Matrix API Reference

Issue 01

Date 2020-05-30

HUAWEI TECHNOLOGIES CO., LTD.

Trademarks and Permissions


and other Huawei trademarks are trademarks of Huawei Technologies Co., Ltd.
All other trademarks and trade names mentioned in this document are the property of their respective holders.

Notice

The purchased products, services and features are stipulated by the contract made between Huawei and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

Contents

1 Overview 1

2 Process Orchestration APIs 3

2.1 Process Connection APIs (C Language) 3
2.1.1 HIAI_Init (Common with C++) 3
2.1.2 HIAI_CreateGraph 4
2.1.3 HIAI_DestroyGraph 5
2.2 Process Connection APIs (C++ Language) 5
2.2.1 Graph::GetInstance 5
2.2.2 Graph::CreateGraph (Creating a Graph Based on the Configuration File) 6
2.2.3 Graph::CreateGraph (Creating a Graph Based on the Configuration File and Writing the Generated Graph Back to the List) 7
2.2.4 Graph::CreateGraph (Creating a Graph Based on the Protobuf Data Format) 8
2.2.5 Graph::DestroyGraph 9
2.2.6 Graph::UpdateEngineConfig 10
2.3 Data Sending APIs (C Language) 11
2.3.1 HIAI_SetDataRecvFunctor 11
2.3.2 HIAI_C_SendData 12
2.4 Data Sending APIs (C++ Language) 13
2.4.1 Graph::SetDataRecvFunctor 13
2.4.2 Graph::SendData 14
2.4.3 HIAI_SendData 16
2.5 API Usage Example 17
2.5.1 Example of Using the C APIs 17
2.5.2 Example of Using C++ APIs (Creating a Graph Based on the Configuration File) 19
2.5.3 Usage Example for C++ APIs (Creating a Graph Based on the Configuration File and Writing the Generated Graph to the List) 21

3 Engine Implementation APIs (C++ Language) 24

3.1 Constructor and Destructor Functions of an Engine 24
3.2 Engine::Init 24
3.3 Macro: HIAI_DEFINE_PROCESS 26
3.4 Macro: HIAI_IMPL_ENGINE_PROCESS 26
3.5 Engine::SetDataRecvFunctor 28
3.6 Engine::SendData 29

3.7 Calling Example 30
4 Model Manager APIs (C++ Language) 33
4.1 Offline Model Manager 33
4.1.1 AIModelManager::Init 33
4.1.2 AIModelManager::SetListener 35
4.1.3 AIModelManager::Process 35
4.1.4 AIModelManager::CreateOutputTensor 36
4.1.5 AIModelManager::CreateInputTensor 37
4.1.6 AIModelManager::IsPreAllocateOutputMem 38
4.1.7 AIModelManager::GetModelIOTensorDim 38
4.1.8 AIModelManager::GetMaxUsedMemory 39
4.1.9 AISimpleTensor::SetBuffer 40
4.1.10 AITensorFactory::CreateTensor 41
4.1.11 Calling Example 42
4.2 AIPP Configuration APIs 43
4.2.1 Overview 44
4.2.2 SetDynamicInputIndex 44
4.2.3 SetDynamicInputEdgeIndex 45
4.2.4 SetInputFormat 45
4.2.5 SetCscParams (Setting the Default Parameter Value) 46
4.2.6 SetCscParams (Setting a Parameter Value as Required) 48
4.2.7 SetRbuvSwapSwitch 50
4.2.8 SetAxSwapSwitch 51
4.2.9 SetSrcImageSize 52
4.2.10 SetCropParams 52
4.2.11 SetPaddingParams 53
4.2.12 SetDtcPixelMean 55
4.2.13 SetDtcPixelMin 56
4.2.14 SetPixelVarReci 57
4.2.15 SetInputDynamicAIPP 58
4.2.16 GetDynamicInputIndex 59
4.2.17 GetDynamicInputEdgeIndex 60
4.2.18 Calling Example 61
4.3 Data Types 63
4.3.1 AIConfigItem 63
4.3.2 AIConfig 63
4.3.3 AITensorParaDescription 63
4.3.4 AITensorDescription 64
4.3.5 AITensorDescriptionList 64
4.3.6 AIModelDescription 64
4.3.7 AINNNodeDescription 65
4.3.8 AINNNodeDescriptionList 65

4.3.9 AIAPIDescription ... 65
4.3.10 AIAPIDescriptionList ... 66
4.3.11 AIOPDescription ... 66
4.3.12 AIOPDescriptionList ... 66
4.3.13 NodeDesc ... 66
4.3.14 EngineDesc ... 66
4.3.15 GraphInitDesc ... 67
4.3.16 GeneralFileBuffer ... 67
4.3.17 AIContext ... 67
4.3.18 TensorDimension ... 68
4.3.19 IAIListener ... 69
4.3.20 IAITensor ... 69
4.4 Other Compilation Dependent APIs ... 70
4.4.1 AIAlgAPIFactory ... 70
4.4.2 AIAlgAPIRegisterar ... 71
4.4.3 REGISTER_ALG_API_UNIQUE ... 71
4.4.4 REGISTER_ALG_API ... 72
4.4.5 AIModelManager ... 72
4.4.6 getModelInfo ... 73
4.4.7 IAINNNode ... 73
4.4.8 AINNNodeFactory ... 74
4.4.9 AINNNodeRegisterar ... 75
4.4.10 REGISTER_NN_NODE ... 76
4.4.11 AITensorGetBytes ... 76
4.4.12 AITensorFactory ... 76
4.4.13 REGISTER_TENSOR_CREATER_UNIQUE ... 77
4.4.14 REGISTER_TENSOR_CREATER ... 78
4.4.15 AISimpleTensor ... 78
4.4.16 AINeuralNetworkBuffer ... 79
4.4.17 AIImageBuffer ... 81
4.4.18 HIAILog ... 83
4.4.19 HIAI_ENGINE_LOG ... 84
4.5 Exception Handling ... 84

5 Auxiliary APIs ... 85

5.1 Data Receiving APIs (C++ Language) ... 85
5.1.1 Obtaining the Number of Devices ... 85
5.1.2 Obtaining the ID of the First Device ... 86
5.1.3 Obtaining the ID of the First Graph ... 87
5.1.4 Obtaining the ID of the Next Device ... 87
5.1.5 Obtaining the ID of the Next Graph ... 88
5.1.6 Obtaining the Engine Pointer ... 89
5.1.7 Obtaining the Graph ID ... 89

5.1.8 Obtaining the Device ID of a Graph 90
5.1.9 Obtaining the Graph ID of an Engine 90
5.1.10 Obtaining the Maximum Queue Size of an Engine 90
5.1.11 Obtaining the Current Queue Size of a Specified Port of the Engine 91
5.1.12 Parsing the Matrix Configuration File 91
5.1.13 Obtaining the PCIe Information 92
5.1.14 Obtaining the Version Number 93
5.1.15 Obtaining the OamConfig Smart Pointer 93
5.2 Data Type Serialization and Deserialization (C++ Language) 93
5.2.1 Macro: HIAI_REGISTER_DATA_TYPE 93
5.2.2 Macro: HIAI_REGISTER_TEMPLATE_DATA_TYPE 94
5.2.3 Macro: HIAI_REGISTER_SERIALIZE_FUNC 95
5.2.4 Graph::ReleaseDataBuffer 97
5.2.5 API Usage Example 98
5.3 Memory Management (C Language) 100
5.3.1 HIAI_DMalloc 101
5.3.2 HIAI_DFree 103
5.3.3 HIAI_DVPP_DMalloc 104
5.3.4 HIAI_DVPP_DFree 105
5.4 Memory Management (C++ Language) 106
5.4.1 HIAIMemory::HIAI_DMalloc 107
5.4.2 HIAIMemory::HIAI_DFree 110
5.4.3 HIAIMemory:: IsDMalloc 111
5.4.4 HIAIMemory::HIAI_DVPP_DMalloc 111
5.4.5 HIAIMemory::HIAI_DVPP_DFree 113
5.5 Logs (C++) 114
5.5.1 Error Code Registration 114
5.5.1.1 HIAI_DEF_ERROR_CODE 114
5.5.1.2 API Usage Example 114
5.5.2 Log Printing 115
5.5.2.1 Log Printing Format 1 115
5.5.2.2 Log Printing Format 2 116
5.5.2.3 Log Printing Format 3 116
5.5.2.4 Log Printing Format 4 117
5.5.2.5 Log Printing Format 5 117
5.5.2.6 Log Printing Format 6 118
5.5.2.7 Log Printing Format 7 119
5.5.2.8 Log Printing Format 8 119
5.6 MultiTypeQueue API for Queue Management (C++ Language) 120
5.6.1 MultiTypeQueue Constructor 120
5.6.2 PushData 121
5.6.3 FrontData 121

5.6.4 PopData....122
5.6.5 PopAllData....122
5.6.6 API Usage Example....123
5.7 Event Registration API (C++ Language)....123
5.7.1 Graph::RegisterEventHandle....123
5.8 Others (C++ Language)....124
5.8.1 DataRecvInterface::RecvData....124
5.8.2 Graph::SetPublicKeyForSignatureEncryption....125
6 Python APIs....127
7 Appendix....130
7.1 APIs to Be Deleted in Later Versions....130
7.1.1 Low-Power APIs (C++ Language)....130
7.1.1.1 PowerState::SubscribePowerStateEvent....130
7.1.1.2 PowerState::UnsubscribePowerStateEvent....131
7.1.2 Data Types....131
7.1.2.1 Enumeration: PowerState::DEVICE_POWER_STATE....132
7.1.2.2 PowerState::PowerStateNotifyCallbackT....132
7.2 Matrix Data Types....132
7.2.1 Protobuf Data Types....132
7.2.2 Matrix Custom Data Types....136
7.3 Data Structures Registered in the Matrix....136
7.4 Example....152
7.4.1 Orchestration Configuration Example....152
7.4.2 Example of Data Transmission Based on Optimized Performance....154
7.5 Change History....155

1 Overview

Matrix runs above the operating system (OS) and below service applications. It provides unified standard application programming interfaces (APIs) for applications, shielding differences between OSs. It also provides multi-node scheduling and multi-process management. In addition, Matrix can set up and run service processes based on configuration files and summarize statistics.

The Matrix logic consists of three parts: Matrix Agent (running on the host side), Matrix Daemon (running on the device side), and Matrix Service (running on the device side).

NOTE

- The host side refers to the x86 server, ARM server, or Windows PC connected to the device side. The host side uses the neural-network (NN) computing capability provided by the device side to implement services.
- The device side refers to an Ascend 310–powered hardware device that provides the host side with the NN computing capability over the PCIe interface.
- The digital vision pre-processing (DVPP) implements video encoding and decoding, JPEG encoding and decoding, PNG decoding, and vision pre-processing.
- Framework is a deep learning framework that can convert models under open source frameworks such as Caffe into models supported by Mind Studio.
- The cube-based computing engine (CCE) acceleration library accelerates upper-layer applications (frameworks or applications provided for machine learning) through APIs.
- Runtime runs in the application process space and provides Ascend 310–powered devices with various functions, such as memory management, device management, stream management, event management, and kernel execution.

Matrix is a general-purpose service process execution engine that consists of the following parts:

- **Matrix Agent**: It runs on the host side and provides the following functions:
  - Interacts with the host application for controlling commands and processing data.
  - Implements interprocess communication (IPC) with the device side.
- **Matrix Daemon**: It runs on the device side and provides the following functions:
  - Set ups a service process based on the configuration file.

  - Destroys the service process and reclaims resources according to commands.
  - Serves as the Daemon process, which is used to start the Matrix process.
- **Matrix Service**: It runs on the device side and provides the following functions:
  - Implements media preprocessing by using the DVPP APIs called by engines.
  - Implements model inference by using the AIModelManger APIs called by engines.

You can view the API definition files in **ddk/include/inc/hiaiengine** under the installation directory of the device development kit (DDK). If you install both Mind Studio and the DDK by following the installation wizard, you can log in to the Mind Studio server as the Mind Studio installation user and view the API definition file in **~/tools/che/ddk/ddk/include/inc/hiaiengine**. For details about the mapping between the definition files and the APIs, see the API description.

2 Process Orchestration APIs

2.1 Process Connection APIs (C Language)
2.2 Process Connection APIs (C++ Language)
2.3 Data Sending APIs (C Language)
2.4 Data Sending APIs (C++ Language)
2.5 API Usage Example

2.1 Process Connection APIs (C Language)

2.1.1 HIAI_Init (Common with C++)

Initializes the host device communication (HDC) module of the Matrix. The device ID specified in the graph configuration file is preferred. Otherwise, use the value of **deviceID** configured below. This API is defined in **init.h**.

Syntax

HIAI_StatusT HIAI_Init(uint32_t deviceID)

NOTE

For details about the definition of the **HIAI_StatusT** structure, see **7.2.2 Matrix Custom Data Types**.

Parameter Description

Parameter	Description	Value Range
deviceID	Device ID. The default value is **0**.	0–63

Return Value

For details about the returned error codes, see "Error Codes."

Error Codes

No.	Error Code	Description
1	HIAI_OK	The running is OK.
2	HIAI_INIT_FAIL	HiAI initialization fails.

2.1.2 HIAI_CreateGraph

Creates and starts the entire graph. This API supports the connection between graphs in a single configuration file rather than in different configuration files. It can be called only on the host or Ascend 310 RC. This API is defined in **c_graph.h**.

Syntax

HIAI_StatusT HIAI_CreateGraph(const char* configFile, size_t len)

Parameter Description

Parameter	Description	Value Range
configFile	Configuration file path. Ensure that the transferred file path is correct.	A single configuration file supports a maximum of 2048 graphs. However, restricted by system resources, only several or dozens of graphs are supported based on different hardware configurations and graph sizes.
len	Name length of the configuration file	1–255

Return Value

For details about the returned error codes, see "Error Codes."

Error Codes

No.	Error Code	Description
1	HIAI_OK	The running is OK.

No.	Error Code	Description
2	HIAI_GRAPH_PROTO_FILE_PARSE_FAILED	ParseFromString return failed.

2.1.3 HIAI_DestroyGraph

Destroys a graph. It can be called only on the host or Ascend 310 RC. This API is defined in **c_graph.h**.

Syntax

HIAI_StatusT HIAI_DestroyGraph (uint32_t graphID)

Parameter Description

Parameter	Description	Value Range
graphID	Specified graph ID	-

Return Value

For details about the returned error codes, see "Error Codes."

Error Codes

No.	Error Code	Description
1	HIAI_OK	The running is OK.
2	HIAI_GRAPH_DESTROY_TIMEOUT	Destroying a graph timed out.
3	HIAI_GRAPH_ENGINE_DESTROY_FAILED	Destroying an engine failed.

2.2 Process Connection APIs (C++ Language)

2.2.1 Graph::GetInstance

Obtains a graph instance. After a graph is created, this API can be used. This API is defined in **graph.h**.

Syntax

static std::shared_ptr<Graph> Graph::GetInstance(uint32_t graphID)

Parameter Description

Parameter	Description	Value Range
graphID	ID of the target graph	Graph ID configured in the configuration file

Return Value

Smart pointer to the graph

Error Codes

No.	Error Code	Description
1	HIAI_OK	The running is OK.
2	HIAI_GRAPH_PROTO_FILE_PARSE_FAILED	ParseFromString return failed.

2.2.2 Graph::CreateGraph (Creating a Graph Based on the Configuration File)

Creates and starts the entire graph. This API supports the connection between graphs in a single configuration file rather than in different configuration files. This API is defined in **graph.h**.

It can be called only on the host or Ascend 310 RC.

Syntax

static HIAI_StatusT Graph::CreateGraph(const std::string& configFile)

Parameter Description

Parameter	Description	Value Range
configFile	Path of the graph configuration file. Ensure that the transferred file path is correct.	A single configuration file supports a maximum of 2048 graphs. However, restricted by system resources, only several or dozens of graphs are supported based on different hardware configurations and graph sizes.

Return Value

For details about the returned error codes, see "Error Codes."

Error Codes

No.	Error Code	Description
1	HIAI_OK	The running is OK.
2	HIAI_GRAPH_GET_INSTANCE_NULL	A null instance is obtained.

2.2.3 Graph::CreateGraph (Creating a Graph Based on the Configuration File and Writing the Generated Graph Back to the List)

Overloads **2.2.2 Graph::CreateGraph (Creating a Graph Based on the Configuration File)**. The Matrix creates a graph based on the configuration file and writes the generated graph back to the list. This API is defined in **graph.h**.

It can be called only on the host or Ascend 310 RC.

Syntax

static HIAI_StatusT Graph::CreateGraph(const std::string& configFile, std::list<std::shared_ptr<Graph>>& graphList)

Parameter Description

Parameter	Description	Value Range
configFile	Path of the graph configuration file. Ensure that the transferred file path is correct.	A single configuration file supports a maximum of 2048 graphs. However, restricted by system resources, only several or dozens of graphs are supported based on different hardware configurations and graph sizes.
graphList	User-defined list. The Matrix writes the generated graph back to the list.	-

Return Value

For details about the returned error codes, see "Error Codes."

Error Codes

No.	Error Code	Description
1	HIAI_OK	The running is OK.
2	HIAI_GRAPH_GET_INSTANCE_NULL	A null instance is obtained.

2.2.4 Graph::CreateGraph (Creating a Graph Based on the Protobuf Data Format)

Creates and starts the entire graph based on the Protobuf data format. This API is defined in **graph.h**.

It can be called only on the host or Ascend 310 RC.

Syntax

static HIAI_StatusT Graph::CreateGraph(const GraphConfigList& graphConfig)

Parameter Description

Parameter	Description	Value Range
graphConfig	Protobuf data format. When writing code, you can define the **graphConfig** parameter as follows: • Define a parameter of the **GraphConfigList** type as the input parameter **graphConfig** of the **Graph::CreateGraph(const GraphConfigList& graphConfig)** API. • After the API in **5.1.12 Parsing the Matrix Configuration File** is called, obtain the **graphConfigList** parameter and use it as the input parameter **graphConfig** of the **Graph::CreateGraph(const GraphConfigList& graphConfig)** API.	The number of engines in a graph must be less than or equal to 512.

Return Value

For details about the returned error codes, see "Error Codes."

Error Codes

No.	Error Code	Description
1	HIAI_OK	The running is OK.
2	HIAI_GRAPH_GET_INSTANCE_NULL	A null instance is obtained.

2.2.5 Graph::DestroyGraph

Destroys the entire graph. This API is defined in **graph.h**.

It can be called only on the host or Ascend 310 RC.

Syntax

static HIAI_StatusT Graph::DestroyGraph(uint32_t graphID)

Parameter Description

Parameter	Description	Value Range
graphID	ID of the graph to be destroyed	Graph ID configured in the configuration file

Return Value

For details about the returned error codes, see "Error Codes."

Error Codes

No.	Error Code	Description
1	HIAI_OK	The running is OK.
2	HIAI_GRAPH_DESTROY_TIMEOUT	Destroying a graph timed out.
3	HIAI_GRAPH_ENGINE_DESTROY_FAILED	Destroying a graph failed.

2.2.6 Graph::UpdateEngineConfig

Updates the parameters of a specified engine. This API is defined in **graph.h**.

Application scenario:

This API allows you to update the specified running parameters in the running process.

For example, in the video snapshot scenario, the running process needs to be updated based on the change of day and night. Therefore, you need to update the values for day and night to the engine by using the **UpdataEngineConfig** function.

Syntax

static HIAI_StatusT Graph::UpdateEngineConfig(const uint32_t& graphId, const uint32_t& engineId, const AIConfig& aiConfig, const bool& syncFlag = false)

Parameter Description

Parameter	Description	Value Range
graphId	Graph ID	-
engineId	engine ID	-
aiConfig	Configuration parameter	-
syncFlag	Synchronous or asynchronous execution. For synchronous execution, wait for the returned result.	**true**/**false**

Return Value

For details about the returned error codes, see "Error Codes."

Error Codes

No.	Error Code	Description
1	HIAI_OK	The running is OK.
2	HIAI_GRAPH_NOT_EXIST	The graph does not exist.
3	HIAI_GRAPH_MEMORY_POOL_NOT_EXISTED	The memory pool does not exist.
4	HIAI_GRAPH_MALLOC_LARGER	The buffer fails to be allocated because the size is larger than 256 MB.
5	HIAI_MEMORY_POOL_UPDATE_FAILED	The memory pool fails to be updated.
6	HIAI_GRAPH_SENDMSG_FAILED	The HDC module fails to send a message.
7	HIAI_GRAPH_MEMORY_POOL_INITED	The memory pool has been initialized.
8	HIAI_GRAPH_NO_MEMORY	There is no memory.

2.3 Data Sending APIs (C Language)

2.3.1 HIAI_SetDataRecvFunctor

Sets the callback function for an engine in the graph to receive messages. This API is defined in **c_graph.h**.

NOTE

Callback function type: **typedef HIAI_StatusT (*HIAI_RecvDataCallbackT)(void*)**

Syntax

HIAI_StatusT HIAI_SetDataRecvFunctor(HIAI_PortID_t targetPortConfig, HIAI_RecvDataCallbackT recvCallback)

Parameter Description

Parameter	Description	Value Range
targetPortConfig	Graph ID, engine ID, and port ID of the data receiver	-
recvCallback	Callback function for setting the mode of receiving the user-defined result	-

Return Value

For details about the returned error codes, see "Error Codes."

Error Codes

No.	Error Code	Description
1	HIAI_OK	The running is OK.
2	HIAI_PORT_ID_ERROR	The port ID has an error.

2.3.2 HIAI_C_SendData

Sends data or messages to the Matrix from an external system. This API is defined in **c_graph.h**.

NOTE

- The user-defined message type needs to be registered by calling the API in **5.2 Data Type Serialization and Deserialization (C++ Language)**.
- The message pointer **dataPtr** is controlled and released by the caller. The Matrix does not manage this pointer.
- The **SendData** interface uses the DMA transfer mode, which may affect the CPU's timely response to and processing of interrupt requests, for example, memory allocation by calling the **new** or **malloc** function.

Syntax

HIAI_StatusT HIAI_C_SendData(HIAI_PortID_t targetPortConfig, const char* messageName, size_t len, void* dataPtr)

Parameter Description

Parameter	Description	Value Range
targetPortConfig	Graph ID, engine ID, and port ID of the data receiver	-
messageName	Name of the message that is being sent	-
len	Length of a message name	-
dataPtr	Pointer to a specific message. The message pointer is controlled and released by the caller. The Matrix does not manage this pointer.	-

Return Value

For details about the returned error codes, see "Error Codes."

