



CNStream Developer Guide

Release 2020-05-28 (Version 4.5.0)

May 27, 2020



Table of Contents

Table of Contents	i
1 Copyright	1
2 CNStream Datatypes Reference	3
2.1 Data Source	3
2.1.1 DataSource	3
2.1.2 DataSourceParam	4
2.1.3 DecoderType	6
2.1.4 OutputType	6
2.1.5 SourceType	7
2.2 EventBus	7
2.2.1 Event	7
2.2.2 EventBus	8
2.2.3 EventHandleFlag	8
2.2.4 EventType	8
2.3 Frame	9
2.3.1 CNDataFormat	9
2.3.2 CNDataFrame	10
2.3.3 CNFrameFlag	12
2.3.4 CNFrameInfo	13
2.3.5 CNInferAttr	14
2.3.6 CNInferBoundingBox	14
2.3.7 CNInferFeature	14
2.3.8 CNInferObject	15
2.3.9 DevContext	17
2.3.10 ICNMediaImageMapper	17
2.3.11 IDataDeallocator	18
2.3.12 MemMapType	18
2.4 Inferencer	18
2.4.1 CNFrameInfoPtr	18
2.4.2 Inferencer	19
2.5 Module	19
2.5.1 Module	19
2.5.2 ModuleCreator	19
2.5.3 ModuleCreatorWorker	20
2.5.4 ModuleEx	20
2.5.5 ModuleFactory	20

2.5.6	ModuleParamSet	20
2.5.7	ParametersChecker	20
2.5.8	ParamRegister	20
2.6	Performance Calculator	21
2.6.1	PerfCalculator	21
2.6.2	PerfStats	21
2.7	Performance Manager	21
2.7.1	PerfInfo	21
2.7.2	PerfManager	22
2.8	Pipeline	22
2.8.1	CNModuleConfig	22
2.8.2	LinkStatus	24
2.8.3	Pipeline	24
2.8.4	StreamMsg	24
2.8.5	StreamMsgObserver	25
2.8.6	StreamMsgType	25
2.9	RTSP Sink	26
2.9.1	ColorFormat	26
2.9.2	RtspParam	27
2.9.3	RtspSink	28
2.10	SyncMem	28
2.10.1	CNSyncedMemory	28
2.10.2	SyncedHead	29
2.11	Tracker	29
2.11.1	Tracker	29
3	CNStream API Reference	30
3.1	Data Source	30
3.1.1	CheckParamSet	30
3.1.2	Close	30
3.1.3	CreateSource	30
3.1.4	GetSourceParam	31
3.1.5	Open	31
3.2	Eventbus	32
3.2.1	AddBusWatch	32
3.2.2	BusWatcher	32
3.2.3	ClearAllWatchers	32
3.2.4	GetBusWatchers	32
3.2.5	IsRunning	32
3.2.6	PollEvent	33
3.2.7	PostEvent	33
3.3	Frame	33
3.3.1	AddAttribute	33
3.3.2	AddAttribute	33
3.3.3	AddExtraAttribute	34
3.3.4	AddExtraAttribute	34
3.3.5	AddFeature	34

3.3.6	CNGetPlanes	34
3.3.7	CopyToSharedMem	35
3.3.8	CopyToSyncMem	35
3.3.9	CopyToSyncMemOnDevice	35
3.3.10	Create	35
3.3.11	GetAttribute	36
3.3.12	GetBytes	36
3.3.13	GetExtraAttribute	36
3.3.14	GetFeatures	36
3.3.15	GetMediaImage	36
3.3.16	GetPlanes	37
3.3.17	GetPlaneBytes	37
3.3.18	ImageBGR	37
3.3.19	~ICNMediaImageMapper	37
3.3.20	MmapSharedMem	37
3.3.21	ReleaseSharedMem	38
3.3.22	UnMapSharedMem	38
3.4	Inferencer	38
3.4.1	CheckParamSet	38
3.4.2	Close	38
3.4.3	Open	38
3.4.4	Process	39
3.5	Module	40
3.5.1	CheckParamSet	40
3.5.2	CheckPath	40
3.5.3	Close	40
3.5.4	Create	40
3.5.5	Create	41
3.5.6	CreateObject	41
3.5.7	ClearPerfManagers	41
3.5.8	DoProcess	41
3.5.9	GetModuleDesc	42
3.5.10	GetName	42
3.5.11	GetOutputFrame	42
3.5.12	GetParams	42
3.5.13	GetPathRelativeToTheJSONFile	42
3.5.14	GetPerfManager	43
3.5.15	GetRegistered	43
3.5.16	HasTransmit	43
3.5.17	Instance	43
3.5.18	IsNum	43
3.5.19	IsRegistered	44
3.5.20	Open	44
3.5.21	PostEvent	44
3.5.22	Process	44
3.5.23	Regist	45
3.5.24	Register	45

3.5.25	SetContainer	45
3.5.26	SetModuleDesc	45
3.5.27	SetPerfManagers	46
3.5.28	ShowPerfInfo	46
3.5.29	ShowPerfInfo	46
3.5.30	TransmitData	46
3.6	Performance Calculator	46
3.6.1	CalcLatency	46
3.6.2	CalcThroughputByTotalTime	47
3.6.3	PrintLatency	47
3.6.4	PrintPerfStats	47
3.6.5	PrintThroughput	48
3.7	Performance Manager	48
3.7.1	CalculatePerfStats	48
3.7.2	CalculatePipelinePerfStats	48
3.7.3	Init	48
3.7.4	Record	49
3.7.5	RegisterPerfType	49
3.7.6	SqlBeginTrans	49
3.7.7	SqlCommitTrans	49
3.7.8	Stop	49
3.8	Pipelines	50
3.8.1	AddModule	50
3.8.2	AddModuleConfig	50
3.8.3	BuildPipeline	50
3.8.4	BuildPipelineByJSONFile	50
3.8.5	CalculateModulePerfStats	51
3.8.6	CalculatePerfStats	51
3.8.7	CalculatePipelinePerfStats	51
3.8.8	Close	51
3.8.9	CreatePerfManager	51
3.8.10	GetEventBus	52
3.8.11	GetLinkIds	52
3.8.12	GetModule	52
3.8.13	GetModuleConfig	52
3.8.14	GetModuleParallelism	52
3.8.15	GetModuleParamSet	53
3.8.16	GetStreamMsgObserver	53
3.8.17	IsRunning	53
3.8.18	LinkModules	53
3.8.19	NotifyStreamMsg	54
3.8.20	Open	54
3.8.21	ParseByJSONFile	54
3.8.22	ParseByJSONStr	54
3.8.23	PerfSqlCommitLoop	54
3.8.24	Process	54
3.8.25	ProvideData	55

3.8.26	QueryLinkStatus	55
3.8.27	RegistIPCFrameDoneCallBack	55
3.8.28	SetModuleAttribute	55
3.8.29	SetStreamMsgObserver	56
3.8.30	Start	56
3.8.31	Stop	56
3.9	RTSP Sink	56
3.9.1	CheckParamSet	56
3.9.2	Close	57
3.9.3	Open	57
3.9.4	Process	57
3.10	Syncmem	57
3.10.1	CNStreamFreeHost	57
3.10.2	CNStreamMallocHost	57
3.10.3	CNSyncedMemory	58
3.10.4	CNSyncedMemory	58
3.10.5	GetCpuData	58
3.10.6	GetHead	58
3.10.7	GetMluData	58
3.10.8	GetMluDdrChnId	59
3.10.9	GetMluDevId	59
3.10.10	GetMutableCpuData	59
3.10.11	GetMutableMluData	59
3.10.12	GetSize	59
3.10.13	SetCpuData	59
3.10.14	SetMluCpuData	60
3.10.15	SetMluData	60
3.10.16	SetMluDevContext	60
3.10.17	ToCpu	60
3.10.18	ToMlu	60
3.11	Tracker	61
3.11.1	CheckParamSet	61
3.11.2	Close	61
3.11.3	Open	61
3.11.4	Process	61
4	Release Notes	62
4.1	CNStream Release 2020-05-25 (Version 4.5.0)	62
4.1.1	API Updates	62
4.2	Release 2020-04-16 (Version 4.4.0)	62
4.2.1	API Updates	62
4.2.2	Doc Updates	63
4.3	Release 2020-02-24	63
4.3.1	API Updates	63
4.4	Release 2019-12-31	63
4.4.1	API Updates	63



1 Copyright

The Information in this guide and all other information contained in Cambricon Documentation Referenced in this guide is provided “AS IS.” Cambricon Makes no Warranties, Expressed, Implied, Statutory, or otherwise with respect to the information and expressly disclaims all implied warranties of noninfringement merchantability, title, noninfringement of intellectual property or fitness for a particular purpose. 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.

IN no event shall Cambricon be liable for any damages whatsoever (Including, without limitation, damages for loss of profits, business interruption, 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.

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 are measured using specific chip systems and/or components. The results reflect the approximate performance of Cambricon products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Cambricon makes no representation or warranty that the product described in this guide will be suitable for any specified use without further testing or modification. Testing of all parameters of each product is not necessarily performed by Cambricon. It is customer’s sole responsibility to ensure the product is suitable and fit for the application planned by 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 the Cambricon product and may result in additional or different conditions and/ or requirements beyond those contained in this guide. Cambricon does not accept any liability related to any default, damage, costs or problem 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.

