Cambricon makes no

representation, warranty or guarantee that the product described in this guide will be suitable for any specified use. Cambricon does not represent or warrant that it tests all parameters of each product. It is customer's sole responsibility to ensure that the product is suitable and fit for the application planned by the customer and to do the necessary testing for the application in order to avoid a default of the application or the product.

Weaknesses in customer's product designs may affect the quality and reliability of Cambricon product and may result in additional or different conditions and/or requirements beyond those contained in this guide.

### **IP NOTICES**

Cambricon and the Cambricon logo are trademarks and/or registered trademarks of Cambricon Corporation in China and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

This guide is copyrighted and is protected by worldwide copyright laws and treaty provisions. This guide may not be copied, reproduced, modified, published, uploaded, posted, transmitted, or distributed in any way, without Cambricon's prior written permission. Other than the right for customer to use the information in this guide with the product, no other right or license, either express or implied, is granted by Cambricon under this guide. For the avoidance of doubt, Cambricon does not grant any right or license (express or implied) to customer under any patents, copyrights, trademarks, trade secret or any other intellectual property or proprietary rights of Cambricon.

- Copyright
- © 2021 Cambricon Corporation. All rights reserved.



## 2 Datatypes Reference

### 2.1 enum EventType

```
enum EventType {  
    EVENT_INVALID = 0,  
    EVENT_ERROR = 1,  
    EVENT_WARNING = 2,  
    EVENT_EOS = 3,  
    EVENT_STOP = 4,  
    EVENT_STREAM_ERROR = 5,  
    EVENT_TYPE_END = 6,  
};
```

```
enum cnstream::EventType
```

Enumeration variables describing the type of event.

Values:

```
enumerator EVENT_INVALID
```

An invalid event type.

```
enumerator EVENT_ERROR
```

An error event.

```
enumerator EVENT_WARNING
```

A warning event.

```
enumerator EVENT_EOS
```

An EOS event.

```
enumerator EVENT_STOP
```

A stop event.



enumerator EVENT\_STREAM\_ERROR

A stream error event.

enumerator EVENT\_TYPE\_END

Reserved for users custom events.

## 2.2 enum EventHandleFlag

enum EventHandleFlag {

EVENT\_HANDLE\_NULL = 0,

EVENT\_HANDLE\_INTERCEPTION = 1,

EVENT\_HANDLE\_SYNCED = 2,

EVENT\_HANDLE\_STOP = 3,

};

enum cnstream::EventHandleFlag

Enumeration variables describing the way how bus watchers handle an event.

Values:

enumerator EVENT\_HANDLE\_NULL

The event is not handled.

enumerator EVENT\_HANDLE\_INTERCEPTION

The event has been handled and other bus watchers needn't to handle it.

enumerator EVENT\_HANDLE\_SYNCED

The event has been handled and other bus watchers are going to handle it.

enumerator EVENT\_HANDLE\_STOP

The event has been handled and bus watchers stop all other events' processing.

## 2.3 enum CNFrameFlag

enum CNFrameFlag {

CN\_FRAME\_FLAG\_EOS = 1 << 0,

CN\_FRAME\_FLAG\_INVALID = 1 << 1,

CN\_FRAME\_FLAG\_REMOVED = 1 << 2,

};

```
enum cnstream::CNFrameFlag
```

Enumeration variables describing the mask of [CNDataFrame](#).

Values:

```
enumerator CN_FRAME_FLAG_EOS
```

This enumeration indicates the end of data stream.

```
enumerator CN_FRAME_FLAG_INVALID
```

This enumeration indicates an invalid frame.

```
enumerator CN_FRAME_FLAG_REMOVED
```

This enumeration indicates that the stream has been removed.

## 2.4 enum CNDataFormat

```
enum CNDataFormat {
```

```
    CN_INVALID = - 1,
```

```
    CN_PIXEL_FORMAT_YUV420_NV21 = 0,
```

```
    CN_PIXEL_FORMAT_YUV420_NV12 = 1,
```

```
    CN_PIXEL_FORMAT_BGR24 = 2,
```

```
    CN_PIXEL_FORMAT_RGB24 = 3,
```

```
    CN_PIXEL_FORMAT_ARGB32 = 4,
```

```
    CN_PIXEL_FORMAT_ABGR32 = 5,
```

```
    CN_PIXEL_FORMAT_RGBA32 = 6,
```

```
    CN_PIXEL_FORMAT_BGRA32 = 7,
```

```
};
```

```
enum cnstream::CNDataFormat
```

Enumeration variables describing the pixel format of the data in [CNDataFrame](#).

Values:

```
enumerator CN_INVALID
```

This frame is invalid.

```
enumerator CN_PIXEL_FORMAT_YUV420_NV21
```

This frame is in the YUV420SP(NV21) format.

```
enumerator CN_PIXEL_FORMAT_YUV420_NV12
```

This frame is in the YUV420sp(NV12) format.

```
enumerator CN_PIXEL_FORMAT_BGR24
```

This frame is in the BGR24 format.

```
enumerator CN_PIXEL_FORMAT_RGB24
```

This frame is in the RGB24 format.

```
enumerator CN_PIXEL_FORMAT_ARGB32
```

This frame is in the ARGB32 format.

```
enumerator CN_PIXEL_FORMAT_ABGR32
```

This frame is in the ABGR32 format.

```
enumerator CN_PIXEL_FORMAT_RGBA32
```

This frame is in the RGBA32 format.

```
enumerator CN_PIXEL_FORMAT_BGRA32
```

This frame is in the BGRA32 format.

## 2.5 enum StreamMsgType

```
enum StreamMsgType {
```

```
    EOS_MSG = 0,
```

```
    ERROR_MSG = 1,
```

```
    STREAM_ERR_MSG = 2,
```

```
    FRAME_ERR_MSG = 3,
```

```
    USER_MSG0 = 32,
```

```
    USER_MSG1 = 33,
```

```
    USER_MSG2 = 34,
```

```
    USER_MSG3 = 35,
```

```
    USER_MSG4 = 36,
```

```
    USER_MSG5 = 37,
```

```
    USER_MSG6 = 38,
```

```
    USER_MSG7 = 39,
```

```
    USER_MSG8 = 40,
```

```
    USER_MSG9 = 41,
```

```
};
```

`enum cnstream::StreamMsgType`

Enumeration variables describing the data stream message type.

Values:

`enumerator EOS_MSG`

The end of a stream message. The stream has received EOS message in all modules.

`enumerator ERROR_MSG`

An error message. The stream process has failed in one of the modules.

`enumerator STREAM_ERR_MSG`

Stream error message.

`enumerator FRAME_ERR_MSG`

Frame error message.

`enumerator USER_MSG0`

Reserved message. You can define your own messages.

`enumerator USER_MSG1`

Reserved message. You can define your own messages.

`enumerator USER_MSG2`

Reserved message. You can define your own messages.

`enumerator USER_MSG3`

Reserved message. You can define your own messages.

`enumerator USER_MSG4`

Reserved message. You can define your own messages.

`enumerator USER_MSG5`

Reserved message. You can define your own messages.

`enumerator USER_MSG6`

Reserved message. You can define your own messages.

`enumerator USER_MSG7`

Reserved message. You can define your own messages.

`enumerator USER_MSG8`

Reserved message. You can define your own messages.

`enumerator USER_MSG9`

Reserved message. You can define your own messages.

## 2.6 struct CNConfigBase

```
struct CNConfigBase {
```

```
    std::string config_root_dir = "";
```

```
};
```

```
struct cnstream::CNConfigBase
```

CNConfigBase is a base structure for configurations.

Subclassed by cnstream::CNGraphConfig, cnstream::CNModuleConfig, cnstream::CNSubgraphConfig, cnstream::ProfilerConfig

### Public Functions

```
bool ParseByJSONFile(const std::string &jfname)
```

Parses members from a JSON file.

#### Parameters

- [in] jfname: JSON configuration file path.

**Return** Returns true if the JSON file has been parsed successfully. Otherwise, returns false.

```
bool ParseByJSONStr(const std::string &jstr) = 0
```

Parses members from JSON string.

#### Parameters

- [in] jstr: JSON string of a configuration.

**Return** Returns true if the JSON string has been parsed successfully. Otherwise, returns false.

### Public Members

```
std::string config_root_dir = ""
```

The directory where a configuration file is stored.

## 2.7 struct ProfilerConfig

```
struct ProfilerConfig {  
    bool enable_profiling = false;  
    bool enable_tracing = false;  
    size_t trace_event_capacity = 100000;  
};
```

```
struct cnstream::ProfilerConfig: public cnstream::CNConfigBase
```

`ProfilerConfig` is a structure for profiler configuration.

The profiler configuration can be a JSON file.

```
{  
    "profiler_config" : {  
        "enable_profiling" : true,  
        "enable_tracing" : true  
    }  
}
```

**Note** It will not take effect when the profiler configuration is in the subgraph configuration.

### Public Functions

```
bool ParseByJSONStr(const std::string &jstr) override
```

Parses members from JSON string.

#### Parameters

- [in] `jstr`: JSON configuration string.

**Return** Returns true if the JSON string has been parsed successfully. Otherwise, returns false.

### Public Members

`bool enable_profiling = false`

Whether to enable profiling.

`bool enable_tracing = false`

Whether to enable tracing.

```
size_t trace_event_capacity = 100000
```

The maximum number of cached trace events.

## 2.8 struct CNModuleConfig

```
struct CNModuleConfig {  
    std::string name;  
    std::unordered_map<std::string, std::string> parameters;  
    int parallelism;  
    int maxInputQueueSize;  
    std::string className;  
    std::set<std::string> next;  
};
```

```
struct cnstream::CNModuleConfig: public cnstream::CNConfigBase
```

`CNModuleConfig` is a structure for module configuration. The module configuration can be a JSON file.

```
{  
  "name": {  
    "parallelism": 3,  
    "max_input_queue_size": 20,  
    "class_name": "cnstream::Inferencer",  
    "next_modules": ["module_name/subgraph:subgraph_name",  
                     "module_name/subgraph:subgraph_name", ...],  
    "custom_params" : {  
      "param_name" : "param_value",  
      "param_name" : "param_value",  
      ...  
    }  
  }  
}
```

### Public Functions

```
bool ParseByJSONStr(const std::string &jstr) override
```

Parses members except `CNModuleConfig::name` from the JSON file.

#### Parameters

- [in] `jstr`: JSON string of a configuration.

**Return** Returns true if the JSON string has been parsed successfully. Otherwise, returns false.

### Public Members

```
std::string name
```

The name of the module.

```
std::unordered_map<std::string, std::string> parameters
```

The key-value pairs. The pipeline passes this value to the `CNModuleConfig::name` module.

```
int parallelism
```

`Module` parallelism. It is equal to module thread number or the data queue of input data.

```
int maxInputQueueSize
```

The maximum size of the input data queues.

```
std::string className
```

The class name of the module.

```
std::set<std::string> next
```

The name of the downstream modules/subgraphs.

## 2.9 struct CNSubgraphConfig

```
struct CNSubgraphConfig {
```

```
    std::string name;
```

```
    std::string config_path;
```

```
    std::set<std::string> next;
```

```
};
```

```
struct cnstream::CNSubgraphConfig: public cnstream::CNConfigBase
```

`CNSubgraphConfig` is a structure for subgraph configuration.



The subgraph configuration can be a JSON file.

```
{
  "subgraphs:name" : {
    "config_path" : "/your/path/to/config_file.json",
    "next_modules": ["module_name/subgraph:subgraph_name",
                     "module_name/subgraph:subgraph_name", ...]
  }
}
```

### Public Functions

```
bool ParseByJSONStr(const std::string &jstr) override
```

Parses members except *CNSubgraphConfig::name* from the JSON file.

#### Parameters

- [in] *jstr*: JSON string of a configuration.

**Return** Returns true if the JSON string has been parsed successfully. Otherwise, returns false.

### Public Members

`std::string name`

The name of the subgraph.

`std::string config_path`

The path of configuration file.

`std::set<std::string> next`

The name of the downstream modules/subgraphs.

## 2.10 struct CNGraphConfig

```
struct CNGraphConfig {
    std::string name = "";
    ProfilerConfig profiler_config;
    std::vector<CNModuleConfig> module_configs;
    std::vector<CNSubgraphConfig> subgraph_configs;
};
```

`struct cnstream::CNGraphConfig: public cnstream::CNConfigBase`

`CNGraphConfig` is a structure for graph configuration.

You can use `CNGraphConfig` to initialize a `CNGraph` instance. The graph configuration can be a JSON file.

```
{
  "profiler_config" : {
    "enable_profiling" : true,
    "enable_tracing" : true
  },
  "module1": {
    "parallelism": 3,
    "max_input_queue_size": 20,
    "class_name": "cnstream::DataSource",
    "next_modules": ["subgraph:subgraph1"],
    "custom_params" : {
      "param_name" : "param_value",
      "param_name" : "param_value",
      ...
    }
  },
  "subgraph:subgraph1" : {
    "config_path" : "/your/path/to/subgraph_config_file.json"
  }
}
```

### Public Functions

`bool ParseByJSONStr(const std::string &jstr) override`

Parses members except `CNGraphConfig::name` from the JSON file.

#### Parameters

- [in] `jstr`: Json configuration string.

**Return** Returns true if the JSON string has been parsed successfully. Otherwise, returns false.

**Public Members**

`std::string name = ""`

Graph name.

`ProfilerConfig profiler_config`

Configuration of profiler.

`std::vector<CNModuleConfig> module_configs`

Configurations of modules.

`std::vector<CNSubgraphConfig> subgraph_configs`

Configurations of subgraphs.

## 2.11 struct Event

```
struct Event {
```

```
    EventType type;
```

```
    std::string stream_id;
```

```
    std::string message;
```

```
    std::string module_name;
```

```
    std::thread::id thread_id;
```

```
};
```

```
struct cnstream::Event
```

The [Event](#) is a structure describing the event information.

**Public Members**

`EventType type`

The event type.

`std::string stream_id`

The stream that posts this event.

`std::string message`

More detailed messages describing the event.

`std::string module_name`

The module that posts this event.

```
std::thread::id thread_id
```

The thread ID from which the event is posted.

## 2.12 struct DevContext

```
struct DevContext {
```

```
    DevType dev_type = DevType::INVALID;
```

```
    int dev_id = 0;
```

```
    int ddr_channel = 0;
```

```
};
```

```
struct cnstream::DevContext
```

[DevContext](#) is a structure holding the information that [CNDataFrame](#) data is allocated by CPU or MLU.

### Public Members

```
enum cnstream::DevContext::DevType dev_type = DevType::INVALID
```

Device type. The default value is `INVALID`.

```
int dev_id = 0
```

Ordinal device ID.

```
int ddr_channel = 0
```

Ordinal channel ID for MLU. The value should be in the range [0, 4).

## 2.13 struct CNInferBoundingBox

```
struct CNInferBoundingBox {
```

```
    float x;
```

```
    float y;
```

```
    float w;
```

```
    float h;
```

```
};
```

```
struct cnstream::CNInferBoundingBox
```

[CNInferBoundingBox](#) is a structure holding the bounding box information of a detected object in normalized coordinates.

**Public Members**

float **x**

The x-axis coordinate in the upper left corner of the bounding box.

float **y**

The y-axis coordinate in the upper left corner of the bounding box.

float **w**

The width of the bounding box.

float **h**

The height of the bounding box.

## 2.14 struct CNInferAttr

```
typedef struct {
```

```
    int id = - 1;
```

```
    int value = - 1;
```

```
    float score = 0;
```

```
} CNInferAttr;
```

```
struct cnstream::CNInferAttr
```

[CNInferAttr](#) is a structure holding the classification properties of an object.

**Public Members**

int **id** = -1

The unique ID of the classification. The value -1 means invalid.

int **value** = -1

The label value of the classification.

float **score** = 0

The label score of the classification.

## 2.15 struct CNInferObjs

```
struct CNInferObjs {  
    std::vector<std::shared_ptr<CNInferObject>> objs_;  
    std::mutex mutex_;  
};
```

```
struct cnstream::CNInferObjs : public NonCopyable
```

[CNInferObjs](#) is a structure holding inference results.

### Public Members

```
std::mutex mutex_;
```

The objects storing inference results.

## 2.16 struct InferData

```
struct InferData {  
    CNDataFormat input_fmt_;  
    int input_width_;  
    int input_height_;  
    std::shared_ptr<void> input_cpu_addr_;  
    size_t input_size_;  
    std::vector<std::shared_ptr<void>> output_cpu_addr_;  
    std::vector<size_t> output_sizes_;  
    size_t output_num_;  
};
```

```
struct cnstream::InferData
```

[InferData](#) is a structure holding the information of raw inference input & outputs.

**Public Members****CNDataFormat** `input_fmt_`

The input image's pixel format.

`int` `input_width_`

The input image's width.

`int` `input_height_`

The input image's height.

`std::shared_ptr<void>` `input_cpu_addr_`

The input data's CPU address.

`size_t` `input_size_`

The input data's size.

`std::vector<std::shared_ptr<void>>` `output_cpu_addr_`

The corresponding inference outputs to the input data.

`std::vector<size_t>` `output_sizes_`

The inference outputs' sizes.

`size_t` `output_num_`

The inference output count.

## 2.17 struct CNInferData

**struct CNInferData {****`std::unordered_map<std::string,` `std::vector<std::shared_ptr<InferData>>>`****`datas_map_;`****`std::mutex mutex_;`****`};`****`struct cnstream::CNInferData : public NonCopyable`****`CNInferData`** is a structure holding a map between module name and **InferData**.

**Public Members**

`std::unordered_map<std::string, std::vector<std::shared_ptr<InferData>>> datas_map_`

The map between module name and [InferData](#).

`std::mutex mutex_`

Inference data mutex.