Error Codes

No.	Error Code	Description
1	HIAI_OK	The running is OK.
2	HIAI_GRAPH_ENGINE_NOT_EXIST	The engine does not exist.
3	HIAI_GRAPH_NOT_EXIST	The graph does not exist.
4	HIAI_GRAPH_NO_USEFUL_MEMORY	There is no useful memory.
5	HIAI_GRAPH_SENDMSG_FAILED	The HDC module fails to send a message.
6	HIAI_GRAPH_INVALID_VALUE	The graph has an invalid value.

2.4 Data Sending APIs (C++ Language)

2.4.1 Graph::SetDataRecvFunctor

Sets the callback function for an engine in the graph to receive messages. The running environment of the code that calls the API must be the same as that of

the engine. For example, if the engine runs on the device side, the code that calls the function must also run in the device side. This API is defined in **graph.h**.

This API, which must be used together with the **DataRecvInterface::RecvData** API, is generally used to return the inference result to the user. The detailed principles are as follows:

1. Define a subclass (for example, **DdkDataRecvInterface**) of class **DataRecvInterface** and initialize the object of the subclass.
2. Set the object of class **DdkDataRecvInterface** to a callback function by calling **Graph::SetDataRecvFunctor**.
3. Send the input data to the corresponding output port by calling **Engine::SendData**.
4. Return the inference result by calling the **RecvData** function of class **DdkDataRecvInterface**.

Syntax

static HIAI_StatusT Graph::SetDataRecvFunctor(const EnginePortID& targetPortConfig, const std::shared_ptr<DataRecvInterface>& dataRecv);

Parameter Description

Parameter	Description	Value Range
targetPortConfig	Graph ID, engine ID, and port ID of the data receiver	-
dataRecv	Callback function for receiving user-defined data	-

Return Value

For details about the returned error codes, see "Error Codes."

Error Codes

No.	Error Code	Description
1	HIAI_OK	The running is OK.
2	HIAI_PORT_ID_ERROR	The port ID has an error.

2.4.2 Graph::SendData

Sends **void** data from an external system to a specified port on the Matrix. This API is defined in **graph.h**.

NOTE

The **SendData** interface uses the DMA transfer mode, which may affect the CPU's timely response to and processing of interrupt requests, for example, memory allocation by calling the **new** or **malloc** function.

Syntax

HIAI_StatusT Graph::SendData(const EnginePortID& targetPortConfig, const std::string& messageName, const std::shared_ptr<void>& dataPtr, const uint32_t timeOut = 500)

Parameter Description

Parameter	Description	Value Range
targetPortConfig	Graph ID, engine ID, and port ID of the data receiver	-
messageName	Name of a message	-
dataPtr	Pointer to a message	-
timeOut	Timeout for calling this API to send data. The default timeout is 500 ms. If data sending times out, the system background attempts to send data again. A maximum of 16 attempts are allowed.	-

Return Value

For details about the returned error codes, see "Error Codes."

Error Codes

No.	Error Code	Description
1	HIAI_OK	The running is OK.
2	HIAI_GRAPH_ENGINE_NOT_EXIST	The engine does not exist.
3	HIAI_GRAPH_NOT_EXIST	The graph does not exist.
4	HIAI_GRAPH_NO_USEFUL_MEMORY	There is no useful memory.

No.	Error Code	Description
5	HIAI_GRAPH_SENDMSG_FAILED	The HDC module fails to send a message.
6	HIAI_GRAPH_INVALID_VALUE	The graph has an invalid value.

2.4.3 HIAI_SendData

Sends **void** data from an external system to a specified port on the Matrix. This API is defined in **c_graph.h**.

NOTE

The **SendData** interface uses the DMA transfer mode, which may affect the CPU's timely response to and processing of interrupt requests, for example, memory allocation by calling the **new** or **malloc** function.

Syntax

HIAI_StatusT HIAI_SendData (HIAI_PortID_t targetPortConfig, const std::string& messageName, std::shared_ptr<void>& dataPtr, uint32_t timeOut=0)

Parameter Description

Parameter	Description	Value Range
targetPortConfig	Graph ID, engine ID, and port ID of the data receiver	-
messageName	Name of a message	-
dataPtr	Pointer to a message	-
timeOut	Timeout for data sending, in ms. If the TX queue is full or the memory pool is insufficient, blocking is enabled based on the timeout configuration. The default value is **0**, indicating no blocking.	-

Return Value

For details about the returned error codes, see "Error Codes."

Error Codes

No.	Error Code	Description
1	HIAI_OK	The running is OK.
2	HIAI_GRAPH_ENGINE_NOT_EXIST	The engine does not exist.
3	HIAI_GRAPH_NOT_EXIST	The graph does not exist.
4	HIAI_GRAPH_NO_USEFUL_MEMORY	There is no useful memory.
5	HIAI_GRAPH_SENDMSG_FAILED	The HDC module fails to send a message.
6	HIAI_GRAPH_INVALID_VALUE	The graph has an invalid value.

2.5 API Usage Example

2.5.1 Example of Using the C APIs

```
/**
* @file graph_c_api_example.cpp
*
* Copyright(C), 2017 - 2017, Huawei Tech. Co., Ltd. ALL RIGHTS RESERVED.
*
* @Source Files for HIAI Graph Orchestration
*
* @version 1.0
*
*/
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

#include "hiaiengine/c_api.h"
#include "user_def_data_type.h"

/*****************************************************************************
* Normally the mandatory steps to set up the whole HIAI graph include:
*  1) Call the HIAI_Init() API to perform global system initialization;
*  2) Call HIAI_CreateGraph to create and start a graph.
*  3) Set profile config if necessary.
*  4) Call HIAI_DestoryGraph to destroy the graph finally.
*
*****************************************************************************/


// Just define variables which are used in only this example
// and are not required in real user case.
#define MATRIX_USER_SPECIFY_ENGINE_PRIORITY (10)

static uint32_t g_graph_id = 100;
static uint32_t src_engine_id = 1000; // Defined in graph.prototxt
static uint32_t dest_engine_id = 1002; // Defined in graph.prototxt
static const char graph_config_proto_file[] = "llt/hiaiengine/st/examples/config/graph.prototxt";
static const char graph_config_len = sizeof(graph_config_proto_file);

// Define End
```

```
// Receive user-defined data (recv_data is automatically released by the framework).
HIAI_StatusT UserDefDataRecvCallBack(void* recv_data)
{
// Process the received data.
    return HIAI_OK;
}


/**************************************************************************
* HIAIEngine Example
* Graph:
* SrcEngine<-->Engine<-->DestEngine
*
**************************************************************************/
HIAI_StatusT HIAI_InitAndStartGraph()
{
    // Step1: Global System Initialization before using Matrix
    HIAI_StatusT status = HIAI_Init(0);

    // Step2: Create and Start the Graph
    status = HIAI_CreateGraph(graph_config_proto_file, graph_config_len);
    if (status != HIAI_OK)
    {
        return status;
    }

    // Step3: Set the callback function to receive data.
    HIAI_PortID_t engine_id;
    engine_id.graph_id = g_graph_id;
    engine_id.engine_id = dest_engine_id;
    engine_id.port_id = 0;
    HIAI_SetDataRecvFunctor(engine_id, UserDefDataRecvCallBack);

    return HIAI_OK;
}


// User Application Main Example
int main()
{
    HIAI_StatusT ret = HIAI_OK;

    // Creation process
    ret = HIAI_InitAndStartGraph();

    if (HIAI_OK != ret)
    {
        return -1;
    }

    // Read data and inject data into the process.
    HIAI_PortID_t engine_id;
    engine_id.graph_id = g_graph_id;
    engine_id.engine_id = src_engine_id;
    engine_id.port_id = 0;

    int is_end_of_data = 0;
    while (!is_end_of_data)
    {
        UserDefDataTypeT* user_def_msg = (UserDefDataTypeT*)malloc(sizeof(UserDefDataTypeT));

        // Read data and pad UserDefDataType. For details about the data type registration, see the
corresponding section in this document.

        const char * message_name = "UserDefDataType";
        size_t msg_len = strlen(message_name);
        if (HIAI_OK != HIAI_C_SendData(engine_id, message_name, msg_len, user_def_msg))
        {
```



```
            // When the queue is full, a message is returned indicating that sending data fails. The service logic
determines whether to resend or discard the data.
            break;
        }

        free(user_def_msg);
        user_def_msg = nullptr;
    }

   // After the processing is complete, delete the entire graph.
    HIAI_DestroyGraph(g_graph_id);
    return 0;
}
```

2.5.2 Example of Using C++ APIs (Creating a Graph Based on the Configuration File)

```
/**
* @file graph_cplusplus_api_example.cpp
*
* Copyright(C), 2017 - 2017, Huawei Tech. Co., Ltd. ALL RIGHTS RESERVED.
*
* @Source Files for HIAI Graph Orchestration
*
* @version 1.0
*
*/
#include <unistd.h>
#include <thread>
#include <fstream>
#include <algorithm>

#include "hiaiengine/api.h"
#include "user_def_data_type.h"

/**************************************************************************
* Normally the mandatory steps to set up the whole HIAI graph include:
*  1) Call the HIAI_Init() API to perform global system initialization;
*  2) Call HIAI_CreateGraph or Graph::CreateGraph to create and start a graph.
*  3) Set profile config if necessary.
*  4) Call HIAI_DestoryGraph or Graph::DestroyGraph to destroy the graph finally.
*
**************************************************************************/


// Just define variables which are used in only this example
// and are not required in real user case.
#define MATRIX_USER_SPECIFY_ENGINE_PRIORITY (10)

static uint32_t g_graph_id = 100;
static uint32_t src_engine_id = 1000; // Defined in graph.prototxt
static uint32_t dest_engine_id = 1002; // Defined in graph.prototxt
static std::string graph_config_proto_file = "./config/graph.prototxt";
// Define End

namespace hiai {

class GraphDataRecvInterface: public DataRecvInterface
{
public:
    GraphDataRecvInterface()
    {
    }

    /**
    * @ingroup hiaiengine
    * @brief   Read and save the data.
    * @param [in] in_data   Input data
```

```
    * @return HIAI Status
    */
    HIAI_StatusT RecvData(const std::shared_ptr<void>& message)
    {
        // Convert the data to a specific message type and process the related message. For example:
        // shared_ptr<std::string> data =
        // std::static_pointer_cast<std::string>(message);
        return HIAI_OK;
    }
private:
};
}

/**************************************************************************
* HIAIEngine Example
* Graph:
* SrcEngine<-->Engine<-->DestEngine
*
**************************************************************************/
HIAI_StatusT HIAI_InitAndStartGraph()
{
    // Step1: Global System Initialization before using Matrix
    HIAI_StatusT status = HIAI_Init(0);

    // Step2: Create and Start the Graph
    status = hiai::Graph::CreateGraph(graph_config_proto_file);
    if (status != HIAI_OK)
    {
        HIAI_ENGINE_LOG(status, "Fail to start graph");
        return status;
    }

    // Step3: Set the callback function to receive data.
    std::shared_ptr<hiai::Graph> graph = hiai::Graph::GetInstance(g_graph_id);
    if (nullptr == graph)
    {
        HIAI_ENGINE_LOG("Fail to get the graph-%u", g_graph_id);
        return status;
    }

    // Specify the port id (default to zero)
    hiai::EnginePortID target_port_config;
    target_port_config.graph_id = g_graph_id;
    target_port_config.engine_id = dest_engine_id;
    target_port_config.port_id = 0;

    graph->SetDataRecvFunctor(target_port_config,
        std::shared_ptr<hiai::GraphDataRecvInterface>(
        new hiai::GraphDataRecvInterface()));

    return HIAI_OK;
}


// User Application Main Example
int main()
{
    HIAI_StatusT ret = HIAI_OK;

    // Creation process
    ret = HIAI_InitAndStartGraph();

    if(HIAI_OK != ret)
    {
        HIAI_ENGINE_LOG("Fail to start graph");;
        return -1;
    }

    // Read data and inject data into the process.
```

```
    std::shared_ptr<hiai::Graph> graph = hiai::Graph::GetInstance(g_graph_id);
    if (nullptr == graph)
    {
        HIAI_ENGINE_LOG("Fail to get the graph-%u", g_graph_id);
        return -1;
    }

    hiai::EnginePortID engine_id;
    engine_id.graph_id = g_graph_id;
    engine_id.engine_id = src_engine_id;
    engine_id.port_id = 0;

    bool is_end_of_data = false;
    while (!is_end_of_data)
    {
        std::shared_ptr<UserDefDataType> user_def_msg(new UserDefDataType);

        // Read data and pad UserDefDataType. For details about the data type registration, see the
corresponding section in this document.
       // Inject data to the graph.
        if (HIAI_OK != graph->SendData(engine_id, "UserDefDataType",
            std::static_pointer_cast<void>(user_def_msg)))
        {
            // When the queue is full, a message is returned indicating that sending data fails. The service logic
determines whether to resend or discard the data.
            HIAI_ENGINE_LOG("Fail to send data to the graph-%u", g_graph_id);
            break;
        }
    }

   // After the processing is complete, delete the entire graph.
    hiai::Graph::DestroyGraph(g_graph_id);
    return 0;
}
```

2.5.3 Usage Example for C++ APIs (Creating a Graph Based on the Configuration File and Writing the Generated Graph to the List)

```
/**
* @file graph_cplusplus_api_example.cpp
*
* Copyright(C), 2017 - 2017, Huawei Tech. Co., Ltd. ALL RIGHTS RESERVED.
*
* @Source Files for HIAI Graph Orchestration
*
* @version 1.0
*
*/
#include <unistd.h>
#include <thread>
#include <fstream>
#include <algorithm>
#include "hiaiengine/api.h"
#include "user_def_data_type.h"
/**************************************************************************
* Normally the mandatory steps to setup the whole HIAI graph include:
*  1) Call the HIAI_Init() API to perform global system initialization;
*  2) Call HIAI_CreateGraph or Graph::CreateGraph to create and start a graph.
*  3) Set profile config if necessary.
*  4) Call HIAI_DestoryGraph or Graph::DestroyGraph to destroy the graph finally.
*
**************************************************************************/
// Just define variables which are used in only this example
// and are not required in real user case.
#define MATRIX_USER_SPECIFY_ENGINE_PRIORITY (10)
```

```
static uint32_t src_engine_id = 1000; // Defined in graph.prototxt
static uint32_t dest_engine_id = 1002; // Defined in graph.prototxt
static std::string graph_config_proto_file = "./config/graph.prototxt";
// Define End
namespace hiai {
class GraphDataRecvInterface: public DataRecvInterface
{
public:
    GraphDataRecvInterface()
    {
    }
    /**
    * @ingroup hiaiengine
    * @brief   Read and save the data.
    * @param [in] in_data   Input data
    * @return HIAI Status
    */
    HIAI_StatusT RecvData(const std::shared_ptr<void>& message)
    {
        // Convert the data to a specific message type and process the related message. For example:
        // shared_ptr<std::string> data =
        // std::static_pointer_cast<std::string>(message);
        return HIAI_OK;
    }
private:
};
}
/**************************************************************************
* Matrix Example
* Graph:
* SrcEngine<-->Engine<-->DestEngine
*
**************************************************************************/
HIAI_StatusT HIAI_InitAndStartGraph(std::list<std::shared_ptr<hiai::Graph>>& graphList)
{
    // Step1: Global System Initialization before using Matrix
    HIAI_StatusT status = HIAI_Init(0);
    // Step2: Create and Start the Graph
    status = hiai::Graph::CreateGraph(graph_config_proto_file, graphList);
    if (status != HIAI_OK)
    {
        HIAI_ENGINE_LOG(status, "Fail to start graph");
        return status;
    }
    // Step3: Set the callback function to receive data.
   // If multiple graphs are configured in a configuration file, you can traverse the graph list to obtain the
corresponding graph and perform the expected operation.
    std::list<std::shared_ptr<hiai::Graph>>::iterator iter = graphList.begin();
    std::shared_ptr<hiai::Graph> graph = hiai::Graph::GetInstance((*iter)->GetGraphId());
    if (nullptr == graph)
    {
        HIAI_ENGINE_LOG("Fail to get the graph");
        return status;
    }
    // Specify the port id (default to zero)
    hiai::EnginePortID target_port_config;
    target_port_config.graph_id = (*iter)->GetGraphId();
    target_port_config.engine_id = dest_engine_id;
    target_port_config.port_id = 0;
    graph->SetDataRecvFunctor(target_port_config,
        std::shared_ptr<hiai::GraphDataRecvInterface>(
        new hiai::GraphDataRecvInterface()));
    return HIAI_OK;
}
// User Application Main Example
int main()
{
    HIAI_StatusT ret = HIAI_OK;
    // Store the graphs created by the user.
```



```
    std::list<std::shared_ptr<hiai::Graph>> graphList;
    // Creation process
    ret = HIAI_InitAndStartGraph(graphList);
    if(HIAI_OK != ret)
    {
        HIAI_ENGINE_LOG("Fail to start graph");;
        return -1;
    }
    // Read data and inject data into the process.
    std::list<std::shared_ptr<hiai::Graph>>::iterator iter = graphList.begin();
    std::shared_ptr<hiai::Graph> graph = hiai::Graph::GetInstance((*iter)->GetGraphId());
    if (nullptr == graph)
    {
        HIAI_ENGINE_LOG("Fail to get the graph");
        return -1;
    }
    hiai::EnginePortID engine_id;
    engine_id.graph_id = (*iter)->GetGraphId();
    engine_id.engine_id = src_engine_id;
    engine_id.port_id = 0;
    bool is_end_of_data = false;
    while (!is_end_of_data)
    {
        std::shared_ptr<UserDefDataType> user_def_msg(new UserDefDataType);
        // Read data and pad UserDefDataType. For details about the data type registration, see the
corresponding section in this document.
       // Inject data to the graph.
        if (HIAI_OK != graph->SendData(engine_id, "UserDefDataType",
            std::static_pointer_cast<void>(user_def_msg)))
        {
            // When the queue is full, a message is returned indicating that sending data fails. The service logic
determines whether to resend or discard the data.
            HIAI_ENGINE_LOG("Fail to send data to the graph-%u", (*iter)->GetGraphId());
            break;
        }
    }
   // After the processing is complete, delete all graphs.
    std::list<std::shared_ptr<hiai::Graph>>::iterator graphIter;
    for (graphIter = graphList.begin(); graphIter != graphList.end(); ++graphIter) {
        hiai::Graph::DestroyGraph((*graphIter)->GetGraphId());
    }
    return 0;
}
```


3 Engine Implementation APIs (C++ Language)

3.1 Constructor and Destructor Functions of an Engine
3.2 Engine::Init
3.3 Macro: HIAI_DEFINE_PROCESS
3.4 Macro: HIAI_IMPL_ENGINE_PROCESS
3.5 Engine::SetDataRecvFunctor
3.6 Engine::SendData
3.7 Calling Example

3.1 Constructor and Destructor Functions of an Engine

This API is defined in **engine.h**.

Syntax

Engine()

virtual ~Engine()

NOTE

The constructor and destructor functions are optional. You can determine whether to overload and implement them based on the actual situation.

3.2 Engine::Init

Initializes the configuration of an engine instance. This API is defined in **engine.h**.

NOTE

The **Init** API is optional. You can determine whether to overload and implement it based on the actual situation.

Syntax

HIAI_StatusT Engine::Init(const AIConfig &config, const vector<AIModelDescription> &modelDesc)

Parameter Description

Parameter	Description	Value Range
config	Engine configuration item	-
modelDesc	Model description	-

Return Value

The returned error codes are registered by the user in advance. For details about the registration method, see **5.5.1 Error Code Registration**.

Error Codes

The error codes of this API are registered by the user.

Example of Overload and Implementation

Initialize the inference engine. During the initialization, load the model through the API of the model manager (AIModelManager), as shown in **4.1.1 AIModelManager::Init**.

```
HIAI_StatusT FrameworkerEngine::Init(const hiai::AIConfig& config,
   const  std::vector<hiai::AIModelDescription>& model_desc)
{
    hiai::AIStatus ret = hiai::SUCCESS;
    // init ai_model_manager_
    if (nullptr == ai_model_manager_)
    {
        ai_model_manager_ = std::make_shared<hiai::AIModelManager>();
    }
    std::cout<<"FrameworkerEngine Init"<<std::endl;
    HIAI_ENGINE_LOG("FrameworkerEngine Init");

    for (int index = 0; index < config.items_size(); ++index)
    {

        const ::hiai::AIConfigItem& item = config.items(index);
        // loading model
        if(item.name() == "model_path")
        {
            const char* model_path = item.value().data();
            std::vector<hiai::AIModelDescription> model_desc_vec;
            hiai::AIModelDescription model_desc_;
            model_desc_.set_path(model_path);
            model_desc_vec.push_back(model_desc_);
            ret = ai_model_manager_->Init(config, model_desc_vec);

            if (hiai::SUCCESS != ret)
            {
                HIAI_ENGINE_LOG(this, HIAI_AI_MODEL_MANAGER_INIT_FAIL, "[DEBUG] fail to init ai_model");
                return HIAI_AI_MODEL_MANAGER_INIT_FAIL;
```



```
        }
      }
    }
    HIAI_ENGINE_LOG("FrameworkerEngine Init success");
    return HIAI_OK;
}
```

3.3 Macro: HIAI_DEFINE_PROCESS

You can directly call the number of input and output ports of the engine in the macro definition. This macro is defined in **engine.h**.

This macro encapsulates the following function:

```
HIAI_StatusT Engine::InitQueue(const uint32_t& in_port_num, const uint32_t& out_port_num);
```

Related macro:

This macro is called before **HIAI_IMPL_ENGINE_PROCESS(name, engineClass, inPortNum)**.

Syntax

HIAI_DEFINE_PROCESS (inputPortNum, outputPortNum)

Parameter Description

Parameter	Description	Value Range
inputPortNum	Number of input ports of an engine	-
outputPortNum	Number of output ports of an engine	-

Calling Example

```
#define FRAMEWORK_ENGINE_INPUT_SIZE  1
#define FRAMEWORK_ENGINE_OUTPUT_SIZE  1

* @[in]: Defines an input port and an output port. */
HIAI_DEFINE_PROCESS(FRAMEWORK_ENGINE_INPUT_SIZE, FRAMEWORK_ENGINE_OUTPUT_SIZE)
```

3.4 Macro: HIAI_IMPL_ENGINE_PROCESS

You need to overload and implement this macro to define the implementation of the engine. This macro is defined in **engine.h**.

This macro encapsulates the following functions:

```
static HIAIEngineFactory* GetInstance();
HIAI_StatusT HIAIEngineFactory::RegisterEngineCreator(const std::string&
engine_name,HIAI_ENGINE_FUNCTOR_CREATOR engineCreatorFunc);
HIAI_StatusT HIAIEngineFactory::UnRegisterEngineCreator(const std::string& engine_name);
```

Related macro:

This macro is called after **HIAI_DEFINE_PROCESS (inputPortNum, outputPortNum)**.