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. Except as expressly provided herein, Cambricon and its suppliers do not grant any express or implied right to you under any patents, copyrights, trademarks, trade secret or any other intellectual property or proprietary right. Other than the right for customer to use the information in this guide with the product, no other license, either expressed or implied, is hereby granted by Cambricon under this guide.

1. COPYRIGHT

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

Copyright © 2020 Cambricon Corporation. All rights reserved.



2 CNStream Datatypes Reference

CNStream data types support both on MLU270 and MLU220.

2.1 Data Source

2.1.1 DataSource

class cnstream::DataSource

```
class cnstream::DataSource : public SourceModule, public cnstream::ModuleCreator<DataSource>
```

The class for handling input data.

Public Functions

```
DataSource(const std::string &moduleName)
```

Constructs [DataSource](#) object with a given module name.

Parameters

- `moduleName[in]`: A defined module name.

```
~DataSource()
```

Deconstructs [DataSource](#) object.

```
bool Open(ModuleParamSet paramSet) override
```

Called by pipeline when the pipeline is started.

Return Returns true if `paramSet` are supported and valid. Otherwise, returns false.

Parameters

- `paramSet`:
 - `source_type`: Optional. The demuxer type. The default value is `SOURCE_RAW`. Supported values are `raw` and `ffmpeg`.
 - `output_type`: Optional. The output type. The default `output_type` is `cpu`. Supported values are `mlu` and `cpu`.
 - `interval`: Optional. Process one frame for every `interval` frames. Process every frame by default.
 - `decoder_type`: Optional. The decoder type. The default `decoder_type` is `cpu`. Supported values are `mlu` and `cpu`.
 - `reuse_cndec_buf`: Optional. Whether the codec buffer will be reused. The default value is `false`. This parameter is used when decoder type is `mlu`. Supported values are `true` and `false`.

- `device_id`: Required when MLU is used. Device id. Set the value to -1 for CPU. Set the value for MLU in the range 0 - N.
- `chunk_size`: Required when `source_type` is set to `raw`. The size of the input data chunk.
- `width`: Required when `source_type` is set to `raw`. The width of the frame.
- `height`: Required when `source_type` is set to `raw`. The height of the frame.
- `interlaced`: Required when `source_type` is set to `raw`. Interlaced mode.
- `input_buf_number`: Optional. The input buffer number. The default value is 2.
- `output_buf_number`: Optional. The output buffer number. The default value is 3.

`void Close()` *override*

Called by pipeline when the pipeline is stopped.

`bool CheckParamSet(const ModuleParamSet ¶mSet) const` *override*

Checks parameters for a module.

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

Parameters

- `paramSet`: Parameters for this module.

`std::shared_ptr<cnstream::SourceHandler> CreateSource(const std::string &stream_id, const std::string &filename, int framerate, bool loop = false)`

Adds one stream to `DataSource` module. This function should be called after the pipeline has been started.

Return Returns the source handler instance.

Parameters

- `stream_id[in]`: The unique stream identifier.
- `filename[in]`: The source path that supports local-file-path, rtsp-url, jpg-sequences, and so on.
- `framerate[in]`: The input frequency of the source data.
- `loop[in]`: Whether to reload source when EOF is reached.

`DataSourceParam GetSourceParam() const`

Gets module parameters. This function should be called after `Open()` has been invoked.

2.1.2 DataSourceParam

`struct DataSourceParam {`

`SourceType source_type_ = SOURCE_RAW;`

`OutputType output_type_ = OUTPUT_CPU;`

`size_t interval_ = 1;`

`DecoderType decoder_type_ = DECODER_CPU;`

`bool reuse_cndec_buf = false;`

`int device_id_ = -1;`

```

size_t chunk_size_ = 0;
size_t width_ = 0;
size_t height_ = 0;
bool interlaced_ = false;
size_t output_w = 0;
size_t output_h = 0;
uint32_t input_buf_number_ = 2;
uint32_t output_buf_number_ = 3;
};

```

```

struct cnstream::DataSourceParam
    A structure for private usage.

```

Public Members

SourceType source_type_ = SOURCE_RAW

The demuxer type. The SOURCE_RAW value is set for debugging.

OutputType output_type_ = OUTPUT_CPU

Outputs data to CPU or MLU.

size_t interval_ = 1

Outputs image for every interval frames.

DecoderType decoder_type_ = DECODER_CPU

The decoder type.

bool reuse_cndec_buf = false

Valid when DECODER_MLU is used.

int device_id_ = -1

The MLU device ID. To disable MLU, set the value to -1.

size_t chunk_size_ = 0

Valid when SOURCE_RAW is used. For H264 and H265 only.

size_t width_ = 0

Valid when SOURCE_RAW is used. For H264 and H265 only.

size_t height_ = 0

Valid when SOURCE_RAW is used. For H264 and H265 only.

bool interlaced_ = false

Valid when SOURCE_RAW is used. For H264 and H265 only.

size_t output_w = 0

Invalid for MLU200 series.

size_t output_h = 0

Invalid for MLU200 series.

```
uint32_t input_buf_number_ = 2
    Valid when decoder_type is set to DECODER_MLU.

uint32_t output_buf_number_ = 3
    Valid when decoder_type is set to DECODER_MLU.
```

2.1.3 DecoderType

```
enum DecoderType {
    DECODER_CPU,
    DECODER_MLU
};
```

`enum cnstream::DecoderType`
The decoder type used in the source module.

Values:

```
enumerator DECODER_CPU
    CPU decoder with FFmpeg.

enumerator DECODER_MLU
    MLU decoder with CNCodec.
```

2.1.4 OutputType

```
enum OutputType {
    OUTPUT_CPU,
    OUTPUT_MLU
};
```

`enum cnstream::OutputType`
The storage type of the output frame data that are stored for modules on CPU or MLU.

Values:

```
enumerator OUTPUT_CPU
    Outputs to CPU.

enumerator OUTPUT_MLU
    Outputs to MLU.
```

2.1.5 SourceType

```
enum SourceType {
```

```
    SOURCE_RAW,
```

```
    SOURCE_FFmpeg
```

```
};
```

```
enum cnstream::SourceType
```

The type of stream or source to be processed.

Values:

```
enumerator SOURCE_RAW
```

Represents the raw stream. The source is sent for decoding directly.

```
enumerator SOURCE_FFmpeg
```

Represents the normal stream. The source is demuxed with FFmpeg before sent for decoding.

2.2 EventBus

2.2.1 Event

```
struct Event {
```

```
    EventType type;
```

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

```
    const Module *module;
```

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

```
};
```

```
struct cnstream::Event
```

The structure holding the event information.

Public Members

[EventType type](#)

The event type.

[std::string message](#)

Additional event messages.

[const Module *module](#)

The module that posts this event.

[std::thread::id thread_id](#)

The thread ID from which the event is posted.

2.2.2 EventBus

class cnstream::EventBus

```
class EventBus
```

The event bus that transmits events from modules to a pipeline.

2.2.3 EventHandleFlag

enum EventHandleFlag {

```
    EVENT_HANDLE_NULL,  
    EVENT_HANDLE_INTERCEPTION,  
    EVENT_HANDLE_SYNCED,  
    EVENT_HANDLE_STOP
```

```
};
```

```
enum cnstream::EventHandleFlag
```

Flags to specify the way in which bus watcher handled one event.

Values:

```
enumerator EVENT_HANDLE_NULL
```

Event is not handled.

```
enumerator EVENT_HANDLE_INTERCEPTION
```

Watcher is informed and intercepts the event.

```
enumerator EVENT_HANDLE_SYNCED
```

Watcher is informed and informs other watchers.

```
enumerator EVENT_HANDLE_STOP
```

Stops a poll event.

2.2.4 EventType

enum EventType {

```
    EVENT_INVALID,  
    EVENT_ERROR,  
    EVENT_WARNING,  
    EVENT_EOS,  
    EVENT_STOP,  
    EVENT_TYPE_END
```

```
};
```

```
enum cnstream::EventType
    Flags to specify how bus watchers handle a single 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 (End of Stream) event.

    enumerator EVENT_STOP
        Stops an event that is called by an application.

    enumerator EVENT_TYPE_END
        Reserved for your custom events.
```

2.3 Frame

2.3.1 CNDataFormat

```
enum CNDataFormat {
    CN_INVALID = -1,
    CN_PIXEL_FORMAT_YUV420_NV21 = 0,
    CN_PIXEL_FORMAT_YUV420_NV12,
    CN_PIXEL_FORMAT_BGR24,
    CN_PIXEL_FORMAT_RGB24
};

enum cnstream::CNDataFormat
    An enumerated type that is used to identify the pixel format of the data in CNDataFrame.

    Values:

    enumerator CN_INVALID = -1
        This frame is invalid.

    enumerator CN_PIXEL_FORMAT_YUV420_NV21 = 0
        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.
```

2.3.2 CNDataFrame

```
typedef struct {
    std::string stream_id;
    size_t flags = 0;
    int64_t frame_id;
    int64_t timestamp;
} CNDataFrame;
```

```
struct cnstream::CNDataFrame
    The structure holding a data frame and the frame description.
```

Public Functions

```
int GetPlanes() const
    Gets plane count for a specified frame.

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

```
size_t GetPlaneBytes(int plane_idx) const
    Gets the number of bytes in a specified plane.