## 2.18 struct StreamMsg

```
struct StreamMsg {
```

```
    StreamMsgType type;
```

```
    std::string stream_id;
```

```
    std::string module_name;
```

```
    int64_t pts = - 1;
```

```
};
```

```
struct cnstream::StreamMsg
```

The [StreamMsg](#) is a structure holding the information of a stream message.

**See** [StreamMsgType](#).

**Public Members**

[StreamMsgType](#) type

The type of a message.

`std::string stream_id`

Stream ID, set in [CNFrameInfo::stream\\_id](#).

`std::string module_name`

The module that posts this event.

`int64_t pts = -1`

The PTS (Presentation Timestamp) of this frame.



## 2.19 struct ESPacket

```
struct ESPacket {  
    unsigned char * data = nullptr;  
    int size = 0;  
    uint64_t pts = 0;  
    uint32_t flags = 0;  
};
```

```
struct cnstream::ESPacket
```

The [ESPacket](#) is a structure describing the elementary stream data packet.

### Public Members

`unsigned char *data = nullptr`

The video data.

`int size = 0`

The size of the data.

`uint64_t pts = 0`

The presentation time stamp of the data.

`uint32_t flags = 0`

The flags of the data.

## 2.20 struct MaximumVideoResolution

```
struct MaximumVideoResolution {  
    bool enable_variable_resolutions = false;  
    uint32_t maximum_width;  
    uint32_t maximum_height;  
};
```

```
struct cnstream::MaximumVideoResolution
```

The [MaximumVideoResolution](#) (not supported on MLU220/MLU270) is a structure describing the maximum video resolution parameters.

**Public Members**

`bool enable_variable_resolutions = false`  
Whether to enable variable resolutions.

`uint32_t maximum_width`  
The maximum video width.

`uint32_t maximum_height`  
The maximum video height.

## 2.21 struct StreamProfile

```
struct StreamProfile {
    std::string stream_name;
    uint64_t counter = 0;
    uint64_t completed = 0;
    int64_t dropped = 0;
    double latency = 0.0;
    double maximum_latency = 0.0;
    double minimum_latency = 0.0;
    double fps = 0.0;
};
```

```
struct cnstream::StreamProfile
```

The [StreamProfile](#) is a structure describing the performance statistics of streams.

**Public Functions**

```
StreamProfile() = default
```

Constructs a [StreamProfile](#) object with default constructor.

**Return** No return value.

```
StreamProfile(const StreamProfile &it) = default
```

Constructs a [StreamProfile](#) object with the copy of the contents of another object.

**Parameters**

- [in] `it`: Another object used to initialize an object.

**Return** No return value.

```
StreamProfile &operator=(const StreamProfile &it) = default
```

Replaces the contents with a copy of the contents of another `StreamProfile` object.

#### Parameters

- [in] `it`: Another object used to initialize the current object.

**Return** Returns a lvalue reference to the current instance.

```
StreamProfile(StreamProfile &&it)
```

Constructs a `StreamProfile` object with the contents of another object using move semantics.

#### Parameters

- [in] `it`: Another object used to initialize an object.

**Return** No return value.

```
StreamProfile &operator=(StreamProfile &&it)
```

Replaces the contents with those of another `StreamProfile` object using move semantics.

#### Parameters

- [in] `it`: Another object used to initialize the current object.

**Return** Returns a lvalue reference to the current instance.

### Public Members

`std::string stream_name`

The stream name.

`uint64_t counter = 0`

The frame counter, it is equal to `completed` plus `dropped`.

`uint64_t completed = 0`

The completed frame counter.

`int64_t dropped = 0`

The dropped frame counter.

`double latency = 0.0`

The average latency. (unit:ms)

`double maximum_latency = 0.0`

The maximum latency. (unit:ms)

```
double minimum_latency = 0.0
    The minimum latency. (unit:ms)

double fps = 0.0
    The throughput.
```

## 2.22 struct ProcessProfile

```
struct ProcessProfile {
    std::string process_name;

    uint64_t counter = 0;

    uint64_t completed = 0;

    int64_t dropped = 0;

    int64_t ongoing = 0;

    double latency = 0.0;

    double maximum_latency = 0.0;

    double minimum_latency = 0.0;

    double fps = 0.0;

    std::vector<StreamProfile> stream_profiles;
};
```

```
struct cnstream::ProcessProfile
```

The [ProcessProfile](#) is a structure describing the performance statistics of process.

### Public Functions

```
ProcessProfile() = default
```

Constructs a [ProcessProfile](#) object with default constructor.

**Return** No return value.

```
ProcessProfile(const ProcessProfile &it) = default
```

Constructs a [ProcessProfile](#) object with the copy of the contents of another object.

### Parameters

- [in] `it`: Another object used to initialize an object.

**Return** No return value.

`ProcessProfile &operator=(const ProcessProfile &it) = default`

Replaces the contents with a copy of the contents of another `ProcessProfile` object.

#### Parameters

- [in] `it`: Another object used to initialize the current object.

**Return** Returns a lvalue reference to the current instance.

`ProcessProfile(ProcessProfile &&it)`

Constructs a `ProcessProfile` object with the contents of another object using move semantics.

#### Parameters

- [in] `it`: Another object used to initialize an object.

**Return** No return value.

`ProcessProfile &operator=(ProcessProfile &&it)`

Replaces the contents with those of another `ProcessProfile` object using move semantics.

#### Parameters

- [in] `it`: Another object used to initialize the current object.

**Return** Returns a lvalue reference to the current instance.

### Public Members

`std::string process_name`

The process name.

`uint64_t counter = 0`

The frame counter, it is equal to completed plus dropped frames.

`uint64_t completed = 0`

The completed frame counter.

`int64_t dropped = 0`

The dropped frame counter.

`int64_t ongoing = 0`

The number of frame being processed.

`double latency = 0.0`

The average latency. (unit:ms)

`double maximum_latency = 0.0`

The maximum latency. (unit:ms)

```
double minimum_latency = 0.0
    The minimum latency. (unit:ms)

double fps = 0.0
    The throughput.

std::vector<StreamProfile> stream_profiles
    The stream profiles.
```

## 2.23 struct ModuleProfile

```
struct ModuleProfile {
    std::string module_name;

    std::vector<ProcessProfile> process_profiles;
};
```

```
struct cnstream::ModuleProfile
```

The [ModuleProfile](#) is a structure describing the performance statistics of module.

### Public Functions

```
ModuleProfile() = default
```

Constructs a [ModuleProfile](#) object with default constructor.

**Return** No return value.

```
ModuleProfile(const ModuleProfile &it) = default
```

Constructs a [ModuleProfile](#) object with the copy of the contents of another object.

### Parameters

- [in] *it*: Another object used to initialize an object.

**Return** No return value.

```
ModuleProfile &operator=(const ModuleProfile &it) = default
```

Replaces the contents with a copy of the contents of another [ModuleProfile](#) object.

### Parameters

- [in] *it*: Another object used to initialize the current object.

**Return** Returns a lvalue reference to the current instance.

```
ModuleProfile(ModuleProfile &&it)
```

Constructs a [ModuleProfile](#) object with the contents of another object using move semantics.

#### Parameters

- [in] `it`: Another object used to initialize an object.

**Return** No return value.

```
ModuleProfile &operator=(ModuleProfile &&it)
```

Replaces the contents with those of another [ModuleProfile](#) object using move semantics.

#### Parameters

- [in] `it`: Another object used to initialize the current object.

**Return** Returns a lvalue reference to the current instance.

#### Public Members

```
std::string module_name
```

The module name.

```
std::vector<ProcessProfile> process_profiles
```

The process profiles.

## 2.24 struct PipelineProfile

```
struct PipelineProfile {
```

```
    std::string pipeline_name;
```

```
    std::vector<ModuleProfile> module_profiles;
```

```
    ProcessProfile overall_profile;
```

```
};
```

```
struct cnstream::PipelineProfile
```

The [PipelineProfile](#) is a structure describing the performance statistics of pipeline.

**Public Functions**

```
PipelineProfile() = default
```

Constructs a [PipelineProfile](#) object with default constructor.

**Return** No return value.

```
PipelineProfile(const PipelineProfile &it) = default
```

Constructs a [PipelineProfile](#) object with the copy of the contents of another object.

**Parameters**

- [in] it: Another object used to initialize an object.

**Return** No return value.

```
PipelineProfile &operator=(const PipelineProfile &it) = default
```

Replaces the contents with a copy of the contents of another [PipelineProfile](#) object.

**Parameters**

- [in] it: Another object used to initialize the current object.

**Return** Returns a lvalue reference to the current instance.

```
PipelineProfile(PipelineProfile &&it)
```

Constructs a [PipelineProfile](#) object with the contents of another object using move semantics.

**Parameters**

- [in] it: Another object used to initialize an object.

**Return** No return value.

```
PipelineProfile &operator=(PipelineProfile &&it)
```

Replaces the contents with those of another [PipelineProfile](#) object using move semantics.

**Parameters**

- [in] it: Another object used to initialize the current object.

**Return** Returns a lvalue reference to the current instance.



**Public Members**

`std::string pipeline_name`

The pipeline name.

`std::vector<ModuleProfile> module_profiles`

The module profiles.

`ProcessProfile overall_profile`

The profile of the whole pipeline.

## 2.25 struct TraceElem

```
struct TraceElem {
```

```
    pair<std::string, int64_t> key;
```

```
    time_point time;
```

```
    TraceEvent::Type type;
```

```
};
```

```
struct cstream::TraceElem
```

The `TraceElem` is a structure describing a trace element used by profilers.

**Public Functions**

```
TraceElem() = default
```

Constructs a `TraceElem` object by using default constructor.

**Return** No return value.

```
TraceElem(const TraceElem &other) = default
```

Constructs a `TraceElem` object with the copy of the contents of another object.

**Parameters**

- `[in] other`: Another object used to initialize an object.

**Return** No return value.

```
TraceElem &operator=(const TraceElem &other) = default
```

Replaces the contents with a copy of the contents of another `TraceElem` object.

**Parameters**

- `[in] other`: Another object used to initialize the current object.

**Return** Returns a lvalue reference to the current instance.

```
TraceElem(TraceElem &&other)
```

Constructs a `TraceElem` object with the contents of another object using move semantics.

**Parameters**

- [in] `other`: Another object used to initialize an object.

**Return** No return value.

```
TraceElem &operator=(TraceElem &&other)
```

Replaces the contents with those of another `TraceElem` object using move semantics.

**Parameters**

- [in] `other`: Another object used to initialize the current object.

**Return** Returns a lvalue reference to the current instance.

```
TraceElem(const TraceEvent &event)
```

Constructs a `TraceElem` object with a trace event.

**Parameters**

- [in] `event`: A specific trace event instance.

**Return** No return value.

```
TraceElem(TraceEvent &&event)
```

Constructs a `TraceElem` object with a trace event using move semantics.

**Parameters**

- [in] `event`: A specific trace event instance.

**Return** No return value.

**Public Members**

`RecordKey` `key`

The unique identification of a frame.

`Time` `time`

The timestamp of an event.

`TraceEvent::Type` `type`

The type of an event. It could be START or END.

## 2.26 struct PipelineTrace

```
struct PipelineTrace {
```

```
    std::unordered_map<std::string, ProcessTrace> process_traces;
```

```
    std::unordered_map<std::string, ModuleTrace> module_traces;
```

```
};
```

```
struct cnstream::PipelineTrace
```

The [PipelineTrace](#) is a structure describing the trace data of a pipeline.

### Public Functions

```
PipelineTrace() = default
```

Constructs a [PipelineTrace](#) object by using default constructor.

**Return** No return value.

```
PipelineTrace(const PipelineTrace &other) = default
```

Constructs a [PipelineTrace](#) object with the copy of the contents of another object.

### Parameters

- [in] `other`: Another object used to initialize an object.

**Return** No return value.

```
PipelineTrace &operator=(const PipelineTrace &other) = default
```

Replaces the contents with a copy of the contents of another [PipelineTrace](#) object.

### Parameters

- [in] `other`: Another object used to initialize the current object.

**Return** Returns a lvalue reference to the current instance.

```
PipelineTrace(PipelineTrace &&other)
```

Constructs a [PipelineTrace](#) object with the contents of another object using move semantics.

### Parameters

- [in] `other`: Another object used to initialize an object.

**Return** No return value.

```
PipelineTrace &operator=(PipelineTrace &&other)
```

Replaces the contents with those of another [PipelineTrace](#) object using move semantics.

#### Parameters

- [in] `other`: Another object used to initialize the current object.

**Return** Returns a lvalue reference to the current instance.

#### Public Members

```
std::unordered_map<std::string, ProcessTrace> process_traces
```

The trace data of processes.

```
std::unordered_map<std::string, ModuleTrace> module_traces
```

The trace data of modules.

## 2.27 typedef ModuleParamSet

```
typedef std::unordered_map<std::string, std::string> cnstream::ModuleParamSet;
```

```
using cnstream::ModuleParamSet = std::unordered_map<std::string, std::string>
```

Defines an alias for `std::unordered_map<std::string, std::string>`. `ModuleParamSet` now denotes an unordered map which contains the pairs of parameter name and parameter value.

## 2.28 typedef BusWatcher

```
typedef std::function<EventHandleFlag ( const Event & )> cnstream::BusWatcher;
```

```
using cnstream::BusWatcher = std::function<EventHandleFlag(const Event&)>
```

Defines an alias of bus watcher function.

#### Parameters

- [in] `event`: The event is polled from the event bus.

**Return** Returns the flag that specifies how the event is handled.

## 2.29 typedef CNFrameInfoPtr

**typedef std::shared\_ptr<CNFrameInfo> cnstream::CNFrameInfoPtr;**

**using cnstream::CNFrameInfoPtr = std::shared\_ptr<CNFrameInfo>**

Defines an alias for the `std::shared_ptr<CNFrameInfo>`. `CNFrameInfoPtr` now denotes a shared pointer of frame information.

## 2.30 typedef CNInferFeature

**typedef std::vector<float> cnstream::CNInferFeature;**

**using cnstream::CNInferFeature = std::vector<float>**

Defines an alias for `std::vector<float>`. `CNInferFeature` contains one kind of inference feature.

## 2.31 typedef CNInferFeatures

**typedef std::vector<std::pair<std::string, CNInferFeature>> cnstream::CNInferFeatures;**

**using cnstream::CNInferFeatures = std::vector<std::pair<std::string, CNInferFeature>>**

Defines an alias for `std::vector<std::pair<std::string, std::vector<float>>>`. `CNInferFeatures` contains all kinds of features for one object.

## 2.32 typedef StringPairs

**typedef std::vector<std::pair<std::string, std::string>> cnstream::StringPairs;**

**using cnstream::StringPairs = std::vector<std::pair<std::string, std::string>>**

Defines an alias for `std::vector<std::pair<std::string, std::string>>`.

## 2.33 typedef CNInferObjectPtr

**typedef std::shared\_ptr<CNInferObject> cnstream::CNInferObjectPtr;**

**using cnstream::CNInferObjectPtr = std::shared\_ptr<CNInferObject>**

Defines an alias for the `std::shared_ptr<CNInferObject>`. `CNInferObjectPtr` now denotes a shared pointer of inference objects.

## 2.34 typedef CNDataFramePtr

**typedef std::shared\_ptr<CNDataFrame> cnstream::CNDataFramePtr;**

**using cnstream::CNDataFramePtr = std::shared\_ptr<CNDataFrame>**

Defines an alias for the std::shared\_ptr<CNDataFrame>.

## 2.35 typedef CNInferObjsPtr

**typedef std::shared\_ptr<CNInferObjs> cnstream::CNInferObjsPtr;**

**using cnstream::CNInferObjsPtr = std::shared\_ptr<CNInferObjs>**

Defines an alias for the std::shared\_ptr<CNInferObjs>.

## 2.36 typedef CNObjsVec

**typedef std::vector<std::shared\_ptr<CNInferObject>> cnstream::CNObjsVec;**

**using cnstream::CNObjsVec = std::vector<std::shared\_ptr<CNInferObject>>**

Defines an alias for the std::vector<std::shared\_ptr<CNInferObject>>.

## 2.37 typedef CNInferDataPtr

**typedef std::shared\_ptr<CNInferData> cnstream::CNInferDataPtr;**

**using cnstream::CNInferDataPtr = std::shared\_ptr<CNInferData>**

Defines an alias for the std::shared\_ptr<CNInferData>.

## 2.38 typedef Clock

**typedef std::chrono::steady\_clock cnstream::Clock;**

**using cnstream::Clock = std::chrono::steady\_clock**

Defines an alias for the std::chrono::steady\_clock.

## 2.39 typedef Duration

**typedef std::chrono::duration<double, std::milli> cnstream::Duration;**

using cnstream::Duration = std::chrono::duration<double, std::milli>

Defines an alias for the std::chrono::duration<double, std::milli>.

## 2.40 typedef Time

**typedef Clock::time\_point cnstream::Time;**

using cnstream::Time = Clock::time\_point

Defines an alias for the std::chrono::steady\_clock::timepoint.

## 2.41 typedef RecordKey

**typedef std::pair<std::string, int64\_t> cnstream::RecordKey;**

using cnstream::RecordKey = std::pair<std::string, int64\_t>

Defines an alias for the std::pair<std::string, int64\_t>. RecordKey now denotes a pair of the stream name *CNFrameInfo::stream\_id* and pts *CNFrameInfo::timestamp*.

## 2.42 typedef ProcessTrace

**typedef std::vector<TraceElem> cnstream::ProcessTrace;**

using cnstream::ProcessTrace = std::vector<TraceElem>

Defines an alias for the std::vector<TraceElem>. ProcessTrace now denotes a vector which contains trace elements for a process.

## 2.43 typedef ModuleTrace

**typedef std::unordered\_map<std::string, ProcessTrace> cnstream::ModuleTrace;**

using cnstream::ModuleTrace = std::unordered\_map<std::string, ProcessTrace>

Defines an alias for the std::unordered\_map<std::string, ProcessTrace>. ModuleTrace now denotes an unordered map which contains the pairs of the process name and the ProcessTrace object for a module.