Syntax

HIAI_IMPL_ENGINE_PROCESS(name, engineClass, inPortNum)

Parameter Description

Parameter	Description	Value Range
name	Engine name in the configuration	-
engineClass	Name of the engine implementation class	-
inPortNum	Number of input ports	-

Return Value

The returned error codes are registered by the user in advance.

Error Codes

The error codes of this API are registered by the user.

Example of Overload and Implementation

Define the implementation of the inference engine. During implementation, the model inference is executed through the API of the model manager (AIModelManager), as shown in **4.1.3 AIModelManager::Process**.

```
HIAI_IMPL_ENGINE_PROCESS("FrameworkerEngine", FrameworkerEngine,
FRAMEWORK_ENGINE_INPUT_SIZE)
{
    hiai::AIStatus ret = hiai::SUCCESS;
    HIAI_StatusT hiai_ret = HIAI_OK;
    // receive data
   // arg0 indicates the input port (numbered 0) of the engine. If there are multiple input ports, you can use
arg1 (numbered 1) and arg2 (numbered 2) to match the input ports. The data sent by the previous engine
is obtained through the input port.
    std::shared_ptr<std::string> input_arg =
        std::static_pointer_cast<std::string>(arg0);
    if (nullptr == input_arg)
    {
        HIAI_ENGINE_LOG(this, HIAI_INVALID_INPUT_MSG, "[DEBUG] input arg is invalid");
        return HIAI_INVALID_INPUT_MSG;
    }
    std::cout<<"FrameworkerEngine Process"<<std::endl;

    //  prapare for calling the process of ai_model_manager_
    std::vector<std::shared_ptr<hiai::IAITensor>> input_data_vec;

    uint32_t len = 75264;

    HIAI_ENGINE_LOG("FrameworkerEngine:Go to Process");
    std::cout << "HIAIAippOp::Go to process" << std::endl;
```



```
    std::shared_ptr<hiai::AINeuralNetworkBuffer> neural_buffer =
std::shared_ptr<hiai::AINeuralNetworkBuffer>(new hiai::AINeuralNetworkBuffer());//
std::static_pointer_cast<hiai::AINeuralNetworkBuffer>(input_data);
    neural_buffer->SetBuffer((void*)(input_arg->c_str()), (uint32_t)(len));
    std::shared_ptr<hiai::IAITensor> input_data = std::static_pointer_cast<hiai::IAITensor>(neural_buffer);
    input_data_vec.push_back(input_data);

    //  call Process and inference
    hiai::AIContext ai_context;
    std::vector<std::shared_ptr<hiai::IAITensor>> output_data_vec;
    ret = ai_model_manager_->CreateOutputTensor(input_data_vec, output_data_vec);
    if (hiai::SUCCESS != ret)
    {
        HIAI_ENGINE_LOG(this, HIAI_AI_MODEL_MANAGER_PROCESS_FAIL, "[DEBUG] fail to process
ai_model");
        return HIAI_AI_MODEL_MANAGER_PROCESS_FAIL;
    }

    ret = ai_model_manager_->Process(ai_context, input_data_vec, output_data_vec, 0);

    if (hiai::SUCCESS != ret)
    {
        HIAI_ENGINE_LOG(this, HIAI_AI_MODEL_MANAGER_PROCESS_FAIL, "[DEBUG] fail to process
ai_model");
        return HIAI_AI_MODEL_MANAGER_PROCESS_FAIL;
    }
    std::cout<<"[DEBUG] output_data_vec size is "<< output_data_vec.size()<<std::endl;
    for (uint32_t index = 0; index < output_data_vec.size(); index++)
    {
        // send data of inference to destEngine
        std::shared_ptr<hiai::AINeuralNetworkBuffer> output_data =
std::static_pointer_cast<hiai::AINeuralNetworkBuffer>(output_data_vec[index]);
        std::shared_ptr<std::string> output_string_ptr = std::shared_ptr<std::string>(new
std::string((char*)output_data->GetBuffer(), output_data->GetSize()));

        hiai_ret = SendData(0, "string", std::static_pointer_cast<void>(output_string_ptr));
        if (HIAI_OK != hiai_ret)
        {
            HIAI_ENGINE_LOG(this, HIAI_SEND_DATA_FAIL, "fail to send data");
            return HIAI_SEND_DATA_FAIL;
        }
    }
    return HIAI_OK;
}
```

3.5 Engine::SetDataRecvFunctor

Sets the callback function for an engine to receive messages. This API is defined in **engine.h**.

It must be used together with the **DataRecvInterface::RecvData** API. The detailed principles are as follows:

1. Define a subclass (for example, **DdkDataRecvInterface**) of class **DataRecvInterface** and initialize the object of the subclass.
2. Set the object of class **DdkDataRecvInterface** to a callback function by calling **Engine::SetDataRecvFunctor**.
3. Send the input data to the corresponding output port by calling **Engine::SendData**.
4. Return data by calling the **RecvData** function of class **DdkDataRecvInterface**.

Syntax

HIAI_StatusT Engine::SetDataRecvFunctor(const uint32_t portId, const shared_ptr<DataRecvInterface>& userDefineDataRecv)

Parameter Description

Parameter	Description	Value Range
portId	Port ID	-
userDefineDataRecv	Callback function for receiving user-defined data	-

Return Value

For details about the returned error codes, see "Error Codes."

Error Codes

No.	Error Code	Description
1	HIAI_OK	The running is OK.
2	HIAI_PORT_ID_ERROR	The port ID has an error.

3.6 Engine::SendData

Sends data from the engine to the specified port_id. This API is defined in **engine.h**.

NOTE

The **SendData** interface uses the DMA transfer mode, which may affect the CPU's timely response to and processing of interrupt requests, for example, memory allocation by calling the **new** or **malloc** function.

Syntax

HIAI_StatusT Engine::SendData(uint32_t portId, const std::string& messageName, const shared_ptr<void>& dataPtr, uint32_t timeOut = TIME_OUT_VALUE)

Parameter Description

Parameter	Description	Value Range
portId	ID of the output port of the engine	-

Parameter	Description	Value Range
messageName	Name of the message that is being sent. The message must be registered in advance by calling the macro provided by HiAI.	-
dataPtr	Pointer that points to a specific message	-
timeOut	Timeout for calling this API to send data. The default timeout is 500 ms. If data sending times out, the system background attempts to send data again. A maximum of 16 attempts are allowed.	-

Return Value

For details about the returned error codes, see "Error Codes."

Error Codes

No.	Error Code	Description
1	HIAI_OK	The running is OK.
2	HIAI_ENGINE_NULL_POINTER	The pointer is null.
3	HIAI_GRAPH_ENGINE_NOT_EXIST	The engine does not exist.
4	HIAI_GRAPH_SRC_PORT_NOT_EXIST	The source port does not exist.

3.7 Calling Example

```
/**
* @file multi_input_output_engine_example.h
*
* Copyright(c)<2018>, <Huawei Technologies Co.,Ltd>
*
* @version 1.0
*
* @date 2018-4-25
*/
```

```
#ifndef MULTI_INPUT_OUTPUT_ENGINE_EXAMPLE_H_
#define MULTI_INPUT_OUTPUT_ENGINE_EXAMPLE_H_

#include "hiaiengine/engine.h"
#include "hiaiengine/data_type.h"
#include "hiaiengine/multitype_queue.h"

#define MULTI_INPUT_OUTPUT_ENGINE_EXAMPLE_INPUT_SIZE 3
#define MULTI_INPUT_OUTPUT_ENGINE_EXAMPLE_OUTPUT_SIZE 2

namespace hiai {
// Define New Engine
class HIAIMultiEngineExample : public Engine {

public:
    HIAIMultiEngineExample() :
        input_que_(MULTI_INPUT_OUTPUT_ENGINE_EXAMPLE_INPUT_SIZE) {}

    // Overload the Init of the parent engine.
    HIAI_StatusT Init(const AIConfig& config,
        const std::vector<AIModelDescription>& model_desc)
    {
        return HIAI_OK;
    }

    /**
    * @ingroup hiaiengine
    * @brief HIAI_DEFINE_PROCESS
    * @[in]: Defines an input port and an output port.
    */
    HIAI_DEFINE_PROCESS(MULTI_INPUT_OUTPUT_ENGINE_EXAMPLE_INPUT_SIZE,
MULTI_INPUT_OUTPUT_ENGINE_EXAMPLE_OUTPUT_SIZE)
private:

   // Implement a member variable in a private manner to cache the input queue.
    hiai::MultiTypeQueue input_que_;
};

}

#endif //MULTI_INPUT_OUTPUT_ENGINE_EXAMPLE_H_
```

```
/**
* @file multi_input_output_engine_example.h
*
* Copyright(c)<2018>, <Huawei Technologies Co.,Ltd>
*
* @version 1.0
*
* @date 2018-4-25
*/

#include "multi_input_output_engine_example.h"
#include "use_def_data_type.h"
#include "use_def_errorcode.h"

namespace hiai {

/**
* @ingroup hiaiengine
* @brief HIAI_DEFINE_PROCESS: implements the input and output processing flow of multiple ports.
* @[in]: Defines an input port and an output port,
*        The engine is registered and named "HIAIMultiEngineExample".
*/
HIAI_IMPL_ENGINE_PROCESS("HIAIMultiEngineExample", HIAIMultiEngineExample,
MULTI_INPUT_OUTPUT_ENGINE_EXAMPLE_INPUT_SIZE)
{
    // This Engine has three input args and two output
```

```
  // Each port may receive data at different time points. Therefore, some input ports may be empty.
   input_que_.PushData(0,arg0);
   input_que_.PushData(1,arg1);
   input_que_.PushData(2,arg2);

   std::shared_ptr<void> input_arg1;
   std::shared_ptr<void> input_arg2;
   std::shared_ptr<void> input_arg3;

  // The subsequent processing is performed only when all the three ports have input data. Method 1:
   if (!input_que_.PopAllData(input_arg1, input_arg2, input_arg3))
   {
       HIAI_ENGINE_LOG(this, HIAI_INVALID_INPUT_MSG, "fail to process");
       return HIAI_INVALID_INPUT_MSG;
   }

  // The subsequent processing is performed only when all the three ports have input data. Method 2:
/*
   if (!(input_que_.FrontData(0, input_arg1) && input_que_.FrontData(1, input_arg2) &&
input_que_.FrontData(2, input_arg3)))
   {
       HIAI_ENGINE_LOG(this, HIAI_INVALID_INPUT_MSG, "fail to process");
       return HIAI_INVALID_INPUT_MSG;
   } else {
        input_que_.PopData(0, input_arg1);
        input_que_.PopData(1, input_arg2);
        input_que_.PopData(2, input_arg3);
   }
*/
    //
  // Intermediate service logic can directly call the DVPP API or AIModelManger API for processing.
    //

  // Allocate memory to save results.
   std::shared_ptr<UseDefDataTypeT> output1 =
       std::make_shared<UseDefDataTypeT>();

   std::shared_ptr<UseDefTemplateDataType<uint64_t, uint64_t, uint64_t>> output2 =
       std::make_shared<UseDefTemplateDataType<uint64_t, uint64_t, uint64_t>>();

  // Fill in the output data structure, for example, assign a value.
   *output1 = *input_arg2;
   *output2 = *input_arg3;

  // Send data to output port 0.
   hiai::Engine::SendData(0, "UseDefDataTypeT", std::static_pointer_cast<void>(output1));

  // Send data to output port 1.
   hiai::Engine::SendData(1, "UseDefTemplateDataType_uint64_t_uint64_t_uint64_t",
       std::static_pointer_cast<void>(output2));

   return HIAI_OK;
}

}
```

4 Model Manager APIs (C++ Language)

4.1 Offline Model Manager

4.2 AIPP Configuration APIs

4.3 Data Types

4.4 Other Compilation Dependent APIs

4.5 Exception Handling

4.1 Offline Model Manager

The APIs marked by the following macros are not implemented in the current version and cannot be directly called: __ANDROID__, ANDROID, and __LITE__.

4.1.1 AIModelManager::Init

Initializes the model manager to load a model.

Syntax

virtual AIStatus AIModelManager::Init(const AIConfig &config, const std::vector<AIModelDescription> &model_descs = {}) override;

Parameter Description

Parameter	Description	Value Range
config	Configuration information For details about the syntax of the AIConfig data type, see **4.3.2 AIConfig**.	-

Parameter	Description	Value Range
model_descs	Model description list. A model can be loaded from the memory (prior) or file. For details about the syntax of the AIModelDescription data type, see **4.3.6 AIModelDescription**.	-

Return Value

SUCCESS indicates that initialization succeeds, while **FAILED** indicates that initialization fails.

Calling Example

Note that the value of **MODEL_NAME** can contain only uppercase letters, lowercase letters, digits, underscores (_), and dots (.). The value of **MODEL_PATH** can contain only uppercase letters, lowercase letters, digits, and underscores (_).

- Load a model from a file:

```
AIModelManager model_mngr;
AIModelDescription model_desc;
AIConfig config;
/* Check whether the input file path is valid.
   Path: The value can contain uppercase letters, lowercase letters, digits, and underscores (_).
   File name: The value can contain uppercase letters, lowercase letters, digits, underscores (_), and dots (.). */
model_desc.set_path(MODEL_PATH);
model_desc.set_name(MODEL_NAME);
model_desc.set_type(0);
vector<AIModelDescription> model_descs;
model_descs.push_back(model_desc);
// AIModelManager Init
AIStatus ret = model_mngr.Init(config, model_descs);
if (SUCCESS != ret)
{
    printf("AIModelManager Init failed. ret = %d\n", ret);
    return -1;
}
```

- Load a model from memory:

```
AIModelManager model_mngr;
AIModelDescription model_desc;
vector<AIModelDescription> model_descs;
AIConfig config;
model_desc.set_name(MODEL_NAME);
model_desc.set_type(0);
char *model_data = nullptr;
uint32_t model_size = 0;
ASSERT_EQ(true, Utils::ReadFile(MODEL_PATH.c_str(), model_data, model_size));
model_desc.set_data(model_data,model_size);
model_desc.set_size(model_size);
AIStatus ret = model_mngr.Init(config, model_descs);
if (SUCCESS != ret)
{
    printf("AIModelManager Init failed. ret = %d\n", ret);
```



```
        return -1;
    }
}
```

NOTE

The value of **model_size** must be the same as the actual size of the model.

4.1.2 AIModelManager::SetListener

Sets the model manager.

NOTE

If this API is not called or **listener** is set to **nullptr** when this API is called, the **Process** interface is called synchronously. Otherwise, the **Process** interface is called asynchronously.

Syntax

virtual AIStatus AIModelManager::SetListener(std::shared_ptr<IAIListener> listener) override;

Parameter Description

Parameter	Description	Value Range
listener	Callback function For details about the syntax of the IAIListener data type, see **4.3.19 IAIListener**.	-

Return Value

SUCCESS indicates that initialization succeeds, while **FAILED** indicates that initialization fails.

4.1.3 AIModelManager::Process

Processes input data with a model. When inference is performed on a single model, the buffer queue length of the process is limited to 2048 bytes. If the buffer queue length exceeds 2048 bytes, a failure message is returned.

Syntax

virtual AIStatus AIModelManager::Process(AIContext &context, const std::vector<std::shared_ptr<IAITensor>> &in_data, std::vector<std::shared_ptr<IAITensor>> &out_data, uint32_t timeout);

Parameter Description

Parameter	Description	Value Range
context	Context information, including the configurations of variable parameters when the engine is running For details about the syntax of the AIContext data type, see **4.3.17 AIContext**.	-
in_data	List of input tensors for a model For details about the syntax of the IAITensor data type, see **4.3.20 IAITensor**.	-
out_data	List of output tensors for a model For details about the syntax of the IAITensor data type, see **4.3.20 IAITensor**.	-
timeout	Calculation timeout. This parameter is reserved. The default value is **0**, indicating that the configuration is invalid.	-

Return Value

SUCCESS indicates that initialization succeeds, while **FAILED** indicates that initialization fails.

4.1.4 AIModelManager::CreateOutputTensor

Creates a list of output tensors.

NOTE

If you use this API to create tensors, the memory start address returned by this API must be 512-byte aligned. You are advised to use the AITensorFactory::CreateTensor interface to create a tensor. For details, see **4.1.10 AITensorFactory::CreateTensor**.

Syntax

virtual AIStatus AIModelManager::CreateOutputTensor(const std::vector<std::shared_ptr<IAITensor>> &in_data, std::vector<std::shared_ptr<IAITensor>> &out_data) override;

Parameter Description

Parameter	Description	Value Range
in_data	List of input tensors For details about the syntax of the IAITensor data type, see **4.3.20 IAITensor**.	-
out_data	List of output tensors For details about the syntax of the IAITensor data type, see **4.3.20 IAITensor**.	-

Return Value

SUCCESS indicates that initialization succeeds, while **FAILED** indicates that initialization fails.

4.1.5 AIModelManager::CreateInputTensor

Creates a list of input tensors.

NOTE

If you use this API to create tensors, the memory start address returned by this API must be 512-byte aligned. You are advised to use the AITensorFactory::CreateTensor interface to create a tensor. For details, see **4.1.10 AITensorFactory::CreateTensor**.

Syntax

virtual AIStatus AIModelManager::CreateInputTensor(std::vector<std::shared_ptr<IAITensor>> &in_data);

Parameter Description

Parameter	Description	Value Range
in_data	List of input tensors For details about the syntax of the IAITensor data type, see **4.3.20 IAITensor**.	-

Return Value

SUCCESS indicates that initialization succeeds, while **FAILED** indicates that initialization fails.

4.1.6 AIModelManager::IsPreAllocateOutputMem

Determines whether the output memory can be pre-allocated.

Related function:

AIStatus AIModelManager::CreateOutputTensor(const std::vector<std::shared_ptr<IAITensor>> &in_data, std::vector<std::shared_ptr<IAITensor>> &out_data);

Syntax

virtual bool AIModelManager::IsPreAllocateOutputMem() override ;

Parameter Description

None

Return Value

- **True:** You can call CreateOutputTensor to apply for output memory.
- **False**: You cannot call CreateOutputTensor.

  NOTE

  True is returned only when the model manager loads a model.

4.1.7 AIModelManager::GetModelIOTensorDim

Obtains the input and output dimensions of the loaded model.

Syntax

virtual AIStatus AIModelManager::GetModelIOTensorDim(const std::string& model_name, std::vector<TensorDimension>& input_tensor, std::vector<TensorDimension>& output_tensor) ;

Parameter Description

Parameter	Description	Value Range
model_name	Model name	-
input_tensor	List of input dimensions for a model For details about the syntax of the TensorDimension data type, see **4.3.18 TensorDimension**.	-
output_tensor	List of output dimensions for a model For details about the syntax of the TensorDimension data type, see **4.3.18 TensorDimension**.	-

Return Value

SUCCESS indicates that initialization succeeds, while **FAILED** indicates that initialization fails.

Example

The input and output tensors of the ResNet-50 model are described as follows.

If **size** of **output_tensor** is not **0**, the obtained output_tensor data is appended to the original output_tensor data.

```
input_tensor
{
name = "data"                # Name of the input layer
data_type = 0                # Reserved data type, which is not used currently
size = 20                    # Memory size, in bytes
format = 0                   # Reserved tensor format, which is not used currently
dims = {1,3,224,224}
}
output_tensor
{
name =  "output_0_prob_0"     # Name of the output tensor. The format is as follows: output_{digit}_{name of the output node}_{index of the output node}
data_type = 0                # Reserved data type, which is not used currently
size = 20                    # Memory size, in bytes
format = 0                   # Reserved tensor format, which is not used currently
dims = {1,1000,1,1}
}
```

4.1.8 AIModelManager::GetMaxUsedMemory

Queries the size of the memory used by the model based on the model name.

Syntax

int32_t AIModelManager::GetMaxUsedMemory(std::string model_name);

Parameter Description

Parameter	Description	Value Range
model_name	Model name	-

Return Value

Size of the memory used by the model

4.1.9 AISimpleTensor::SetBuffer

Sets the tensor data address.

Syntax

void SetBuffer(void *data, const int32_t size, bool isown=false);

Parameter Description

Parameter	Description	Value Range
data	Data address	-
size	Data length **NOTE** The unit is byte. The value of **size** must be the same as the actual data size.	-
isown	Whether the tensor frees the memory of the data address after the tensor life cycle expires. • **false** (default): You can perform the free operation after the tensor life cycle expires. • **true**: After the tensor life cycle expires, the tensor frees the memory. Do not perform the free operation manually in this case. Otherwise, the memory is repeatedly freed.	-

Return Value

None

4.1.10 AITensorFactory::CreateTensor

Creates a model tensor. This API is defined in **ai_tensor.h**.

Syntax

std::shared_ptr<IAITensor> CreateTensor(const AITensorDescription &tensor_desc, void *buffer, int32_t size);

std::shared_ptr<IAITensor> CreateTensor(const AITensorDescription &tensor_desc);

std::shared_ptr<IAITensor> CreateTensor(const std::string &type);

Parameter Description

Parameter	Description	Value Range
tensor_desc	Tensor description For details about the syntax of the AITensorDescription data type, see **4.3.4 AITensorDescription**.	-
buffer	Data address **NOTE** • You are advised to allocate the memory for the input data and output data through the **HIAI_DMalloc** interface. This enables a zero-copy mechanism for algorithm inference to optimize the processing time. • The memory address is applied for and released by the customer.	-
size	Data length **NOTE** The unit is byte. The value of **size** must be the same as the actual data size.	-
type	Type of registering a tensor Note: This API should be used only when no memory is pre-allocated. For details, see the description of **type** in **4.3.4 AITensorDescription**.	-

Return Value

If a model tensor is created successfully, the tensor pointer is returned. If a model tensor fails to be created, a null pointer is returned.

4.1.11 Calling Example

Example 1: Synchronous Calling

The current example is the implementation code of the inference engine in the case of a single model.