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

Parameters

- `plane_idx`: The index of the plane. The index increments from 0.

```
size_t GetBytes() const
    Gets the number of bytes in a frame.

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

```
void CopyToSyncMem()
    Synchronizes the source-data to CNSyncedMemory.
```

```
void CopyToSyncMemOnDevice(int device_id)
    Synchronizes source data to specific device, and reset ctx.dev_id to device_id when
    synced, for multi-device case.
```

Return Void.

Parameters

- `device_id`: The device id.

```
void MmapSharedMem(MemMapType type)
    Maps shared memory for multi-process.
```

Return Void.

Parameters

- `memory`: The type of the mapped or shared memory.

`void UnMapSharedMem(MemMapType type)`

Unmaps the shared memory for multi-process.

Return Void.

Parameters

- `memory`: The type of the mapped or shared memory.

`void CopyToSharedMem(MemMapType type)`

Copies source-data to shared memory for multi-process.

Return Void.

Parameters

- `memory`: The type of the mapped or shared memory.

`void ReleaseSharedMem(MemMapType type)`

Releases shared memory for multi-process.

Return Void.

Parameters

- `memory`: The type of the mapped or shared memory.

`cv::Mat *ImageBGR()`

Converts data from RGB to BGR. This API should be called after `CopyToSyncMem()` is invoked.

If the data is not in RGB format but in BGR, YUV420NV12, or YUV420NV21 format, its color mode will not be converted.

Return Returns data with opencv mat type.

Public Members

`std::string stream_id`

The data stream aliases where this frame is located to.

`size_t flags = 0`

The mask for this frame, `CNFrameFlag`.

`int64_t frame_id = -1`

The frame index that incremented from 0.

`int64_t timestamp = -1`

The time stamp of this frame.

`CNDataFormat fmt`

The format of the frame.

The source data information. You need to set the information before calling `CopyToSyncMem()`.

`int width`

The width of the frame.

`int height`

The height of the frame.

```
int stride[CN_MAX_PLANES]
```

The strides of the frame.

```
DevContext ctx
```

The device context of this frame.

```
void *ptr_mlu[CN_MAX_PLANES]
```

The MLU data addresses for planes.

```
void *ptr_cpu[CN_MAX_PLANES]
```

The CPU data addresses for planes.

```
void *mlu_mem_handle = nullptr
```

The MLU memory handle for MLU data.

```
std::shared_ptr<IDataDeallocator> deAllocator_ = nullptr
```

The dedicated deallocator for CNDecoder Buffer.

```
std::shared_ptr<CNMediaImageMapper> mapper_ = nullptr
```

The dedicated Mapper for M220 CNDecoder.

```
void *cpu_data = nullptr
```

CPU data pointer. You need to allocate it by calling `CNStreamMallocHost()`.

```
void *mlu_data = nullptr
```

A pointer to the MLU data.

```
std::shared_ptr<CNSyncedMemory> data[CN_MAX_PLANES]
```

Synchronizes data helper.

2.3.3 CNFrameFlag

```
enum CNFrameFlag {
```

```
    CN_FRAME_FLAG_EOS = 1 << 0
```

```
};
```

```
enum cnstream::CNFrameFlag
```

An enumerated type that specifies the mask of `CNDataFrame`.

Values:

```
enumerator CN_FRAME_FLAG_EOS = 1 << 0
```

Identifies the end of data stream.

2.3.4 CNFrameInfo

typedef struct {

```
    static std::shared_ptr<CNFrameInfo> Create(const std::string& stream_id, bool eos
    = false);
```

```
    uint32_t channel_idx = INVALID_STREAM_IDX;
```

```
    CNDataFrame frame;
```

```
    std::vector<std::shared_ptr<CNInferObject>> objs;
```

```
    ~CNFrameInfo();
```

```
private:
```

```
    CNFrameInfo() {}
```

```
    DISABLE_COPY_AND_ASSIGN(CNFrameInfo);
```

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

```
    static std::map<std::string, int> stream_count_map_;
```

```
public:
```

```
    static int parallelism_;
```

```
} CNFrameInfo;
```

```
struct cnstream::CNFrameInfo
```

A structure holding the information of a frame.

Public Members

```
uint32_t channel_idx = INVALID_STREAM_IDX
```

The index of the channel, stream_index.

```
CNDataFrame frame
```

The data of the frame.

```
ThreadSafeVector<std::shared_ptr<CNInferObject>> objs
```

Structured information of the objects for this frame.

Public Static Functions

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

Creates a [CNFrameInfo](#) instance.

Return Returns `shared_ptr` of [CNFrameInfo](#) if this function has run successfully. Otherwise, returns NULL.

Parameters

- `stream_id`: The data stream alias. Identifies which data stream the frame data comes from.

- `eos`: If 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.

2.3.5 CNInferAttr

```
typedef struct {
```

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

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

```
    float score = 0;
```

```
} CNInferAttr;
```

```
struct cnstream::CNInferAttr
```

A structure holding the classification properties of an object.

Public Members

```
int id = -1
```

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

```
int value = -1
```

The label value of the classification.

```
float score = 0
```

The label score of the classification.

2.3.6 CNInferBoundingBox

```
typedef struct {
```

```
    float x, y, w, h;
```

```
} CNInferBoundingBox;
```

```
struct CNInferBoundingBox
```

A structure holding the bounding box for detection information of an object. Normalized coordinates.

2.3.7 CNInferFeature

```
typedef std::vector<float> cnstream::CNInferFeature
```

```
typedef std::vector<float> cnstream::CNInferFeature
```

The feature value for one object.

2.3.8 CNInferObject

```
typedef struct {
    public:
        std::string id;
        std::string track_id;
        float score;
        CNInferBoundingBox bbox;
        void* user_data_ = nullptr;
    private:
        std::map<std::string, CNInferAttr> attributes_;
        std::map<std::string, std::string> extra_attributes_;
        std::vector<CNInferFeature> features_;
        std::mutex attribute_mutex_;
        std::mutex feature_mutex_;
} CNInferObject;
```

```
struct cnstream::CNInferObject
```

A structure holding the information for an object.

Public Functions

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

Adds the key of an attribute to a specified object.

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

Note This is a thread-safe function.

Parameters

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

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

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

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

Note This is a thread-safe function.

Parameters

- `attribute`: The attribute pair (key, value) to be added.

```
CNInferAttr GetAttribute(const std::string &key)
```

Gets an attribute by key.

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.

Parameters

- `key`: The key of an attribute you want to query. See [AddAttribute\(\)](#).

`bool AddExtraAttribute(const std::string &key, const std::string &value)`

Adds the key of the extended attribute to a specified object.

Return Returns true if the attribute has been added successfully. Returns false if the attribute already exists in the object.

Note This is a thread-safe function.

Parameters

- `key`: The key of an attribute. You can get this attribute by key. See [GetExtraAttribute\(\)](#).
- `value`: The value of the attribute.

`bool AddExtraAttribute(const std::vector<std::pair<std::string, std::string>> &attributes)`

Adds the key pairs of the extended attributes to a specified object.

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

Note This is a thread-safe function.

Parameters

- `attributes`: Attributes to be added.

`std::string GetExtraAttribute(const std::string &key)`

Gets the extended attribute by key.

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.

Parameters

- `key`: The key of an identified attribute. See [AddExtraAttribute\(\)](#).

`void AddFeature(const CNInferFeature &features)`

Adds the feature value to a specified object.

Return Void.

Note This is a thread-safe function.

Parameters

- `features`: The feature value you want to add to.

`std::vector<CNInferFeature> GetFeatures()`

Gets the features of an object.

Return Returns the features of an object.

Note This is a thread-safe function.

Public Members`std::string id`

The ID of the classification. (label value).

`std::string track_id`

The tracking result.

`float score`

The label score.

`CNInferBoundingBox bbox`

The object normalized coordinates.

`void *user_data_ = nullptr`

User data. You can store your own data in this parameter.

2.3.9 DevContext**typedef struct {****DevType dev_type = INVALID;****int dev_id = 0;****int ddr_channel = 0****} DevContext;**`struct cnstream::DevContext`Identifies if the `CNDataFrame` data is allocated by CPU or MLU.**Public Members**`enum cnstream::DevContext::DevType dev_type = INVALID`

Device type.

`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).

2.3.10 ICNMediaImageMapper**class cnstream::ICNMediaImageMapper**`class ICNMediaImageMapper``ICNMediaImageMapper` is an abstract class.

2.3.11 IDataDeallocator

class cnstream::IDataDeallocator

```
class IDataDeallocator
    Dedicated deallocator for the CNDecoder buffer.
```

2.3.12 MemMapType

enum MemMapType {

```
    MEMMAP_INVALID = 0,
```

```
    MEMMAP_CPU = 1,
```

```
    MEMMAP_MLU = 2
```

```
};
```

enum cnstream::MemMapType

Identifies memory shared type for multi-process.

Values:

```
enumerator MEMMAP_INVALID = 0
    Invalid memory shared type.
```

```
enumerator MEMMAP_CPU = 1
    CPU memory is shared.
```

```
enumerator MEMMAP_MLU = 2
    MLU memory is shared.
```

2.4 Inferencer

2.4.1 CNFrameInfoPtr

typedef std::shared_ptr<cnstream::CNFrameInfo> cnstream::CNFrameInfoPtr