## 3 Class

### 3.1 Framework

#### 3.1.1 Class Collection

`class Collection : public NonCopyable`

`Collection` is a class storing structured data of variable types.

**Note** This class is thread safe.

#### API Reference

##### Collection

```
cnstream::Collection::Collection() = default
```

Constructs an instance with empty value.

**Return** No return value.

##### ~Collection

```
cnstream::Collection::~~Collection() = default
```

Destructs an instance.

**Return** No return value.



### Get

```
template<typename ValueT>
ValueT &cnstream::Collection::Get(const std::string &tag)
```

Gets the reference to the object of typename ValueT if it exists, otherwise crashes.

#### Parameters

- [in] tag: The unique identifier of the data.

**Return** Returns the reference to the object of typename ValueT which is tagged by tag.

### Add

```
template<typename ValueT>
ValueT &cnstream::Collection::Add(const std::string &tag,
                                const ValueT &value)
```

Adds data tagged by tag. Crashes when there is already a piece of data tagged by tag.

#### Parameters

- [in] tag: The unique identifier of the data.
- [in] value: Value to be add.

**Return** Returns the reference to the object of typename ValueT which is tagged by tag.

```
template<typename ValueT>
ValueT &cnstream::Collection::Add(const std::string &tag,
                                ValueT &&value)
```

Adds data tagged by tag using move semantics. Crashes when there is already a piece of data tagged by tag.

#### Parameters

- [in] tag: The unique identifier of the data.
- [in] value: Value to be add.

**Return** Returns the reference to the object of typename ValueT which is tagged by tag.

### AddIfNotExists

```
template<typename ValueT>
bool cnstream::Collection::AddIfNotExists(const std::string &tag,
                                         const ValueT &value)
```

Adds data tagged by `tag`, only if there is no piece of data tagged by `tag`.

#### Parameters

- [in] `tag`: The unique identifier of the data.
- [in] `value`: Value to be add.

**Return** Returns true if the data is added successfully, otherwise returns false.

```
template<typename ValueT>
bool cnstream::Collection::AddIfNotExists(const std::string &tag,
                                         ValueT &&value)
```

Adds data tagged by `tag` using move semantics, only if there is no piece of data tagged by `tag`.

#### Parameters

- [in] `tag`: The unique identifier of the data.
- [in] `value`: Value to be add.

**Return** Returns true if the data is added successfully, otherwise returns false.

### HasValue

```
bool cnstream::Collection::HasValue(const std::string &tag)
```

Checks whether there is the data tagged by `tag`.

#### Parameters

- [in] `tag`: The unique identifier of the data.

**Return** Returns true if there is already a piece of data tagged by `tag`, otherwise returns false.

### Type

```
const std::type_info &cnstream::Collection::Type(const std::string &tag)
```

Gets type information for data tagged by `tag`.

#### Parameters

- [in] `tag`: The unique identifier of the data.

**Return** Returns type information of the data tagged by `tag`.

### TaggedIsOfType

```
template<typename ValueT>
bool cnstream::Collection::TaggedIsOfType(const std::string &tag)
```

Checks if the type of data tagged by `tag` is `ValueT` or not.

#### Parameters

- `tag`: The unique identifier of the data.

**Return** Returns true if the type of data tagged by `tag` is `ValueT`, otherwise returns false.

### 3.1.2 Class ParametersChecker

```
class ParametersChecker
```

ParameterChecker is a class used to check module parameters.

#### API Reference

#### CheckPath

```
bool cnstream::ParametersChecker::CheckPath(const std::string &path,
                                             const ModuleParamSet &paramSet)
```

Checks if a path exists.

#### Parameters

- [in] `path`: The path relative to JSON file or an absolute path.
- [in] `paramSet`: The module parameters. The JSON file path is one of the parameters.

**Return** Returns true if the path exists. Otherwise, returns false.

#### IsNum

```
bool cnstream::ParametersChecker::IsNum(const std::list<std::string> &check_list,
                                         const ModuleParamSet &paramSet,
                                         std::string &err_msg,
                                         bool greater_than_zero = false)
```

Checks if the parameters are number, and the value is specified in the correct range.

#### Parameters

- [in] `check_list`: A list of parameter names.
- [in] `paramSet`: The module parameters.
- [out] `err_msg`: The error message.

- [in] `greater_than_zero`: If this parameter is set to `true`, the parameter set should be greater than or equal to zero. If this parameter is set to `false`, the parameter set is less than zero.

**Return** Returns true if the parameters are number and the value is in the correct range. Otherwise, returns false.

### 3.1.3 Class ParamRegister

`class ParamRegister`

`ParamRegister` is a class for module parameter registration.

Each module registers its own parameters and descriptions. This is used in CNStream Inspect tool to detect parameters of each module.

#### API Reference

##### Register

```
void cnstream::ParamRegister::Register(const std::string &key,
                                       const std::string &desc)
```

Registers a paramter and its description.

This is used in CNStream Inspect tool.

##### Parameters

- [in] `key`: The parameter name.
- [in] `desc`: The description of the paramter.

**Return** Void.

##### GetParams

```
std::vector<std::pair<std::string, std::string>> cnstream::ParamRegister::GetParams()
```

Gets the registered paramters and the parameter descriptions.

This is used in CNStream Inspect tool.

**Return** Returns the registered paramters and the parameter descriptions.

### IsRegistered

```
bool cnstream::ParamRegister::IsRegistered(const std::string &key) const
```

Checks if the paramter is registered.

This is used in CNStream Inspect tool.

#### Parameters

- [in] key: The parameter name.

**Return** Returns true if the parameter has been registered. Otherwise, returns false.

### SetModuleDesc

```
void cnstream::ParamRegister::SetModuleDesc(const std::string &desc)
```

Sets the description of the module.

This is used in CNStream Inspect tool.

#### Parameters

- [in] desc: The description of the module.

**Return** Void.

### GetModuleDesc

```
std::string cnstream::ParamRegister::GetModuleDesc()
```

Gets the description of the module.

This is used in CNStream Inspect tool.

**Return** Returns the description of the module.

### 3.1.4 Class EventBus

```
class EventBus : private NonCopyable
```

[EventBus](#) is a class that transmits events from modules to a pipeline.

## API Reference

### ~EventBus

```
cnstream::EventBus::~~EventBus()
```

Destructor. A destructor to destruct event bus.

**Return** No return value.

### Start

```
bool cnstream::EventBus::Start()
```

Starts an event bus thread.

**Return** Returns true if start successfully, otherwise false.

### Stop

```
void cnstream::EventBus::Stop()
```

Stops an event bus thread.

**Return** No return values.

### AddBusWatch

```
uint32_t cnstream::EventBus::AddBusWatch(BusWatcher func)
```

Adds a watcher to the event bus.

#### Parameters

- [in] func: The bus watcher to be added.

**Return** The number of bus watchers that has been added to this event bus.

### PostEvent

```
bool cnstream::EventBus::PostEvent(Event event)
```

Posts an event to a bus.

#### Parameters

- [in] event: The event to be posted.

**Return** Returns true if this function run successfully. Otherwise, returns false.

### 3.1.5 Class CNFrameInfo

`class CNFrameInfo : private NonCopyable`

`CNFrameInfo` is a class holding the information of a frame.

#### Variables

##### `stream_id`

`std::string cnstream::CNFrameInfo::stream_id`

The data stream aliases where this frame is located to.

##### `timestamp`

`int64_t cnstream::CNFrameInfo::timestamp = -1`

The time stamp of this frame.

##### `flags`

`size_t cnstream::CNFrameInfo::flags = 0`

The mask for this frame, `CNFrameFlag`.

##### `datas_lock`

`std::mutex cnstream::CNFrameInfo::datas_lock_`

(Deprecated) Uses `CNFrameInfo::collection` instead.

##### `datas`

`std::unordered_map<int, any> cnstream::CNFrameInfo::datas`

(Deprecated) Uses `CNFrameInfo::collection` instead.

#### API Reference

##### Create

```
std::shared_ptr<CNFrameInfo> cnstream::CNFrameInfo::Create(const std::string
                                                         &stream_id,
                                                         bool eos = false,
                                                         std::shared_ptr<CNFrameInfo>
                                                         payload = nullptr)
```

Creates a `CNFrameInfo` instance.

#### Parameters

- [in] `stream_id`: The data stream alias. Identifies which data stream the frame data comes from.
- [in] `eos`: Whether this is the end of the stream. This parameter is set to false by default to create a `CNFrameInfo` instance. If you set this parameter to true, `CNDataFrame::flags` will be set to `CN_FRAME_FLAG_EOS`. Then, the modules do not have permission to process this frame. This frame should be handed over to the pipeline for processing.

**Return** Returns `shared_ptr` of `CNFrameInfo` if this function has run successfully. Otherwise, returns NULL.

#### CNFrameInfo

```
cnstream::CNFrameInfo::CNFrameInfo() = default
```

#### ~CNFrameInfo

```
cnstream::CNFrameInfo::~CNFrameInfo()
```

Destructs `CNFrameInfo` object.

**Return** No return value.

#### IsEos

```
bool cnstream::CNFrameInfo::IsEos()
```

Checks whether `DataFrame` is end of stream (EOS) or not.

**Return** Returns true if the frame is EOS. Returns false if the frame is not EOS.



### IsRemoved

```
bool cnstream::CNFrameInfo::IsRemoved()
```

Checks whether DataFrame is removed or not.

**Return** Returns true if the frame is EOS. Returns false if the frame is not EOS.

### IsValid

```
bool cnstream::CNFrameInfo::IsValid()
```

Checks if DataFrame is valid or not.

**Return** Returns true if frame is invalid, otherwise returns false.

### SetStreamIndex

```
void cnstream::CNFrameInfo::SetStreamIndex(uint32_t index)
```

Sets index (usually the index is a number) to identify stream.

#### Parameters

- [in] index: Number to identify stream.

**Return** No return value.

**Note** This is only used for distributing each stream data to the appropriate thread. We do not recommend SDK users to use this API because it will be removed later.

### GetStreamIndex

```
uint32_t cnstream::CNFrameInfo::GetStreamIndex() const
```

Gets index number which identifies stream.

**Return** Index number.

**Note** This is only used for distributing each stream data to the appropriate thread. We do not recommend SDK users to use this API because it will be removed later.

### 3.1.6 Class IModuleObserver

class IModuleObserver

`IModuleObserver` is an interface class. Users need to implement an observer based on this, and register it to one module.

#### API Reference

##### notify

```
void cnstream::IModuleObserver::notify(std::shared_ptr<CNFrameInfo> data) = 0
```

Notifies "data" after being processed by this module.

#### Parameters

- [in] data: The frame that is notified to observer.

**Return** No return value.

##### ~IModuleObserver

```
cnstream::IModuleObserver::~IModuleObserver() = default
```

Default destructor. A destructor to destruct module observer.

**Return** No return value.

### 3.1.7 Class Module

class Module: private NonCopyable

`Module` is the parent class of all modules. A module could have configurable number of upstream links and downstream links. Some modules are already constructed with a framework, such as source, inferencer, and so on. You can also design your own modules.

Subclassed by `cnstream::ModuleEx`, `cnstream::SourceModule`

#### API Reference

##### Module

```
cnstream::Module::Module(const std::string &name)
```

Constructor. A constructor to construct module object.

#### Parameters

- [in] name: The name of a module. Modules defined in a pipeline must have different names.

**Return** No return value.

#### ~Module

```
cnstream::Module::~~Module()
```

Destructor. A destructor to destruct module instance.

**Return** No return value.

#### SetObserver

```
void cnstream::Module::SetObserver(IModuleObserver *observer)
```

Registers an observer to the module.

##### Parameters

- [in] observer: An observer you defined.

**Return** No return value.

#### Open

```
bool cnstream::Module::Open(ModuleParamSet param_set) = 0
```

Opens resources for a module.

##### Parameters

- [in] param\_set: A set of parameters for this module.

**Return** Returns true if this function has run successfully. Otherwise, returns false.

**Note** You do not need to call this function by yourself. This function is called by pipeline automatically when the pipeline is started. The pipeline calls the `Process` function of this module automatically after the `Open` function is done.

#### Close

```
void cnstream::Module::Close() = 0
```

Closes resources for a module.

**Return** No return value.

**Note** You do not need to call this function by yourself. This function is called by pipeline automatically when the pipeline is stopped. The pipeline calls the `Close` function of this

module automatically after the `Open` and `Process` functions are done.

### Process

```
int cnstream::Module::Process(std::shared_ptr<CNFrameInfo> data) = 0
```

Processes data.

#### Parameters

- [in] `data`: The data to be processed by the module.

#### Return Value

- 0: The data is processed successfully. The data should be transmitted in the framework then.
- >0: The data is processed successfully. The data has been handled by this module. The `hasTransmit_` must be set. The `Pipeline::ProvideData` should be called by `Module` to transmit data to the next modules in the pipeline.
- <0: `Pipeline` will post an event with the `EVENT_ERROR` event type and return number.

### OnEos

```
void cnstream::Module::OnEos(const std::string &stream_id)
```

Notifies flow-EOS arriving, the module should reset internal status if needed.

#### Parameters

- [in] `stream_id`: The stream identification.

**Note** This function will be invoked when flow-EOS is forwarded by the framework.

### GetName

```
std::string cnstream::Module::GetName() const
```

Gets the name of this module.

**Return** Returns the name of this module.

### PostEvent

```
bool cnstream::Module::PostEvent(EventType type,
                                const std::string &msg)
```

Posts an event to the pipeline.

#### Parameters

- [in] type: The type of an event.
- [in] msg: The event message string.

**Return** Returns true if this function has run successfully. Returns false if this module has not been added to the pipeline.

```
bool cnstream::Module::PostEvent(Event e)
```

Posts an event to the pipeline.

#### Parameters

- *Event*: with event type, stream\_id, message, module name and thread\_id.

**Return** Returns true if this function has run successfully. Returns false if this module has not been added to the pipeline.

### TransmitData

```
bool cnstream::Module::TransmitData(std::shared_ptr<CNFrameInfo> data)
```

Transmits data to the following stages.

Valid when the module has permission to transmit data by itself.

#### Parameters

- [in] data: A pointer to the information of the frame.

**Return** Returns true if the data has been transmitted successfully. Otherwise, returns false.

### CheckParamSet

```
bool cnstream::Module::CheckParamSet(const ModuleParamSet &paramSet) const
```

Checks parameters for a module, including parameter name, type, value, validity, and so on.

#### Parameters

- [in] paramSet: Parameters for this module.

**Return** Returns true if this function has run successfully. Otherwise, returns false.

### GetContainer

```
Pipeline *cnstream::Module::GetContainer() const
```

Gets the pipeline this module belongs to.

**Return** Returns the pointer to pipeline instance.

### GetProfiler

```
ModuleProfiler *cnstream::Module::GetProfiler()
```

Gets module profiler.

**Return** Returns a pointer to the module's profiler.

### HasTransmit

```
bool cnstream::Module::HasTransmit() const
```

Checks if this module has permission to transmit data by itself.

**Return** Returns true if this module has permission to transmit data by itself. Otherwise, returns false.

**See** [Process](#)

## 3.1.8 Class ModuleEx

```
class ModuleEx: public cnstream::Module
```

[ModuleEx](#) is the base class of the modules who have permission to transmit processed data by themselves.

### API Reference

#### ModuleEx

```
cnstream::ModuleEx::ModuleEx(const std::string &name)
```

Constructor. A constructor to construct the module which has permission to transmit processed data by itself.

#### Parameters

- [in] `name`: The name of a module. Modules defined in a pipeline must have different names.

**Return** No return value.

### 3.1.9 Class Pipeline

`class Pipeline : private NonCopyable`

`Pipeline` is the manager of the modules, which manages data transmission between modules and controls messages delivery.

#### API Reference

##### Pipeline

```
cnstream::Pipeline::Pipeline(const std::string &name)
```

A constructor to construct one pipeline.

##### Parameters

- [in] `name`: The name of the pipeline.

**Return** No return value.

##### ~Pipeline

```
cnstream::Pipeline::~Pipeline()
```

A destructor to destruct one pipeline.

##### Parameters

- [in] `name`: The name of the pipeline.

**Return** No return value.

##### GetName

```
const std::string &cnstream::Pipeline::GetName() const
```

Gets the pipeline's name.

**Return** Returns the pipeline's name.

### BuildPipeline

```
bool cnstream::Pipeline::BuildPipeline(const std::vector<CNModuleConfig>
                                     &module_configs,
                                     const ProfilerConfig &profiler_config =
                                     ProfilerConfig())
```

Builds a pipeline by module configurations.

#### Parameters

- [in] module\_configs: The configurations of a module.
- [in] profiler\_config: The configuration of a profiler.

**Return** Returns true if this function has run successfully. Otherwise, returns false.

```
bool cnstream::Pipeline::BuildPipeline(const CNGraphConfig &graph_config)
```

Builds a pipeline by graph configuration.

#### Parameters

- [in] graph\_config: The configuration of a graph.

**Return** Returns true if this function has run successfully. Otherwise, returns false.

### BuildPipelineByJSONFile

```
bool cnstream::Pipeline::BuildPipelineByJSONFile(const std::string &config_file)
```

Builds a pipeline from a JSON file. You can learn to write a configuration file by looking at the description of [CNGraphConfig](#).

#### Parameters

- [in] config\_file: The configuration file in JSON format.

**See** [CNGraphConfig](#)

**Return** Returns true if this function has run successfully. Otherwise, returns false.

### Start

```
bool cnstream::Pipeline::Start()
```

Starts a pipeline. Starts data transmission in a pipeline. Calls the `Open` function for all modules. See [Module::Open](#).

**Return** Returns true if this function has run successfully. Returns false if the `Open` function did not run successfully in one of the modules, or the link modules failed.