In the case of multiple models, pay attention to the following points if you need to refer to this example:

- The **preOutBuffer** variable cannot contain the **static** keyword. You need to define the variable as follows:
  ```
  bool preOutBuffer = false;
  ```
- Before calling the **Process** function, call the **AddPara** function to set the name of each model.
  ```
  ai_context.AddPara("model_name", modelName);// Multiple models. You must set the seperate model name for each model.
  ret = ai_model_manager_->Process(ai_context,
        inDataVec, outDataVec_, 0);
  ```

The sample code is as follows:

```
// Process function implementation of the inference engine
HIAI_IMPL_ENGINE_PROCESS("ClassifyNetEngine", ClassifyNetEngine, CLASSIFYNET_ENGINE_INPUT_SIZE)
{
    HIAI_ENGINE_LOG(this, HIAI_OK, "ClassifyNetEngine Process");
    HIAI_StatusT ret = HIAI_OK;
    static bool preOutBuffer = false;
    std::vector<std::shared_ptr<hiai::IAITensor>> inDataVec;

    // Obtain the data input by the previous engine.
    std::shared_ptr<EngineTransNewT> input_arg =
        std::static_pointer_cast<EngineTransNewT>(arg0);
    // If the input data is a null pointer, an error code is returned
    if (nullptr == input_arg)
    {
        HIAI_ENGINE_LOG(this, HIAI_INVALID_INPUT_MSG,
            "fail to process invalid message");
        return HIAI_INVALID_INPUT_MSG;
    }

    // Prepare output data. If the synchronization mechanism is used, use the HIAI_DMalloc interface to allocate memory for the output data, and use the CreateTensor interface to provide memory for algorithm inference.
    //If the Process function for inference is called, the output memory needs to be allocated for only once.
    if (preOutBuffer == false) {
        std::vector<hiai::TensorDimension> inputTensorVec;
        std::vector<hiai::TensorDimension> outputTensorVec;
        ret = ai_model_manager_->GetModelIOTensorDim(modelName, inputTensorVec, outputTensorVec);
        if (ret != hiai::SUCCESS)
        {
            HIAI_ENGINE_LOG(this, HIAI_AI_MODEL_MANAGER_INIT_FAIL,
                "hiai ai model manager init fail");
            return HIAI_AI_MODEL_MANAGER_INIT_FAIL;
        }
        // allocate OutData in advance
        HIAI_StatusT hiai_ret = HIAI_OK;
        for (uint32_t index = 0; index < outputTensorVec.size(); index++) {
            hiai::AITensorDescription outputTensorDesc = hiai::AINeuralNetworkBuffer::GetDescription();
            uint8_t* buffer = nullptr;
            //Allocate the memory by calling the HIAI_Dmalloc interface. The memory is used for algorithm inference and can be released by calling the HIAI_DFree interface.
            //Release the memory during engine destruction.
            hiai_ret = hiai::HIAIMemory::HIAI_DMalloc(outputTensorVec[index].size, (void*&)buffer, 1000);
            if (hiai_ret != HIAI_OK || buffer == nullptr) {
```



```
            std::cout<<"HIAI_DMalloc failed"<< std::endl;
            continue;
        }
        outData_.push_back(buffer);
        shared_ptr<hiai::IAITensor> outputTensor =
            hiai::AITensorFactory::GetInstance()->CreateTensor(outputTensorDesc, buffer,
outputTensorVec[index].size);
        outDataVec_.push_back(outputTensor);
    }
    preOutBuffer = true;
  }

  // Transfer buffer to Framework directly, only one inputsize
  hiai::AITensorDescription inputTensorDesc =
    hiai::AINeuralNetworkBuffer::GetDescription();
  shared_ptr<hiai::IAITensor> inputTensor =
    hiai::AITensorFactory::GetInstance()->CreateTensor(inputTensorDesc,
    input_arg->trans_buff.get(), input_arg->buffer_size);
  // AIModelManager. fill in the input data.
  inDataVec.push_back(inputTensor);

  hiai::AIContext ai_context;
  // Process work
  ret = ai_model_manager_->Process(ai_context,
    inDataVec, outDataVec_, 0);
  if (hiai::SUCCESS != ret)
  {
    HIAI_ENGINE_LOG(this, HIAI_AI_MODEL_MANAGER_PROCESS_FAIL,
      "Fail to process ai model manager");
    return HIAI_AI_MODEL_MANAGER_PROCESS_FAIL;
  }

  // Convert the generated data to the buffer of the string type and send the data.
  for (uint32_t index = 0; index < outDataVec_.size(); index++)
  {
    HIAI_ENGINE_LOG(this, HIAI_OK, "ClassifyNetEngine SendData");
    std::shared_ptr<hiai::AINeuralNetworkBuffer> output_data =
std::static_pointer_cast<hiai::AINeuralNetworkBuffer>(outDataVec_[index]);
    std::shared_ptr<std::string> output_string_ptr =
      std::shared_ptr<std::string>(new std::string((char*)output_data->GetBuffer(), output_data-
>GetSize()));
    hiai::Engine::SendData(0, "string",
      std::static_pointer_cast<void>(output_string_ptr));
  }
  inDataVec.clear();
  return HIAI_OK;
}

ClassifyNetEngine::~ClassifyNetEngine() {
  //Release the pre-allocated memory of outData.
  HIAI_StatusT ret = HIAI_OK;
  for (auto buffer : outData_) {
    if (buffer != nullptr) {
      ret = hiai::HIAIMemory::HIAI_DFree(buffer);
      buffer = nullptr;
    }
  }
 }
```

Example 2: Asynchronous Calling

For details, see **ddk/sample/customop/customop_app/main.cpp** under the DDK installation directory in the DDK sample.

4.2 AIPP Configuration APIs

4.2.1 Overview

Artificial intelligence pre-processing (AIPP) involves image resizing (including cropping and padding), color space conversion (CSC), and subtracting the mean value and multiplying a factor (pixel changing). All these functions are implemented by the AI Core.

For dynamic AIPP, the AIPP mode is only set to dynamic during model conversion, when different AIPP parameters can be used. Before model inference, however, the dynamic AIPP parameter values must be set in the code of the inference engine. To set the dynamic AIPP parameter values in the code of the inference engine, the AIPP configuration APIs provided in this section must be called. These APIs are defined in **ai_tensor.h**.

4.2.2 SetDynamicInputIndex

Syntax

void SetDynamicInputIndex(uint32_t dynamicInputIndex = 0);

Function Description

Specifies the sequence number of the original input of a model to implement AIPP.

Parameter Description

Parameter	Input / Output	Type	Description
dynamicInputIndex	Input	uint32_t	Subscript of the original input of a model, starting from 0. The default value is **0**. For example, if the model has two inputs. To have AIPP start from the second input, set **dynamicInputIndex** to **1**.

Return Value

None

Exception Handling

None

Restriction

None

4.2.3 SetDynamicInputEdgeIndex

Syntax

void SetDynamicInputEdgeIndex(uint32_t dynamicInputEdgeIndex = 0);

Function Description

If a model input is shared by multiple operators, that is, the **Data** operator is followed by more operators, set this parameter to perform AIPP differently for different output edges of the **Data** operator.

Parameter Description

Parameter	Input / Output	Type	Description
dynamicInputEdg-eIndex	Input	uint32_t	Subscript of the output edge corresponding to the input of a model, starting from 0. The default value is **0**.

Return Value

None

Exception Handling

None

Restriction

None

4.2.4 SetInputFormat

Syntax

AIStatus SetInputFormat(AippInputFormat inputFormat);

Function Description

Sets the original input type of a model.

Parameter Description

Parameter	Input/Output	Type	Description

inputFormat	Input	AippInputFormat	Original input type of a model.
			enum AippInputFormat
			{
			YUV420SP_U8 = 1,
			XRGB8888_U8,
			RGB888_U8,
			YUV400_U8,
			RESERVED
			};

Return Value

Parameter	Type	Description
-	AIStatus	If the parameter is set successfully, **0** is returned. If the input is invalid, other values are returned. The AIStatus type is defined as follows: AIStatus = uint32_t

Exception Handling

None

Restriction

None

4.2.5 SetCscParams (Setting the Default Parameter Value)

Syntax

AIStatus SetCscParams(AippInputFormat srcFormat,

AippModelFormat dstFormat,

ImageFormat imageFormat = BT_601NARROW);

Function Description

You can call this API to implement the CSC function. The default values of CSC parameters in AIPP are automatically generated based on the original input type,

target input type, and image type specified in the input parameters. In addition, the RB/UV channel switching function is enabled or disabled based on the original input type and target input type.

Parameter Description

Parameter	Input/Output	Type	Description

srcFormat	Input	AippInputFormat	Original input type of a model.
			enum AippInputFormat
			{
			YUV420SP_U8 = 1,
			XRGB8888_U8,
			RGB888_U8,
			YUV400_U8,
			RESERVED
			};

dstFormat	Input	AippModelFormat	Target input type after CSC in AIPP.
			enum AippModelFormat
			{
			MODEL_RGB888_U8 = 1,
			MODEL_BGR888_U8,
			MODEL_GRAY,
			MODEL_YUV444SP_U8,
			MODEL_YVU444SP_U8
			};

imageFormat	Input	ImageFormat	Image type. Currently, only JPEG and BT_601NARROW are supported.
			enum ImageFormat
			{
			BITMAP,
			PNG,
			JPEG,
			BT_601NARROW
			};

Return Value

Parameter	Type	Description
-	AIStatus	If the parameter is set successfully, **0** is returned. If the input is invalid, other values are returned. The **AIStatus** type is defined as follows: AIStatus = uint32_t

Exception Handling

None

Restriction

This API is used to quickly set CSC parameters. The system provides a group of default CSC parameters. For details, see "AIPP Configuration" in *Model Conversion Guide*.

The supported input formats of images before and after CSC in AIPP are as follows:

AippInputFormat::YUV420SP_U8 to AippModelFormat::MODEL_YVU444SP_U8

AippInputFormat::YUV420SP_U8 to AippModelFormat::MODEL_RGB888_U8

AippInputFormat::YUV420SP_U8 to AippModelFormat::MODEL_BGR888_U8

AippInputFormat::YUV420SP_U8 to AippModelFormat::MODEL_GRAY

AippInputFormat::XRGB8888_U8 to AippModelFormat::MODEL_YUV444SP_U8

AippInputFormat::XRGB8888_U8 to AippModelFormat::MODEL_YVU444SP_U8

AippInputFormat::XRGB8888_U8 to AippModelFormat::MODEL_GRAY

AippInputFormat::RGB888_U8 to AippModelFormat::MODEL_BGR888_U8

AippInputFormat::RGB888_U8 to AippModelFormat::MODEL_YUV444SP_U8

AippInputFormat::RGB888_U8 to AippModelFormat::MODEL_YVU444SP_U8

AippInputFormat::RGB888_U8 to AippModelFormat::MODEL_GRAY

If the supported image types or image processing formats do not meet the requirements, call another API in **4.2.5 SetCscParams (Setting the Default Parameter Value)** to set CSC parameters.

4.2.6 SetCscParams (Setting a Parameter Value as Required)

Syntax

void SetCscParams(bool csc_switch = false,

int16_t cscMatrixR0C0 = 0,

int16_t cscMatrixR0C1 = 0,

int16_t cscMatrixR0C2 = 0,

int16_t cscMatrixR1C0 = 0,

int16_t cscMatrixR1C1 = 0,

int16_t cscMatrixR1C2 = 0,

int16_t cscMatrixR2C0 = 0,

int16_t cscMatrixR2C1 = 0,

int16_t cscMatrixR2C2 = 0,

uint8_t cscOutputBiasR0 = 0,

uint8_t cscOutputBiasR1 = 0,

uint8_t cscOutputBiasR2 = 0,

uint8_t cscInputBiasR0 = 0,

uint8_t cscInputBiasR1 = 0,

uint8_t cscInputBiasR2 = 0);

Function Description

You can call this API to flexibly configure CSC parameters in AIPP as required.

Parameter Description

Parameter	Input/Output	Type	Description
csc_switch	Input	bool	CSC switch. It is set to **false** by default.
cscMatrixR0C0	Input	int16_t	CSC matrix parameter.
cscMatrixR0C1	Input	int16_t	CSC matrix parameter.
cscMatrixR0C2	Input	int16_t	CSC matrix parameter.
cscMatrixR1C0	Input	int16_t	CSC matrix parameter.
cscMatrixR1C1	Input	int16_t	CSC matrix parameter.
cscMatrixR1C2	Input	int16_t	CSC matrix parameter.
cscMatrixR2C0	Input	int16_t	CSC matrix parameter.
cscMatrixR2C1	Input	int16_t	CSC matrix parameter.
cscMatrixR2C2	Input	int16_t	CSC matrix parameter.

Parameter	Input/Output	Type	Description
cscOutputBiasR0	Input	uint8_t	Output bias for RGB-to-YUV conversion. The default value is **0**. You can set only some configuration items.
cscOutputBiasR1	Input	uint8_t	Output bias for RGB-to-YUV conversion. The default value is **0**. You can set only some configuration items.
cscOutputBiasR2	Input	uint8_t	Output bias for RGB-to-YUV conversion. The default value is **0**. You can set only some configuration items.
cscInputBiasR0	Input	uint8_t	Input bias for YUV-to-RGB conversion. The default value is **0**. You can set only some configuration items.
cscInputBiasR1	Input	uint8_t	Input bias for YUV-to-RGB conversion. The default value is **0**. You can set only some configuration items.
cscInputBiasR2	Input	uint8_t	Input bias for YUV-to-RGB conversion. The default value is **0**. You can set only some configuration items.

Return Value

None

Exception Handling

None

Restriction

None

4.2.7 SetRbuvSwapSwitch

Syntax

void SetRbuvSwapSwitch(bool rbuvSwapSwitch = false);

Function Description

Sets whether to enable R/B swap or U/V channel swap before CSC.

Parameter Description

Parameter	Input/Output	Type	Description
rbuvSwapSwitch	Input	bool	**true**: Swap is supported. **false**: Swap is not supported.

Return Value

None

Exception Handling

None

Restriction

None

4.2.8 SetAxSwapSwitch

Syntax

void SetAxSwapSwitch(bool axSwapSwitch = false);

Function Description

Sets whether to enable GBA->ARGB swap or YUVA->AYUV swap before CSC.

Parameter Description

Parameter	Input/Output	Type	Description
axSwapSwitch	Input	bool	**true**: Swap is supported. **false**: Swap is not supported.

Return Value

None

Exception Handling

None

Restriction

None

4.2.9 SetSrcImageSize

Syntax

void SetSrcImageSize(int32_t srcImageSizeW = 0, int32_t srcImageSizeH = 0);

Function Description

Sets the width and height of the source image before AIPP.

Parameter Description

Parameter	Input/Output	Type	Description
srcImageSizeW	Input	int32_t	Width of the source image.
srcImageSizeH	Input	int32_t	Height of the source image.

Return Value

None

Exception Handling

None

Restriction

None

4.2.10 SetCropParams

Syntax

AIStatus SetCropParams(bool cropSwitch,

int32_t cropStartPosW, int32_t cropStartPosH,

int32_t cropSizeW, int32_t cropSizeH,

uint32_t batch_index = 0);

Function Description

Sets the crop parameters. Dynamic AIPP supports the configuration of different crop parameters for each batch. **batchIndex** indicates the sequence number of the

batch for which the crop parameters are set. The value range of **batchIndex** is [0, batchNum). If the value exceeds the range, a failure message will be returned.

Parameter Description

Parameter	Input/Output	Type	Description
cropSwitch	Input	Bool	**true**: Crop is supported. **false**: Crop is not supported.
cropStartPosW	Input	int32_t	Horizontal coordinate of the start point in the image during cropping
cropStartPosH	Input	int32_t	Vertical coordinate of the start point in the image during cropping
cropSizeW	Input	int32_t	Crop width
cropSizeH	Input	int32_t	Crop height
batchIndex	Input	uint32_t	Sequence number of the batch for which cropping is performed. The default value is **0**. The value range is [0, batchNum).

Return Value

Parameter	Type	Description
-	AIStatus	If the parameter is set successfully, **0** is returned. If the input is invalid, other values are returned. The AIStatus type is defined as follows: AIStatus = uint32_t

Exception Handling

None

Restriction

None

4.2.11 SetPaddingParams

Syntax

AIStatus SetPaddingParams(int8_t paddingSwitch,

int32_t paddingSizeTop, int32_t paddingSizeBottom,

int32_t paddingSizeLeft, int32_t paddingSizeRight,

uint32_t batch_index = 0);

Function Description

Sets the padding parameters. The **paddingSizeTop**, **paddingSizeBottom**, **paddingSizeLeft**, and **paddingSizeRight4** parameters indicate that padding is added to the top, bottom, left, and right of an image, respectively.

Parameter Description

Parameter	Input/Output	Type	Description
paddingSwitch	Input	Bool	**true**: Padding is supported. **false**: Padding is not supported.
paddingSizeTop	Input	int32_t	Padding to the top of an image
paddingSizeBottom	Input	int32_t	Padding to the bottom of an image
paddingSizeLeft	Input	int32_t	Padding to the left of an image
paddingSizeRight	Input	int32_t	Padding to the right of an image
batchIndex	Input	uint32_t	Sequence number of the batch for which padding is performed. The default value is **0**. The value range is [0, batchNum).

Return Value

Parameter	Type	Description
-	AIStatus	If the parameter is set successfully, **0** is returned. If the input is invalid, other values are returned. The AIStatus type is defined as follows: AIStatus = uint32_t

Exception Handling

None

Restriction

None

4.2.12 SetDtcPixelMean

Syntax

AIStatus SetDtcPixelMean(int16_t dtcPixelMeanChn0 = 0,

int16_t dtcPixelMeanChn1 = 0,

int16_t dtcPixelMeanChn2 = 0,

int16_t dtcPixelMeanChn3 = 0,

uint32_t batch_index = 0);

Function Description

Sets the parameters for the mean values of DTC channels.

Parameter Description

Parameter	Input/Output	Type	Description
dtcPixelMeanChn0	Input	int16_t	Mean value of channel 0
dtcPixelMeanChn1	Input	int16_t	Mean value of channel 1
dtcPixelMeanChn2	Input	int16_t	Mean value of channel 2
dtcPixelMeanChn3	Input	int16_t	Mean value of channel 3
batchIndex	Input	uint32_t	Sequence number of the batch for which the mean value of a DTC channel is set. The default value is **0**. The value range is [0, batchNum).

Return Value

Parameter	Type	Description
-	AIStatus	If the parameter is set successfully, **0** is returned. If the input is invalid, other values are returned. The AIStatus type is defined as follows: AIStatus = uint32_t

Exception Handling

None

Restriction

None

4.2.13 SetDtcPixelMin

Syntax

AIStatus SetDtcPixelMin(float dtcPixelMinChn0 = 0,

float dtcPixelMinChn1 = 0,

float dtcPixelMinChn2 = 0,

float dtcPixelMinChn3 = 0,

uint32_t batch_index = 0);

Function Description

Sets the parameters for the minimum values of DTC channels.

Parameter Description

Parameter	Input/Output	Type	Description
dtcPixelMinChn0	Input	float	Minimum value of DTC channel 0
dtcPixelMinChn1	Input	float	Minimum value of DTC channel 1
dtcPixelMinChn2	Input	float	Minimum value of DTC channel 2
dtcPixelMinChn3	Input	float	Minimum value of DTC channel 3
batchIndex	Input	uint32_t	Sequence number of the batch for which the minimum value of a DTC channel is set. The default value is **0**. The value range is [0, batchNum).

Return Value

Parameter	Type	Description
-	AIStatus	If the parameter is set successfully, **0** is returned. If the input is invalid, other values are returned. The AIStatus type is defined as follows: AIStatus = uint32_t

Exception Handling

None

Restriction

None

4.2.14 SetPixelVarReci

Syntax

AIStatus SetPixelVarReci(float dtcPixelVarReciChn0 = 0,

float dtcPixelVarReciChn1 = 0,

float dtcPixelVarReciChn2 = 0,

float dtcPixelVarReciChn3 = 0,

uint32_t batch_index = 0);

Function Description

Sets the parameters for the variance values or (max-min) reciprocal values of DTC channels.

Parameter Description

Parameter	Input / Output	Type	Description
dtcPixelVarReciChn0	Input	float	Variance of DTC channel 0
dtcPixelVarReciChn1	Input	float	Variance of DTC channel 1
dtcPixelVarReciChn2	Input	float	Variance of DTC channel 2
dtcPixelVarReciChn3	Input	float	Variance of DTC channel 3

Parameter	Input / Output	Type	Description
batchIndex	Input	uint32_t	Sequence number of the batch for which the variance of a DTC channel is set. The default value is **0**. The value range is [0, batchNum).

Return Value

Parameter	Type	Description
-	AIStatus	If the parameter is set successfully, **0** is returned. If the input is invalid, other values are returned. The AIStatus type is defined as follows: AIStatus = uint32_t

Exception Handling

None

Restriction

None

4.2.15 SetInputDynamicAIPP

Syntax

```
AIStatus SetInputDynamicAIPP(std::vector<std::shared_ptr<IAITensor>>& inData,
std::shared_ptr<AippDynamicParaTensor> aippParms);
```

Function Description

Inserts the generated dynamic AIPP parameter **AippDynamicParaTensor** into the input tensor list. The insertion position is determined by the **dynamicInputIndex** and **dynamicInputEdgeIndex2** attributes in the AIPP tensor. The two attributes are set by calling APIs in **4.2.2 SetDynamicInputIndex** and **4.2.15 SetInputDynamicAIPP**.

The AIPP tensor will be inserted after the tensor of the original input.

If multiple AippDynamicParaTensors need to be inserted, this API needs to be called for multiple times. In this case, if the **dynamicInputIndex** value of the newly inserted AippDynamicParaTensor is smaller than that of the existing AippDynamicParaTensor in the tensor list, or if the **dynamicInputIndex** value of

the newly inserted AippDynamicParaTensor is equal to that of the existing AippDynamicParaTensor but the **dynamicInputEdgeIndex** value is smaller, the AIPP tensor will be inserted before the existing AippDynamicParaTensor. If the values of **dynamicInputIndex** and **dynamicInputEdgeIndex** are equal to those of the existing AippDynamicParaTensor in the tensor list, the newly inserted AippDynamicParaTensor overwrites the existing AippDynamicParaTensor.

When this API is called, the validity of **dynamicInputIndex** and **dynamicInputEdgeIndex** is not checked. That is, if the value of **dynamicInputIndex** is greater than the number of outputs of the model or the value of **dynamicInputEdgeIndex** is greater than the number of output edges corresponding to the input, no error is returned.

Parameter Description

Parameter	Input/ Output	Type	Description
inData	Input	std::vector<std::shared_ptr<IAITensor>>&	List of input tensors
aippParms	Input	std::shared_ptr<AippDynamicParaTensor>	Dynamic AIPP tensor

Return Value

Parameter	Type	Description
-	AIStatus	If the parameter is set successfully, **0** is returned. If the input is invalid, other values are returned. The AIStatus type is defined as follows: AIStatus = uint32_t

Exception Handling

None

Restriction

None

4.2.16 GetDynamicInputIndex

Syntax

uint32_t GetDynamicInputIndex() const;

Function Description

Obtains the subscript (specified by **SetDynamicInputIndex**) of the original input of a model.

Parameter Description

None

Return Value

Parameter	Type	Description
-	uint32_t	Subscript of the original input of a model

Exception Handling

None

Restriction

None

4.2.17 GetDynamicInputEdgeIndex

Syntax

uint32_t GetDynamicInputEdgeIndex() const;

Function Description

Obtains the subscript (specified by **SetDynamicInputEdgeIndex**) of the output edge corresponding to the input of a model.