```
typedef std::shared_ptr<CNFrameInfo> cnstream::CNFrameInfoPtr
```

Constructs a pointer to [CNFrameInfo](#).

Pointer for frame information.

2.4.2 Inferencer

class cnstream::Inferencer

class Inferencer : public cnstream::Module, public cnstream::ModuleCreator<Inferencer>
 Inferencer is a module for running offline model inference.

@detail The input could come from Decoder or other plugins, in MLU memory or CPU memory. Also, if the `preproc_name` parameter is set to `PreprocCpu` in the Open function or configuration file, CPU is used for image preprocessing. Otherwise, if the `preproc_name` parameter is not set, MLU is used for image preprocessing. The image preprocessing includes data shape resizing and color space conversion. Afterwards, you can infer with offline model loading from the model path.

2.5 Module

2.5.1 Module

class cnstream::Module

class Module
 Module virtual base class.

Module is the parent class of all modules. A module should 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::Inferencer`, `cnstream::ModuleEx`, `cnstream::Pipeline`, `cnstream::RtspSink`, `cnstream::Tracker`

2.5.2 ModuleCreator

class cnstream::ModuleCreator

template<typename T>
 class ModuleCreator
 ModuleCreator A concrete ModuleClass needs to inherit ModuleCreator to enable reflection mechanism. ModuleCreator provides CreateFunction, and registers ModuleClassName and CreateFunction to ModuleFactory().

2.5.3 ModuleCreatorWorker

class cnstream::ModuleCreatorWorker

```
class ModuleCreatorWorker
    ModuleCreatorWorker, a dynamic-creator helper.
```

2.5.4 ModuleEx

class cnstream::ModuleEx

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

2.5.5 ModuleFactory

class cnstream::ModuleFactory

```
class ModuleFactory
    ModuleCreator, ModuleFactory, and ModuleCreatorWorker: Implements reflection mechanism to create a module instance dynamically with the ModuleClassName and moduleName parameters. See ActorFactory&DynamicCreator in https://github.com/Bwar/Nebula (under Apache2.0 license)
```

ModuleFactory Provides functions to create instances with the ModuleClassName and moduleName parameters.

2.5.6 ModuleParamSet

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

```
typedef std::unordered_map<std::string, std::string> cnstream::ModuleParamSet
    Module parameter set.
```

2.5.7 ParametersChecker

class cnstream::ParametersChecker

```
class ParametersChecker
    Checks the module parameters.
```

2.5.8 ParamRegister

class cnstream::Module::ParamRegister

```
class ParamRegister
    ParamRegister.
```

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

2.6 Performance Calculator

2.6.1 PerfCalculator

class cnstream::PerfCalculator

`class PerfCalculator`

Calculates performance statistics.

Reads useful data from database and then calculates performance statistics including latency, throughput, and so on.

2.6.2 PerfStats

typedef struct {

size_t latency_avg;

size_t latency_max;

size_t frame_cnt;

double fps;

} PerfStats;

`struct cnstream::PerfStats`

The basic data structure of performance statistics, including latency, frame count, and throughput.

Public Members

`size_t latency_max`
Average latency.

`size_t frame_cnt`
Max latency.

`double fps`
Frame count.

2.7 Performance Manager

2.7.1 PerfInfo

struct PerfInfo {

std::string perf_type;

std::string primary_key;

std::string primary_value;

```

    std::string key;
    std::string value;
};

```

```
struct cnstream::PerfInfo
```

The basic data structure for performance measurements.

Records performance information at the start-time point and end-time point.

Public Members

```
std::string primary_key
```

The type of perf.

```
std::string primary_value
```

pts of each data frame.

```
std::string key
```

pts of each data frame.

2.7.2 PerfManager

```
class cnstream::PerfManager
```

```
class PerfManager
```

PerfManager class.

Records [PerfInfo](#) and calculates the performance statistics of modules and pipeline .

2.8 Pipeline

2.8.1 CNModuleConfig

```
typedef struct {
```

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

```
    ModuleParamSet parameters;
```

```
    int parallelism;
```

```
    int maxInputQueueSize;
```

```
    std::string className;
```

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

```
    void ParseByJSONStr(const std::string& jstr) noexcept(false);
```

```
    void ParseByJSONFile(const std::string& jfname) noexcept(false);
```

```
} CNModuleConfig;
```

`struct cnstream::CNModuleConfig`

The configuration parameters of a module.

You can use `CNModuleConfig` to add modules in a pipeline. The module configuration can be in JSON file.

```
"name(CNModuleConfig::name)": {
  custom_params(CNModuleConfig::parameters): {
    "key0": "value",
    "key1": "value",
    ...
  }
  "parallelism(CNModuleConfig::parallelism)": 3,
  "max_input_queue_size(CNModuleConfig::maxInputQueueSize)": 20,
  "class_name(CNModuleConfig::className)": "Inferencer",
  "next_modules": ["module0(CNModuleConfig::name)", "module1(CNModuleConfig::name)", ...],
}
```

See `Pipeline::AddModuleConfig`.

Public Functions

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

Parses members from JSON string except `CNModuleConfig::name`.

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

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

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

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

Public Members

`std::string name`

The name of the module.

`ModuleParamSet parameters`

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

`int parallelism`

`Module` parallelism. The module thread number should be the same as this parameter.

`int maxInputQueueSize`

The maximum size of the input data queues.

`std::string className`

The class name of the module.

`std::vector<std::string> next`
The name of the downstream modules.

`bool showPerfInfo`
Whether to show performance information.

2.8.2 LinkStatus

```
typedef struct {
    bool stopped;
    std::vector<uint32_t> cache_size;
} LinkStatus;
```

`struct cnstream::LinkStatus`
The link status between modules.

Public Members

`bool stopped`
Whether the data transmissions between the modules are stopped.

`std::vector<uint32_t> cache_size`
The size of each queue that is used to cache data between modules.

2.8.3 Pipeline

class cnstream::Pipeline

```
class Pipeline : public cnstream::Module
```

The manager of the modules. Manages data transmission between modules, and controls message delivery.

2.8.4 StreamMsg

```
typedef struct {
    StreamMsgType type;
    int32_t chn_idx;
    std::string stream_id = "" ;
} StreamMsg;
```

`struct cnstream::StreamMsg`
Specifies a stream message.

See [StreamMsgType](#).

Public Members**StreamMsgType** type

The type of a message.

int32_t chn_idx

The index of a stream channel that incremented from 0.

std::string stream_id = ""Stream ID. You can set in `CNDataFrame::stream_id`.**2.8.5 StreamMsgObserver****class cnstream::StreamMsgObserver**`class StreamMsgObserver`

Stream message observer.

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 are bounded to the pipeline using the `Pipeline::SetStreamMsgObserver` function .

See `Pipeline::SetStreamMsgObserver` `StreamMsg` `StreamMsgType`.

2.8.6 StreamMsgType**enum StreamMsgType {****EOS_MSG = 0,****ERROR_MSG,****USER_MSG0 = 32,****USER_MSG1,****USER_MSG2,****USER_MSG3,****USER_MSG4,****USER_MSG5,****USER_MSG6,****USER_MSG7,****USER_MSG8,****USER_MSG9****};****enum cnstream::StreamMsgType**

Data stream message type.

Values:

`enumerator EOS_MSG = 0`

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 USER_MSG0 = 32`

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.

2.9 RTSP Sink

2.9.1 ColorFormat

`enum ColorFormat {`

`YUV420 = 0,`

`RGB24,`

`BGR24,`

`NV21,`

`NV12,`

`};`


```
enum cnstream::ColorFormat
  Values:

  enumerator YUV420 = 0
    Planar Y4-U1-V1.

  enumerator RGB24

  enumerator BGR24
    Packed B8G8R8.

  enumerator NV21
    Semi-Planar Y4-V1U1.

  enumerator NV12
    Semi-Planar Y4-U1V1.
```

2.9.2 RtspParam

```
typedef struct {
  int frame_rate = 25;
  int udp_port = 9554;
  int http_port = 8080;
  int src_width = 1920;
  int src_height = 1080;
  int dst_width = 1920;
  int dst_height = 1080;
  int gop = 20;
  int kbps = 2 * 1024;
  ColorFormat color_format = NV21;
  VideoCodecType codec_type = H264;
  EncoderType enc_type = FFMPEG;
  int device_id;
  int view_rows;
  int view_cols;
  std::string view_mode;
  std::string color_mode;
  std::string preproc_type;
  std::string encoder_type;
} RtspParam;

struct cnstream::RtspParam
```

Public Members

`int frame_rate = 25`
Target fps.

`int src_width = 1920`
Source width.

`int src_height = 1080`
Source height.

`int dst_width = 1920`
Target width, preferred size same with input.

`int dst_height = 1080`
Target height, preferred size same with input.

`int gop = 20`
Target gop, default is 10.

`int kbps = 2 * 1024`
Target Kbps, default is 2*1024(2M).

2.9.3 RtspSink**class cnstream::RtspSink**

`class RtspSink : public cnstream::Module, public cnstream::ModuleCreator<RtspSink>`
`RtspSink` is a module to deliver stream by RTSP protocol.

2.10 SyncMem**2.10.1 CNSyncedMemory****class cnstream::CNSyncedMemory**

`class CNSyncedMemory`
Synchronizes 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 `CNSyncedMemory::UNINITIALIZED` when memory size is 0.

2.10.2 SyncedHead

```
enum SyncedHead {
```

```
    UNINITIALIZED,
```

```
    HEAD_AT_CPU,
```

```
    HEAD_AT_MLU,
```

```
    SYNCED
```

```
};
```

```
enum cnstream::CNSyncedMemory::SyncedHead
```

Head synchronization.

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.

2.11 Tracker

2.11.1 Tracker