### Stop

```
bool cnstream::Pipeline::Stop()
```

Stops data transmissions in a pipeline.

**Return** Returns true if this function has run successfully. Otherwise, returns false.

### IsRunning

```
bool cnstream::Pipeline::IsRunning() const
```

The running status of a pipeline.

**Return** Returns true if the pipeline is running. Returns false if the pipeline is not running.

### GetModule

```
Module *cnstream::Pipeline::GetModule(const std::string &module_name) const
```

Gets a module in current pipeline by name.

#### Parameters

- [in] `module_name`: The module name specified in the module configuration. If you specify a module name written in the module configuration, the first module with the same name as the specified module name in the order of DFS will be returned. When there are modules with the same name as other graphs in the subgraph, you can also find the module by adding the graph name prefix divided by slash. eg. `pipeline_name/subgraph1/module1`.

**Return** Returns the module pointer if the module has been added to the current pipeline. Otherwise, returns nullptr.

### GetModuleConfig

```
CNModuleConfig cnstream::Pipeline::GetModuleConfig(const std::string &module_name)
const
```

Gets the module configuration by the module name.

#### Parameters

- [in] `module_name`: The module name specified in module configuration. The module name can be specified by two ways, see [Pipeline::GetModule](#) for detail.

**Return** Returns module configuration if this function has run successfully. Returns NULL if

the module specified by `module_name` has not been added to the current pipeline.

### IsProfilingEnabled

```
bool cnstream::Pipeline::IsProfilingEnabled() const
```

Checks if profiling is enabled.

**Return** Returns true if profiling is enabled.

### IsTracingEnabled

```
bool cnstream::Pipeline::IsTracingEnabled() const
```

Checks if tracing is enabled.

**Return** Returns true if tracing is enabled.

### ProvideData

```
bool cnstream::Pipeline::ProvideData(const Module *module,  
                                     std::shared_ptr<CNFrameInfo> data)
```

Provides data for the pipeline that is used in source module or the module transmitted by itself.

#### Parameters

- [in] `module`: The module that provides data.
- [in] `data`: The data that is transmitted to the pipeline.

**Return** Returns true if this function has run successfully. Returns false if the module is not added in the pipeline or the pipeline has been stopped.

**Note** `ProvideData` can be only called by the head modules in pipeline. A head module means the module has no parent modules.

**See** [Module::Process](#).

### GetEventBus

```
EventBus *cnstream::Pipeline::GetEventBus() const
```

Gets the event bus in the pipeline.

**Return** Returns the event bus.

### SetStreamMsgObserver

```
void cnstream::Pipeline::SetStreamMsgObserver(StreamMsgObserver*observer)
```

Binds the stream message observer with a pipeline to receive stream message from this pipeline.

#### Parameters

- [in] observer: The stream message observer.

**Return** No return value.

**See** [StreamMsgObserver](#).

### GetStreamMsgObserver

```
StreamMsgObserver*cnstream::Pipeline::GetStreamMsgObserver() const
```

Gets the stream message observer that has been bound with this pipeline.

**Return** Returns the stream message observer that has been bound with this pipeline.

**See** [Pipeline::SetStreamMsgObserver](#).

### GetProfiler

```
PipelineProfiler*cnstream::Pipeline::GetProfiler() const
```

Gets this pipeline's profiler.

**Return** Returns profiler.

### GetTracer

```
PipelineTracer*cnstream::Pipeline::GetTracer() const
```

Gets this pipeline's tracer.

**Return** Returns tracer.

### IsRootNode

```
bool cnstream::Pipeline::IsRootNode(const std::string &module_name) const
```

Checks if module is root node of pipeline or not. The module name can be specified by two ways, see [Pipeline::GetModule](#) for detail.

#### Parameters

- [in] module\_name: module name.

**Return** Returns true if it's root node, otherwise returns false.

### IsLeafNode

```
bool cnstream::Pipeline::IsLeafNode(const std::string &module_name) const
```

Checks if module is leaf node of pipeline. The module name can be specified by two ways, see [Pipeline::GetModule](#) for detail.

#### Parameters

- [in] module\_name: module name.

**Return** Returns true if it's leaf node, otherwise returns false.

### RegisterFrameDoneCallBack

```
void cnstream::Pipeline::RegisterFrameDoneCallBack(const std::function<void>std::shared_ptr<CNFrameInfo>  
> &callback
```

Registers a callback to be called after the frame process is done.

#### Parameters

- [in] callback: The call back function.

**Return** No return value.

### 3.1.10 Class StreamMsgObserver

```
class StreamMsgObserver
```

Receives stream messages from a pipeline. To receive stream messages from the pipeline, you can define a class to inherit the [StreamMsgObserver](#) class and call the `Update` function. The observer instance is bounded to the pipeline using the [Pipeline::SetStreamMsgObserver](#) function .

**See** [Pipeline::SetStreamMsgObserver](#) [StreamMsg](#) [StreamMsgType](#).

## API Reference

### Update

```
void cnstream::StreamMsgObserver::Update(const StreamMsg &msg) = 0
```

Receives stream messages from a pipeline passively.

#### Parameters

- [in] msg: The stream message from a pipeline.

**Return** No return value.

### ~StreamMsgObserver

```
cnstream::StreamMsgObserver::~~StreamMsgObserver() = default
```

Default destructor to destruct stream message observer.

**Return** No return value.

### 3.1.11 Class SourceHandler

```
class SourceHandler : private NonCopyable
```

**SourceHandler** is a class that handles various sources, such as RTSP and video file.

Subclassed by `cnstream::ESJpegMemHandler`, `cnstream::ESMemHandler`, `cnstream::FileHandler`, `cnstream::RawImgMemHandler`, `cnstream::RtspHandler`

## API Reference

### SourceHandler

```
cnstream::SourceHandler::SourceHandler(SourceModule *module,  
                                       const std::string &stream_id)
```

Constructs a source handler.

#### Parameters

- [in] module: The source module this handler belongs to.
- [in] stream\_id: The name of the stream.

**Return** No return value.

### ~SourceHandler

```
cnstream::SourceHandler::~SourceHandler()
```

Destructs a source module.

**Return** No return value.

### Open

```
bool cnstream::SourceHandler::Open() = 0
```

Opens a decoder.

**Return** Returns true if a decoder is opened successfully, otherwise returns false.

### Close

```
void cnstream::SourceHandler::Close() = 0
```

Closes a decoder.

**Return** No return value.

### GetStreamId

```
std::string cnstream::SourceHandler::GetStreamId() const
```

Gets the stream identification.

**Return** Returns the name of stream.

### CreateFrameInfo

```
std::shared_ptr<CNFrameInfo> cnstream::SourceHandler::CreateFrameInfo(
    bool eos = false,
    std::shared_ptr<CNFrameInfo> payload = nullptr)
```

Creates the context of CNFrameInfo .

#### Parameters

- [in] eos: The flag marking the frame is end of stream.
- [in] payload: The payload of CNFrameInfo. It's useless now.

**Return** Returns the context of `CNFameInfo`.

### SendData

```
bool cnstream::SourceHandler::SendData(std::shared_ptr<CNFrameInfo> data)
```

Sends data to next module.

#### Parameters

- [in] `data`: The data need to be sent to next modules.

**Return** Returns true if send data successfully, otherwise returns false.

### 3.1.12 Class SourceModule

```
class SourceModule : public cnstream::Module
```

`SourceModule` is the base class of source modules.

Subclassed by `cnstream::DataSource`

### API Reference

#### SourceModule

```
cnstream::SourceModule::SourceModule(const std::string &name)
```

Constructs a source module.

#### Parameters

- [in] `name`: The name of the source module.

**Return** No return value.

#### ~SourceModule

```
cnstream::SourceModule::~SourceModule()
```

Destructs a source module.

**Return** No return value.

### AddSource

```
int cnstream::SourceModule::AddSource(std::shared_ptr<SourceHandler> handler)
```

Adds one stream to [DataSource](#) module. This function should be called after pipeline starts.

#### Parameters

- [in] handler: The source handler

#### Return Value

- Returns: 0 for success, otherwise returns -1.

### GetSourceHandler

```
std::shared_ptr<SourceHandler> cnstream::SourceModule::GetSourceHandler(const
                                                                    std::string
                                                                    &stream_id)
```

Destructs a source module.

#### Parameters

- [in] stream\_id: The stream identifier.

**Return** Returns the handler of the stream.

### RemoveSource

```
int cnstream::SourceModule::RemoveSource(std::shared_ptr<SourceHandler> handler,
                                          bool force = false)
```

Removes one stream from [DataSource](#) module with given handler. This function should be called before pipeline stops.

#### Parameters

- [in] handler: The handler of one stream.
- [in] force: The flag describing the removing behaviour.

#### Return Value

- 0: success (always success by now).

**Note** If `force` sets to true, the stream will be removed immediately, otherwise the stream will be removed after all cached frames are processed.

```
int cnstream::SourceModule::RemoveSource(const std::string &stream_id,
                                          bool force = false)
```

Removes one stream from [DataSource](#) module with given the stream identification. This function should be called before pipeline stops.



**Parameters**

- [in] `stream_id`: The stream identification.
- [in] `force`: The flag describing the removing behaviour.

**Return Value**

- 0: success (always success by now).

**Note** If `force` sets to true, the stream will be removed immediately, otherwise the stream will be removed after all cached frames are processed.

**RemoveSources**

```
int cnstream::SourceModule::RemoveSources(bool force = false)
```

Removes all streams from `DataSource` module.

**Parameters**

- [in] `force`: The flag describing the removing behaviour.

**Return Value**

- 0: success (always success by now).

**Note** If `force` sets to true, the stream will be removed immediately, otherwise the stream will be removed after all cached frames are processed.

**3.1.13 Class CNSyncedMemory**

```
class CNSyncedMemory : private NonCopyable
```

`CNSyncedMemory` is a class synchronizing memory between CPU and MLU.

If the data on MLU is the latest, the data on CPU should be synchronized before processing the data on CPU. Vice versa, if the data on CPU is the latest, the data on MLU should be synchronized before processing the data on MLU.

**Note** `CNSyncedMemory::Head()` always returns `SyncedHead::UNINITIALIZED` when memory size is 0.

**Datatypes Reference****enum SyncedHead****enum SyncedHead {**

**UNINITIALIZED = 0,**

**HEAD\_AT\_CPU = 1,**

**HEAD\_AT\_MLU = 2,**

```

    SYNCED = 3,
};

enum cnstream: CNSyncedMemory::SyncedHead
    An enumerator describing the synchronization status.

    Values:

    enumerator UNINITIALIZED
        The memory is not allocated.

    enumerator HEAD_AT_CPU
        The data is updated to CPU but is not synchronized to MLU yet.

    enumerator HEAD_AT_MLU
        The data is updated to MLU but is not synchronized to CPU yet.

    enumerator SYNCED
        The data is synchronized to both CPU and MLU.

```

## API Reference

### CNSyncedMemory

```
cnstream: CNSyncedMemory::CNSyncedMemory(size_t size)
```

Constructor to construct synchronized memory object.

#### Parameters

- [in] size: The size of the memory.

**Return** No return value.

```
cnstream: CNSyncedMemory::CNSyncedMemory(size_t size,
                                           int mlu_dev_id,
                                           int mlu_ddr_chn = -1)
```

Constructor to construct synchronized memory object.

#### Parameters

- [in] size: The size of the memory.
- [in] mlu\_dev\_id: MLU device ID that is incremented from 0.
- [in] mlu\_ddr\_chn: The MLU DDR channel that is greater than or equal to 0, and is less than 4. It specifies which piece of DDR channel the memory allocated on.

**Return** No return value.

### ~CNSyncedMemory

```
cnstream::CNSyncedMemory::~CNSyncedMemory()
```

Destructor to destruct synchronized memory object.

**Return** No return value.

### GetCpuData

```
const void *cnstream::CNSyncedMemory::GetCpuData()
```

Gets the CPU data.

#### Parameters

- No: return value.

**Return** Returns the CPU data pointer.

**Note** If the size is 0, nullptr is always returned.

### SetCpuData

```
void cnstream::CNSyncedMemory::SetCpuData(void *data)
```

Sets the CPU data.

#### Parameters

- [in] data: The data pointer on CPU.

**Return** Void.

### GetMluData

```
const void *cnstream::CNSyncedMemory::GetMluData()
```

Gets the MLU data.

**Return** Returns the MLU data pointer.

**Note** If the size is 0, nullptr is always returned.

### SetMluData

```
void cnstream::CNSyncedMemory::SetMluData(void *data)
```

Sets the MLU data.

#### Parameters

- [out] data: The data pointer on MLU.

**Return** No return value.

### SetMluDevContext

```
void cnstream::CNSyncedMemory::SetMluDevContext(int dev_id,  
                                                  int ddr_chn = -1)
```

Sets the MLU device context.

#### Parameters

- [in] dev\_id: The MLU device ID that is incremented from 0.
- [in] ddr\_chn: The MLU DDR channel ID that is greater than or equal to 0, and less than a. It specifies which piece of DDR channel the memory is allocated on.

**Return** No return value.

**Note** You need to call this API before all getters and setters.

### GetMluDevId

```
int cnstream::CNSyncedMemory::GetMluDevId() const
```

Gets the MLU device ID.

**Return** Returns the ID of the device that the MLU memory is allocated on.

### GetMluDdrChnId

```
int cnstream::CNSyncedMemory::GetMluDdrChnId() const
```

Gets the channel ID of the MLU DDR.

**Return** Returns the DDR channel ID that the MLU memory is allocated on.

### GetMutableCpuData

```
void *cnstream::CNSyncedMemory::GetMutableCpuData()
```

Gets the mutable CPU data.

**Return** Returns the CPU data pointer.

### GetMutableMluData

```
void *cnstream::CNSyncedMemory::GetMutableMluData()
```

Gets the mutable MLU data.

**Return** Returns the MLU data pointer.

### GetHead

```
SyncedHead cnstream::CNSyncedMemory::GetHead() const
```

Gets synchronization status.

**Return** Returns synchronization status .

**See** [SyncedHead](#).

### GetSize

```
size_t cnstream::CNSyncedMemory::GetSize() const
```

Gets data bytes.

**Return** Returns data bytes.

## 3.2 Profiler

### 3.2.1 Class ModuleProfiler

`class ModuleProfiler : private NonCopyable`

`ModuleProfiler` is a class of the performance statistics of a module. It contains multiple `cnstream::ProcessProfiler` instances to support multiple process profilings.

The trace events of each process will be recorded when `ProfilerConfig::enable_tracing` is true. Profiling

and tracing of customized process is supported. See `ModuleProfiler::RegisterProcessName` for details.

**Note** This class is thread safe.

#### API Reference

##### ModuleProfiler

```
cnstream::ModuleProfiler::ModuleProfiler(const ProfilerConfig &config,
                                         const std::string &module_name,
                                         PipelineTracer *tracer)
```

Constructs a `ModuleProfiler` object.

##### Parameters

- [in] `config`: The configuration of the profiler.
- [in] `module_name`: The name of the module.
- [in] `tracer`: The tracer for tracing events.

**Return** No return value.

##### RegisterProcessName

```
bool cnstream::ModuleProfiler::RegisterProcessName(const std::string &process_name)
```

Registers process named by `process_name` for this profiler.

##### Parameters

- [in] `process_name`: The process name is the unique identification of a function or a piece of code that needs to do profiling.

**Return** Returns true if the registration is successful. Returns false if the process name has been registered.

### RecordProcessStart

```
bool cnstream::ModuleProfiler::RecordProcessStart(const std::string &process_name,
                                                  const RecordKey &key)
```

Records the start of a process named `process_name`.

#### Parameters

- [in] `process_name`: The name of the process. It should be registered by `RegisterProcessName`.
- [in] `key`: The unique identifier of a `CNFrameInfo` instance.

**Return** Returns true if recording is successful. Returns false if the process named by `process_name` is not registered by `RegisterProcessName`.

**See** `cnstream::ModuleProfiler::RegisterProcessName`

**See** `cnstream::ModuleProfiler::RecordKey`

### RecordProcessEnd

```
bool cnstream::ModuleProfiler::RecordProcessEnd(const std::string &process_name,
                                                  const RecordKey &key)
```

Records the end of a process named `process_name`.

#### Parameters

- [in] `process_name`: The name of the process. It should be registered by `RegisterProcessName`.
- [in] `key`: The unique identifier of a `CNFrameInfo` instance.

**Return** Returns true if record successfully. Returns false if the process named by `process_name` has not been registered by `RegisterProcessName`.

**See** `cnstream::ModuleProfiler::RegisterProcessName`

**See** `cnstream::ModuleProfiler::RecordKey`

### OnStreamEos

```
void cnstream::ModuleProfiler::OnStreamEos(const std::string &stream_name)
```

Clears profiling data of the stream named by `stream_name`, as the end of the stream is reached.

#### Parameters

- [in] `stream_name`: The name of the stream, usually the `CNFrameInfo::stream_id`.

**Return** No return value.

### GetName

```
std::string cnstream::ModuleProfiler::GetName() const
```

Gets the name of the module.

**Return** Returns the name of the module.

### GetProfile

```
ModuleProfile cnstream::ModuleProfiler::GetProfile()
```

Gets profiling results of the module during the execution of the program.

**Return** Returns the profiling results.

```
ModuleProfile cnstream::ModuleProfiler::GetProfile(const ModuleTrace &trace)
```

Gets profiling results according to the trace data.

#### Parameters

- [in] trace: Gets profiling results according to the trace data.

**Return** Returns the profiling results.

### 3.2.2 Class PipelineProfiler

```
class PipelineProfiler: private NonCopyable
```

**PipelineProfiler** is responsible for the performance statistics of a pipeline. It contains multiple `cnstream::ModuleProfiler` instances to support multiple module profilings.