Parameter Description

None

Return Value

Parameter	Type	Description
-	uint32_t	Subscript of the output edge corresponding to the input of a model

Exception Handling

None

Restriction

None

4.2.18 Calling Example

```
/**
* @file mngr_sample.h
*
* Copyright(c)<2018>, <Huawei Technologies Co.,Ltd>
*
* @version 1.0
*
* @date 2018-4-25
*/
#include <stdio.h>
#include <string>
#include <vector>
#include <iostream>
#include <fstream>
#include <assert.h>
#include "hiaiengine/ai_model_manager.h"
using namespace std;
using namespace hiai;
// Image data path
static const std::string IMAGE_FILE_PATH = "/data/input_zebra.bin";
static const char* MODEL_PATH = "/data/ResNet.davincimodel";
static const char* MODEL_NAME = "resnet18";
/*
*** brief: Read input data.
*/
char* ReadBinFile(const char *file_name, uint32_t *fileSize)
{
    std::filebuf *pbuf;
    std::ifstream filestr;
    size_t size;
    filestr.open(file_name, std::ios::binary);
    if (!filestr)
    {
        return nullptr;
    }
    pbuf = filestr.rdbuf();
    size = pbuf->pubseekoff(0, std::ios::end, std::ios::in);
    pbuf->pubseekpos(0, std::ios::in);
    char * buffer = (char*)malloc(size);
    if (nullptr == buffer)
    {
        return nullptr;
    }
    pbuf->sgetn(buffer, size);
    *fileSize = size;
    filestr.close();
    return buffer;
}
int main(int argc, char* argv[])
{
    vector<shared_ptr<IAITensor>> model_input;
    vector<shared_ptr<IAITensor>> model_output;
    AIModelManager model_mngr;
    AIModelDescription model_desc;
    AIConfig config;
    AIContext context;
    model_desc.set_path(MODEL_PATH);
    model_desc.set_name(MODEL_NAME);
    model_desc.set_type(0);
    vector<AIModelDescription> model_descs;
    model_descs.push_back(model_desc);
    // AIModelManager Init
```

```
    AIStatus ret = model_mngr.Init(config, model_descs);
    if (SUCCESS != ret)
    {
        printf("AIModelManager Init failed. ret = %d\n", ret);
        return -1;
    }
    // Input tensor
    // The input tensor will be reset after the image data is read. The function here is only for initialization.
    AITensorDescription tensor_desc = AINeuralNetworkBuffer::GetDescription();
    shared_ptr<IAITensor> input_tensor = AITensorFactory::GetInstance()->CreateTensor(tensor_desc);
    if (nullptr == input_tensor)
    {
        printf("Create input_tensor failed.\n");
        return -1;
    }
    // Read image data.
    uint32_t image_data_size = 0;
    float* image_data = (float*)ReadBinFile(IMAGE_FILE_PATH.c_str(), &image_data_size);
    if (nullptr == image_data)
    {
        printf("ReadBinFile failed bin file path= %s \n", IMAGE_FILE_PATH.c_str());
        return -1;
    }
    // Set the pointer and length of the image data address to input_simple_tensor.
    shared_ptr<AISimpleTensor> input_simple_tensor = static_pointer_cast<AISimpleTensor>(input_tensor);
    input_simple_tensor->SetBuffer((void*)image_data, image_data_size);
    model_input.push_back(input_tensor);
    // Create an output tensor.
    if(model_mngr.IsPreAllocateOutputMem())
    {
        ret = model_mngr.CreateOutputTensor(model_input, model_output);
        if (SUCCESS != ret)
        {
            printf("CreateOutputTensor failed.ret = %d\n", ret);
            delete image_data;
            return -1;
        }
    }
    else
    {
       // Create a tensor.
        ret = model_mngr.GetModelIOTensorDim(MODEL_NAME, input_tensor_dims, output_tensor_dims);
        std::vector<TensorDimension> input_tensor_dims;
        std::vector<TensorDimension> output_tensor_dims;
        for(TensorDimension & dims : output_tensor_dims)
        {
            shared_ptr<IAITensor> output_tensor = AITensorFactory::GetInstance()->CreateTensor(tensor_desc);
            shared_ptr<AISimpleTensor> output_simple_tensor =
static_pointer_cast<AISimpleTensor>(output_tensor);
            output_simple_tensor->setBuffer((void*) new char[dims.size], dims.size);
            model_output.push_back(output_tensor)
        }
    }

    bool dynamic_aipp_flag = true;
    if (dynamic_aipp_flag)
    {
        const int batch_number = 2;
        AITensorDescription desc = AippDynamicParaTensor::GetDescription(std::to_string(batch_number));
        shared_ptr<IAITensor> tensor = AITensorFactory::GetInstance()->CreateTensor(desc);
        shared_ptr<AippDynamicParaTensor> aipp_params_tensor =
static_pointer_cast<AippDynamicParaTensor>(tensor);
       // Set the original input type of a model.
        aipp_params_tensor->SetInputFormat(hiai::YUV420SP_U8);
       // Set the CSC matrix parameters.
        aipp_params_tensor->SetCscParams(hiai::YUV420SP_U8, hiai::MODEL_BGR888_U8, hiai::JPEG);
        // Set the width and height of the source image.
        aipp_params_tensor->SetSrcImageSize(256, 224);
       // If there are multiple batchs, set the AIPP parameters for each batch.
```



```
        for (int i = 0; i < batch_number; i++) {
            // Set the crop parameters.
            aipp_params_tensor->SetCropParams(true, 0, 0, 200, 200, i);
            // Set the padding parameters.
            aipp_params_tensor->SetPaddingParams(true, 12, 12, 12, 12, i);
            // Set the parameters for the mean values of DTC channels.
            aipp_params_tensor->SetDtcPixelMean(104, 117, 123, 0, i);
            // Set the parameters for the variance values or (max-min) reciprocal values of DTC channels.
            aipp_params_tensor->SetPixelVarReci(1.0, 1.0, 1.0, 1.0, i);
        }

    model_mngr.SetInputDynamicAIPP(model_input, aipp_params_tensor);
    }

    // Start model inference.
    printf("Start process.\n");
    ret = model_mngr.Process(context, model_input, model_output, 0);
    if (SUCCESS != ret)
    {
        printf("Process failed.ret = %d\n", ret);
        return -1;
    }
   // Because the listener is not set, the model inference is synchronously processed. The model inference output data can be obtained directly.
    shared_ptr<AISimpleTensor> result_tensor = static_pointer_cast<AISimpleTensor>(model_output[0]);
    printf("Get Result, buffsize is %d",result_tensor->GetSize());
    for(TensorDimension & dims : output_tensor_dims)
    {
    }
    printf("predict ok.\n");
    return 0;
}
```

4.3 Data Types

4.3.1 AIConfigItem

Provides the AIConfig item description. For details, see **ai_type.proto**.

```
message AIConfigItem
{
    string name = 1; // Configuration item name
    string value = 2; // Configuration item value
    repeated AIConfigItem sub_items = 3; // Configuration subitem
};
```

4.3.2 AIConfig

Serves as the input parameter when the Init interface of the model manager is called. For details, see **ai_types.proto**.

```
message AIConfig
{
    repeated AIConfigItem items = 1; // Configuration item list
};
```

4.3.3 AITensorParaDescription

Provides the tensor parameter description. For details, see **ai_types.proto**.

```
message AITensorParaDescription
{
    string name = 1; // Parameter name
```



```
    string type = 2; // Parameter type
    string value = 3;  // Parameter value
    string desc = 4; // Parameter description
    repeated AITensorParaDescription sub_paras = 5; // Sub-parameter list
};
```

4.3.4 AITensorDescription

Provides the tensor description, mainly about the input and output information of a model. For details, see **ai_types.proto**.

```
// Tensor description
message AITensorDescription
{
    string name = 1; // Tensor name
    string type = 2; // Tensor type
    repeated string compatible_type = 3; // Specifies all types of parent classes that can be compatible.
   repeated AITensorParaDescription paras = 4; // Parameter list
};
```

4.3.5 AITensorDescriptionList

Provides the tensor description list, mainly used for describing the input and output information list of a model. For details, see **ai_types.proto**.

```
message AITensorDescriptionList
{
    repeated AITensorDescription tensor_descs = 1; // Tensor list
}
```

4.3.6 AIModelDescription

Serves as the input when the **Init** interface is called. For details, see **ai_types.proto**.

```
messAIPPDynaage AIModelDescription
{
    string name = 1; // Model name, which can contain uppercase letters, lowercase letters, digits, underscores (_), and dots (.)
    int32 type = 2; // Model type. Currently, only the DAVINCI_OFFLINE_MODEL type is supported. The value is 0.
                    // The model parsing capability has been added to the model manager. Therefore, it does not matter if this field is not set. To ensure forward compatibility, this field is reserved.
    string version = 3; // Model version
    int32 size = 4; // Model size
    string path = 5; // Model path, which can contain uppercase letters, lowercase letters, digits, and underscores (_).
    repeated string sub_path = 6; // Auxiliary model path, which is used when there are multiple model files, for example, Caffe online model
                                  // To be forward compatible, the "path" field is not changed to "repeated". Instead, the "sub_path" field is added.
    string key = 7; // Model key
    repeated string sub_key = 8; // Auxiliary model key, which is used when there are multiple model keys, for example, Caffe online model
                                 // To be forward compatible, the "key" field is not changed to "repeated". Instead, the "sub_key" field is added.
    enum Frequency
    {
        UNSET   =0;
        LOW     =1;
        MEDIUM  =2;
        HIGH    =3;
    }
    Frequency frequency = 9;
    enum DeviceType
    {
```

```
        NPU  = 0;
        IPU  = 1;
        MLU  = 2;
        CPU  = 3;
        NONE = 255;
    }
    DeviceType device_type = 10;
    enum Framework
    {
        OFFLINE    =0;
        CAFFE      =1;
        TENSORFLOW =2;
    }
    Framework framework = 11;
    bytes data = 100; // Model data
    repeated AITensorDescription inputs = 12; // Description of input tensors
    repeated AITensorDescription outputs = 13; // Description of output tensors
};
```

4.3.7 AINNNodeDescription

NN node description, which is used for describing the model, input tensor, and output tensor required by an NN node. For details, see **ai_types.proto**.

```
message AINNNodeDescription
{
    string name = 1; // NN node name
    string desc = 2; // NN node description
   bool isPreAllocateOutputMem = 3; // Whether to pre-allocate the output memory
   AIConfig config = 4; // Configuration parameter
   repeated AIModelDescription model_list = 5;  // List of models required by an NN node
    repeated AITensorDescription inputs = 6; // Description of input tensors
    repeated AITensorDescription outputs = 7; // Description of output tensors
    bool need_verify = 8; // Whether to verify the tensor match during serial connection
    repeated string ignored_check_aitensor = 9; // Specifies the list of tensors for which the mechanism for
verifying whether they match the tensors of the inputs is ignored during serial connection.
};
```

4.3.8 AINNNodeDescriptionList

NN node description list, which is used for describing the model, input tensor, and output tensor required by an NN node. For details, see **ai_types.proto**.

```
message AINNNodeDescriptionList
{
    repeated AINNNodeDescription nnnode_descs = 1; // NN node list
}
```

4.3.9 AIAPIDescription

API description, which is used for describing the name, input tensor, and output tensor of an API. For details, see **ai_types.proto**.

```
message AIAPIDescription
{
    string name = 1; // API name
    string desc = 2; // API description
    bool isPreAllocateOutputMem = 3; // Whether to pre-allocate the output memory
   AIConfig config = 4; // Configuration parameter
    repeated AITensorDescription inputs = 5; // Description of input tensors
    repeated AITensorDescription outputs = 6; // Description of output tensors
    bool need_verify = 7; // Whether to verify the tensor match during serial connection
    repeated string ignored_check_aitensor = 8; // Specifies the list of tensors for which the mechanism for
verifying whether they match the tensors of the inputs is ignored during serial connection.
};
```


4.3.10 AIAPIDescriptionList

API description list, which is used for describing the name, input tensor, and output tensor of an API. For details, see **ai_types.proto**.

```
message AIAPIDescriptionList
{
    repeated AIAPIDescription api_descs = 1; // API list
}
```

4.3.11 AIOPDescription

This data type is defined in **ai_types.proto**.

```
// AI operation description
message AIOPDescription
{
    enum OP_Type
    {
        AI_API = 0;
        AI_NNNODE = 1;
    }

    OP_Type type = 1;
    AINNNodeDescription nnnode_desc = 2;
    AIAPIDescription api_desc = 3;
};
```

4.3.12 AIOPDescriptionList

This data type is defined in **ai_types.proto**.

```
// AI operation description list
message AIOPDescriptionList
{
    repeated AIOPDescription op_descs = 1; // AI operation list
}
```

4.3.13 NodeDesc

This data type is defined in **ai_types.proto**.

```
message NodeDesc
{
   string name=1; // IAINNNode or ALG_API name
   AIConfig config=2; // Initialization parameter required by IAINNNode or ALG_API
   repeated AIModelDescription mode_desc=3; // Initialization parameter required by IAINNNode
}
```

4.3.14 EngineDesc

This data type is defined in **ai_types.proto**.

```
message EngineDesc
{
    enum RunSide
    {
        DEVICE=0;
        HOST=1;
    }
    enum EngineType
    {
        NORMAL=0;
        SOURCE=1;
        DEST=2;
```



```
    }
    uint32 id =1; // Engine ID (node)
    EngineType type=2;
    string name =3; // Engine node name
    repeated string so_name=4; // List of all required DLL .so files
    RunSide side=5; // Run on the host or device.
    int32 priority=6; // Node priority
    uint32 instance_cnt=7; // Number of instances (equivalent to the number of threads)
    repeated uint32 next_node_id=8; // Next node list
    bool user_input_cb=9; // IDE can be ignored.
    bool user_output_cb=10; // IDE can be ignored.
    repeated NodeDesc oper=11; // HIAIEngine node list
}
```

4.3.15 GraphInitDesc

This data type is defined in **ai_types.proto**.

```
message GraphInitDesc
{
    int32 priority=1; // Priority of the entire process for a graph
}
```

4.3.16 GeneralFileBuffer

This data type is defined in **ai_types.proto**.

```
message GeneralFileBuffer
{
    bytes raw_data = 1;
    string file_name = 2;
}
```

4.3.17 AIContext

Saves the process context when the **Process** interface of the model manager is called asynchronously, including the key-value pair of the string type. This data type is defined in **ai_types.h**.

```
class AIContext
{
public:
    /*
    * @brief   Obtain a parameter.
    * @param [in] key   Key corresponding to the parameter
    * @return sring   Value corresponding to the key. If the value does not exist, a null string is returned.
    */
    const std::string GetPara(const std::string &key) const;

    /*
    * @brief   Set a parameter.
    * @param [in] key   Key corresponding to the parameter
    * @param [in] value   Value corresponding to the parameter
    */
    void AddPara(const std::string &key, const std::string &value);

    /*
    * @brief   Delete a parameter.
    * @param [in] key   Key corresponding to the parameter to be deleted
    */
    void DeletePara(const std::string &key);

    /*
     * @brief    Obtain all parameters.
    * @param [out] keys  Key corresponding to all parameters that have been set
    */
    void GetAllKeys(std::vector<std::string> &keys);
```



```
#if defined( __ANDROID__) || defined(ANDROID)
        std::string  Serialize();

        AIStatus  Deserialize(std::string str);
#endif

private:
    std::map<std::string, std::string> paras_; /** Definition of the name-value pairs of parameters */
};
```

4.3.18 TensorDimension

The **AIModelManager::GetModelIOTensorDim** API can be called to obtain the information about the model input/output size, including the tensor dimension information, data type, buffer size, and tensor name. This data type is defined in **ai_types.h**.

```
/*
* Describe the dimensions of a tensor.
   */
   struct TensorDimension
   {
       uint32_t n; // Batch size
       uint32_t c; // Feature map channels
       uint32_t h; // Feature map height
       uint32_t w; // Feature map width
       uint32_t data_type;
       uint32_t size;
       std::string  name;
   };
```

- If the parameters related to AIPP are not configured during model conversion, the information about the model input/output dimensions obtained by calling **AIModelManager::GetModelIOTensorDim** is as follows:
  - **n**, **c**, **h**, and **w** are those before model conversion.
  - The enumerated value of **data_type** is defined as follows:

    ```
    enum DataType
    {
    DT_UNDEFINED = 0; // Used to indicate a DataType field has not been set.
    DT_FLOAT = 1; // float type, 4 bytes
    DT_FLOAT16 = 2; // fp16 type, 2 bytes
    DT_INT8 = 3; // int8 type, 1 byte
    DT_UINT8 = 4; // uint8 type, 1 byte
    DT_INT16 = 5; // int16 type, 2 bytes
    DT_UINT16 = 6; // uint16 type, 2 bytes
    DT_INT32 = 7; // int32 type, 4 bytes
    DT_INT64 = 8; // int64 type, 8 bytes
    DT_UINT32 = 9; // unsigned int32, 4 bytes
    DT_UINT64 = 10; // unsigned int64, 8 bytes
    DT_BOOL = 11; // boolean type, 1 byte
    DT_DOUBLE = 12; // double type, 8 bytes
    }
    ```

  - size = n*c*h*w*Number of bytes of each data type
- If AIPP parameters are set during model conversion, the information about the model input/output size obtained by **AIModelManager::GetModelIOTensorDim** is as follows:
  - **n**, **c**, **h**, and **w** are those before model conversion, instead of **h** (corresponding to the **src_image_size_h** parameter) and **w** (corresponding to the **src_image_size_w** parameter) configured in AIPP.
  - The value of **data_type** is fixed at **3**, indicating the DT_INT8 type.

– The parameter **size** indicates the size of the picture data processed by AIPP. The size varies according to the input format. For details, see **Table 4-1**.

Table 4-1 Size formula

input_format	size
YUV400_U8	n * src_image_size_w * src_image_size_h
YUV420SP_U8	n * src_image_size_w * src_image_size_h * 1.5
XRGB8888_U8	n * src_image_size_w * src_image_size_h * 4
RGB888_U8	n * src_image_size_w * src_image_size_h * 3

NOTE

For details about model conversion or AIPP, see *Model Conversion Guide*.

4.3.19 IAIListener

Provides IAIListener when the Process interface is asynchronously called, used for callback notification after model execution is complete. The details are as follows. This data type is defined in **ai_nn_node.h**.

```
/*
* Asynchronous callback interface, which is implemented by the caller
*/
class IAIListener
{
public:
    virtual ~IAIListener(){}
    /*
   * @brief  Asynchronous callback interface
   * @param [in] context   Context information, including variable parameter configurations when the
NNNode is running
    * @param [in] result  Task status when the execution is complete
    * @param [in] out_data   Output data when the execution is complete
    */
    virtual void OnProcessDone(const AIContext &context, int result,-
        const std::vector<std::shared_ptr<IAITensor>> &out_data) = 0;
    / *
     * @brief   Service death callback interface. When the client is suspended from the server, the
application is notified.
     */
    virtual void OnServiceDied() {};
};
```

4.3.20 IAITensor

Serves as the input and output of a model when the process function is used. This data type is defined in **ai_tensor.h**.

```
/**
* Tensor    Input and output data
*/
class IAITensor
{
public:
    IAITensor() {}
    virtual ~IAITensor() {};
```

```
    /*
    * @brief   Set parameters through AITensorDescription.
    * @param [in] tensor_desc   Tensor description
    * @return true: Initialization success
    * @return false: Initialization failure. The possible cause is that tensor_desc is inconsistent with the
current tensor.
    */
    virtual bool Init(const AITensorDescription &tensor_desc) = 0;
   /*
    * @brief   Obtain the type name.
    */
    virtual const char* const GetTypeName() = 0;
    /*
    * @brief   Obtain the byte length after serialization.
    */
    virtual uint32_t ByteSizeLong() = 0;
    virtual void SetBufferAttr(void *data, int32_t size, bool isowner, bool is_support_mem_share) = 0;

    virtual bool IsSupportZerocpy() = 0;

#if defined( __ANDROID__) || defined(ANDROID)
        /*
        * @brief   Serialize data to the buffer for cross-process data interaction.
        * @param [in] buffer   Address of the buffer to which the data is serialized, allocated by the caller
        * @param [in] size   Size of the buffer to which the data is output
        * @return SUCCESS   Success
        *         FAILED: failure. If the tensor does not support cross-process, this API does not need to be
implemented and a failure message is returned.
        */
        virtual AIStatus SerializeTo(void* buffer, const uint32_t size) = 0;

        /*
        * @brief   Deserialize data from the buffer to a tensor for cross-process data interaction.
        * @param [in] buffer  Size of the buffer to which the data is input
        * @param [in] size   Size of the buffer to which the data is input
        * @return SUCCESS   Success
        *         FAILED: failure. If the tensor does not support cross-process, this API does not need to be
implemented and a failure message is returned.
        */
        virtual AIStatus DeserializeFrom(const void* buffer, const uint32_t size) = 0;
#endif
};
```

4.4 Other Compilation Dependent APIs

This section describes the compilation dependent APIs. It recommended that these APIs not be directly called.