```
class cnstream::Tracker
```

```
class Tracker : public cnstream::Module, public cnstream::ModuleCreator<Tracker>
```

[Tracker](#) is a module for realtime tracking. Extracts feature on MLU if the model path is provided. Otherwise, it would be done on CPU.



3 CNStream API Reference

CNStream APIs support both on MLU270 and MLU220.

3.1 Data Source

3.1.1 CheckParamSet

```
bool cnstream::DataSource::CheckParamSet(const ModuleParamSet &paramSet) const
                                         override
```

Checks parameters for a module.

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

Parameters

- paramSet: Parameters for this module.

3.1.2 Close

```
void cnstream::DataSource::Close() override
```

Called by pipeline when the pipeline is stopped.

3.1.3 CreateSource

```
std::shared_ptr<cnstream::SourceHandler> cnstream::DataSource::CreateSource(const
                                                                              std::string
                                                                              &stream_id,
                                                                              const
                                                                              std::string
                                                                              &filename,
                                                                              int fram-
                                                                              erate,
                                                                              bool
                                                                              loop    =
                                                                              false)
```

Adds one stream to `DataSource` module. This function should be called after the pipeline has been started.

Return Returns the source handler instance.

Parameters

- stream_id[in]: The unique stream identifier.

- `filename[in]`: The source path that supports local-file-path, rtsp-url, jpg-sequences, and so on.
- `framerate[in]`: The input frequency of the source data.
- `loop[in]`: Whether to reload source when EOF is reached.

3.1.4 GetSourceParam

`DataSourceParam` `cnstream::DataSource::GetSourceParam()` `const`

Gets module parameters. This function should be called after `Open()` has been invoked.

3.1.5 Open

`bool` `cnstream::DataSource::Open(ModuleParamSet paramSet)` `override`

Called by pipeline when the pipeline is started.

Return Returns true if `paramSet` are supported and valid. Otherwise, returns false.

Parameters

- `paramSet`:
 - `source_type`: Optional. The demuxer type. The default value is `SOURCE_RAW`. Supported values are `raw` and `ffmpeg`.
 - `output_type`: Optional. The output type. The default `output_type` is `cpu`. Supported values are `mlu` and `cpu`.
 - `interval`: Optional. Process one frame for every `interval` frames. Process every frame by default.
 - `decoder_type`: Optional. The decoder type. The default `decoder_type` is `cpu`. Supported values are `mlu` and `cpu`.
 - `reuse_cndec_buf`: Optional. Whether the codec buffer will be reused. The default value is `false`. This parameter is used when decoder type is `mlu`. Supported values are `true` and `false`.
 - `device_id`: Required when MLU is used. Device id. Set the value to -1 for CPU. Set the value for MLU in the range 0 - N.
 - `chunk_size`: Required when `source_type` is set to `raw`. The size of the input data chunk.
 - `width`: Required when `source_type` is set to `raw`. The width of the frame.
 - `height`: Required when `source_type` is set to `raw`. The height of the frame.
 - `interlaced`: Required when `source_type` is set to `raw`. Interlaced mode.
 - `input_buf_number`: Optional. The input buffer number. The default value is 2.
 - `output_buf_number`: Optional. The output buffer number. The default value is 3.

3.2 Eventbus

3.2.1 AddBusWatch

```
uint32_t cnstream::EventBus::AddBusWatch(BusWatcher func, Module *watch_module)
```

Adds the watcher to the event bus.

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

Parameters

- `func`: The bus watcher to be added.
- `watch_module`: The module that adds this bus watcher.

3.2.2 BusWatcher

```
typedef std::function<EventHandleFlag(const Event&, Module*)> cnstream::BusWatcher
```

The bus watcher function.

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

Parameters

- `event`: The event polled from the event bus.
- `module`: The module that is watching.

3.2.3 ClearAllWatchers

```
void cnstream::EventBus::ClearAllWatchers()
```

Removes all bus watchers.

3.2.4 GetBusWatchers

```
const std::list<std::pair<BusWatcher, Module*>> &cnstream::EventBus::GetBusWatchers() const
```

Gets all bus watchers from the event bus.

Return A list with pairs of bus watcher and module.

3.2.5 IsRunning

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

Checks if the event bus is running.

Return Returns true if the event bus is running. Otherwise, returns false.

3.2.6 PollEvent

`Event cnstream::EventBus::PollEvent()`

Polls an event from a bus [block].

Note This function is blocked until an event or a bus is stopped.

3.2.7 PostEvent

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

Posts an event to a bus.

Return Returns true if this function has run successfully.

Parameters

- `event`: The event to be posted.

3.3 Frame

3.3.1 AddAttribute

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

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

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

Note This is a thread-safe function.

Parameters

- `attribute`: The attribute pair (key, value) to be added.

3.3.2 AddAttribute

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

Adds the key of an attribute to a specified object.

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

Note This is a thread-safe function.

Parameters

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

3.3.3 AddExtraAttribute

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

Adds the key of the extended attribute to a specified object.

Return Returns true if the attribute has been added successfully. Returns false if the attribute already exists in the object.

Note This is a thread-safe function.

Parameters

- `key`: The key of an attribute. You can get this attribute by key. See [GetExtraAttribute\(\)](#).
- `value`: The value of the attribute.

3.3.4 AddExtraAttribute

`bool cnstream::CNInferObject::AddExtraAttribute(const std::vector<std::pair<std::string, std::string>> &attributes)`

Adds the key pairs of the extended attributes to a specified object.

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

Note This is a thread-safe function.

Parameters

- `attributes`: Attributes to be added.

3.3.5 AddFeature

`void cnstream::CNInferObject::AddFeature(const CNInferFeature &features)`

Adds the feature value to a specified object.

Return Void.

Note This is a thread-safe function.

Parameters

- `features`: The feature value you want to add to.

3.3.6 CNGetPlanes

`int cnstream::CNGetPlanes(CNDataFormat fmt)`

Gets image plane number by a specified image format.

Return

Parameters

- `fmt`: The format of the image.

Return Value

- 0: Unsupported image format.
- >0: Image plane number.

3.3.7 CopyToSharedMem

void `cnstream::CNDataFrame::CopyToSharedMem(MemMapType type)`
Copies source-data to shared memory for multi-process.

Return Void.

Parameters

- `memory`: The type of the mapped or shared memory.

3.3.8 CopyToSyncMem

void `cnstream::CNDataFrame::CopyToSyncMem()`
Synchronizes the source-data to `CNSyncedMemory`.

3.3.9 CopyToSyncMemOnDevice

void `cnstream::CNDataFrame::CopyToSyncMemOnDevice(int device_id)`
Synchronizes source data to specific device, and reset `ctx.dev_id` to `device_id` when synced, for multi-device case.

Return Void.

Parameters

- `device_id`: The device id.

3.3.10 Create

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

Creates a `CNFrameInfo` instance.

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

Parameters

- `stream_id`: The data stream alias. Identifies which data stream the frame data comes from.
- `eos`: If 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.

3.3.11 GetAttribute

`CNInferAttr cnstream::CNInferObject::GetAttribute(const std::string &key)`

Gets an attribute by key.

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.

Parameters

- `key`: The key of an attribute you want to query. See [AddAttribute\(\)](#).

3.3.12 GetBytes

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

Gets the number of bytes in a frame.

Return Returns the number of bytes in a frame.

3.3.13 GetExtraAttribute

`std::string cnstream::CNInferObject::GetExtraAttribute(const std::string &key)`

Gets the extended attribute by key.

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.

Parameters

- `key`: The key of an identified attribute. See [AddExtraAttribute\(\)](#).

3.3.14 GetFeatures

`std::vector<CNInferFeature> cnstream::CNInferObject::GetFeatures()`

Gets the features of an object.

Return Returns the features of an object.

Note This is a thread-safe function.

3.3.15 GetMediaImage

`void *cnstream::ICNMediaImageMapper::GetMediaImage() = 0`

Gets an image.

Return Returns the image address.

3.3.16 GetPlanes

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

Gets plane count for a specified frame.

Return Returns the plane count of this frame.

3.3.17 GetPlaneBytes

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

Gets the number of bytes in a specified plane.

Return Returns the number of bytes in the plane.

Parameters

- `plane_idx`: The index of the plane. The index increments from 0.

3.3.18 ImageBGR

`cv::Mat *cnstream::CNDataFrame::ImageBGR()`

Converts data from RGB to BGR. This API should be called after `CopyToSyncMem()` is invoked.

If the data is not in RGB format but in BGR, YUV420NV12, or YUV420NV21 format, its color mode will not be converted.

Return Returns data with opencv mat type.

3.3.19 ~ICNMediaImageMapper

`cnstream::ICNMediaImageMapper::~~ICNMediaImageMapper()`

Destructor of class `ICNMediaImageMapper`.

3.3.20 MmapSharedMem

`void cnstream::CNDataFrame::MmapSharedMem(MemMapType type)`

Maps shared memory for multi-process.

Return Void.

Parameters

- `memory`: The type of the mapped or shared memory.

3.3.21 ReleaseSharedMem

void `cnstream::CNDataFrame::ReleaseSharedMem(MemMapType type)`
 Releases shared memory for multi-process.

Return Void.

Parameters

- `memory`: The type of the mapped or shared memory.

3.3.22 UnMapSharedMem

void `cnstream::CNDataFrame::UnMapSharedMem(MemMapType type)`
 Unmaps the shared memory for multi-process.

Return Void.

Parameters

- `memory`: The type of the mapped or shared memory.

3.4 Inferencer

3.4.1 CheckParamSet

bool `cnstream::Inferencer::CheckParamSet(const ModuleParamSet ¶mSet) const` `override`

Checks parameters for a module.

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

Parameters

- `paramSet`: Parameters of this module.

3.4.2 Close

void `cnstream::Inferencer::Close() override`
 Called by pipeline when the pipeline is stopped.

Return Void.

3.4.3 Open

bool `cnstream::Inferencer::Open(ModuleParamSet paramSet) override`
 Called by pipeline when the pipeline is started.

Return Returns true if the inferencer has been opened successfully.

Parameters

- `paramSet`:
 - `model_path`: Required. The path of the offline model.
 - `func_name`: Required. The function name that is defined in the offline model. It could be found in Cambricon twins file. For most cases, it is “subnet0” .

- `postproc_name`: Required. The class name for postprocess. The class specified by this name must inherited from class `cnstream::Postproc` when `[object_infer]` is false, otherwise the class specified by this name must inherit from class `cnstream::ObjPostproc`.
- `preproc_name`: Optional. The class name for preprocessing on CPU. The class specified by this name must inherited from class `cnstream::Preproc` when `[object_infer]` is false, otherwise the class specified by this name must inherit from class `cnstream::ObjPreproc`. Preprocessing will be done on MLU by `ResizeYuv2Rgb` (cambricon Bang op) when this parameter not set.
- `use_scaler`: Optional. Whether use the scaler to preprocess the input. The scaler will not be used by default.
- `device_id`: Optional. MLU device ordinal number. The default value is 0.
- `batching_timeout`: Optional. The batching timeout. The default value is 3000.0[ms]. type[float]. unit[ms].
- `data_order`: Optional. Data format. The default format is NHWC.
- `threshold`: Optional. The threshold of the confidence. By default it is 0.
- `infer_interval`: Optional. Process one frame for every `infer_interval` frames.
- `show_stats`: Optional. Whether show inferencer performance statistics. It will not be shown by default.
- `stats_db_name`: Required when `show_stats` is set to `true`. The directory to store the db file. e.g., `dir1/dir2/detect.db`.
- `object_infer`: Optional. if `object_infer` is set to `true`, the detection target is used as the input to inferencing. if it is set to `false`, the video frame is used as the input to inferencing. False by default.
- `obj_filter_name`: Optional. The class name for object filter. See `cnstream::ObjFilter`. This parameter is valid when `object_infer` is `true`. When this parameter not set, no object will be filtered.

3.4.4 Process