By default, it will perform profiling of two processes for all modules. They are named `kPROCESS_PROFILER_NAME` and `kINPUT_PROFILER_NAME`. The start of the first process is before `cnstream::Module::Process` being called, and the end is before `cnstream::Module::Transmit` being called. The time when data is pushed into the data queue of the module is the start of the second process and the end is when data starts to be processed by the module.

It also does profiling of the data processing process from entering to exiting the pipeline.

The start and end trace events of each process are recorded when the `config.enable_tracing` is true.

**Note** This class is thread safe.



## API Reference

### PipelineProfiler

```
cstream::PipelineProfiler::PipelineProfiler(const ProfilerConfig &config,  
                                             const std::string &pipeline_name,  
                                             const std::vector<std::shared_ptr<Module>>  
                                             &modules)
```

Constructs a `PipelineProfiler` object.

#### Parameters

- [in] `config`: The configuration of the profiler.
- [in] `pipeline_name`: The name of the pipeline.
- [in] `modules`: All modules of the pipeline named `pipeline_name`.

**Return** No return value.

### GetName

```
std::string cstream::PipelineProfiler::GetName() const
```

Gets the name of the pipeline.

**Return** Returns the name of the pipeline.

### GetConfig

```
ProfilerConfig cstream::PipelineProfiler::GetConfig() const
```

Gets profiler configuration.

**Return** Returns profiler configuration.

### GetTracer

```
PipelineTracer *cstream::PipelineProfiler::GetTracer() const
```

Gets tracer.

**Return** Returns the tracer of the pipeline.

**GetModuleProfiler**

```
ModuleProfiler *cnstream::PipelineProfiler::GetModuleProfiler(const std::string
                                                             &module_name) const
```

Gets the module profiler by the name of the module.

**Parameters**

- [in] module\_name: The name of the module.

**Return** Returns the module profiler.

**GetProfile**

```
PipelineProfile cnstream::PipelineProfiler::GetProfile()
```

Gets profiling results of the pipeline during the execution of the program.

**Return** Returns the profiling results.

```
PipelineProfile cnstream::PipelineProfiler::GetProfile(const Time &start,
                                                         const Time &end)
```

Gets profiling results between the start time and the end time.

**Parameters**

- [in] start: The start time.
- [in] end: The end time.

**Return** Returns the profiling results.

**GetProfileBefore**

```
PipelineProfile cnstream::PipelineProfiler::GetProfileBefore(const Time &end,
                                                             const Duration
                                                             &duration)
```

Gets profiling results during a specified period time.

**Parameters**

- [in] end: The end time.
- [in] duration: The duration in milliseconds. The start time is the end time minus duration.

**Return** Returns the profiling results.

### GetProfileAfter

```
PipelineProfile cnstream::PipelineProfiler::GetProfileAfter(const Time &start,
                                                         const Duration &duration)
```

Gets profiling results for a specified period time.

#### Parameters

- [in] start: The start time.
- [in] duration: The duration in milliseconds. The end time is the start time plus duration.

**Return** Returns the profiling results.

### RecordInput

```
void cnstream::PipelineProfiler::RecordInput(const RecordKey &key)
```

Records the time when the data enters the pipeline.

#### Parameters

- [in] key: The unique identifier of a *CNFrameInfo* instance.

**Return** No return value.

**See** cnstream::RecordKey

### RecordOutput

```
void cnstream::PipelineProfiler::RecordOutput(const RecordKey &key)
```

Records the time when the data exits the pipeline.

#### Parameters

- [in] key: The unique identifier of a *CNFrameInfo* instance.

**Return** No return value.

**See** cnstream::RecordKey

### OnStreamEos

```
void cnstream::PipelineProfiler::OnStreamEos(const std::string &stream_name)
```

Clears profiling data of the stream named by *stream\_name*, as the end of the stream is reached.

#### Parameters

- [in] stream\_name: The name of the stream, usually the *CNFrameInfo::stream\_id*.

**Return** No return value.

### 3.2.3 Class PipelineTracer

class PipelineTracer : private NonCopyable

PipelineTracer is a class for recording trace events of the pipeline.

#### API Reference

##### PipelineTracer

```
cnstream::PipelineTracer::PipelineTracer(size_t capacity = 100000)
```

Constructs a PipelineTracer object.

#### Parameters

- [in] capacity: The capacity to store trace events.

**Return** No return value.

##### ~PipelineTracer

```
cnstream::PipelineTracer::~PipelineTracer()
```

Destructs a PipelineTracer object.

**Return** No return value.

##### RecordEvent

```
void cnstream::PipelineTracer::RecordEvent(const TraceEvent &event)
```

Records a trace event using value reference semantics.

#### Parameters

- [in] event: The trace event.

**Return** No return value.

```
void cnstream::PipelineTracer::RecordEvent(TraceEvent &&event)
```

Records a trace event using move semantics.

#### Parameters

- [in] event: The trace event.

**Return** No return value.

### GetTrace

```
PipelineTrace cnstream::PipelineTracer::GetTrace(const Time &start,  
                                                const Time &end) const
```

Gets the trace data of the pipeline for a specified period of time.

#### Parameters

- [in] start: The start time.
- [in] end: The end time.

**Return** Returns the trace data of the pipeline.

### GetTraceBefore

```
PipelineTrace cnstream::PipelineTracer::GetTraceBefore(const Time &end,  
                                                       const Duration &duration)  
const
```

Gets the trace data of the pipeline for a specified period of time.

#### Parameters

- [in] end: The end time
- [in] duration: The duration in milliseconds. The start time is the end time minus duration.

**Return** Returns the trace data of the pipeline.

### GetTraceAfter

```
PipelineTrace cnstream::PipelineTracer::GetTraceAfter(const Time &start,  
                                                       const Duration &duration) const
```

Gets the trace data of the pipeline for a specified period of time.

#### Parameters

- [in] start: The start time.
- [in] duration: The duration in milliseconds. The end time is the start time plus duration.

**Return** Returns the trace data of the pipeline.

### 3.2.4 Class ProcessProfiler

`class ProcessProfiler : private NonCopyable`

`ProcessProfiler` is the profiler for a process. A process can be a function call or a piece of code.

**Note** This class is thread safe.

#### API Reference

##### ProcessProfiler

```
cnstream::ProcessProfiler::ProcessProfiler(const ProfilerConfig &config,
                                           const std::string &process_name,
                                           PipelineTracer *tracer)
```

Constructs a `ProcessProfiler` object.

##### Parameters

- [in] `config`: The configuration of the profiler.
- [in] `process_name`: The name of the process.
- [in] `tracer`: The tracer for tracing events.

**Return** No return value.

##### ~ProcessProfiler

```
cnstream::ProcessProfiler::~~ProcessProfiler()
```

Destructs a `ProcessProfiler` object.

**Return** No return value.

##### SetModuleName

```
ProcessProfiler &cnstream::ProcessProfiler::SetModuleName(const std::string
                                                         &module_name)
```

Sets the module name to identify which module this profiler belongs to. The module name takes effect when the trace level is `TraceEvent::MODULE`. The trace level can be set by `cnstream::ProcessProfiler::SetTraceLevel`.

##### Parameters

- [in] `module_name`: The name of the module.

**Return** Returns this profiler itself.

### SetTraceLevel

```
ProcessProfiler &cnstream::ProcessProfiler::SetTraceLevel(const TraceEvent::Level
                                                         &level)
```

Set the trace level for this profiler. Trace level identifies whether this profiler belongs to a module or a pipeline.

**Parameters**

- [in] level: Trace level.

**Return** Returns the [ProcessProfiler](#) object itself.

**See** [cnstream::TraceEvent::Level](#).

### RecordStart

```
void cnstream::ProcessProfiler::RecordStart(const RecordKey &key)
```

Records the start of the process.

**Parameters**

- [in] key: The unique identifier of a [CNFrameInfo](#) instance.

**Return** No return value.

**See** [cnstream::RecordKey](#).

### RecordEnd

```
void cnstream::ProcessProfiler::RecordEnd(const RecordKey &key)
```

Records the end of the process.

**Parameters**

- [in] key: The unique identifier of a [CNFrameInfo](#) instance.

**Return** No return value.

**See** [cnstream::RecordKey](#).

### GetName

```
std::string cnstream::ProcessProfiler::GetName() const
```

Gets the name of the process.

**Return** The name of the process.

### GetProfile

```
ProcessProfile cnstream::ProcessProfiler::GetProfile()
```

Gets profiling results of the process during the execution of the program.

**Return** Returns the profiling results.

```
ProcessProfile cnstream::ProcessProfiler::GetProfile(const ProcessTrace &trace) const
```

Gets profiling results according to the trace data.

#### Parameters

- [in] trace: The trace data of the process.

**Return** Returns the profiling results.

### OnStreamEos

```
void cnstream::ProcessProfiler::OnStreamEos(const std::string &stream_name)
```

Clears profiling data of the stream named by `stream_name`, as the end of the stream is reached.

#### Parameters

- [in] stream\_name: The name of the stream, usually the `CNFrameInfo::stream_id`.

**Return** No return value.

### 3.2.5 Class StreamProfiler

```
class StreamProfiler
```

`StreamProfiler` is responsible for the performance statistics of a certain processing process of a stream. It is used by `ProcessProfiler`.

**See** `cnstream::ProcessProfiler`.

#### Datatypes Reference

```
typedef Duration
```

```
typedef std::chrono::duration<double, std::milli> cnstream::Duration;
```

```
using cnstream::StreamProfiler::Duration = std::chrono::duration<double, std::milli>
```



## API Reference

### StreamProfiler

```
cnstream::StreamProfiler::StreamProfiler(const std::string &stream_name)
```

Constructs a [StreamProfiler](#) object.

#### Parameters

- [in] `stream_name`: The name of a stream.

**Return** No return value.

### AddLatency

```
StreamProfiler &cnstream::StreamProfiler::AddLatency(const Duration &latency)
```

Accumulates latency to total latency.

#### Parameters

- [in] `latency`: The latency to be added. The latency will be accumulated to total latency.

**Return** Returns a lvalue reference to the current instance.

### UpdatePhysicalTime

```
StreamProfiler &cnstream::StreamProfiler::UpdatePhysicalTime(const Duration &time)
```

Updates physical time that a stream costs.

#### Parameters

- [in] `time`: The physical time a stream costs.

**Return** Returns a lvalue reference to the current instance.

### AddDropped

```
StreamProfiler &cnstream::StreamProfiler::AddDropped(uint64_t dropped)
```

Accumulates dropped frame count.

#### Parameters

- [in] `dropped`: The dropped frame count.

**Return** Returns a lvalue reference to the current instance.

**AddCompleted**

```
StreamProfiler &cnstream::StreamProfiler::AddCompleted()
```

Accumulates completed frame count with 1.

**Return** Returns a lvalue reference to the current instance.

**GetName**

```
std::string cnstream::StreamProfiler::GetName() const
```

Gets the name of the stream.

**Return** Returns the name of the stream.

**GetProfile**

```
StreamProfile cnstream::StreamProfiler::GetProfile()
```

Gets the performance statistics for this stream.

**Return** Returns the performance statistics for this stream.

**3.2.6 Class TraceSerializeHelper**

```
class TraceSerializeHelper
```

Serializes trace data into JSON format. You can load JSON file by chrome-tracing to show the trace data.

**API Reference****DeserializeFromJSONStr**

```
bool cnstream::TraceSerializeHelper::DeserializeFromJSONStr(const      std::string
                                                                &jsonstr,
                                                                TraceSerializeHelper
                                                                *pout)
```

Deserializes a JSON string.

**Parameters**

- [in] jsonstr: The JSON string.
- [out] pout: The output pointer stores the results.

**Return** Returns true if the JSON string is deserialized successfully, otherwise returns false.

### DeserializeFromJSONFile

```
bool cnstream::TraceSerializeHelper::DeserializeFromJSONFile(const      std::string
                                                             &filename,
                                                             TraceSerializeHelper
                                                             *pout)
```

Deserializes a JSON file.

#### Parameters

- [in] jsonstr: The JSON file path.
- [out] pout: The output pointer stores the results.

**Return** Returns true if the JSON string is deserialized successfully, otherwise returns false.

### TraceSerializeHelper

```
cnstream::TraceSerializeHelper::TraceSerializeHelper()
```

Constructs a `TraceSerializeHelper` object.

**Return** No return value.

```
cnstream::TraceSerializeHelper::TraceSerializeHelper(const      TraceSerializeHelper
                                                             &other)
```

Constructs a `TraceSerializeHelper` object with the copy of the contents of another object.

#### Parameters

- [in] other: Another object used to initialize an object.

**Return** No return value.

```
cnstream::TraceSerializeHelper::TraceSerializeHelper(TraceSerializeHelper &&other)
```

Constructs a `TraceSerializeHelper` object with the contents of another object using move semantics.

#### Parameters

- [in] other: Another object used to initialize an object.

**Return** No return value.

**operator=**

```
TraceSerializeHelper &cnstream::TraceSerializeHelper::operator=(const
                                                                TraceSerializeHelper
                                                                &other)
```

Replaces the contents with a copy of the contents of another [TraceSerializeHelper](#) object.

**Parameters**

- [in] other: Another object used to initialize the current object.

**Return** Returns a lvalue reference to the current instance.

```
TraceSerializeHelper &cnstream::TraceSerializeHelper::operator=(TraceSerializeHelper
                                                                &&other)
```

Replaces the contents with those of another [TraceSerializeHelper](#) object using move semantics.

**Parameters**

- [in] other: Another object used to initialize the current object.

**Return** Returns a lvalue reference to the current instance.

**~TraceSerializeHelper**

```
cnstream::TraceSerializeHelper::~~TraceSerializeHelper() = default
```

Destructs a [TraceSerializeHelper](#) object by using default constructor.

**Return** No return value.

**Serialize**

```
void cnstream::TraceSerializeHelper::Serialize(const PipelineTrace &pipeline_trace)
```

Serializes trace data.

**Parameters**

- [in] pipeline\_trace: The trace data. Get it by `pipeline.GetTracer()->GetTrace()`.

**Return** No return value.

### Merge

```
void cnstream::TraceSerializeHelper::Merge(const TraceSerializeHelper &t)
```

Merges another trace serialization helper tool data.

#### Parameters

- [in] t: The trace serialization helper tool to be merged.

**Return** No return value.

### ToJsonStr

```
std::string cnstream::TraceSerializeHelper::ToJsonStr() const
```

Serializes to a JSON string.

**Return** Returns a JSON string.

### ToFile

```
bool cnstream::TraceSerializeHelper::ToFile(const std::string &filename) const
```

Serializes to a JSON file.

#### Parameters

- [in] filename: The JSON file name.

**Return** Returns true if the serialization is successful, otherwise returns false.

**Note** the possible reason of serialization failure is that writing to the file is not permitted.

### Reset

```
void cnstream::TraceSerializeHelper::Reset()
```

Resets serialization helper. Clears data and frees up memory.

**Return** No return value.

### 3.2.7 Class `TraceEvent`

`class TraceEvent`

`TraceEvent` is a class representing a trace event used by Profile.

#### Datatypes Reference

`enum Level`

`enum Level {`

`PIPELINE = 0,`

`MODULE = 1,`

`};`

`enum cnstream::TraceEvent::Level`

Enumeration variables describing the level of an event. The default level is 0 (pipeline's event).

Values:

`enumerator PIPELINE`

A event of a pipeline.

`enumerator MODULE`

An event of a module.

`enum Type`

`enum Type {`

`START = 1 << 0,`

`END = 1 << 1,`

`};`

`enum cnstream::TraceEvent::Type`

Enumeration variables describing the type of an event. The default type is 1 (START).

Values:

`enumerator START`

A process-start event.

`enumerator END`

A process-end event.

## API Reference

### TraceEvent

```
cnstream::TraceEvent::TraceEvent() = default
```

Constructs a [TraceEvent](#) object by using default constructor.

**Return** No return value.

```
cnstream::TraceEvent::TraceEvent(const RecordKey &key)
```

Constructs a [TraceEvent](#) object with a [RecordKey](#) instance.

#### Parameters

- [in] key: The unique identification of a frame.

**Return** No return value.

```
cnstream::TraceEvent::TraceEvent(RecordKey &&key)
```

Constructs a [TraceEvent](#) object with a [RecordKey](#) using move semantics.

#### Parameters

- [in] key: The unique identification of a frame.

**Return** No return value.

```
cnstream::TraceEvent::TraceEvent(const TraceEvent &other) = default
```

Constructs a [TraceEvent](#) object with the copy of the contents of another object.

#### Parameters

- [in] other: Another object used to initialize an object.

**Return** No return value.

```
cnstream::TraceEvent::TraceEvent(TraceEvent &&other)
```

Constructs a [TraceEvent](#) object with the contents of another object using move semantics.

#### Parameters

- [in] other: Another object used to initialize an object.

**Return** No return value.

**operator=**

```
TraceEvent &cnstream::TraceEvent::operator=(const TraceEvent &other) = default
```

Replaces the contents with a copy of the contents of another [TraceEvent](#) object.

**Parameters**

- [in] other: Another object used to initialize the current object.

**Return** Returns a lvalue reference to the current instance.

```
TraceEvent &cnstream::TraceEvent::operator=(TraceEvent &&other)
```

Replaces the contents with those of another [TraceEvent](#) object using move semantics.

**Parameters**

- [in] other: Another object used to initialize the current object.

**Return** Returns a lvalue reference to the current instance.

**SetKey**

```
TraceEvent &cnstream::TraceEvent::SetKey(const RecordKey &key)
```

Sets a unique identification for a frame.

**Parameters**

- [in] key: The unique identification of a frame.

**Return** Returns a lvalue reference to the current instance.

```
TraceEvent &cnstream::TraceEvent::SetKey(RecordKey &&key)
```

Sets a unique identification for a frame using move semantics.

**Parameters**

- [in] key: The unique identification of a frame.