4.4.1 AIAlgAPIFactory

ALG API registration factory class. This class is defined in **ai_alg_api.h**.

```
    class AIAlgAPIFactory
    {
    public:
        static AIAlgAPIFactory* GetInstance();

        /*
        * @brief   Obtain the API.
        * @param [in] name    API name
        * @return   API prototype pointer
        */
        AI_ALG_API GetAPI(const std::string &name);

        /*
        * @brief   Register an API.
```

```
    * @param [in] desc   API description
    * @param [in] func   API definition
    * @return SUCCESS   Success
    * @return   Other: failure
    */
    AIStatus RegisterAPI(const AIAPIDescription &desc, AI_ALG_API func);

    /*
    * @brief   Obtain all API descriptions.
    * @param [in] api_desc_list   API description list
    */
    void GetAllAPIDescription(AIAPIDescriptionList &api_desc_list);

    /*
    * @brief   Deregister an API.
    * @param [in] api_desc   API description
    */
    AIStatus UnRegisterApi(const AIAPIDescription &api_desc);

    /*
    * @brief   Obtain the API description.
    * @param [in] name   API name
    * @param [in] api_desc    API description
    * @return SUCCESS   Success
    * @return   Other: failure
    */
    AIStatus GetAPIDescription(const std::string &name, AIAPIDescription &api_desc);

private:
    std::map<std::string, AI_ALG_API> func_map_;
    std::map<std::string, AIAPIDescription> desc_map_;
    std::mutex api_reg_lock_;
};
```

4.4.2 AIAlgAPIRegisterar

Macro for registering and encapsulating the **AIAlgAPIFactory** class. This class is defined in **ai_alg_api.h**.

```
class AIAlgAPIRegisterar
{
public:
    AIAlgAPIRegisterar(const AIAPIDescription &desc, AI_ALG_API func)
    {
        AIAlgAPIFactory::GetInstance()->RegisterAPI(desc, func);
        api_desc_ = desc;
    }

    ~AIAlgAPIRegisterar()
    {
        AIAlgAPIFactory::GetInstance()->UnRegisterApi(api_desc_);
    }
private:
    AIAPIDescription api_desc_;
};
```

4.4.3 REGISTER_ALG_API_UNIQUE

Macro for API registration, which is used in API implementation. This class is defined in **ai_alg_api.h**.

```
#define REGISTER_ALG_API_UNIQUE(desc, ctr, func) \
    AIAlgAPIRegisterar g_##ctr##_api_registerar(desc, func)
```


4.4.4 REGISTER_ALG_API

Macro for encapsulating **REGISTER_ALG_API_UNIQUE**. This class is defined in **ai_alg_api.h**.

```
#define REGISTER_ALG_API(name, desc, func) \
    REGISTER_ALG_API_UNIQUE(desc, name, func)
```

4.4.5 AIModelManager

Model manager class. This class is defined in **ai_model_manager.h**.

```
    class AIModelManager : public IAINNNode
    {
    public:
        AIModelManager();

        /*
        * @brief    Set the dynamic batch feature. This API does not need to be called by users.
         * @param [in] inputDim    Input dimension of the model
         * @param [in] input    Input data
         * @param [out] inputIndex    Set the sequence number of a dynamic batch for input, starting from 0.
         * @param [in] batchNumber    Dynamic batch size
         * @return SUCCESS    Success
        * @return  Other: failure
         */
        AIStatus SetInputDynamicBatch(const vector<TensorDimension>& inputDim, std::vector
<std::shared_ptr <IAITensor> > &input,
            uint32_t inputIndex, uint32_t batchNumber);

        /*
         * @brief    Whether to pre-allocate the output memory. This API is implemented by the NNNode
service. The default value is true.
         */
        virtual bool IsPreAllocateOutputMem() override;

        /*
         * @brief Obtain the AINNNodeDescription object.
         */
        static AINNNodeDescription GetDescription();

        /*
         * @brief  Obtain the input and output dimensions of the loaded model.
         * @param [in] model_name  Model name
         * @param [out] input_tensor    Input dimension of the model
         * @param [out] output_tensor  Output dimension of the model
         * @return SUCCESS    Success
        * @return  Other: failure
         */
        AIStatus GetModelIOTensorDim(const std::string& model_name,
            std::vector<TensorDimension>& input_tensor, std::vector<TensorDimension>& output_tensor);

        /*
        * @brief  Set the thread inference request ID.
         * @param [in] request_id  Model name
         * @return None
         */
        static void SetRequestId(uint64_t request_id);

        ~AIModelManager();

#ifdef __LITE__
        /*
        * @brief    Release resources in the case of idle state and timeout. UnloadModels is called for
implementation.
         */
        virtual AIStatus IdleRelease();
        /*
```

```
    * @brief recovers   Resume services after timeout, including enabling a device and loading a model.
LoadModels is called for implementation.
    */
    virtual AIStatus IdleResume();
#endif // __LITE__

  private:
    AIModelManagerImpl* impl_;
  };
```

4.4.6 getModelInfo

Macro for parsing a model file to obtain model information. This API is defined in **ai_model_parser.h**.

```
    /**
     * @ingroup hiai
     * @brief   Parse a model file to obtain model information.
     * @param [in] model_path   Model file path
     * @param [in] key   Model decryption key
     * @param [out] model_desc   Model information
     * @return Status   Running result
     */

     hiai::AIStatus getModelInfo(const std::string & model_path,
         const std::string & key, AIModelDescription & model_desc);
```

4.4.7 IAINNNode

NN node API, which is implemented by the service provider. The IAINNNode class is defined in **ai_nn_node.h**.

```
  class IAINNNode
  {
  public:
    virtual  ~IAINNNode(){}
    /*
    * @brief initialization interface, in which the service implements model loading or other initialization
actions
    * @param [in] model_desc  Model information. If the model is not required, a null vector is transferred.
    * @param [in] config  Configuration parameters
    * @return SUCCESS   Success
    * @return   Other: failure
    */
    virtual AIStatus Init(const AIConfig &config,
        const std::vector<AIModelDescription> &model_descs = {}) = 0;

    /*
    * @brief  Set listener.
    * @parm [in]   If the listener is set to a null pointer, the process interface is called synchronously.
Otherwise, the process interface is called asynchronously.
    * @return SUCCESS   Success
    * @return   Other: failure
    */
    virtual AIStatus SetListener(std::shared_ptr<IAIListener> listener) = 0;

    /*
    * @brief  Compute interface
    * @param [in] context   Context information, including variable parameter configurations when the
NNNode is running
    * @param [in] in_data   Input data
    * @param [out] out_data   Output data
    * @param [in] timeout  Timeout interval, which is invalid during calling in sync mode
    * @return SUCCESS   Success
    * @return   Other: failure
    */
    virtual AIStatus Process(AIContext &context,
```

```
            const std::vector<std::shared_ptr<IAITensor>> &in_data,
            std::vector<std::shared_ptr<IAITensor>> &out_data, uint32_t timeout) = 0;

        /*
        * @brief  Create a list of output tensors.
        * @param [in] in_data  List of input tensors, which may be used for calculating output
        * @param [out] out_data  List of output tensors
        * @return SUCCESS   Success
       * @return   Other: failure
        */
        virtual AIStatus CreateOutputTensor(
            const std::vector<std::shared_ptr<IAITensor>> &in_data,
            std::vector<std::shared_ptr<IAITensor>> &out_data) { (void)in_data;(void)out_data;return SUCCESS; }

        /*
        * @brief   Whether to pre-allocate the output memory. This interface is implemented by the NNNode service. The default value is true.
        */
        virtual bool IsPreAllocateOutputMem() { return true; }

        /*
        * @brief  Check whether the NNNode is valid.
        */
        virtual AIStatus IsValid() { return SUCCESS; }

        /*
        * @brief   Query the synchronization mode supported by the node.
        * @return BOTH   Synchronous and asynchronous modes are both supported.
        * @return ASYNC   Only the asynchronous mode is supported.
        * @return SYNC  Only the synchronous mode is supported.
        */
        virtual AI_NODE_EXEC_MODE GetSupportedExecMode() { return AI_NODE_EXEC_MODE::BOTH; }

#ifdef __LITE__
        /*
        * @brief   Release resources when an NN node is in the idle state and times out. This API is user-defined and applies to the lite scenario. If the model manger is used, the corresponding IdleRelease function should be called.
        */
        virtual AIStatus IdleRelease() { return SUCCESS; }
        /*
        * @brief   Resume services after an NN node times out, including enabling a device and loading a model. This API is user-defined and applies to the lite scenario. If the model manger is used, the corresponding IdleResume function should be called.
        */
        virtual AIStatus IdleResume() { return SUCCESS; }
        /*
        * @brief   Set the maximum idle time. If the maximum idle time is exceeded, the system automatically destroys and releases resources. The default idle time is 60s.
        *        Called in the service constructor function
        *        Implemented in AIServiceBase
        * @param [in] time   Maximum idle time, in ms
        */
        virtual void SetMaxIdleTime(const int32_t time) { (void)time;}
#endif // __LITE__

       /*
        * @brief  Obtain the maximum memory used by the service.
        * @return  Maximum size of the memory used by the service
        */
        virtual uint32_t GetMaxUsedMemory() { return 0; }
    };
```

4.4.8 AINNNodeFactory

Macro that supports service engine self-registration, provides the API for creating an NN node, and supports the query of registered NN node description

information (by name or all). The AINNNodeFactory class is defined in **ai_nn_node.h**.

```
class AINNNodeFactory
{
public:
    static AINNNodeFactory* GetInstance();

    /*
    * @brief   Create an NN node based on the NN node name.
    * @param [in] name   NN node name
    * @return std::shared_ptr<IAINNNode>   Pointer to the NN node object corresponding to the name. If
nullptr is returned, the corresponding NN node cannot be found.
    */
    std::shared_ptr<IAINNNode> CreateNNNode(const std::string &name);

    /*
    * @brief   Obtain the descriptions of all registered NN nodes.
    * @param [out] nnnode_desc   Descriptions of all registered NN nodes
    * @return SUCCESS   Success
    * @return   Other: failure
    */
    void GetAllNNNodeDescription(AINNNodeDescriptionList &nnnode_desc);

    /*
    * @brief   Obtain the description of the NN node based on the NN node name.
    * @param [in] name   NN node name
    * @param [out] engin_desc    NN node description
    * @return SUCCESS   Success
    * @return   Other: failure
    */
    AIStatus GetNNNodeDescription(const std::string &name, AINNNodeDescription &engin_desc);

    /*
    * @brief   Register the NN node creation function.
    * @param [in] nnnode_desc   NN node description
    * @param [in] create_func   Function for creating an NN node
    * @return SUCCESS   Success
    * @return   Other: failure
    */
    AIStatus RegisterNNNodeCreator(const AINNNodeDescription &nnnode_desc,
        AINNNODE_CREATE_FUN create_func);
    AIStatus RegisterNNNodeCreator(const string nnnode_str,
        AINNNODE_CREATE_FUN create_func);

    /*
    * @brief   Deregister the NN node.
    * @param [in] name   NN node name
    * @return SUCCESS   Success
    */
    AIStatus UnRegisterNNNode(const AINNNodeDescription &nnnode_desc);
    AIStatus UnRegisterNNNode(const string nnnode_str);

private:
    std::map<std::string, AINNNODE_CREATE_FUN> create_func_map_;
    std::map<std::string, AINNNodeDescription> nnnode_desc_map_;
    std::mutex  map_lock_;
};
```

4.4.9 AINNNodeRegisterar

NN node registration class. This class is defined in **ai_nn_node.h**.

```
class AINNNodeRegisterar
{
public:
    AINNNodeRegisterar(const AINNNodeDescription &nnnode_desc, AINNNODE_CREATE_FUN
create_func)
    {
```

```
        AINNNodeFactory::GetInstance()->RegisterNNNodeCreator(nnnode_desc, create_func);
        nnnode_desc_ = nnnode_desc;
        nnnode_str_.erase(nnnode_str_.begin(),nnnode_str_.end());
    }

    ~AINNNodeRegisterar(){
        AINNNodeFactory::GetInstance()->UnRegisterNNNode(nnnode_desc_);
    }
private:
    AINNNodeDescription nnnode_desc_;
    string nnnode_str_;
};
```

4.4.10 REGISTER_NN_NODE

NN node registration macro.

```
/*
 * @brief NNNode   Registration macro. The service NN node is used in the implementation class.
 * REGISTER_ENGINE(desc, clazz) is used directly.
 * @param [in] desc   Object of the NN node description information
 * @param [in] clazz  Class name of the NN node
 */
#define REGISTER_NN_NODE(desc, name) \
    std::shared_ptr<IAINNNode> NNNode_##name##_Creator() \
    { \
        return std::make_shared<name>(); \
    } \
    AINNNodeRegisterar g_nnnode_##name##_creator(desc,NNNode_##name##_Creator)
```

4.4.11 AITensorGetBytes

Macro for obtaining the byte size of a tensor. This API is defined in **ai_tensor.h**.

```
extern uint32_t AITensorGetBytes(int32_t data_type);
```

4.4.12 AITensorFactory

Tensor factory class, which is used to create tensors. The **AITensorFactory** class is defined in **ai_tensor.h**.

```
    class AITensorFactory
    {
    public:
        static AITensorFactory* GetInstance();

        /*
        * @brief  Create a tensor by using parameters, including memory allocation
       * @param [in] tensor_desc  Description that contains the tensor parameters
       * @return shared_ptr<IAITensor>  Pointer to the created tensor. If the creation fails, a null pointer is
returned.
        */
        std::shared_ptr<IAITensor> CreateTensor(const AITensorDescription &tensor_desc);

        /*
        * @brief  Create a tensor by using the type, not involving memory allocation. It is used in non-pre-
allocation mode.
        * @param [in] type  Type during tensor registration
        * @return shared_ptr<IAITensor>  Pointer to the created tensor. If the creation fails, a null pointer is
returned.
        */
        std::shared_ptr<IAITensor> CreateTensor(const std::string &type);


#if defined( __ANDROID__) || defined(ANDROID)
        /*
        * @brief   Creates a tensor by using parameters and buffers. The content is obtained from buffer
```

```
deserialization.
     * @param [in] tensor_desc  Description that contains the tensor parameters
      * @param [in] buffer   Existing data buffer
      * @param [in] size   Buffer size
     * @return shared_ptr<IAITensor>  Pointer to the created tensor. If the creation fails, a null pointer is
returned.
      */
      std::shared_ptr<IAITensor> CreateTensor(const AITensorDescription &tensor_desc,
          const void *buffer, const int32_t size);
#else
      std::shared_ptr<IAITensor> CreateTensor(const AITensorDescription &tensor_desc,
          void *buffer, int32_t size);
#endif

      /*
      * @brief   Register a tensor.
      * @param [in] tensor_desc  Tensor description
      * @param [in] create_func   Function for creating a tensor
      */
      AIStatus RegisterTensor(const AITensorDescription &tensor_desc,
          CREATE_TENSOR_FUN create_func);
      AIStatus RegisterTensor(const string tensor_str, CREATE_TENSOR_FUN create_func)
      {
          // Add a lock to prevent multi-thread concurrency.
          AITensorDescription tensor_desc;
          tensor_desc.set_type(tensor_str);
          return RegisterTensor(tensor_desc, create_func);
      }

     /*
      * @brief  Unregister a tensor.
      * @param [in] tensor_desc  Tensor description
      */
      AIStatus UnRegisterTensor(const AITensorDescription &tensor_desc);
      AIStatus UnRegisterTensor(const string tensor_str)
      {
          AITensorDescription tensor_desc;
          tensor_desc.set_type(tensor_str);
          return UnRegisterTensor(tensor_desc);
      }

      /*
      * @brief  Obtain the list of all tensors.
      * @param [out] tensor_desc_list  Description list of output tensors
      */
      void GetAllTensorDescription(AITensorDescriptionList &tensor_desc_list);

      /*
      * @brief  Obtain the tensor description.
      */
      AIStatus GetTensorDescription(const std::string& tensor_type_name,
          AITensorDescription &tensor_desc);

  private:
      std::map<std::string, AITensorDescription> tensor_desc_map_;
      std::map<std::string, CREATE_TENSOR_FUN> create_func_map_;
      std::mutex tensor_reg_lock_;
  };
```

4.4.13 REGISTER_TENSOR_CREATER_UNIQUE

Tensor registration macro. This macro is defined in **ai_tensor.h**.

```
#define REGISTER_TENSOR_CREATER_UNIQUE(desc, name, func) \
  AITensorRegisterar g_##name##_tensor_registerar(desc, func)
```


4.4.14 REGISTER_TENSOR_CREATER

Macro for encapsulating the tensor registration macro. This macro is defined in **ai_tensor.h**.

```
#define REGISTER_TENSOR_CREATER(name, desc,func) \
    REGISTER_TENSOR_CREATER_UNIQUE(desc, name, func)
```

4.4.15 AISimpleTensor

Tensor of a simple data type. This class is defined in **ai_tensor.h**.

```
class AISimpleTensor : public IAITensor
{
 public:
    AISimpleTensor();

    ~AISimpleTensor();

    /*
    * @brief   Obtain the type name.
    */
    virtual const char* const GetTypeName() override;

   /* Obtain the data address. */
    void* GetBuffer();

    /*
    @brief   Set the data address.
    @param [in] data  Data pointer
    @param [in] size  Size
    @param [in] isown   Whether the tensor releases the data address memory after the tensor life cycle ends.
    */
    void SetBuffer(void *data, const int32_t size, bool isown=false);

    /*
    @brief   Obtain the space occupied by data.
    */
    uint32_t GetSize();

    /*
    @brief Allocate data space.
    */
    void* MallocDataBuffer(uint32_t size);

    /*
    * @brief   Obtain the byte length after serialization.
    */
    virtual uint32_t ByteSizeLong() override;

    /*
    * @brief   Set parameters through AITensorDescription.
    * @param [in] tensor_desc  Tensor description
    * @return true: Initialization success
    * @return false: Initialization failure. The possible cause is that tensor_desc is inconsistent with the current tensor.
    */
    virtual bool Init(const AITensorDescription &tensor_desc) override;

#if defined( __ANDROID__) || defined(ANDROID)
    /*
    * @brief   Serialize data to the buffer for cross-process data interaction.
    * @param [in] buffer   Address of the buffer to which the data is serialized, allocated by the caller
    * @param [in] size   Size of the buffer to which the data is output
    * @return SUCCESS   Success
    *         FAILED: failure. If the tensor does not support cross-process, this API does not need to be implemented and a failure message is returned.
```

```
    */
    virtual AIStatus SerializeTo(void* buffer, const uint32_t size) override;

    /*
    * @brief   Deserialize data from the buffer to a tensor for cross-process data interaction.
    * @param [in] buffer  Size of the buffer to which the data is input
    * @param [in] size   Size of the buffer to which the data is input
    * @return SUCCESS   Success
    *         FAILED: failure. If the tensor does not support cross-process, this API does not need to be
implemented and a failure message is returned.
    */
    virtual AIStatus DeserializeFrom(const void* buffer, const uint32_t size) override;
#endif

    /*
    @brief  Build tensor description.
    */
    static AITensorDescription BuildDescription(const std::string& size = "");

    /*
    @brief   Create a tensor according to the description.
    */
    static std::shared_ptr<IAITensor> CreateTensor(const AITensorDescription& tensor_desc);

    bool IsSupportZerocpy();
    /*
    @brief   Set the data address.
    @param [in] data  Data pointer
    @param [in] size  Size
    @param [in] isown   Whether the tensor releases the data address memory after the tensor life cycle
ends.
    @param [in] is_data_support_mem_share_   Whether the tensor releases the data address memory
after the tensor life cycle ends.
    */
    void SetBufferAttr(void *data, int32_t size, bool isowner, bool is_support_mem_share) override;

private:
    void* data_;
    uint32_t size_;
    bool isowner_ ;

    static std::string type_name_;
}
```

4.4.16 AINeuralNetworkBuffer

Defines the general buffer of the NN. This class is defined in **ai_tensor.h**.

```
class AINeuralNetworkBuffer : public AISimpleTensor
{
public:
    AINeuralNetworkBuffer()
    {
        data_type_ = 0;
        number_    = 1;
        channel_   = 1;
        height_    = 1;
        width_     = 1;
        name_      = "";
    };

    ~AINeuralNetworkBuffer() {};
    /*
    * @brief   Obtain the type name.
    */
    const char* const GetTypeName();

    /*
    @brief  Obtain the byte size.
```

```
    */
    uint32_t ByteSizeLong();

    /*
    @brief Initialization
    @param [in] tensor_desc  Tensor description
    */
    bool Init(const AITensorDescription &tensor_desc);

#if defined( __ANDROID__) || defined(ANDROID)
    /*
    * @brief   Serialize data to the buffer for cross-process data interaction.
    * @param [in] buffer   Address of the buffer to which the data is serialized, allocated by the caller
    * @param [in] size   Size of the buffer to which the data is output
    * @return SUCCESS   Success
    *         FAILED: failure. If the tensor does not support cross-process, this API does not need to be
implemented and a failure message is returned.
    */
    AIStatus SerializeTo(void* buffer, uint32_t size_);

    /*
    * @brief   Deserialize data from the buffer to a tensor for cross-process data interaction.
    * @param [in] buffer  Size of the buffer to which the data is input
    * @param [in] size   Size of the buffer to which the data is input
    * @return SUCCESS   Success
    *         FAILED: failure. If the tensor does not support cross-process, this API does not need to be
implemented and a failure message is returned.
    */
    AIStatus DeserializeFrom(const void* buffer, uint32_t size_);
#endif
    /*
    * @brief  Obtain the description.
    */
    static AITensorDescription GetDescription(
        const std::string &size = "0",const std::string &data_type="0",
        const std::string &number="0", const std::string &channel="0",
        const std::string &height="0", const std::string &width="0");

    /*
    @brief Create a tensor.
    */
    static std::shared_ptr<IAITensor> CreateTensor(const AITensorDescription& tensor_desc);

    /*
    @brief  Obtain the number.
    */
    int32_t GetNumber();

    /*
    @brief  Set the number.
    */
    void SetNumber(int32_t number);

    /*
    @brief  Obtain the number of channels.
    */
    int32_t GetChannel();

    /*
    @brief  Set the channel.
    */
    void SetChannel(int32_t channel);

    /*
    @brief  Obtain the height.
    */
    int32_t GetHeight();

    /*
```

```
    @brief  Set the height.
     */
     void SetHeight(int32_t height);

     /*
    @brief  Obtain the width.
     */
     int32_t GetWidth();

     /*
    @brief  Set the width.
     */
     void SetWidth(int32_t width);

     /*
    @brief  Obtain the data type.
     */
     int32_t GetData_type();

     /*
    @brief  Set the data type.
     */
     void SetData_type(int32_t data_type);

     /*
    @brief  Obtain the data type.
     */
     const std::string& GetName() const;

     /*
    @brief  Set the data type.
     */
     void SetName(const std::string& value);

private:
    int32_t data_type_;
    int32_t number_;
    int32_t channel_;
    int32_t height_;
    int32_t width_;
    std::string name_;
};
```