```
int cnstream::Inferencer::Process(CNFrameInfoPtr data) final
```

Performs inference for each frame.

Parameters

- `data`: The information and data of frames.

Return Value

- 1: The process has run successfully.
- -1: The process is failed.

3.5 Module

3.5.1 CheckParamSet

`bool cnstream::Module::CheckParamSet(const ModuleParamSet ¶mSet) const`

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

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

Parameters

- `paramSet`: Parameters for this module.

3.5.2 CheckPath

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

Checks if the path exists.

Return Returns true if exists. Otherwise, returns false.

Parameters

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

3.5.3 Close

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

Closes resources for a module.

Return Void.

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.

3.5.4 Create

`Module *cnstream::ModuleFactory::Create(const std::string &strTypeName, const std::string &name)`

Creates a module instance with `ModuleClassName` and `moduleName`.

Return Returns the module instance if this function has run successfully. Otherwise, returns `nullptr` if failed.

Parameters

- `strTypeName`: The module class name.
- `name`: The `CreateFunction` of a `Module` object that has a parameter `moduleName`.

3.5.5 Create

`Module *cnstream::ModuleCreatorWorker::Create(const std::string &strTypeName, const std::string &name)`

Creates a module instance with `ModuleClassName` and `moduleName`.

Return Returns the module instance if the module instance is created successfully. Returns `nullptr` if failed.

See `ModuleFactory::Create`

Parameters

- `strTypeName`: The module class name.
- `name`: The module name.

3.5.6 CreateObject

`T *cnstream::ModuleCreator::CreateObject(const std::string &name)`

Creates an instance of template (T) with specified instance name.

This is a template function.

Return Returns the instance of template (T).

Parameters

- `name`: The name of the instance.

3.5.7 ClearPerfManagers

`void cnstream::Module::ClearPerfManagers()`

Clears PerfManagers.

Return Void.

3.5.8 DoProcess

`int cnstream::Module::DoProcess(std::shared_ptr<CNFrameInfo> data)`

Processes the data.

This function is called by a pipeline.

Return

Parameters

- `data`: A pointer to the information of the frame.

Return Value

- 0: The process has been run successfully. The data should be transmitted by framework then.
- >0: The process has been run 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` posts an event with the `EVENT_ERROR` event type and return number.

3.5.9 GetModuleDesc

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

Gets the description of the module.

This is used in CNStream Inspect tool.

Return Returns the description of the module.

3.5.10 GetName

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

Gets the name of this module.

Return Returns the name of this module.

3.5.11 GetOutputFrame

`std::shared_ptr<CNFrameInfo> cnstream::Module::GetOutputFrame()`

Get output frame data. This function should be used only when the module is not contained by any containers (pipelines).

Return The output data.

3.5.12 GetParams

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

Gets the registered parameters and the parameter description.

This is used in CNStream Inspect tool.

Return Returns the registered parameters and the parameter description.

3.5.13 GetPathRelativeToTheJSONFile

`std::string cnstream::GetPathRelativeToTheJSONFile(const std::string &path, const ModuleParamSet ¶m_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.

Return Returns the complete path of a file.

Parameters

- `path`: The path relative to the JSON file or an absolute path.
- `param_set`: The module parameters. The JSON file path is one of the parameters.

3.5.14 GetPerfManager

```
std::shared_ptr<PerfManager> cnstream::Module::GetPerfManager(const std::string
                                                           &stream_id)
```

Gets [PerfManager](#) by stream ID.

Return Returns the `shared_ptr` object of [PerfManager](#).

Parameters

- `stream_id`: The stream ID.

3.5.15 GetRegistered

```
std::vector<std::string> cnstream::ModuleFactory::GetRegistered()
```

Gets all registered modules.

Return All registered module class names.

3.5.16 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](#)

3.5.17 Instance

```
ModuleFactory *cnstream::ModuleFactory::Instance()
```

Creates or gets the instance of the [ModuleFactory](#) class.

Return Returns the instance of the [ModuleFactory](#) class.

3.5.18 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.

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

Parameters

- `check_list`: A list of parameter names.
- `paramSet`: The module parameters.
- `err_msg`: The error message.
- `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 should be less than zero.

3.5.19 IsRegistered

`bool cnstream::Module::ParamRegister::IsRegistered(const std::string &key) const`
Checks if the parameter is registered.

This is used in CNStream Inspect tool.

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

Parameters

- `key`: The parameter name.

3.5.20 Open

`bool cnstream::Module::Open(ModuleParamSet param_set) = 0`
Opens resources for a 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.

Parameters

- `param_set`: A set of parameters for this module.

3.5.21 PostEvent

`bool cnstream::Module::PostEvent(EventType type, const std::string &msg) const`
Posts an event to the pipeline.

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

Parameters

- `type`: The type of an event.
- `msg`: The event message string.

3.5.22 Process

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

Parameters

- `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.

3.5.23 `Regist`

```
bool cnstream::ModuleFactory::Regist(const std::string &strTypeName,
                                     std::function<Module*> const strTypeDesc,
                                     const std::string&
                                     > pFuncRegisters ModuleClassName and CreateFunction.
```

Return Returns true if this function has run successfully.

Parameters

- `strTypeName`: The module class name.
- `pFunc`: The `CreateFunction` of a `Module` object that has a parameter `moduleName`.

3.5.24 `Register`

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

Registers a parameter and its description.

This is used in CNStream Inspect tool.

Return Void.

Parameters

- `key`: The parameter name.
- `desc`: The description of the parameter.

3.5.25 `SetContainer`

```
void cnstream::Module::SetContainer(Pipeline *container)
```

Sets a container to this module and identifies which pipeline the module is added to.

Note This function is called automatically by the pipeline after this module is added into the pipeline. You do not need to call this function by yourself.

Parameters

- `container`: A pipeline pointer to the container of this module.

3.5.26 `SetModuleDesc`

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

Sets the description of the module.

This is used in CNStream Inspect tool.

Return Void.

Parameters

- `desc`: The description of the module.

3.5.27 SetPerfManagers

```
void cnstream::Module::SetPerfManagers(const          std::unordered_map<std::string,
                                                std::shared_ptr<PerfManager>>
                                                &perf_managers)
```

Sets PerfManagers. The module can then use PerfManagers.

Return Void.

Parameters

- `perf_managers`: A container instance in the `unordered_map` type. The key is the stream ID. The value is a `shared_ptr` object of `PerfManager`.

3.5.28 ShowPerfInfo

```
bool cnstream::Module::ShowPerfInfo()
```

Checks if the display of performance information is enabled.

Return Returns true if the performance information is displayed. Otherwise, returns false.

3.5.29 ShowPerfInfo

```
void cnstream::Module::ShowPerfInfo(bool enable)
```

Enables or disables to display performance information.

Return Void.

Parameters

- `enable`: If it is true, enable to display performance information, otherwise, disable to display the performance information.

3.5.30 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.

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

Parameters

- `data`: A pointer to the information of the frame.

3.6 Performance Calculator

3.6.1 CalcLatency