**Return** Returns a lvalue reference to the current instance.

**SetModuleName**

```
TraceEvent &cnstream::TraceEvent::SetModuleName(const std::string &module_name)
```

Sets the name of a module.

**Parameters**

- [in] module\_name: The name of a module.

**Return** Returns a lvalue reference to the current instance.



```
TraceEvent &cnstream::TraceEvent::SetModuleName(std::string &&module_name)
```

Sets the name of a module using move semantics.

#### Parameters

- [in] module\_name: The name of a module.

**Return** Returns a lvalue reference to the current instance.

### SetProcessName

```
TraceEvent &cnstream::TraceEvent::SetProcessName(const std::string &process_name)
```

Sets the name of a process.

#### Parameters

- [in] process\_name: The name of a process.

**Return** Returns a lvalue reference to the current instance.

```
TraceEvent &cnstream::TraceEvent::SetProcessName(std::string &&process_name)
```

Sets the name of a process using move semantics.

#### Parameters

- [in] process\_name: The name of a process.

**Return** Returns a lvalue reference to the current instance.

### SetTime

```
TraceEvent &cnstream::TraceEvent::SetTime(const Time &time)
```

Sets the timestamp of this event.

#### Parameters

- [in] time: The timestamp of the event.

**Return** Returns a lvalue reference to the current instance.

```
TraceEvent &cnstream::TraceEvent::SetTime(Time &&time)
```

Sets the timestamp of this event using move semantics.

#### Parameters

- [in] time: The timestamp of the event.

**Return** Returns a lvalue reference to the current instance.

### SetLevel

```
TraceEvent &cnstream::TraceEvent::SetLevel(const Level &level)
```

Sets the level of this event.

#### Parameters

- [in] `level`: the level of the event.

**Return** Returns a lvalue reference to the current instance.

### SetType

```
TraceEvent &cnstream::TraceEvent::SetType(const Type &type)
```

Sets the type of this event.

#### Parameters

- [in] `type`: The type of the event.

**Return** Returns a lvalue reference to the current instance.

## 3.3 Video Analysis

### 3.3.1 Class CNDataFrame

```
class CNDataFrame : public NonCopyable
```

`CNDataFrame` is a class holding a data frame and the frame description.

#### API Reference

#### CNDataFrame

```
cnstream::CNDataFrame::CNDataFrame() = default
```

Constructs an object.

**Return** No return value.

**~CNDataFrame**

```
cnstream::CNDataFrame::~~CNDataFrame() = default
```

Destructs an object.

**Return** No return value.

**GetPlanes**

```
int cnstream::CNDataFrame::GetPlanes() const
```

Gets plane count for a specified frame.

**Return** Returns the plane count of this frame.

**GetPlaneBytes**

```
size_t cnstream::CNDataFrame::GetPlaneBytes(int plane_idx) const
```

Gets the number of bytes in a specified plane.

**Parameters**

- [in] plane\_idx: The index of the plane. The index increments from 0.

**Return** Returns the number of bytes in the plane.

**GetBytes**

```
size_t cnstream::CNDataFrame::GetBytes() const
```

Gets the number of bytes in a frame.

**Return** Returns the number of bytes in a frame.

**CopyToSyncMem**

```
void cnstream::CNDataFrame::CopyToSyncMem(void **ptr_src,
                                             bool dst_mlu)
```

Synchronizes the source data into ::CNSyncedMemory.

**Parameters**

- [in] ptr\_src: The source data's address. This API internally judges the address is MLU memory or not.
- [in] dst\_mlu: The flag shows whether synchronizes the data to MLU memory.

**Note** Sets the `width,height,fmt,ctx,stride,dst_device_id,deAllocator_` before calling this function. There are 5 situations:

- Reuse codec's buffer and do not copy anything. Just assign the `ptr_src` to `CNSyncedMemory mlu_ptr_`.
- This API allocates MLU buffer, and copy the source MLU data to the allocated buffer as the MLU destination.
- This API allocates MLU buffer, and copy the source CPU data to the allocated buffer as the MLU destination.
- This API allocates CPU buffer, and copy the source MLU data to the allocated buffer as the CPU destination.
- This API allocates CPU buffer, and copy the source CPU data to the allocated buffer as the CPU destination. Whatever which situation happens, `::CNSyncedMemory` doesn't own the buffer and it isn't responsible for releasing the data.

### ImageBGR

```
cv::Mat cnstream::CNDDataFrame::ImageBGR()
```

Converts data to the BGR format.

**Return** Returns data with OpenCV mat type.

**Note** This function is called after `CNDDataFrame::CopyToSyncMem()` is invoked.

### HasBGRImage

```
bool cnstream::CNDDataFrame::HasBGRImage()
```

Checks whether there is BGR image stored.

**Return** Returns true if has BGR image, otherwise returns false.

### CopyToSyncMemOnDevice

```
void cnstream::CNDDataFrame::CopyToSyncMemOnDevice(int device_id)
```

Synchronizes source data to specific device, and resets `ctx.dev_id` to `device_id` when synced, for multi-device case.

#### Parameters

- [in] `device_id`: The device id.

**Return** No return value.

### 3.3.2 Class CNInferObject

`class CNInferObject`

`CNInferObject` is a class holding the information of an object.

#### Variables

**id**

`std::string cnstream::CNInferObject::id`

The ID of the classification (label value).

**track\_id**

`std::string cnstream::CNInferObject::track_id`

The tracking result.

**score**

`float cnstream::CNInferObject::score`

The label score.

**bbox**

`CNInferBoundingBox cnstream::CNInferObject::bbox`

The object normalized coordinates.

**datas**

`std::unordered_map<int, any> cnstream::CNInferObject::datas`

(Deprecated) User-defined structured information.

**collection**

`Collection cnstream::CNInferObject::collection`

User-defined structured information.

## API Reference

### CNInferObject

```
cnstream::CNInferObject::CNInferObject() = default
```

Constructs an instance storing inference results.

**Return** No return value.

### ~CNInferObject

```
cnstream::CNInferObject::~~CNInferObject() = default
```

Constructs an instance.

**Return** No return value.

### AddAttribute

```
bool cnstream::CNInferObject::AddAttribute(const std::string &key,
                                           const CNInferAttr &value)
```

Adds the key of an attribute to a specified object.

#### Parameters

- [in] `key`: The Key of the attribute you want to add to. See [GetAttribute\(\)](#).
- [in] `value`: The value of the attribute.

**Return** Returns true if the attribute has been added successfully. Returns false if the attribute already existed.

**Note** This is a thread-safe function.

```
bool cnstream::CNInferObject::AddAttribute(const std::pair<std::string,
                                           CNInferAttr> &attribute)
```

Adds the key pairs of an attribute to a specified object.

#### Parameters

- [in] `attribute`: The attribute pair (key, value) to be added.

**Return** Returns true if the attribute has been added successfully. Returns false if the attribute has already existed.

**Note** This is a thread-safe function.

### GetAttribute

```
CNInferAttr cnstream::CNInferObject::GetAttribute(const std::string &key)
```

Gets an attribute by key.

#### Parameters

- [in] key: The key of an attribute you want to query. See [AddAttribute\(\)](#).

**Return** Returns the attribute key. If the attribute does not exist, `CNInferAttr::id` will be set to -1.

**Note** This is a thread-safe function.

### AddExtraAttribute

```
bool cnstream::CNInferObject::AddExtraAttribute(const std::string &key,  
                                                const std::string &value)
```

Adds the key of the extended attribute to a specified object.

#### Parameters

- [in] key: The key of an attribute. You can get this attribute by key. See [GetExtraAttribute\(\)](#).
- [in] value: The value of the attribute.

**Return** Returns true if the attribute has been added successfully. Returns false if the attribute has already existed in the object.

**Note** This is a thread-safe function.

### AddExtraAttributes

```
bool cnstream::CNInferObject::AddExtraAttributes(const std::vector<std::pair<std::string,  
                                                                    std::string>> &attributes)
```

Adds the key pairs of the extended attributes to a specified object.

#### Parameters

- [in] attributes: Attributes to be added.

**Return** Returns true if the attribute has been added successfully. Returns false if the attribute has already existed.

**Note** This is a thread-safe function.

### GetExtraAttribute

```
std::string cnstream::CNInferObject::GetExtraAttribute(const std::string &key)
```

Gets an extended attribute by key.

#### Parameters

- [in] key: The key of an identified attribute. See [AddExtraAttribute\(\)](#).

**Return** Returns the attribute that is identified by the key. If the attribute does not exist, returns NULL.

**Note** This is a thread-safe function.

### RemoveExtraAttribute

```
bool cnstream::CNInferObject::RemoveExtraAttribute(const std::string &key)
```

Removes an attribute by key.

#### Parameters

- [in] key: The key of an attribute you want to remove. See [AddAttribute](#).

**Return** Return true.

**Note** This is a thread-safe function.

### GetExtraAttributes

```
StringPairs cnstream::CNInferObject::GetExtraAttributes()
```

Gets all extended attributes of an object.

**Return** Returns all extended attributes.

**Note** This is a thread-safe function.

### AddFeature

```
bool cnstream::CNInferObject::AddFeature(const std::string &key,  
                                          const CNInferFeature &feature)
```

Adds the key of feature to a specified object.

#### Parameters

- [in] key: The Key of feature you want to add the feature to. See [GetFeature](#).
- [in] value: The value of the feature.

**Return** Returns true if the feature is added successfully. Returns false if the feature identified



by the key already exists.

**Note** This is a thread-safe function.

### GetFeature

```
CNInferFeature cnstream: :CNInferObject::GetFeature(const std::string &key)
```

Gets an feature by key.

#### Parameters

- [in] key: The key of an feature you want to query. See AddFeature.

**Return** Return the feature of the key. If the feature identified by the key is not exists, CNInferFeature will be empty.

**Note** This is a thread-safe function.

### GetFeatures

```
CNInferFeatures cnstream: :CNInferObject::GetFeatures()
```

Gets the features of an object.

**Return** Returns the features of an object.

**Note** This is a thread-safe function.

## 3.3.3 Class DataSource

```
class DataSource : public cnstream::SourceModule, public ModuleCreator<DataSource>
```

DataSource is a class to handle encoded input data.

**Note** It is always the first module in a pipeline.

### API Reference

#### DataSource

```
cnstream: :DataSource::DataSource(const std::string &moduleName)
```

Constructs a DataSource object.

#### Parameters

- [in] moduleName: The name of this module.

**Return** No return value.

### ~DataSource

```
cnstream::DataSource::~DataSource()
```

Destructs a [DataSource](#) object.

**Return** No return value.

### Open

```
bool cnstream::DataSource::Open(ModuleParamSet paramSet) override
```

Initializes the configuration of the [DataSource](#) module.

This function will be called by the pipeline when the pipeline starts.

#### Parameters

- [in] paramSet: The module's parameter set to configure a [DataSource](#) module.

**Return** Returns true if the parameter set is supported and valid, otherwise returns false.

### Close

```
void cnstream::DataSource::Close() override
```

Frees the resources that the object may have acquired.

This function will be called by the pipeline when the pipeline stops.

**Return** No return value.

### CheckParamSet

```
bool cnstream::DataSource::CheckParamSet(const ModuleParamSet &paramSet) const  
override
```

Checks the parameter set for the [DataSource](#) module.

#### Parameters

- [in] paramSet: Parameters for this module.

**Return** Returns true if all parameters are valid. Otherwise, returns false.

### GetSourceParam

```
DataSourceParam cnstream::DataSource::GetSourceParam() const
```

Gets the parameters of the [DataSource](#) module.

**Return** Returns the parameters of this module.

**Note** This function should be called after `Open` function.

### 3.3.4 Class ESJpegMemHandler

```
class ESJpegMemHandler : public cnstream::SourceHandler
```

[ESJpegMemHandler](#) is a class of source handler for Jpeg bitstreams in memory.

#### API Reference

##### Create

```
std::shared_ptr<SourceHandler> cnstream::ESJpegMemHandler::Create(DataSource
                                                                    *module,
                                                                    const std::string
                                                                    &stream_id,
                                                                    int max_width =
                                                                    7680,
                                                                    int max_height =
                                                                    4320)
```

Creates source handler.

#### Parameters

- [in] `module`: The data source module.
- [in] `stream_id`: The stream id of the stream.
- [in] `max_width`: The maximum width of the image.
- [in] `max_height`: The maximum height of the image.

**Return** Returns source handler if it is created successfully, otherwise returns `nullptr`.

### ~ESJpegMemHandler

```
cnstream::ESJpegMemHandler::~ESJpegMemHandler()
```

The destructor of `ESJpegMemHandler`.

**Return** No return value.

### Open

```
bool cnstream::ESJpegMemHandler::Open() override
```

Opens source handler.

**Return** Returns true if the source handler is opened successfully, otherwise returns false.

### Close

```
void cnstream::ESJpegMemHandler::Close() override
```

Closes source handler.

**Return** No return value.

### Write

```
int cnstream::ESJpegMemHandler::Write(ESPacket *pkt)
```

Sends data in frame mode.

#### Parameters

- [in] `pkt`: The data packet.

**Return** Returns 0 if the data is written successfully. Returns -1 if failed to write data. The possible reason is the handler is closed or the data is nullptr.

### 3.3.5 Class ESMemHandler

```
class ESMemHandler : public cnstream::SourceHandler
```

`ESMemHandler` is a class of source handler for H264/H265 bitstreams in memory (with prefix-start-code).

## Datatypes Reference

### enum DataType

```
enum DataType {
```

```
    INVALID = 0,
```

```
    H264 = 1,
```

```
    H265 = 2,
```

```
};
```

```
enum cnstream::ESMemHandler::DataType
```

Enumeration variables describing ES data type.

Values:

```
enumerator INVALID
```

Invalid data type.

```
enumerator H264
```

The data type is H264.

```
enumerator H265
```

The data type is H265.

## API Reference

### Create

```
std::shared_ptr<SourceHandler> cnstream::ESMemHandler::Create(DataSource *module,
                                                             const std::string
                                                             &stream_id,
                                                             const
                                                             MaximumVideoResolution
                                                             &maximum_resolution
                                                             = {})
```

Creates source handler.

### Parameters

- [in] module: The data source module.
- [in] stream\_id: The stream id of the stream.
- [in] maximum\_resolution: The maximum video resolution for variable video resolutions. See *MaximumVideoResolution* for detail.

**Return** Returns source handler if it is created successfully, otherwise returns nullptr.

#### **~ESMemHandler**

```
cnstream::ESMemHandler::~ESMemHandler()
```

The destructor of `ESMemHandler`.

**Return** No return value.

#### **Open**

```
bool cnstream::ESMemHandler::Open() override
```

Opens source handler.

**Return** Returns true if the source handler is opened successfully, otherwise returns false.

#### **Close**

```
void cnstream::ESMemHandler::Close() override
```

Closes source handler.

**Return** No return value.

#### **SetDataType**

```
int cnstream::ESMemHandler::SetDataType(DataType type)
```

Sets data type.

##### **Parameters**

- [in] type: The data type.

**Return** Returns 0 if data type is set successfully, otherwise returns -1.

**Note** This function must be called before `Write` function.

### Write

```
int cnstream::ESMemHandler::Write(ESPacket *pkt)
```

Sends data in frame mode.

#### Parameters

- [in] pkt: The data packet

**Return** Returns 0 if the data is written successfully. Returns -1 if failed to write data. The possible reasons are the handler is closed, the end of the stream is received, the data is nullptr and the data is invalid, so that the video infomations can not be parsed from it.

```
int cnstream::ESMemHandler::Write(unsigned char *buf,
                                   int len)
```

Sends data in chunk mode.

#### Parameters

- [in] buf: The data buffer
- [in] len: The length of the data

**Return** Returns 0 if the data is written successfully. Returns -1 if failed to write data. The possible reasons are the handler is closed, the end of the stream is received and the data is invalid, so that the video infomations can not be parsed from it.

### WriteEos

```
int cnstream::ESMemHandler::WriteEos()
```

Sends the end of the stream.

The data remains in the parser will be dropped. Call this function, when the data of a stream is not completely written and the stream needed to be removed.

**Return** Returns 0 if the end of the stream is written successfully. Returns -1 if failed to write data. The possible reason is the handler is closed.

### 3.3.6 Class FileHandler

```
class FileHandler : public cnstream::SourceHandler
```

`FileHandler` is a class of source handler for video with format mp4, flv, matroska and USB Camera ("/dev/videoxxx").

## API Reference

### Create

```
std::shared_ptr<SourceHandler> cnstream::FileHandler::Create(DataSource *module,
                                                            const      std::string
                                                            &stream_id,
                                                            const      std::string
                                                            &filename,
                                                            int framerate,
                                                            bool loop = false,
                                                            const
MaximumVideoResolution
&maximum_resolution =
{})
```

Creates source handler.

#### Parameters

- [in] module: The data source module.
- [in] stream\_id: The stream id of the stream.
- [in] filename: The filename of the stream.
- [in] framerate: Controls sending the frames of the stream with specific rate.
- [in] loop: Loops the stream.
- [in] maximum\_resolution: The maximum video resolution for variable video resolutions. See *MaximumVideoResolution* for detail.

**Return** Returns source handler if it is created successfully, otherwise returns nullptr.

### ~FileHandler

```
cnstream::FileHandler::~FileHandler()
```

The destructor of *FileHandler*.

**Return** No return value.



### Open

```
bool cnstream::FileHandler::Open() override
```

Opens source handler.

**Return** Returns true if the source handler is opened successfully, otherwise returns false.

### Close

```
void cnstream::FileHandler::Close() override
```

Closes source handler.

**Return** No return value

## 3.3.7 Class RawImgMemHandler

```
class RawImgMemHandler : public cnstream::SourceHandler
```

[RawImgMemHandler](#) is a class of source handler for raw image data in memory.