4.4.17 AIImageBuffer

Defines the general buffer of images. This class is defined in **ai_tensor.h**.

```
class AIImageBuffer : public AISimpleTensor
{
public:
    AIImageBuffer()
    {
        format_ = JPEG;
        width_  = 0;
        height_ = 0;
    };

    /*
   * @brief   Obtain the type name.
    */
    const char* const GetTypeName();

    /*
   @brief  Obtain the byte size.
    */
    uint32_t ByteSizeLong();

    /*
   @brief Initialization
```

```
        @param [in] tensor_desc  Tensor description
        */
        bool Init(const AITensorDescription &tensor_desc);

#if defined( __ANDROID__) || defined(ANDROID)
        /*
        * @brief   Serialize data to the buffer for cross-process data interaction.
        * @param [in] buffer   Address of the buffer to which the data is serialized, allocated by the caller
        * @param [in] size   Size of the buffer to which the data is output
        * @return SUCCESS   Success
        *         FAILED: failure. If the tensor does not support cross-process, this API does not need to be
implemented and a failure message is returned.
        */
        AIStatus SerializeTo(void* buffer, uint32_t size_);

        /*
        * @brief   Deserialize data from the buffer to a tensor for cross-process data interaction.
        * @param [in] buffer  Size of the buffer to which the data is input
        * @param [in] size   Size of the buffer to which the data is input
        * @return SUCCESS   Success
        *         FAILED: failure. If the tensor does not support cross-process, this API does not need to be
implemented and a failure message is returned.
        */
        AIStatus DeserializeFrom(const void* buffer, uint32_t size_);
#endif
        /*
        * @brief  Obtain the description.
        */
        static AITensorDescription GetDescription(
            const std::string &size = "0",const std::string &height="0",
            const std::string &width="0", const std::string format="JPEG");

        /*
       @brief Create a tensor.
        */
        static std::shared_ptr<IAITensor> CreateTensor(const AITensorDescription& tensor_desc);

        /*
        @brief Obtain the image format.
        */
        ImageFormat GetFormat();
        /*
        @brief  Set the image format.
        */
        void SetFormat(ImageFormat format);

        /*
       @brief  Obtain the height.
        */
        int32_t GetHeight();

        /*
       @brief  Set the height.
        */
        void SetHeight(int32_t height);

        /*
       @brief  Obtain the width.
        */
        int32_t GetWidth();

        /*
       @brief  Set the width.
        */
       void SetWidth(int32_t width);
    private:
       ImageFormat format_;
       int32_t width_;
```



```
    int32_t height_;
};
```

4.4.18 HIAILog

Log class of HiAI. This class is defined in **log.h**.

```
class HIAILog {
public:
  /**
  * @ingroup hiaiengine
  * @brief   Obtain the HIAILog pointer.
  */
  HIAI_LIB_INTERNAL static HIAILog* GetInstance();

  /**
  * @ingroup hiaiengine
  * @brief: Whether to initialize HiAILog or not
  * @return: Whether to initialize HiAILog or not. "true" indciates yes.
  */
  HIAI_LIB_INTERNAL bool IsInit() { return isInitFlag; }

  /**
  * @ingroup hiaiengine
  * @brief   Obtain the log output level.
  * @return: Log output level
  */
  HIAI_LIB_INTERNAL uint32_t HIAIGetCurLogLevel();

  /**
  * @ingroup hiaiengine
  * @brief   Check whether the log needs to be output.
  * @param [in]errorCode: Error code of a message
  * @return   Whether a log can be output
  */
  HIAI_LIB_INTERNAL bool HIAILogOutputEnable(const uint32_t errorCode);

  /**
  * @ingroup hiaiengine
  * @brief   Level corresponding to the log
  * @param [in]logLevel: Log level in the macro definition
  */
  HIAI_LIB_INTERNAL std::string HIAILevelName(const uint32_t logLevel);

  /**
  * @ingroup hiaiengine
  * @brief   Output a log.
  * @param [in]moudleID: Component ID defined by enum
  * @param [in]logLevel: Log level in the macro definition
  * @param [in]strLog: Output log content
  */
  HIAI_LIB_INTERNAL void HIAISaveLog(const int32_t moudleID,
      const uint32_t logLevel, const char* strLog);
protected:
  /**
  * @ingroup hiaiengine
  * @brief   HIAILog constructor
  */
  HIAI_LIB_INTERNAL HIAILog();
  HIAI_LIB_INTERNAL ~HIAILog() {}
private:
  /**
  * @ingroup hiaiengine
  * @brief   HIAILog initialization function
  */
  HIAI_LIB_INTERNAL void Init();
  HIAI_LIB_INTERNAL HIAILog(const HIAILog&) = delete;
  HIAI_LIB_INTERNAL HIAILog(HIAILog&&) = delete;
  HIAI_LIB_INTERNAL HIAILog& operator=(const HIAILog&) = delete;
```



```
    HIAI_LIB_INTERNAL HIAILog& operator=(HIAILog&&) = delete;
  private:
    HIAI_LIB_INTERNAL static std::mutex mutexHandle;
    uint32_t outputLogLevel;
    std::map<uint32_t, std::string> levelName;
    std::map<std::string, uint32_t> levelNum;
    static bool isInitFlag;
};
```

4.4.19 HIAI_ENGINE_LOG

Log encapsulation macro, which encapsulates the **HIAI_ENGINE_LOG_IMPL** macro. This macro is defined in **log.h**.

```
#define HIAI_ENGINE_LOG(...) \
    HIAI_ENGINE_LOG_IMPL(__FUNCTION__, __FILE__, __LINE__, ##__VA_ARGS__)
```

4.5 Exception Handling

If the offline model manager API fails to be called and the return value is **FAILED**, you can view logs in the **Log** window on Mind Studio. View the latest logs based on the time and rectify the error based on the log information.

Example: When the **Init** API is called, the model to be loaded does not exist.

```
Init:hiaiengine/node/ai_model_manager_impl.cpp:163:"Load models failed!"
```

NOTE

For details about how to view logs, see section 2.3.1 "Viewing a Log" in *Ascend 310 Mind Studio Auxiliary Tools*.

5 Auxiliary APIs

5.1 Data Receiving APIs (C++ Language)
5.2 Data Type Serialization and Deserialization (C++ Language)
5.3 Memory Management (C Language)
5.4 Memory Management (C++ Language)
5.5 Logs (C++)
5.6 MultiTypeQueue API for Queue Management (C++ Language)
5.7 Event Registration API (C++ Language)
5.8 Others (C++ Language)

5.1 Data Receiving APIs (C++ Language)

You can obtain related parameters by using the following auxiliary APIs. Most APIs for obtaining data are common between the C language and the C++ language. The APIs dedicated to C++ are described in the syntax.

5.1.1 Obtaining the Number of Devices

Obtains the number of devices. This API is defined in **c_graph.h**.

Syntax

```
HIAI_StatusT HIAI_GetDeviceNum(uint32_t *devCount);
```

Parameter Description

Parameter	Description	Value Range
devCount	Pointer to the number of output devices	-

Return Value

For details about the returned error codes, see "Error Codes."

Error Codes

No.	Error Level	Error Code	Description
1	HIAI_INFO	HIAI_OK	The running is OK.
2	HIAI_ERROR	HIAI_GRAPH_GET_DEVNUM_ERROR	There is an error in obtaining the number of devices.

Calling Example

```
uint32_t dev_count;
HIAI_GetDeviceNum(&dev_count);
```

5.1.2 Obtaining the ID of the First Device

Obtains the ID of the first device. This API is defined in **c_graph.h**.

Syntax

HIAI_StatusT HIAI_GetFirstDeviceId(uint32_t *firstDevID);

Parameter Description

Parameter	Description	Value Range
firstDevID	Pointer to the ID of the first device	-

Return Value

For details about the returned error codes, see "Error Codes."

Error Codes

No.	Error Level	Error Code	Description
1	HIAI_INFO	HIAI_OK	The running is OK.
2	HIAI_ERROR	HIAI_GRAPH_GET_DEVID_ERROR	There is an error in obtaining the device ID.

Calling Example

```
uint32_t first_dev_id;
HIAI_GetFirstDeviceId(&first_dev_id);
```

5.1.3 Obtaining the ID of the First Graph

Obtains the graph ID. This API is defined in **c_graph.h**.

Syntax

Common API between C and C++: **HIAI_StatusT HIAI_GetFirstGraphId(uint32_t *firstGraphID);**

Parameter Description

Parameter	Description	Value Range
firstGraphID	Pointer to the ID of the first graph	-

Return Value

Common API between C and C++: Error code

Error Codes

No.	Error Level	Error Code	Description
1	HIAI_INFO	HIAI_OK	The running is OK.
2	HIAI_ERROR	HIAI_GRAPH_GET_GRAPHID_ERROR	There is an error in obtaining the graph ID.

Calling Example

```
uint32_t first_graph_id;
HIAI_GetFirstGraphId(&first_graph_id);
```

5.1.4 Obtaining the ID of the Next Device

Obtains the ID of the next device. This API is defined in **c_graph.h**.

Syntax

HIAI_StatusT HIAI_GetNextDeviceId(const uint32_t curDevID,uint32_t *nextDevID);

Parameter Description

Parameter	Description	Value Range
curDevID	ID of the current device	-
nextDevID	Pointer to the ID of the next device	-

Return Value

For details about the returned error codes, see "Error Codes."

Error Codes

No.	Error Level	Error Code	Description
1	HIAI_INFO	HIAI_OK	The running is OK.
2	HIAI_ERROR	HIAI_GRAPH_GET_DEVID_ERROR	There is an error in obtaining the device ID.

Calling Example

```
uint32_t cur_dev_id = 1;
uint32_t next_dev_id;
HIAI_GetNextDeviceId(cur_dev_id, &next_dev_id);
```

5.1.5 Obtaining the ID of the Next Graph

Obtains the ID of the next graph. This API is defined in **c_graph.h**.

Syntax

Common API between C and C++: **HIAI_StatusT HIAI_GetNextGraphId(const uint32_t curGraphID,uint32_t *nextGraphID);**

Parameter Description

Parameter	Description	Value Range
curGraphID	ID of the current graph	-
nextGraphID	Pointer to the ID of the next graph	-

Return Value

For details about the returned error codes, see "Error Codes."

Error Codes

No.	Error Level	Error Code	Description
1	HIAI_INFO	HIAI_OK	The running is OK.
2	HIAI_ERROR	HIAI_GRAPH_GET_GRAPHID_ERROR	There is an error in obtaining the graph ID.

Calling Example

```
uint32_t cur_graph_id = 1;
uint32_t next_graph_id;
HIAI_GetNextGraphId(cur_graph_id, &next_graph_id);
```

5.1.6 Obtaining the Engine Pointer

Queries the pointer to the engine object based on the engine ID. This API is defined in **graph.h**.

Syntax

std::shared_ptr<Engine> Graph::GetEngine(uint32_t engineID);

Parameter Description

Parameter	Description	Value Range
engineID	ID of the target engine	-

Return Value

Smart pointer to the engine

Calling Example

```
uint32_t engineID= 1000;
auto graphPtr = Graph::GetInstance(100);
auto enginePtr = graphPtr->GetEngine(engineID);
```

5.1.7 Obtaining the Graph ID

Obtains the ID of the current graph. This API is defined in **graph.h**.

Syntax

uint32_t Graph::GetGraphId();

Return Value

Graph ID

5.1.8 Obtaining the Device ID of a Graph

Obtains the ID of the current graph on the device side. This API is defined in **graph.h**.

Syntax

uint32_t Graph::GetDeviceID();

Return Value

ID of the current graph on the device side

Calling Example

```
auto graphPtr = Graph::GetInstance(100);
auto deviceId = graphPtr->GetDeviceID();
```

5.1.9 Obtaining the Graph ID of an Engine

Obtains the graph ID of an engine. This API is defined in **engine.h**.

Syntax

uint32_t Engine::GetGraphId();

Return Value

Graph ID of an engine

Calling Example

```
uint32_t engineID= 1000;
auto graphPtr = Graph::GetInstance(100);
auto enginePtr = graphPtr->GetEngine(engineID);
auto graphID= enginePtr->GetGraphId();
```

5.1.10 Obtaining the Maximum Queue Size of an Engine

Obtains the maximum queue size of an engine. This API is defined in **engine.h**.

Syntax

const uint32_t Engine::GetQueueMaxSize();

Return Value

Maximum queue size of the engine

Calling Example

```
uint32_t engineID= 1000;
auto graphPtr = Graph::GetInstance(100);
auto enginePtr = graphPtr->GetEngine(engineID);
auto engineQueueSize = enginePtr->GetQueueMaxSize();
```


5.1.11 Obtaining the Current Queue Size of a Specified Port of the Engine

Obtains the queue size of a specified port of the engine. This API is defined in **engine.h**.

Syntax

const uint32_t Engine::GetQueueCurrentSize(const uint32_t portID);

Parameter Description

Parameter	Description	Value Range
portID	Port ID	-

Return Value

Queue size corresponding to the port ID of the engine

Calling Example

```
uint32_t engineID= 1000;
uint32_t portID = 0;
auto graphPtr = Graph::GetInstance(100);
auto enginePtr = graphPtr->GetEngine(engineID);
auto engineQueueSize = enginePtr->GetQueueCurrentSize(portID);
```

5.1.12 Parsing the Matrix Configuration File

Parses the configuration file. The Matrix parses the configuration file and writes the generated graph back to the list for the user. This API is defined in **graph.h**.

Syntax

static HIAI_StatusT Graph::ParseConfigFile(const std::string& configFile, GraphConfigList& graphConfigList)

Parameter Description

Parameter	Description	Value Range
configFile	Configuration file path. Ensure that the transferred file path is correct.	-
graphConfigList	Protobuf data format.	-

Return Value

For details about the returned error codes, see "Error Codes."

Error Codes

No.	Error Level	Error Code	Description
1	-	HIAI_GRAPH_GET_INSTANCE_NULL	-

5.1.13 Obtaining the PCIe Information

Obtains the Peripheral Component Interconnect Express (PCIe) information. This API is defined in **c_graph.h**.

Syntax

static HIAI_StatusT HIAI_GetPCIeInfo(const uint32_t devId,int32_t* bus, int32_t* dev, int32_t* func);

Parameter Description

Parameter	Description	Value Range
devId	Device ID to be queried	-
bus	Returned PCIe bus ID	-
dev	Returned PCIe device ID	-
func	Returned PCIe function ID	-

Return Value

For details about the returned error codes, see "Error Codes."

Error Codes

No.	Error Level	Error Code	Description
1	HIAI_INFO	HIAI_OK	The running is OK.
2	HIAI_ERROR	HIAI_GRAPH_GET_PCIEINFO_ERROR	The **bus**, **dev**, or **func** parameter has a null pointer or there is an error in obtaining the PCIe information.

Calling Example

```
uint32_t devId= 1;
uint32_t bus;
```



```
uint32_t dev;
uint32_t func;
HIAI_GetPCIeInfo(devId, &bus, &dev, &func);
```

5.1.14 Obtaining the Version Number

Obtains the version number. This API is defined in **c_graph.h**.

Syntax

HIAI_API_VERSION HIAI_GetAPIVersion();

Return Value

Enumerated information of the version number

Calling Example

```
HIAI_API_VERSION HIAI_GetAPIVersion();
```

5.1.15 Obtaining the OamConfig Smart Pointer

Obtains the OamConfig smart pointer, that is, to obtain OAM configuration information, including the model name, whether to enable the dump function, and layers at which operators need to be dumped in the model. This API is defined in **engine.h**.

Syntax

OAMConfigDef* GetOamConfig();

Parameter Description

None

Return Value

OamConfig smart pointer. For details about the definition of the **OAMConfigDef** type, see **7.2.1 Protobuf Data Types**.

5.2 Data Type Serialization and Deserialization (C++ Language)

Automatic serialization and deserialization mechanisms are provided for various user-defined data types.

The following API is used for encapsulating related macros:

```
static HIAIDataTypeFactory* HIAIDataTypeFactory::GetInstance();
```

5.2.1 Macro: HIAI_REGISTER_DATA_TYPE

Provides automatic serialization and deserialization mechanisms for various user-defined data types. This macro is defined in **data_type_reg.h**.

NOTE

- This API must be registered on both the host side and the device side.
- Due to the restrictions of Cereal, do not use the **long double** type in the struct body. To ensure running across platforms (for example, from Windows to Linux), the portable type is recommended for the struct members, for example, **int32_t**, in the struct body.
- Due to the restrictions of Cereal, the struct body needs to use the shared pointer.

Syntax

HIAI_REGISTER_DATA_TYPE(name, type)

Parameter Description

Parameter	Description	Value Range
name	Name of the user-defined data type. (The name must be unique for each data type.)	-
type	User-defined data type	-

Return Value

For details about the returned error codes, see "Error Codes."

Error Codes

No.	Error Code	Description
1	HIAI_OK	The running is OK.
2	HIAI_ENGINE_FUNCTOR_NULL	The HiAI Engine function is null.
3	HIAI_ENGINE_FUNCTOR_EXIST	The HiAI Engine function exists.

5.2.2 Macro: HIAI_REGISTER_TEMPLATE_DATA_TYPE

Provides automatic serialization and deserialization mechanisms for various data types of a user-defined template. This macro is defined in **data_type_reg.h**.

NOTE

This API needs to be registered on both the host side and the device side.

Syntax

HIAI_REGISTER_TEMPLATE_DATA_TYPE (name, type, basictype1, basictype2, ...)

Parameter Description

Parameter	Description	Value Range
name	Name of the user-defined data type. (The name must be unique for each data type.)	-
type	Data type of a user-defined template	-
basicType1	User-defined data type	-
basicType2	User-defined data type	-
...	...	-

Return Value

For details about the returned error codes, see "Error Codes."

Error Codes

No.	Error Code	Description
1	HIAI_OK	The running is OK.
2	HIAI_ENGINE_FUNCTOR_NULL	The HiAI Engine function is null.
3	HIAI_ENGINE_FUNCTOR_EXIST	The HiAI Engine function exists.

5.2.3 Macro: HIAI_REGISTER_SERIALIZE_FUNC

Provides user-defined serialization and deserialization mechanisms for user-defined data types. This macro is defined in **data_type_reg.h**.

That is, during data sending, the struct pointers transferred by the user are converted into the struct buffer and data buffer. During data receiving, the struct buffer and data buffer obtained from the framework are converted into the structs.

Application scenario:

This API is used to quickly transfer data from the host side to the device side. If you want to improve the transfer performance, you must call this API.

NOTE

- This API is used to quickly transfer data from the host side to the device side.
- If you want to use a high-performance transfer API, you must call this API to register serialization and deserialization functions.

Syntax

HIAI_REGISTER_SERIALIZE_FUNC(name, type, hiaiSerializeFunc, hiaiDeSerializeFunc)

Parameter Description

Parameter	Description	Value Range
name	Name of a registered message	-
type	Type of a user-defined type	-
hiaiSerializeFunc	Serialization function. The function format is as follows: typedef void(*hiaiSerializeFunc) (void* inputPtr, std::string& ctrlStr, uint8_t*& dataPtr, uint32_t& dataLen); The parameters are described as follows: • **inputPtr**: struct pointer • **ctrlStr**: struct buffer • **dataPtr**: buffer of the struct data pointer • **dataLen**: struct data size	-

Parameter	Description	Value Range
hiaiDeSerializeFunc	Deserialization function. The function format is as follows: typedef std::shared_ptr<void>(***hiaiDeSerializeFunc**)(const char* ctrlPtr, const uint32_t& ctrlLen, const uint8_t* dataPtr, const uint32_t& dataLen); The parameters are described as follows: • **ctrlPtr**: struct pointer • **ctrlLen**: control information size in the data structure • **dataPtr**: struct data pointer • **dataLen**: size of the data storage space in the struct. This parameter is used only for verification and does not indicate the size of the original data.	-

Return Value

For details about the returned error codes, see "Error Codes."

Error Codes

No.	Error Code	Description
1	HIAI_OK	The running is OK.
2	HIAI_ENGINE_FUNCTOR_NULL	The HiAI Engine function is null.
3	HIAI_ENGINE_FUNCTOR_EXIST	The HiAI Engine function exists.

5.2.4 Graph::ReleaseDataBuffer

During data deserialization, this function is registered as a deleter when a value is assigned to the smart pointer in the data memory. This API is defined in **graph.h**.

Syntax

static void Graph::ReleaseDataBuffer(void* ptr)

Parameter Description

Parameter	Description	Value Range
ptr	Memory pointer	-

5.2.5 API Usage Example

```
/**********************************************************************************
*                    CopyRight (C) HiSilicon Co., Ltd.
*
*      Filename:  user_def_datatype.h
*   Description:  User defined data type
*
*       Version:  1.0
*       Created:  2018-01-08 10:15:18
*        Author:
*
*      Revision:  initial draft;
**********************************************************************************/
#ifndef USE_DEF_DATA_TYPE_H_
#define USE_DEF_DATA_TYPE_H_

#ifdef __cplusplus
#include <vector>
#include <map>
#endif

// User-defined data type (C language)
typedef struct UseDefDataType
{
   int data1;
   float data2;
}UseDefDataTypeT;

#ifdef __cplusplus
// User-defined data type (with template)
template<typename T1, typename T2, typename T3>
class UseDefTemplateDataType
{
  public:
   std::vector<T1> data_vec_;
   std::map<T2, T3> data_map_;
};
#endif

#endif
/**********************************************************************************
*                    CopyRight (C) HiSilicon Co., Ltd.
*
*      Filename:  use_def_data_type_reg.cpp
*   Description:  User defined Data Type Register
*
*       Version:  1.0
*       Created:  2018-01-08 10:15:18
*        Author:  h00384043
*
*      Revision:  initial draft;
**********************************************************************************/
```

```
#include "use_def_data_type.h"
#include "hiaiengine/data_type_reg.h"

// The user-defined data type must be registered with the Matrix before being used as the communication
interface between engines.
template<class Archive>
void serialize(Archive& ar, UseDefDataTypeT& data)
{
    ar(data.data1, data.data2);
}

// Register UseDefDataTypeT.
HIAI_REGISTER_DATA_TYPE("UseDefDataTypeT", UseDefDataTypeT)


// Register the data of the template type.
// All the required template types must be registered.
template<class Archive, typename T1, typename T2, typename T3>
void serialize(Archive& ar, UseDefTemplateDataType<T1, T2, T3>& data)
{
    ar(data.data_vec_, data.data_map_);
}

// Register UseDefTemplateDataType<int, float>.
HIAI_REGISTER_TEMPLATE_DATA_TYPE("UseDefTemplateDataType_uint64_t_float_string",
UseDefTemplateDataType, uint64_t, float, std::string);

//... Register other types.

// Register UseDefTemplateDataType<int, int>.
HIAI_REGISTER_TEMPLATE_DATA_TYPE("UseDefTemplateDataType_uint64_t_uint64_t_uint64_t",
UseDefTemplateDataType, uint64_t, uint64_t, uint64_t);
// Serialization and deserialization of registered structs
typedef struct tagST_SC_IMAGE
{
    INT32  iWidth;                // image width
    INT32  iHeight;               // image height
    INT32  iWidthStep;              // Size of aligned image row in bytes
    INT32  iChannel;               // channels
    INT32  iDepth;                // depth in bits
    UINT32 uiTimeStampH;
    UINT32 uiTimeStampL;
    UINT32 iSize;
    UINT64 uiID;
    EN_SC_IMAGE_TYPE eType;          // image type
    ST_SC_IMAGE_ROI *roi;           // Image ROI. If NULL, the whole image is selected
    UINT8*  pucImageData;           // Image data
    UINT64 uiPhyAddr;
}ST_SC_IMAGE;

std::shared_ptr<ST_SC_IMAGE> st_image = std::make_shared<ST_SC_IMAGE>();
st_image->iWidth = 1080;
st_image->iHeight = 1080;
st_image->iChannel = 1;
st_image->iWidthStep = 1080;
st_image->iSize = 1080*1080*3/2 ;
st_image->eType = SC_IMAGE_YUV420SP;
st_image->roi = (ST_SC_IMAGE_ROI*)malloc(sizeof(ST_SC_IMAGE_ROI));
st_image->pucImageData = nullptr;
uint8_t* align_buffer = nullptr;
HIAI_StatusT get_ret =  HIAIMemory::HIAI_DMalloc(st_image->iSize,(void*&)align_buffer,10000);
hiai::EnginePortID engine_id;
engine_id.graph_id = 1;
engine_id.engine_id = 1;
engine_id.port_id = 0;
HIAI_SendData(engine_id, "ST_SC_IMAGE", std::static_pointer_cast<void>(st_image),100000);

void GetStScImageSearPtr(void* inputPtr, std::string& ctrlStr, uint8_t*& dataPtr, uint32_t& dataLen)
```



```
{
  // If the struct body contains multiple pointers in addition to image, the pointers are concateneated.
   ST_SC_IMAGE* st_image = (ST_SC_IMAGE*)inputPtr;

   ctrlStr = std::string((char*)inputPtr, sizeof(ST_SC_IMAGE));
   if(nullptr != st_image->roi)
   {
       std::string image_roi_str = std::string((char*)st_image->roi, sizeof(ST_SC_IMAGE_ROI));
       ctrlStr += image_roi_str;
   }

   dataPtr = (UINT8*)st_image->pucImageData;
   dataLen= st_image->iSize;
}

std::shared_ptr<void> GetStScImageDearPtr(const char* ctrlPtr, const uint32_t& ctrlLen, const uint8_t*
dataPtr, const uint32_t& dataLen)
{
   ST_SC_IMAGE* st_sc_image = (ST_SC_IMAGE*)ctrlPtr;
   std::shared_ptr<hiai::BatchImagePara<uint8_t>> ImageInfo(new hiai::BatchImagePara<uint8_t>);
   hiai::ImageData<uint8_t> image_data;
   image_data.width = st_sc_image->iWidth;
   image_data.height = st_sc_image->iHeight;
   image_data.channel = st_sc_image->iChannel;
   image_data.width_step = st_sc_image->iWidthStep;
   if (st_sc_image->eType == SC_IMAGE_U8C3PLANAR)
   {
      image_data.size = st_sc_image->iWidth * st_sc_image->iHeight * 3;
      image_data.format = hiai::BGR888;
   }
   else if (SC_IMAGE_U8C3PACKAGE == st_sc_image->eType)
   {
      image_data.size = st_sc_image->iWidth * st_sc_image->iHeight * 3;
      image_data.format = hiai::BGR888;
   }
   else if (st_sc_image->eType == SC_IMAGE_YUV420SP)
   {
      image_data.size = st_sc_image->iSize;//st_sc_image->iWidth * st_sc_image->iHeight * 3 / 2;
      image_data.format = hiai::YUV420SP;
   }
   image_data.data.reset(dataPtr, hiai::Graph::ReleaseDataBuffer);
   ImageInfo->v_img.push_back(image_data);
   ImageInfo->b_info.frame_ID.push_back(0);
   ImageInfo->b_info.batch_size = ImageInfo->b_info.frame_ID.size();

   return std::static_pointer_cast<void>(ImageInfo);
}

HIAI_REGISTER_SERIALIZE_FUNC("ST_SC_IMAGE", ST_SC_IMAGE, GetStScImageSearPtr,
GetStScImageDearPtr);
```