```
PerfStats cnstream::PerfCalculator::CalcLatency(std::shared_ptr<Sqlite> sql, std::string
                                                type, std::string start_key, std::string
                                                end_key)
```

Calculates latency.

Calculates latency from the previous time that this API is called, to now. Then the performance statistics are updated and saved. The time when this API is called is recorded and will be used in calculation in the call of this API next time.

Return Returns the performance statistics.

Parameters

- `sql`: A shared pointer points to `Sqlite`, which is a class for operating the database.
- `type`: The type of perf.
- `start_key`: The index name of start-time points in database.
- `end_key`: The index name of end-time points in database.

3.6.2 CalcThroughputByTotalTime

```
PerfStats cnstream::PerfCalculator::CalcThroughputByTotalTime(std::shared_ptr<Sqlite>
                                                             sql, std::string type,
                                                             std::string start_key,
                                                             std::string end_key)
```

Calculates throughput. The formula is as follows:

$$\text{throughput} = \text{frame_count} / (\text{maximum_of_the_end_time_points} - \text{minimum_of_the_start_time_points})$$

Return Returns the performance statistics.

Parameters

- `sql`: A shared pointer points to `Sqlite`, which is a class for operating the database.
- `type`: The type of perf.
- `start_key`: The index name of the start-time points in database.
- `end_key`: The index name of the end-time points in database.

3.6.3 PrintLatency

```
void cnstream::PrintLatency(const PerfStats &stats)
    Prints latency.
```

Return Void.

Parameters

- `stats`: The performance statistics.

3.6.4 PrintPerfStats

```
void cnstream::PrintPerfStats(const PerfStats &stats)
    Prints performance statistics.
```

Return Void.

Parameters

- `stats`: The performance statistics.

3.6.5 PrintThroughput

`void cnstream::PrintThroughput(const PerfStats &stats)`

Prints throughput.

Return Void.

Parameters

- `stats`: The performance statistics.

3.7 Performance Manager

3.7.1 CalculatePerfStats

`PerfStats cnstream::PerfManager::CalculatePerfStats(std::string perf_type, std::string module_name)`

Calculates the performance statistics of modules.

Return Returns the performance statistics.

Parameters

- `perf_type`: The type of performance.
- `module_name`: The module name.

3.7.2 CalculatePipelinePerfStats

`std::vector<std::pair<std::string, PerfStats>> cnstream::PerfManager::CalculatePipelinePerfStats(std::string perf_type)`

Calculates the performance statistics of a pipeline.

Return Returns performance statistics of all end modules of a pipeline.

Parameters

- `perf_type`: The type of performance.

3.7.3 Init

`bool cnstream::PerfManager::Init(std::string db_name, std::vector<std::string> module_names, std::string start_node, std::vector<std::string> end_nodes)`

Initializes `PerfManager`.

Creates database, tables, and `PerfCalculator`, and starts the thread function, which inserts `PerfInfo` to database.

Return Returns true if `PerfManager` is initialized successfully, otherwise returns false.

Parameters

- `db_name`: The name of the database.
- `module_names`: All module names in the pipeline.
- `start_node`: The name of the start module in the pipeline.
- `end_nodes`: The names of all end modules in the pipeline.

3.7.4 Record

`bool cnstream::PerfManager::Record(bool is_finished, std::string type, std::string module_name, int64_t pts)`

Records `PerfInfo`.

Creates timestamp and sets it to `PerfInfo`. And then inserts the time stamp into database.

Return Returns true if the information is recorded successfully, otherwise returns false.

Parameters

- `info`: The `PerfInfo`.

3.7.5 RegisterPerfType

`bool cnstream::PerfManager::RegisterPerfType(std::string type)`

Registers performance type.

Return Returns true if the performance type is registered successfully, otherwise returns false.

Parameters

- `type`: The type of performance.

3.7.6 SqlBeginTrans

`void cnstream::PerfManager::SqlBeginTrans()`

Begins a sqlite3 event.

Return Void.

3.7.7 SqlCommitTrans

`void cnstream::PerfManager::SqlCommitTrans()`

Commits a sqlite3 event.

Return Void.

3.7.8 Stop

`void cnstream::PerfManager::Stop()`

Stops to record `PerfInfo`.

Return Void.

3.8 Pipelines

3.8.1 AddModule

bool cnstream::Pipeline::AddModule(std::shared_ptr<Module> module)

Adds the module to a pipeline.

Return Returns true if this function has run successfully. Returns false if the module has been added to this pipeline.

Parameters

- module: The module instance to be added to this pipeline.

3.8.2 AddModuleConfig

int cnstream::Pipeline::AddModuleConfig(const CNModuleConfig &config)

Adds module configurations in a pipeline.

Return Returns 0 if this function has run successfully. Otherwise, returns -1.

Parameters

- The: configuration of a module.

3.8.3 BuildPipeline

int cnstream::Pipeline::BuildPipeline(const std::vector<CNModuleConfig> &configs)

Builds a pipeline by module configurations.

Return Returns 0 if this function has run successfully. Otherwise, returns -1.

Parameters

- configs: The configurations of a module.

3.8.4 BuildPipelineByJSONFile

int cnstream::Pipeline::BuildPipelineByJSONFile(const std::string &config_file)

Builds a pipeline from a JSON file.

```
{
  "source" : {
    "class_name" : "cnstream::DataSource",
    "parallelism" : 0,
    "next_modules" : ["detector"],
    "custom_params" : {
      "source_type" : "ffmpeg",
      "decoder_type" : "mlu",
      "device_id" : 0
    }
  },
  "detector" : {...}
}
```


Return Returns 0 if this function has run successfully. Otherwise, returns -1.

Parameters

- `config_file`: The configuration file in JSON format.

3.8.5 CalculateModulePerfStats

`void cnstream::Pipeline::CalculateModulePerfStats()`

Calculates the performance of modules and prints the performance statistics.

This is called by thread function `CalculatePerfStats`.

Return Void.

3.8.6 CalculatePerfStats

`void cnstream::Pipeline::CalculatePerfStats()`

Calculates performance of modules and pipeline, and prints performance statistics every two seconds.

This is a thread function.

Return Void.

3.8.7 CalculatePipelinePerfStats

`void cnstream::Pipeline::CalculatePipelinePerfStats()`

Calculates the performance of pipeline and prints the performance statistics.

This is called by thread function `CalculatePerfStats`.

Return Void.

3.8.8 Close

`void cnstream::Pipeline::Close() override`

See `Module::Close`.

3.8.9 CreatePerfManager

`bool cnstream::Pipeline::CreatePerfManager(std::vector<std::string> stream_ids,
std::string db_dir)`

Creates `PerfManager` to measure performance of modules and pipeline for each stream.

This function creates database for each stream. One thread is for committing sqlite events to increase the speed of inserting data to the database. Another is for calculating performance of modules and pipeline, and printing performance statistics afterwards.

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

Parameters

- `stream_ids`: The stream IDs.

- `db_dir`: The directory where database files to be saved.

3.8.10 GetEventBus

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

Gets the event bus in the pipeline.

Return Returns the event bus.

3.8.11 GetLinkIds

`std::vector<std::string> cnstream::Pipeline::GetLinkIds()`

Gets link-indexes that is used to query link status between modules. The link-index is the return value of `Pipeline::LinkModules`.

Return Returns all link-indexes in a pipeline.

See `Pipeline::LinkModules` and `Pipeline::QueryLinkStatus`.

3.8.12 GetModule

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

Gets a module in a pipeline by name.

Return Returns the module pointer if the module named `moduleName` has been added to the pipeline. Otherwise, returns `nullptr`.

Parameters

- `moduleName`: The module name specified in the module constructor.

3.8.13 GetModuleConfig

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

Gets the module configuration by the module name.

Return Returns module configuration if this function has run successfully. Returns `NULL` if the module specified by `module_name` has not been added to this pipeline.

Parameters

- `module_name`: The module name specified in module constructor.

3.8.14 GetModuleParallelism

`uint32_t cnstream::Pipeline::GetModuleParallelism(std::shared_ptr<Module> module)`

Gets the module parallelism.

Return Returns the module parallelism if this function has run successfully. Returns 0 if the module has not been added to this pipeline.

Parameters

- `module`: The module you want to query.

3.8.15 GetModuleParamSet

`ModuleParamSet` `cnstream::Pipeline::GetModuleParamSet(const std::string &moduleName)`

Gets parameter set of a module. `Module` parameter set is used in `Module::Open`. It provides the ability for modules to customize parameters.

Return Returns the customized parameters of the module. If the module does not have customized parameters or the module has not been added to this pipeline, then the value of `size (ModuleParamSet::size)` is 0.

See `Module::Open`.

Parameters

- `moduleName`: The module name specified in the module constructor.

3.8.16 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`.

3.8.17 IsRunning

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

Returns the running status of a pipeline.

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

3.8.18 LinkModules

`std::string` `cnstream::Pipeline::LinkModules(std::shared_ptr<Module> up_node, std::shared_ptr<Module> down_node)`

Links two modules. The upstream node will process data before the downstream node.

Return Returns the link-index if this function has run successfully. The link-index can be used to query link status between `up_node` and `down_node`. See `Pipeline::QueryStatus` for details. Returns NULL if one of the two nodes has not been added to this pipeline.

Note Both `up_node` and `down_node` should be added to this pipeline before calling this function.

See `Pipeline::QueryStatus`.

Parameters

- `up_node`: The upstream module.
- `down_node`: The downstream module.

3.8.19 NotifyStreamMsg

void cnstream::Pipeline::NotifyStreamMsg(const StreamMsg &smg)
Passes the stream message to the observer of this pipeline.

Return Void.

See StreamMsg.

Parameters

- smg: The stream message.

3.8.20 Open

bool cnstream::Pipeline::Open(ModuleParamSet paramSet) override

See Module::Open.

3.8.21 ParseByJSONFile

bool cnstream::CNModuleConfig::ParseByJSONFile(const std::string &jfname)

Parses members from JSON file except CNModuleConfig::name.

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

3.8.22 ParseByJSONStr

bool cnstream::CNModuleConfig::ParseByJSONStr(const std::string &jstr)

Parses members from JSON string except CNModuleConfig::name.

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

3.8.23 PerfSqlCommitLoop

void cnstream::Pipeline::PerfSqlCommitLoop()

Commits sqlite events to increase the speed of inserting data to the database.

This is a thread function. The events are committed every second.

Return Void.

3.8.24 Process

int cnstream::Pipeline::Process(std::shared_ptr<CNFrameInfo> data) override

Useless.

See Module::Process.

3.8.25 ProvideData

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

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

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.

See [Module::Process](#).

Parameters

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

3.8.26 QueryLinkStatus