**Note** This handler will not send data to MLU decoder as the raw data has been decoded.

### API Reference

#### Create

```
std::shared_ptr<SourceHandler> cnstream::RawImgMemHandler::Create(DataSource
                                                                    *module,
                                                                    const std::string
                                                                    &stream_id)
```

Creates source handler.

#### Parameters

- [in] module: The data source module.
- [in] stream\_id: The stream id of the stream.

**Return** Returns source handler if it is created successfully, otherwise returns nullptr.

### ~RawImgMemHandler

```
cnstream::RawImgMemHandler::~RawImgMemHandler()
```

The destructor of `RawImgMemHandler`.

**Return** No return value.

### Open

```
bool cnstream::RawImgMemHandler::Open() override
```

Opens source handler.

**Return** Returns true if the source handler is opened successfully, otherwise returns false.

### Close

```
void cnstream::RawImgMemHandler::Close() override
```

Closes source handler.

**Return** No return value.

### Write

```
int cnstream::RawImgMemHandler::Write(const cv::Mat *mat_data,  
                                       const uint64_t pts)
```

Sends raw image with `cv::Mat`. Only BGR data with 8UC3 type is supported, and data is continuous.

#### Parameters

- [in] `mat_data`: The bgr24 format image data.
- [in] `pts`: The pts for `mat_data`, should be different for each image.

**Return** Returns 0 if the data is written successfully. Returns -1 if failed to write data. The possible reason is the end of the stream is received or failed to process the data. Returns -2 if the data is invalid.

**Note** Sends nullptr after all data are sent.

```
int cnstream::RawImgMemHandler::Write(const uint8_t *data,
                                     const int size,
                                     const uint64_t pts,
                                     const int width = 0,
                                     const int height = 0,
                                     const CNDataFormat pixel_fmt =
                                     CNDataFormat::CN_INVALID)
```

Sends raw image with image data and image information, support formats: bgr24, rgb24, nv21 and nv12.

#### Parameters

- [in] data: The data of the image, which is a continuous buffer.
- [in] size: The size of the data.
- [in] pts: The pts for raw image, should be different for each image.
- [in] width: The width of the image.
- [in] height: The height of the image.
- [in] pixel\_fmt: The pixel format of the image. These formats are supported, bgr24, rgb24, nv21 and nv12.

**Return** Returns 0 if the data is written successfully. Returns -1 if failed to write data. The possible reason is the end of the stream is received or failed to process the data. Returns -2 if the data is invalid.

**Note** Sends nullptr as data and passes 0 as size after all data are sent.

### 3.3.8 Class RtspHandler

```
class RtspHandler : public cnstream::SourceHandler
```

**RtspHandler** is a class of source handler for rtsp stream.

#### API Reference

##### Create

```
std::shared_ptr<SourceHandler> cnstream::RtspHandler::Create(DataSource *module,
                                                            const      std::string
                                                            &stream_id,
                                                            const      std::string
                                                            &url_name,
                                                            bool use_ffmpeg = false,
                                                            int reconnect = 10,
                                                            const
MaximumVideoResolution
&maximum_resolution =
                                                            {})
```

Creates source handler.

#### Parameters

- [in] module: The data source module.
- [in] stream\_id: The stream ID of the stream.
- [in] url\_name: The url of the stream.
- [in] use\_ffmpeg: Uses ffmpeg demuxer if it is true, otherwise uses live555 demuxer.
- [in] reconnect: It is valid when "use\_ffmpeg" set false.
- [in] maximum\_resolution: The maximum video resolution for variable video resolutions. See *MaximumVideoResolution* for detail.

**Return** Returns source handler if it is created successfully, otherwise returns nullptr.

#### ~RtspHandler

```
cnstream::RtspHandler::~~RtspHandler()
```

The destructor of *RtspHandler*.

**Return** No return value.

#### Open

```
bool cnstream::RtspHandler::Open() override
```

Opens source handler.

**Return** Returns true if the source handler is opened successfully, otherwise returns false.

**Close**

```
void cnstream::RtspHandler::Close() override
```

Closes source handler.

**Return** No return value.

**3.3.9 Class ObjPostproc**

```
class ObjPostproc : public virtual ReflexObjectEx<ObjPostproc>
```

`ObjPostproc` is the base class of object post processing.

**API Reference****~ObjPostproc**

```
cnstream::ObjPostproc::~~ObjPostproc() = 0
```

Destructs an object.

**Return** No return value.

**Create**

```
ObjPostproc *cnstream::ObjPostproc::Create(const std::string &proc_name)
```

Creates a postprocess object with the given postprocess's class name.

**Parameters**

- [in] `proc_name`: The postprocess class name.

**Return** The pointer to postprocess object.

**SetThreshold**

```
void cnstream::ObjPostproc::SetThreshold(const float threshold)
```

Sets threshold.

**Parameters**

- [in] `threshold`: The value between 0 and 1.

**Return** No return value.

**Execute**

```
int cnstream::ObjPostproc::Execute(const std::vector<float*> &net_outputs,
                                   const std::shared_ptr<edk::ModelLoader> &model,
                                   const CNFrameInfoPtr &finfo,
                                   const std::shared_ptr<CNInferObject> &pobj)
```

Executes post processing on neural network outputs.

**Parameters**

- [in] net\_outputs: Neural network outputs, and the data is stored on the host.
- [in] model: Model information including input shape and output shape.
- [inout] finfo: Smart pointer of *CNFrameInfo* to store processed data.
- [in] pobj: The deduced object information.

**Return** Returns 0 if successful, otherwise returns -1.

**Note**

- This function is called by the Inferencer module when the parameter mem\_on\_mlu\_for\_postproc is set to false and obj\_infer is set to true. See the Inferencer parameter description for details.

```
int cnstream::ObjPostproc::Execute(const std::vector<void*> &net_outputs,
                                   const std::shared_ptr<edk::ModelLoader> &model,
                                   const std::vector<std::pair<CNFrameInfoPtr,
                                   std::shared_ptr<CNInferObject>>> &obj_infos)
```

Execute post processing on neural network outputs.

**Parameters**

- [in] net\_outputs: Neural network outputs, and the data is stored on the MLU.
- [in] model: Model information including input shape and output shape.
- [inout] obj\_infos: The batched frames's result of postprocessing.

**Return** Returns 0 if successful, otherwise returns -1.

**Note**

- This function is called by the Inferencer module when the parameter mem\_on\_mlu\_for\_postproc is set to true and obj\_infer is set to true. See the Inferencer parameter description for details.

### 3.3.10 Class Postproc

`class Postproc : public virtual ReflexObjectEx<Postproc>`

`Postproc` is the base class of post process.

#### API Reference

##### ~Postproc

```
cnstream::Postproc::~~Postproc() = 0
```

Destructs an object.

**Return** No return value.

##### Create

```
Postproc *cnstream::Postproc::Create(const std::string &proc_name)
```

Creates a postprocess object with the given postprocess's class name.

##### Parameters

- [in] `proc_name`: The postprocess class name.

**Return** The pointer to postprocess object.

##### SetThreshold

```
void cnstream::Postproc::SetThreshold(const float threshold)
```

Sets threshold.

##### Parameters

- [in] `threshold`: The value between 0 and 1.

**Return** No return value.

##### Execute

```
int cnstream::Postproc::Execute(const std::vector<float*> &net_outputs,
                                const std::shared_ptr<edk::ModelLoader> &model,
                                const CNFrameInfoPtr &package)
```

Executes postproc on neural network outputs.

##### Parameters

- [in] net\_outputs: Neural network outputs, and the data is stored on the host.
- [in] model: Model information including input shape and output shape.
- [inout] package: Smart pointer of *CNFrameInfo* to store processed data.

**Return** Returns 0 if successful, otherwise returns -1.

**Note**

- This function is called by the Inferencer module when the parameter mem\_on\_mlu\_for\_postproc is set to false and obj\_infer is set to false. See the Inferencer parameter description for details.

```
int cnstream::Postproc::Execute(const std::vector<void*> &net_outputs,
                               const std::shared_ptr<edk::ModelLoader> &model,
                               const std::vector<CNFrameInfoPtr> &packages)
```

Execute post processing on neural network outputs.

**Parameters**

- [in] net\_outputs: Neural network outputs, and the data is stored on the MLU.
- [in] model: Model information including input shape and output shape.
- [inout] packages: The batched frames's result of postprocessing.

**Return** Returns 0 if successful, otherwise returns -1.

**Note**

- This function is called by the Inferencer module when the parameter mem\_on\_mlu\_for\_postproc is set to true and obj\_infer is set to false. See the Inferencer parameter description for details.

### 3.3.11 Class ObjPreproc

```
class ObjPreproc : public virtual ReflexObjectEx<ObjPreproc>
```

*ObjPreproc* is the base class of preprocess for object.

**API Reference**

**~ObjPreproc**

```
cnstream::ObjPreproc::~ObjPreproc()
```

Destructs an object.

**Return** No return value.



### Create

```
ObjPreproc *cnstream::ObjPreproc::Create(const std::string &proc_name)
```

Creates a preprocess object with the given preprocess's class name.

#### Parameters

- [in] `proc_name`: The preprocess class name.

**Return** Returns the pointer to preprocess object.

### Execute

```
int cnstream::ObjPreproc::Execute(const std::vector<float*> &net_inputs,
                                   const std::shared_ptr<edk::ModelLoader> &model,
                                   const CNFrameInfoPtr &finfo,
                                   const std::shared_ptr<CNInferObject> &pobj) = 0
```

Executes preprocess on neural network inputs.

#### Parameters

- [out] `net_inputs`: Neural network inputs.
- [in] `model`: Model information including input shape and output shape.
- [in] `finfo`: Smart pointer of *CNFrameInfo* which stores origin data.
- [in] `obj`: The deduced object information.

**Return** Returns 0 if successful, otherwise returns -1.

### 3.3.12 Class Preproc

```
class Preproc : public virtual ReflexObjectEx<Preproc>
```

*Preproc* is the base class of neural network preprocessing for inference module.

#### API Reference

##### ~Preproc

```
cnstream::Preproc::~Preproc()
```

Destructs an object.

**Return** No return value.

**Create**

```
Preproc *cnstream::Preproc::Create(const std::string &proc_name)
```

Creates a preprocess object with the given preprocess's class name.

**Parameters**

- [in] `proc_name`: The preprocess class name.

**Return** Returns the pointer to preprocess object.

**Execute**

```
int cnstream::Preproc::Execute(const std::vector<float*> &net_inputs,
                               const std::shared_ptr<edk::ModelLoader> &model,
                               const CNFrameInfoPtr &package) = 0
```

Executes preprocess on neural network inputs.

**Parameters**

- [out] `net_inputs`: Neural network inputs.
- [in] `model`: Model information including input shape and output shape.
- [in] `package`: Smart pointer of *CNFrameInfo* which stores origin data.

**Return** Returns 0 if successful, otherwise returns -1.

**3.3.13 Class VideoPostproc**

```
class VideoPostproc : public virtual ReflexObjectEx<VideoPostproc>
```

*VideoPostproc* is the base class of post processing classes for Inference2.

**API Reference****~VideoPostproc**

```
cnstream::VideoPostproc::~VideoPostproc() = 0
```

Destructs an object.

**Return** No return value.

### Create

```
VideoPostproc *cnstream::VideoPostproc::Create(const std::string &proc_name)
```

Creates a postprocess object with the given postprocess's class name.

#### Parameters

- [in] `proc_name`: The postprocess class name.

**Return** Returns the pointer to postprocess object.

### SetThreshold

```
void cnstream::VideoPostproc::SetThreshold(const float threshold)
```

Sets threshold.

#### Parameters

- [in] `threshold`: The value between 0 and 1.

**Return** No return value.

### Execute

```
bool cnstream::VideoPostproc::Execute(infer_server::InferData *output_data,  
                                       const infer_server::ModelIO &model_output,  
                                       const infer_server::ModelInfo &model_info) = 0
```

Executes postprocessing on the model's output data.

#### Parameters

- [out] `output_data`: The postprocessing result. The result of postprocessing should be set to it. You could set any type of data to this parameter and get it in `UserProcess` function.
- [in] `model_output`: The neural network origin output data.
- [in] `model_info`: The model information, such as input/output number and shape.

**Return** Returns true if successful, otherwise returns false.

**Note** This function is executed by infer server postproc processor. You could override it to develop custom postprocessing. To set any type of data to `output_data`, use this statement, e.g., `int example_var = 1; output_data->Set(example_var);`

### 3.3.14 Class VideoPreproc

`class VideoPreproc : public virtual ReflexObjectEx<VideoPreproc>`

`VideoPreproc` is the base class of video preprocessing.

#### API Reference

##### ~VideoPreproc

```
cnstream::VideoPreproc::~VideoPreproc()
```

Destructs an object.

**Return** No return value.

##### Create

```
VideoPreproc *cnstream::VideoPreproc::Create(const std::string &proc_name)
```

Creates a preprocess object with the given preprocess's class name.

##### Parameters

- [in] `proc_name`: The preprocess class name.

**Return** The pointer to preprocess object.

##### SetModelInputPixelFormat

```
void cnstream::VideoPreproc::SetModelInputPixelFormat(infer_server::video::PixelFormat  
fmt)
```

Sets model input pixel format.

##### Parameters

- [in] `fmt`: The model input pixel format.

**Return** No return value.

### Execute

```
bool cnstream::VideoPreproc::Execute(infer_server::ModelIO *model_input,  
                                     const infer_server::InferData &input_data,  
                                     const infer_server::ModelInfo &model_info) = 0
```

Executes preprocessing on the origin data.

#### Parameters

- [out] `model_input`: The input of neural network.
- [in]  
    `input_data`: The raw input data. The user could get `infer_server::video::VideoFrame` object from it.
- [in] `model_info`: The model information, e.g., input/output number, shape and etc.

**Note** The `input_data` holds `infer_server::video::VideoFrame` object. Use the statement to get video frame: `const infer_server::video::VideoFrame& frame = input_data.GetLref<infer_server::video::VideoFrame>();`. After preprocessing, you should set the result to `model_output`. For example, the model only has one input, then you should copy the result to `model_input->buffers[0].MutableData()` which is a void pointer.

**Return** Returns true if successful, otherwise returns false.



## 4 API Reference

### 4.1 Framework Function

#### 4.1.1 GetMaxModuleNumber

```
uint32_t cnstream::GetMaxModuleNumber()
```

Gets the number of modules that a pipeline is able to hold.

**Return** The maximum modules of a pipeline can own.

#### 4.1.2 GetMaxStreamNumber

```
uint32_t cnstream::GetMaxStreamNumber()
```

Gets the number of streams that a pipeline can hold, regardless of the limitation of hardware resources.

**Return** Returns the value of MAX\_STREAM\_NUM.

**Note** The factual stream number that a pipeline can process is always subject to hardware resources, no more than MAX\_STREAM\_NUM.

#### 4.1.3 GetPathRelativeToTheJSONFile

```
std::string cnstream::GetPathRelativeToTheJSONFile(const std::string &path,
                                                    const ModuleParamSet &param_set)
```

Gets the complete path of a file.

If the path you set is an absolute path, returns the absolute path. If the path you set is a relative path, returns the path that appends the relative path to the specified JSON file path.

**Parameters**

- [in] path: The path relative to the JSON file or an absolute path.
- [in] param\_set: The module parameters. The JSON file path is one of the parameters.

**Return** Returns the complete path of a file.

### 4.1.4 VersionString

```
const char *cnstream::VersionString()
```

Gets the CNStream version string.

**Return** Returns the version string formatted as "v%major.%minor.%patch". e.g. "v3.5.1".

### 4.1.5 MajorVersion

```
const int cnstream::MajorVersion()
```

Gets the CNStream major version.

**Return** Returns the major version, [0, MAXINT].

### 4.1.6 MinorVersion

```
const int cnstream::MinorVersion()
```

Gets the CNStream minor version.

**Return** Returns the minor version, [0, MAXINT].

### 4.1.7 PatchVersion

```
const int cnstream::PatchVersion()
```

Gets the CNStream patch version.

**Return** Returns the patch version, [0, MAXINT].

## 4.2 Video Analysis Function

### 4.2.1 CNGetPlanes

```
int cnstream::CNGetPlanes(CNDataFormat fmt)
```

Gets image plane number by a specified image format.

**Parameters**

- [in] `fmt`: The format of the image.

**Return Value**

- 0: Unsupported image format.

- >0: Image plane number.

### 4.2.2 GetCNDataFramePtr

`CNDataFramePtr` `cnstream::GetCNDataFramePtr(std::shared_ptr<CNFrameInfo> frameInfo)`

This helper will be deprecated in the future versions. Uses *Collection::Get*<CNDataFramePtr>(kCNDataFrameTag) instead.

### 4.2.3 GetCNInferObjsPtr

`CNInferObjsPtr` `cnstream::GetCNInferObjsPtr(std::shared_ptr<CNFrameInfo> frameInfo)`

This helper will be deprecated in the future versions. Uses *Collection::Get*<CNInferObjsPtr>(kCNInferObjsTag) instead.

### 4.2.4 GetCNInferDataPtr

`CNInferDataPtr` `cnstream::GetCNInferDataPtr(std::shared_ptr<CNFrameInfo> frameInfo)`

This helper will be deprecated in the future versions. Uses *Collection::Get*<CNInferDataPtr>(kCNInferDataTag) instead.





## 5 Release Notes

This release notes outlines CNStream API updates and documentation updates in CNStream Developer Guide.

### 5.1 CNStream Release Version 6.1.0

#### 5.1.1 API Updates

This section lists API functions and fields that were added, changed, or removed.

- None.

### 5.2 CNStream Release Version 6.0.0

#### 5.2.1 API Updates

This section lists API functions and fields that were added, changed, or removed.

- Changes on the Frame and FrameVa frameworks are as follows:
  - Removed ICNMediaImageMapper class.
  - Removed CNDataframe::user\_data\_.
  - Changed CNDataframe::ImageBGR return value from cv::Mat\* to cv::Mat.
  - Changed CNDataframe::CopyToSyncMem parameters from (bool) to (void\*\*, bool).
  - Added CNDataframe::collection.
- Changes on framework are as follows:
  - Removed Module::SetParentID.
  - Removed Module::SetParentId.
  - Removed Module::GetModuleMask.
  - Removed Module::SetParentID.
  - Removed Module::SetParentId.
  - Removed Module::GetModuleMask.
  - Removed SpinLock class.

- Added Collection class.
- Removed CnstreamError class.
- Added the contents of the following missing head files:
  - \* cnstream\_allocator.hpp
  - \* cnstream\_common.hpp
  - \* cnstream\_config.hpp
  - \* cnstream\_source.hpp
  - \* cnstream\_version.hpp
  - \* postproc.hpp
  - \* preproc.hpp
  - \* video\_postproc.hpp
  - \* video\_preproc.hpp

## 5.3 CNStream Release 2021-01-25 (Version 5.3.0)

### 5.3.1 API Updates

This section lists API functions and fields that were added, changed, or removed.

- Changes on the Frame and FrameVa frameworks are as follows:
  - Add Parameter `CN_FRAME_FLAG_REMOVED` to `CNFrameFlag` enum for identifying the stream to which the frame belongs is removed.
  - Changed the struct `CNFrameInfo` to a class and privately inherits from class `NonCopyable`.
  - Added the new `payload` parameter to the `Create` API, the default value of which is `nullptr`.
  - Added the new `IsRemoved` API for checking whether the stream to which the frame belongs is removed.
  - Changed the struct `CNDataFrame` to a class and privately inherits from class `NonCopyable`.
  - Added the new `dst_mlu` parameter to the `CopyToSyncMem` API, the default value of which is `true`.
  - Added the new struct `CNInferObjs` for holding objects inference result.
  - Added the new struct `InferData` contains the inputs, the outputs and the information of inference.
  - Added the new struct `CNInferData` for holding all `InferData` of one frame.
  - Added the new `GetCNDataFramePtr` API for getting the `CNDataFramePtr` object of one frame.
  - Added the new `GetCNInferObjsPtr` API for getting the `CNInferObjsPtr` object of one frame.
  - Added the new `GetCNInferDataPtr` API for getting the `CNInferDataPtr` object of one frame.
- Changes on the Module framework are as follows:
  - Added the new virtual `OnEos` API to notify the module that the EOS is arrived.
  - Added the new `GetContainer` API to get the container of the module.
  - Added the new `GetProfiler` API to get the profiler of the module.

- Removed the `RecordTime` API due to the `PerfManager` has been replaced to `Profiler`.
- Removed the `GetPerfManager` API due to the `PerfManager` has been replaced to `Profiler`.
- Changes on the Pipeline framework are as follows:
  - Added the new `GetName` API to get the name of the pipeline.
  - Added the new `profiler_config` parameter to the `BuildPipeline` API, the default value of which is a `ProfilerConfig` object created by `ProfilerConfig` constructor.
  - The following APIs are removed due to the `PerfManager` has been replaced by `Profiler`:
    - \* The `CreatePerfManager` API.
    - \* The `RemovePerfManager` API.
    - \* The `AddPerfManager` API.
    - \* The `PerfSqlCommitLoop` API.
    - \* The `CalculatePerfStats` API.
    - \* The `CalculateModulePerfStats` API.
    - \* The `CalculatePipelinePerfStats` API.
    - \* The `GetPerfManagers` API.
  - Added the new `IsProfilingEnabled` API to check if profiling function is enabled.
  - Added the new `IsTracingEnabled` API to check if tracing function is enabled.
  - Added the new `GetProfiler` API to get the profiler.
  - Added the new `GetTracer` API to get the tracer.
  - Added the new `IsRootNode` API to check if the module is the root node of the pipeline.
  - Added the new `IsLeafNode` API to check if the module is the leaf node of the pipeline.
- Supported the `Profiler` with the related APIs.
- Replaced the `PerfManager` and `PerfCalculator` by `Profiler`.
- Changes on the `SyncMem` are as follows:
  - Removed the `CNStreamMallocHost` API.
  - Removed the `CNSyncedMemory` constructor.
  - Set the parameter `mlu_ddr_chn` with default value `-1` of the `CNSyncedMemory` constructor.
  - Changed the default value of parameter `mlu_ddr_chn` of the `SetMluDevContext` API, from `0` to `-1`.
  - Removed the `SetMluCpuData` API which is used on `MLU220_SOC` platform.
- Supported the `Inferencer2` module with the related APIs.
- Changes on the `DataSource` module are as follows:
  - Changes on the `RawImgMemHandler` class are as follows:
    - \* Removed the `Write` API with one parameter `cv::Mat* mat_data`.
    - \* Removed the `Write` API with five parameters `unsigned char *data, int size, int width = 0, int height = 0, CNDataFormat pixel_fmt = CN_INVALID`.
    - \* Changed the parameters from `cv::Mat* mat_data, uint64_t pts` to `const cv::Mat* mat_data, const uint64_t pts` of the `Write` API.
    - \* Changed the parameters from `unsigned char *data, int size, uint64_t pts, int width = 0, int height = 0, CNDataFormat pixel_fmt`

= CN\_INVALID to const uint8\_t \*data, const int size, const uint64\_t pts, const int width = 0, const int height = 0, const CNDataFormat pixel\_fmt = CN\_INVALID of the Write API.

- Removed the UsbHandler class.

## 5.4 CNStream Release 2020-09-18 (Version 5.2.0)

### 5.4.1 API Updates

This section lists API functions and fields that were added, changed, or removed.

- Changes on the FrameVa are as follows:
  - Added the new HasBGRImage API for checking whether data frame is converted to BGR format and saved to CV format.
  - Added the new RemoveExtraAttribute API for removing an attribute by key.
  - Added the new GetExtraAttributes API for retrieving all extended attributes of an object.
  - Added the new GetFeature API for retrieving the feature of an object by key.
  - Added the new key parameter to the AddFeature API.
  - Renamed the AddExtraAttribute to AddExtraAttributes.
  - Changed the return type of the AddFeature API from void to bool.
  - Changed the return type of the GetFeatures API from ThreadSafeVector<CNInferFeature> to CNInferFeatures.
  - Added the new CNInferFeatures type.
  - Added the new StringPairs type.
  - Changed the struct CNInferFeature to vector<float> type.
  - Changed the type of variable datas in struct CNInferObject from ThreadSafeUnorderedMap<int, any> to std::unordered\_map<int, any>.
- Changes on the Frame framework are as follows:
  - Changed the type of variable datas in struct CNFrameInfo from ThreadSafeUnorderedMap<int, any> to std::unordered\_map<int, any>.
- Changes on the Pipeline framework are as follows:
  - Added the new GetEndModule API for retrieving the end module of a pipeline.
- Changes on the PerfCalculator are as follows:
  - Added the new total\_time variable in struct PerfStats.
- Changes on the PerfManager are as follows:
  - Added the new CreateDir API for creating directory.

## 5.5 CNStream Release 2020-07-10 (Version 5.0.0)

### 5.5.1 API Updates

This section lists API functions and fields that were added, changed, or removed.

- Changes on the DataSource module are as follows:
  - The following new data types are supported:
    - \* Added the new ESPacket struct.
    - \* Added the new FileHandler class.
    - \* Added the new RtspHandler class.
    - \* Added the new ESMemHandler class.
  - Parameter changes in DataSourceParam struct.
  - The following data type and API are removed due to function changes:
    - \* The SourceType enum.
    - \* The CreateSource API.
- Changes on the EventBus framework are as follows:
  - The `cnstream_eventbus.hpp` file is moved from the `modules/core/include` directory to the `framework/core/include` directory.
  - Added the new Start and Stop APIs to support starting and stopping an event bus thread.
  - Parameter changes in Event struct.
  - Removed the `module` parameter from the BusWatcher API.
  - Removed the `watch_module` parameter from the AddBusWatch API.
  - Removed EventType enum due to function changes.
- Changes on the Frame framework are as follows:
  - The `cnstream_frame.hpp` file is moved from the `modules/core/include` directory to the `framework/core/include` directory.
  - Added the new IsEos API to check if this is an eos frame.
  - Added the new SetStreamIndex API to support setting stream index.
  - Parameter changes in CNFrameInfo struct.
  - The following enums, structs, classes, and APIs are moved from the `cnstream_frame.hpp` file to the `cnstream_frame_va.hpp` file:
    - \* The CNDataFormat enum.
    - \* The DevContext struct.
    - \* The MemMapType enum.
    - \* The CNGetPlanes API.
    - \* The IDataDeallocator class.
    - \* The ICNMediaImageMapper class.
    - \* The CNDataFrame struct.

- \* The CNInferBoundingBox struct.
- \* The CNInferAttr struct.
- \* The CNInferFeature struct.
- \* The CNInferObject struct.
- Added the new `stream_id` parameter to the `MmapSharedMem`, `CopyToSharedMem`, and `ReleaseSharedMem` APIs.
- Parameter changed in `CNDataFrame` struct.
- Changed `CNInferFeature` from a type to struct.
- Changed return value type of `GetFeatures` API.
- Changes on the Module framework are as follows:
  - The `cnstream_module.hpp` file is moved from the `modules/core/include` directory to the `framework/core/include` directory.
  - Added the new `IModuleObserver` class to support observing modules.
  - Added the new `SetObserver`, `ParseByJSONStr`, `ParseByJSONFile`, and `ConfigsFromJsonFile` APIs.
  - Removed `SetPerfManagers` and `ClearPerfManagers` APIs due to function changes.
  - The following enums, structs, classes, and APIs are moved from the `cnstream_module.hpp` file to the `cnstream_config.hpp` file:
    - \* The `ParamRegister` class.
    - \* The `ParametersChecker` class.
    - \* The `ModuleParamSet` struct.
    - \* The `GetPathRelativeToTheJSONFile` API.
    - \* The `Register` API.
    - \* The `GetParams` API.
    - \* The `IsRegistered` API.
    - \* The `SetModuleDesc` API.
- Changes on the Pipeline framework are as follows:
  - The `cnstream_pipeline.hpp` file is moved from the `modules/core/include` directory to the `framework/core/include` directory.
  - Added the new `IdxManager` class to support managing stream index.
  - Added the new `final_print` parameter to the `CalculateModulePerfStats` and `CalculatePipelinePerfStats` APIs.
  - Parameters are changed in `StreamMsg` struct.
  - Removed the following APIs due to the function changes:
    - \* The `Open` API.
    - \* The `Close` API.
    - \* The `Process` API.
    - \* The `GetLinkIds` API.
    - \* The `GetModuleParallelism` API.

- \* The `NotifyStreamMsg` API.
- Moved the `CNModuleConfig` struct from the `cnstream_pipeline.hpp` file to the `cnstream_config.hpp` file.
- Changes on the `PerfManager` are as follows:
  - The `perf_manager.hpp` file is moved from the `modules/core/include` directory to the `framework/core/include` directory.
  - The following new APIs are supported:
    - \* Added the new `GetSql` API to support getting SQL handler.
    - \* Added the new `GetKeys` API to support generating keys.
    - \* Added the new `GetEndTimeSuffix` API to support getting the end time suffix.
    - \* Added the new `GetStartTimeSuffix` API to support getting the start time suffix.
    - \* Added the new `GetPrimaryKey` API to support getting the default primary key.
    - \* Added the new `GetDefaultType` API to support getting the default perf type.
  - Removed the following data types and APIs due to the function changes:
    - \* The `PerfInfo` struct.
    - \* The `Init` API that contains the `db_name`, `module_names`, `start_node` and `end_nodes` parameters.
    - \* The `RegisterPerfType` API that contains the `type` parameter.
    - \* The `CalculatePipelinePerfStats` API.
    - \* The `GetCalculator` API.
    - \* The `SetModuleNames` API.
    - \* The `SetStartNode` API.
    - \* The `SetEndNodes` API.
    - \* All `CreatePerfCalculator` APIs.
    - \* All `CalculatePerfStats` APIs.
    - \* All `CalculateThroughput` APIs.
- Changes on the `PerfCalculator` are as follows:
  - The `perf_calculator.hpp` file is moved from the `modules/core/include` directory to the `framework/core/include` directory.
  - The following new data types, classes, APIs are supported:
    - \* The `PerfCalculatorForModule`, `PerfCalculatorForPipeline`, and `PerfCalculatorForInfer` classes, which inherits from `PerfCalculator` class.
    - \* The `PerfCalculationMethod` class.
    - \* The `PerfUtils` class.
    - \* The `PrintStreamId` API to print stream id.
    - \* The `PrintStr` API to print string.
    - \* The `PrintTitle` API to print title.
    - \* The `PrintTitleForLatestThroughput` API to print title for latest throughput.
    - \* The `PrintTitleForAverageThroughput` API to print title for average throughput.

- \* The `PrintTitleForTotal` API to print 'total'.
- \* The `SetPerfUtils` API to set the `PerfUtils` for getting data from database.
- \* The `GetPerfUtils` API to get the `PerfUtils`.
- \* The `CalcAvgThroughput` API to calculate average throughput.
- \* The `GetAvgThroughput` API to get average throughput.
- \* The `CalculateFinalThroughput` API to calculate final throughput.
- \* The virtual `CalcLatency` API to calculate latency.
- \* The virtual `CalcThroughput` API to calculate throughput.
- \* The `SetPrintThroughput` API to set whether print throughput inside perf calculator.
- Added the new width parameter to the `PrintLatency` API.
- Added the new width parameter to the `PrintThroughput` API.
- Added the new `sql_name` and `perf_type` parameters to the `GetLatency` API.
- Added the new `sql_name` and `perf_type` parameters to the `GetThroughput` API.
- Parameter changes in `PerfStats` struct.
- Changed the return type of the `GetThroughput` API from `PerfStats` to `std::vector<PerfStats>`.
- The following APIs are removed due to function changes:
  - \* The `PrintPerfStats` API.
  - \* The `CalcLatency` API.
  - \* The `CalcThroughputByTotalTime` API.
  - \* The `CalcThroughputByEachFrameTime` API.
  - \* The `SearchFromDatabase` API.

## 5.6 CNStream Release 2020-05-25 (Version 4.5.0)

### 5.6.1 API Updates

This section lists API functions and fields that were added, changed, or removed.

- The following API is supported in the Frame framework:
  - Added the new `CopyToSyncMemOnDevice` API to synchronize source data to a specified device.
- The following APIs are supported in the Module framework:
  - Added the new `ClearPerfManagers` API to clear all performance managers.
- Supported the `RtspSink` module with the related APIs.



## 5.7 Release 2020-04-16 (Version 4.4.0)

### 5.7.1 API Updates

This section lists API functions and fields that were added, changed, or removed.

- The following APIs are supported in Frame framework for multi-process function:
  - Added the new `MmapSharedMem` API to map shared memory.
  - Added the new `UnMapSharedMem` API to unmap shared memory.
  - Added the new `CopyToSharedMem` API to copy source-data to shared memory.
  - Added the new `ReleaseSharedMem` API to release shared memory.
- The following APIs are supported in Module framework for the performance measurement function:
  - Added the new `SetPerfManagers` API to set PerfManagers.
  - Added the new `GetPerfManager` API to retrieve PerfManager by stream id.
  - Added the new `ClearPerfManagers` API to clear PerfManagers.
- The following APIs are supported in Pipeline framework for the performance measurement function:
  - Added the new `CreatePerfManager` API to create PerfManager for each stream to measure performance of modules and pipeline.
  - Added the new `PerfSqlCommitLoop` API to commit sqlite events to increase the speed of inserting data to the database.
  - Added the new `CalculatePerfStats` API to calculate performance of modules and pipeline, and print performance statistics.
  - Added the new `CalculateModulePerfStats` API to calculate performance of modules, and print performance statistics.
  - Added the new `CalculatePipelinePerfStats` API to calculate performance of pipeline, and print performance statistics.
  - Removed the `PrintPerformanceInformation` API due to function changes.

### 5.7.2 Doc Updates

This section lists the documentation updates that were made in this version:

- Optimized the description of the APIs.
- Added the missing description of APIs and data types.

## 5.8 Release 2020-02-24

### 5.8.1 API Updates

This section lists API functions and fields that were added, changed, or removed.

- The following APIs are supported in Frame framework:
  - Supported the new virtual `GetMediaImage` API.
  - Supported the new virtual `GetPitch` API.
  - Supported the new virtual `GetCpuAddress` API.
  - Supported the new virtual `GetDevAddress` API.
  - Supported the new virtual `~ICNMediaImageMapper` API.
  - Parameter changes in `DevContext` struct.
- The following APIs are supported in SyncMem:
  - Supported the new `SetMluCpuData` API to set the CPU and MLU data for MLU220SOC only.
  - Supported the new `mlu_data` and `cpu_data` parameters to the `SetMluCpuData` API.

## 5.9 Release 2019-12-31

### 5.9.1 API Updates

This section lists API functions and fields that were added, changed, or removed.

- The following APIs are supported in Module framework:
  - Supported the new `IsRegistered` API for checking if a module parameter is registered or not.
  - Supported the new `SetModuleDesc` API for setting module description.
  - Supported the new `GetModuleDesc` API for getting module description.
  - Supported the new `CheckParamSet` API for checking ParamSet in a module.
  - Supported the new `GetRegistered` API for getting all registered modules name.
  - Supported the new `CheckPath` API for checking path of a configuration file.
  - Supported the new `IsNum` API for checking if a parameter is a number.
- The following APIs are supported in Inferencer module:
  - Supported the new `CheckParamSet` API for checking ParamSet in Inferencer module.
- The following APIs are supported in DataSource module:
  - Supported the new `CheckParamSet` API for checking ParamSet in DataSource module.
- The following APIs are supported in Tracker module:
  - Supported the new `CheckParamSet` API for checking ParamSet in Tracker module.