```
bool cnstream::Pipeline::QueryLinkStatus(LinkStatus *status, const std::string &link_id)
```

Queries the link status by link-index. link-index is returned by [Pipeline::LinkModules](#).

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

See [Pipeline::LinkModules](#).

Parameters

- `status`: The link status to query.
- `link_id`: The Link-index returned by [Pipeline::LinkModules](#).

3.8.27 RegistIPCFrameDoneCallback

```
void cnstream::Pipeline::RegistIPCFrameDoneCallBack(std::function<void> std::shared_ptr<CNFrameInfo>
> callback)
```

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

Return Void.

3.8.28 SetModuleAttribute

```
bool cnstream::Pipeline::SetModuleAttribute(std::shared_ptr<Module> module, uint32_t
parallelism, size_t queue_capacity = 20)
```

Sets the parallelism and conveyor capacity attributes of the module.

The `SetModuleParallelism` function is deprecated. Please use the `SetModuleAttribute` function instead.

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

Note You must call this function before calling [Pipeline::Start](#).

See [CNModuleConfig::parallelism](#).

Parameters

- `module`: The module to be configured.
- `parallelism`: [Module](#) parallelism, as well as [Module](#)' s conveyor number of input connector.

- `queue_capacity`: The queue capacity of the `Module` input conveyor.

3.8.29 SetStreamMsgObserver

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

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

Return Void.

See `StreamMsgObserver`.

Parameters

- `observer`: The stream message observer.

3.8.30 Start

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

Starts a pipeline. Starts data transmission in a pipeline. Calls the `Open` function for all modules. See `Module::Open`. Links modules.

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.

3.8.31 Stop

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

Stops data transmissions in a pipeline.

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

3.9 RTSP Sink

3.9.1 CheckParamSet

`bool cnstream::RtspSink::CheckParamSet(const ModuleParamSet ¶mSet) const`
`override`

Checks ParamSet for a module.

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

Parameters

- `paramSet`: Parameters for this module.

3.9.2 Close

void `cnstream::RtspSink::Close()` override

Called by a pipeline when the pipeline is stopped.

3.9.3 Open

bool `cnstream::RtspSink::Open(ModuleParamSet paramSet)` override

Called by a pipeline when the pipeline is started.

Return Returns true if a module is opened succeed. Otherwise, returns false.

Parameters

- `paramSet`: Parameter set.

3.9.4 Process

int `cnstream::RtspSink::Process(CNFrameInfoPtr data)` override

Encodes for each frame.

Return Whether the process is succeed.

Parameters

- `data`: Data to be processed.

Return Value

- 0: Succeed and has no intercept data.
- <0: Failed.

3.10 Syncmem

3.10.1 CNStreamFreeHost

void `cnstream::CNStreamFreeHost(void *ptr)`

Frees the data allocated by `CNStreamMallocHost`.

Parameters

- `ptr`: The data address to be freed.

3.10.2 CNStreamMallocHost

void `cnstream::CNStreamMallocHost(void **ptr, size_t size)`

Allocates data on a host.

Parameters

- `ptr`: Outputs data pointer.
- `size`: The size of the data to be allocated.

3.10.3 CNSyncedMemory

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

Parameters

- `size`: The size of the memory.

3.10.4 CNSyncedMemory

`cnstream::CNSyncedMemory::CNSyncedMemory(size_t size, int mlu_dev_id, int mlu_ddr_chn)`
Constructor.

Parameters

- `size`: The size of the memory.
- `mlu_dev_id`: MLU device ID that is incremented from 0.
- `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.

3.10.5 GetCpuData

`const void *cnstream::CNSyncedMemory::GetCpuData()`
Gets the CPU data.

Return Returns the CPU data pointer.

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

3.10.6 GetHead

`SyncedHead cnstream::CNSyncedMemory::GetHead() const`
Gets synchronized head.

Return Returns synchronized head.

3.10.7 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.

3.10.8 GetMluDdrChnId

int cnstream::CNSyncedMemory::GetMluDdrChnId() const

Gets the channel ID of the MLU DDR.

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

3.10.9 GetMluDevId

int cnstream::CNSyncedMemory::GetMluDevId() const

Gets the MLU device ID.

Return Returns the device that the MLU memory allocated on.

3.10.10 GetMutableCpuData

void *cnstream::CNSyncedMemory::GetMutableCpuData()

Gets the mutable CPU data.

Return Returns the CPU data pointer.

3.10.11 GetMutableMluData

void *cnstream::CNSyncedMemory::GetMutableMluData()

Gets the mutable MLU data.

Return Returns the MLU data pointer.

3.10.12 GetSize

size_t cnstream::CNSyncedMemory::GetSize() const

Gets data bytes.

Return Returns data bytes.

3.10.13 SetCpuData

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

Sets the CPU data.

Return Void.

Parameters

- data: The data pointer on CPU.

3.10.14 SetMluCpuData

```
void cnstream::CNSyncedMemory::SetMluCpuData(void *mlu_data, void *cpu_data)
```

Sets the MLU data and the CPU data.

Return Void.

Parameters

- `mlu_data`: The data pointer on MLU.
- `cpu_data`: The data pointer on CPU.

3.10.15 SetMluData

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

Sets the MLU data.

Parameters

- `data`: The data pointer on MLU.

3.10.16 SetMluDevContext

```
void cnstream::CNSyncedMemory::SetMluDevContext(int dev_id, int ddr_chn = 0)
```

Sets the MLU device context.

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

Parameters

- `dev_id`: The MLU device ID that is incremented from 0.
- `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 allocated on.

3.10.17 ToCpu

```
void cnstream::CNSyncedMemory::ToCpu()
```

Synchronizes the memory data to CPU.

3.10.18 ToMlu

```
void cnstream::CNSyncedMemory::ToMlu()
```

Synchronizes the memory data to MLU.

3.11 Tracker

3.11.1 CheckParamSet

```
bool cnstream::Tracker::CheckParamSet(const ModuleParamSet &paramSet) const
                                     override
```

Checks parameters for a module.

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

Parameters

- paramSet: Parameters for this module.

3.11.2 Close

```
void cnstream::Tracker::Close() override
```

Called by pipeline when pipeline is stopped.

Return None.

3.11.3 Open

```
bool cnstream::Tracker::Open(cnstream::ModuleParamSet paramSet) override
```

Called by pipeline when pipeline is started.

Return Returns true if the module has been opened successfully.

Parameters

- paramSet:
 - track_name: Optional. The algorithm name for track. It is “FeatureMatch” by default.
 - model_path: Optional. The path of the offline model.
 - func_name: Optional. The function name defined in the offline model. It can be found in the Cambricon twins description file. It is “subnet0” for the most cases.

3.11.4 Process

```
int cnstream::Tracker::Process(std::shared_ptr<CNFrameInfo> data) override
```

Processes each frame.

Return Whether the process succeed.

Parameters

- data: : Pointer to the frame information.

Return Value

- 0: The process has run successfully and has no intercepted data.
- <0: The process is failed.



4 Release Notes

This release notes outlines CNStream API updates and documentation updates in CNStream Developer Guide.

4.1 CNStream Release 2020-05-25 (Version 4.5.0)

4.1.1 API Updates

This section lists API functions and fields that were added, changed, or removed.

- The following API is supported in the Frame module:
 - Added the new `CopyToSyncMemOnDevice` API to synchronize source data to a specified device.
- The following APIs are supported in the Module module:
 - Added the new `ClearPerfManagers` API to clear all performance managers.
- Supported the RTSP Sink module with the related APIs.

4.2 Release 2020-04-16 (Version 4.4.0)

4.2.1 API Updates

This section lists API functions and fields that were added, changed, or removed.

- The following APIs are supported in Frame module 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 module 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 module 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.

4.2.2 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 datatypes.

4.3 Release 2020-02-24

4.3.1 API Updates

This section lists API functions and fields that were added, changed, or removed.

- The following APIs are supported in `Frame` module:
 - 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` module:
 - 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.

4.4 Release 2019-12-31

4.4.1 API Updates

This section lists API functions and fields that were added, changed, or removed.

- The following APIs are supported in `Module` module:
 - 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 Data Source 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.