5.3 Memory Management (C Language)

Precautions for API Usage

When memory is allocated by using **HIAI_DMalloc**, pay attention to the following issues about memory management:

- When memory to be automatically freed is allocated for host-device or device-host data sending, if a smart pointer is used, Matrix automatically frees the memory, so the destructor specified by the smart pointer must be null. If the pointer is not a smart pointer, Matrix automatically frees the memory.

- When memory to be manually freed is allocated for host-device or device-host data sending, if a smart pointer is used, you need to set the destructor to **HIAI_DFree**. If the pointer is not a smart pointer, **HIAI_DFree** must be called to free the memory after data sending is complete.
- When memory to be manually freed is allocated, the **SendData** API cannot be called repeatedly to send data in the memory.
- When memory to be manually freed is allocated for host-device or device-host data sending, do not reuse the data in the memory before the memory is freed. If the memory is used for host-host or device-device data sending, the data in the memory can be reused before the memory is freed.
- When memory to be manually freed is allocated, if the **SendData** API is called to asynchronously send data, data in the memory cannot be modified after being sent.

If the **HIAI_DVPP_MAlloc** API is called to allocate memory for device-host data sending, the **HIAI_DVPP_DFree** API must be called to manually free the memory, because the **HIAI_DVPP_MAlloc** API does not automatically free the memory. If a smart pointer is used to store the allocated memory address, the destructor must be set to **HIAI_DVPP_DFree**.

5.3.1 HIAI_DMalloc

Allocates memory blocks on the host side or device side to support efficient data transmission. This API is defined in **c_graph.h**.

To allocate memory for DVPP on the device side, see **5.3.3 HIAI_DVPP_DMalloc**.

Application scenario:

With Matrix, this API is used to send large data from the host side to the device side. It must be used together with **5.2.3 Macro: HIAI_REGISTER_SERIALIZE_FUNC**.

For example, a 1080p or 4K image needs to be sent to the device side. To improve the data sending performance, you must register the structure serialization and deserialization functions through **HIAI_REGISTER_SERIALIZE_FUNC**. In addition, the large data block memory is allocated through **HIAI_DMalloc**. This greatly improves the data sending performance.

NOTE

This API is used to send large data between the host side and the device side. If there is no performance requirement or no data sending is involved, this API is not recommended.

Syntax

void* HIAI_DMalloc (const uint32_t dataSize, const uint32_t timeOut, uint32_t flag)

Parameter Description

Parameter	Description	Value Range
dataSize	Size of a memory block	• If this API is called to allocate memory on the host side, the value range is 0–256 MB, excluding 0. The recommended range is 256 KB–256 MB. • If this API is called to allocate memory on the device side, the value range is 0 to (256 MB – 96 bytes), excluding 0. The recommended range is (256 KB – 96 bytes) to (256 MB – 96 bytes). The model manager on the device side occupies 96 bytes.
timeOut	Timeout when memory fails to be allocated. The default value is 500 ms.	-

Parameter	Description	Value Range

flag	• If **flag** is set to **MEMORY_ATTR_AUTO_FREE** (default value), it indicates that if the memory is allocated and data is sent to the peer end using **SendData**, the **HIAI_DFree** API does not need to be called. After the programming is complete, the memory is automatically freed. If the memory is allocated but the data is not sent to the peer end using **SendData**, the **HIAI_DFree** API needs to be called to free the memory.
	• If **flag** is set to **MEMORY_ATTR_MANUAL_FREE**, the **HIAI_DFree** API must be called to free the memory in any case.
	• **MEMORY_ATTR_NONE** and **MEMORY_ATTR_MAX** are not used currently.
		typedef enum {
		MEMORY_ATTR_NONE = 0,
		// The framework automatically frees the memory allocated by using **DMalloc**.
		MEMORY_ATTR_AUTO_FREE = (0x1 << 1),
		// **DFree** needs to be called manually to free the memory.
		MEMORY_ATTR_MANUAL_FREE = (0x1 << 2),
		MEMORY_ATTR_MAX
		} HIAI_MEMORY_ATTR;

Return Value

Address of the memory allocated by using **HIAI_DMalloc**. If the memory fails to be allocated, a null pointer is returned.

5.3.2 HIAI_DFree

Releases the memory that is allocated through the HiAI interface. This API is defined in **c_graph.h**.

Application scenario:

When memory is allocated using **5.3.1 HIAI_DMalloc** but the **SendData** interface is not called to send data, the memory must be released using the **HIAI_DFree** interface, unless the **SendData** interface has been called.

Syntax

HIAI_StatusT HIAI_DFree (void* dataBuffer)

Parameter Description

Parameter	Description	Value Range
dataBuffer	Pointer to the memory to be released	-

Return Value

For details about the returned error codes, see "Error Codes."

Error Codes

No.	Error Code	Description
1	HIAI_OK	The running is OK.
2	HIAI_GRAPH_NOT_EXIST	The graph does not exist.
3	HIAI_GRAPH_INVALID_VALUE	The graph has an invalid value.
4	HIAI_GRAPH_NOT_FIND_MEMORY	The memory cannot be found.

5.3.3 HIAI_DVPP_DMalloc

Allocates memory for DVPP on the device side. This API is defined in **c_graph.h**. After this API is called to allocate memory, **HIAI_DVPP_DFree** must be called to free the memory.

Syntax

void* HIAI_DVPP_DMalloc(const uint32_t dataSize)

Parameter Description

Parameter	Description	Value Range
dataSize	Size of a memory block	0 to (256 MB – 96 bytes), excluding 0. The recommended range is (256 KB – 96 bytes) to (256 MB – 96 bytes). The model manager on the device side occupies 96 bytes.

Return Value

Address of the memory allocated by using **HIAI_DVPP_DMalloc**. If the memory fails to be allocated, a null pointer is returned.

Remarks

HIAI_DVPP_DMalloc is a new API for allocating memory to DVPP. The memory allocated by using this API can meet the various memory requirements of DVPP.

If **HIAI_DMalloc(1000, buffer, 1000, HIAI_MEMORY_HUGE_PAGE)** is called to allocate memory for DVPP in the earlier version, use **HIAI_DVPP_DMalloc** instead. For details, see **5.3.1 HIAI_DMalloc**.

Usage Example:

```
// Allocate memory by using HIAI_DVPP_DMalloc.
uint32_t allocSize = 1000;
char* allocBuffer = (char*)HIAI_DVPP_DMalloc(allocSize);
if (allocBuffer == nullptr) {
   // If the allocation fails, handle the exception.
}
```

5.3.4 HIAI_DVPP_DFree

Releases the memory allocated by the **HIAI_DVPP_DMalloc** interface. This API is defined in **c_graph.h**.

Syntax

HIAI_StatusT HIAI_DVPP_DFree(void* dataBuffer)

Parameter Description

Parameter	Description	Value Range
dataBuffer	Address of the memory to be released	-

Return Value

For details about the returned error codes, see "Error Codes."

Error Codes

No.	Error Code	Description
1	HIAI_OK	The running is OK.
2	HIAI_GRAPH_NOT_EXIST	The graph does not exist.
3	HIAI_GRAPH_INVALID_VALUE	The graph has an invalid value.
4	HIAI_GRAPH_NOT_FIND_MEMORY	The memory cannot be found.

5.4 Memory Management (C++ Language)

Precautions for API Usage

When memory is allocated by using **HIAIMemory::HIAI_DMalloc**, pay attention to the following issues about memory management:

- When memory to be automatically freed is allocated for host-device or device-host data sending, if a smart pointer is used, Matrix automatically frees the memory, so the destructor specified by the smart pointer must be null. If the pointer is not a smart pointer, Matrix automatically frees the memory.
- When memory to be manually freed is allocated for host-device or device-host data sending, if a smart pointer is used, you need to set the destructor to **HIAIMemory::HIAI_DFree**. If the pointer is not a smart pointer, you need to call **HIAIMemory::HIAI_DFree** to free the memory after data sending is complete.
- When memory to be manually freed is allocated, the **SendData** API cannot be called repeatedly to send data in the memory.
- When memory to be manually freed is allocated for host-device or device-host data sending, do not reuse the data in the memory before the memory is freed. If the memory is used for host-host or device-device data sending, the data in the memory can be reused before the memory is freed.
- When memory to be manually freed is allocated, if the **SendData** API is called to asynchronously send data, data in the memory cannot be modified after being sent.

If the **HIAIMemory::HIAI_DVPP_DMalloc** API is called to allocate memory for device-host data sending, the **HIAIMemory::HIAI_DVPP_DFree** API must be called to manually free the memory, because the **HIAIMemory::HIAI_DVPP_DMalloc** API does not automatically free the memory. If a smart pointer is used to store the allocated memory address, the destructor must be set to **HIAIMemory::HIAI_DVPP_DFree**.

5.4.1 HIAIMemory::HIAI_DMalloc

Allocates memory blocks on the host side or device side to support efficient data transmission. This API is defined in **ai_memory.h**.

To allocate memory for DVPP on the device side, see **5.4.4 HIAIMemory::HIAI_DVPP_DMalloc**.

Application scenario:

With Matrix, this API is used to send large data from the host side to the device side. It must be used together with **5.2.3 Macro: HIAI_REGISTER_SERIALIZE_FUNC**.

For example, a 1080p or 4K image needs to be sent to the device side. To improve the data sending performance, you must register the structure serialization and deserialization functions through **HIAI_REGISTER_SERIALIZE_FUNC**. In addition, the large data block memory is allocated through **HIAI_DMalloc**. This greatly improves the data sending performance.

NOTE

- This API deals with sending of large data between the host side and device side. You are advised not to use it as a common **malloc** API.
- For the sake of performance, this API pre-allocates memory. The actual memory size is different from the allocated memory size. The running time of this API may also change.

Syntax

static HIAI_StatusT HIAIMemory::HIAI_DMalloc (const uint32_t dataSize, void*& dataBuffer, const uint32_t timeOut = MALLOC_DEFAULT_TIME_OUT, uint32_t flag = MEMORY_ATTR_AUTO_FREE)

Parameter Description

Parameter	Description	Value Range
dataSize	Size of a memory block	• If this API is called to allocate memory on the host side, the value range is 0–256 MB, excluding 0. The recommended range is 256 KB–256 MB. • If this API is called to allocate memory on the device side, the value range is 0 to (256 MB – 96 bytes), excluding 0. The recommended range is (256 KB – 96 bytes) to (256 MB – 96 bytes). The model manager on the device side occupies 96 bytes.
dataBuffer	Memory pointer	-
timeOut	Timeout when memory fails to be allocated. The default value is **MALLOC_DEFAULT_TIME_OUT** (indicating 500 ms).	-

Parameter	Description	Value Range

flag	• If **flag** is set to **MEMORY_ATTR_AUTO_FREE** (default value), it indicates that if the memory is allocated and data is sent to the peer end using **SendData**, the **HIAIMemory::HIAI_DFree** API does not need to be called. After the programming is complete, the memory is automatically freed. If the memory is allocated but the data is not sent to the peer end using **SendData**, the **HIAIMemory::HIAI_DFree** API needs to be called to free the memory.
	• If **flag** is set to **MEMORY_ATTR_MANUAL_FREE**, the **HIAIMemory::HIAI_DFree** API must be called to free the memory in any case.
	• **MEMORY_ATTR_NONE** and **MEMORY_ATTR_MAX** are not used currently.
		typedef enum {
		MEMORY_ATTR_NONE = 0,
		// The framework automatically frees the memory allocated by using **DMalloc**.
		MEMORY_ATTR_AUTO_FREE = (0x1 << 1),
		// **DFree** needs to be called manually to free the memory.
		MEMORY_ATTR_MANUAL_FREE = (0x1 << 2),
		MEMORY_ATTR_MAX
		} HIAI_MEMORY_ATTR;

Return Value

For details about the returned error codes, see "Error Codes."

Error Codes

No.	Error Code	Description
1	HIAI_OK	The running is OK.

No.	Error Code	Description
2	HIAI_GRAPH_NOT_EXIST	The graph does not exist.
3	HIAI_GRAPH_MEMORY_POOL_NOT_EXISTED	The memory pool does not exist.
4	HIAI_GRAPH_MALLOC_LARGER	The buffer fails to be allocated because the size is larger than 256 MB.
5	HIAI_MEMORY_POOL_UPDATE_FAILED	The memory pool fails to be updated.
6	HIAI_GRAPH_SENDMSG_FAILED	The HDC module fails to send a message.
7	HIAI_GRAPH_MEMORY_POOL_INITED	The memory pool has been initialized.
8	HIAI_GRAPH_NO_MEMORY	There is no memory.

5.4.2 HIAIMemory::HIAI_DFree

Releases the memory that is allocated through the HiAI interface. This API is defined in **ai_memory.h**.

Application scenario:

When memory is allocated using **5.4.1 HIAIMemory::HIAI_DMalloc** but the **SendData** interface is not called to send data, the memory must be released using the **HIAI_DFree** interface, unless the **SendData** interface has been called.

Syntax

static HIAI_StatusT HIAIMemory::HIAI_DFree (void* dataBuffer)

Parameter Description

Parameter	Description	Value Range
dataBuffer	Pointer to the memory to be released	-

Return Value

For details about the returned error codes, see "Error Codes."

Error Codes

No.	Error Code	Description
1	HIAI_OK	The running is OK.
2	HIAI_GRAPH_NOT_EXIST	The graph does not exist.
3	HIAI_GRAPH_INVALID_VALUE	The graph has an invalid value.
4	HIAI_GRAPH_NOT_FIND_MEMORY	The memory cannot be found.

5.4.3 HIAIMemory:: IsDMalloc

Checks whether the memory is allocated by **DMalloc**. This API is defined in **ai_memory.h**.

Syntax

static bool IsDMalloc(const void* dataBuffer, const uint32_t& dataSize)

Parameter Description

Parameter	Description	Value Range
dataBuffer	Pointer to the memory to be determined	-
dataSize	Size of the memory address, in bytes	-

Return Value

true: The memory is applied for by using **DMalloc**.

false: The memory is not applied for by using **DMalloc**.

5.4.4 HIAIMemory::HIAI_DVPP_DMalloc

Allocates memory for DVPP on the device side. This API is defined in **ai_memory.h**. After this API is called to allocate memory, **HIAIMemory::HIAI_DVPP_DFree** must be called to free the memory.

Syntax

HIAI_StatusT HIAI_DVPP_DMalloc(const uint32_t dataSize, void*& dataBuffer)

Parameter Description

Parameter	Description	Value Range
dataSize	Size of a memory block	0 to (256 MB – 96 bytes), excluding 0. The recommended range is (256 KB – 96 bytes) to (256 MB – 96 bytes). The model manager on the device side occupies 96 bytes.
dataBuffer	Allocated memory address. If the allocation fails, a null pointer is returned.	-

Return Value

For details about the returned error codes, see "Error Codes."

Error Codes

No.	Error Code	Description
1	HIAI_OK	The running is OK.
2	HIAI_GRAPH_NOT_EXIST	The graph does not exist.
3	HIAI_GRAPH_MEMORY_POOL_NOT_EXISTED	The memory pool does not exist.
4	HIAI_GRAPH_MALLOC_LARGER	The buffer fails to be allocated because the size is larger than 256 MB.
5	HIAI_MEMORY_POOL_UPDATE_FAILED	The memory pool fails to be updated.
6	HIAI_GRAPH_SENDMSG_FAILED	The HDC module fails to send a message.
7	HIAI_GRAPH_MEMORY_POOL_INITED	The memory pool has been initialized.
8	HIAI_GRAPH_NO_MEMORY	There is no memory.

Remarks

HIAIMemory::HIAI_DVPP_DMalloc is a new API for allocating memory to DVPP. The memory allocated by using this API can meet the various memory requirements of DVPP.

If **HIAIMemory::HIAI_DMalloc(1000, buffer, 1000, HIAI_MEMORY_HUGE_PAGE)** is called to allocate memory for DVPP in the earlier version, use **HIAIMemory::HIAI_DVPP_DMalloc** instead. For details, see **5.4.1 HIAIMemory::HIAI_DMalloc**.

Usage Example:

```
// Allocate memory by using HIAI_DVPP_DMalloc.
uint32_t allocSize = 1000;
char* allocBuffer  = nullptr;
HIAI_StatusT ret = hiai::HIAIMemory::HIAI_DVPP_DMalloc(allocSize, (void*&)allocBuffer);
if (ret != HIAI_OK || allocBuffer == nullptr) {
   // If the allocation fails, handle the exception.
}
```

5.4.5 HIAIMemory::HIAI_DVPP_DFree

Releases the memory allocated by the **HIAIMemory::HIAI_DVPP_DMalloc** interface. This API is defined in **ai_memory.h**.

Syntax

HIAI_StatusT HIAI_DVPP_DFree(void* dataBuffer)

Parameter Description

Parameter	Description	Value Range
dataBuffer	Address of the memory to be released	-

Return Value

For details about the returned error codes, see "Error Codes."

Error Codes

No.	Error Code	Description
1	HIAI_OK	The running is OK.
2	HIAI_GRAPH_NOT_EXIST	The graph does not exist.
3	HIAI_GRAPH_INVALID_VALUE	The graph has an invalid value.
4	HIAI_GRAPH_NOT_FIND_MEMORY	The memory cannot be found.

5.5 Logs (C++)

5.5.1 Error Code Registration

5.5.1.1 HIAI_DEF_ERROR_CODE

Registers error codes. This macro is defined in **status.h**.

This macro encapsulates the following functions:

```
static StatusFactory* StatusFactory::StatusFactory::GetInstance();
void RegisterErrorNo(const uint32_t err, const std::string& desc);
```

Syntax

HIAI_DEF_ERROR_CODE(moduleId, logLevel, codeName, codeDesc)

Parameter Description

Parameter	Description	Value Range
moduleId	module ID	-
logLevel	Error level	-
codeName	Error code name	-
codeDesc	Description of an error code, which is a character string	-

Return Value

None

5.5.1.2 API Usage Example

```
/*****************************************************************************
*                    Copyright (C) HiSilicon Co., Ltd.
*
*      Filename:  user_def_errorcode.h
*   Description:  User defined Error Code
*
*       Version:  1.0
*       Created:  2018-01-08 10:15:18
*        Author:
*
*      Revision:  initial draft;
*****************************************************************************/
#ifndef USE_DEF_ERROR_CODE_H_
#define USE_DEF_ERROR_CODE_H_

#include "hiaiengine/status.h"
#define USE_DEFINE_ERROR 0x6001   // ID of the custom module. The value 0x6001 is for reference only.
```

```
You can set the value as required. The value cannot be the same as MODID_GRAPH in the status.h file.
enum
{
HIAI_INVALID_INPUT_MSG_CODE = 0   // error code name. The value 0 is for reference only. You can set the value as required.
};
HIAI_DEF_ERROR_CODE(USE_DEFINE_ERROR, HIAI_ERROR, HIAI_INVALID_INPUT_MSG, "invalid input message pointer");
#endif
```

5.5.2 Log Printing

Use HIAI_ENGINE_LOG to invoke different HIAI_ENGINE_LOG_IMPL functions based on the input parameters. When the formatting function is invoked, the type and number of parameters in the format must be the same as the actual parameter type and number. The APIs associated with log printing are defined in **log.h**.

After **HIAI_ENGINE_LOG** is called, the system records logs in the **/var** directory on the host side or developer board. Device logs are recorded in log files whose names start with **device-*id***. Host logs are recorded in log files whose names start with **host-0**.

5.5.2.1 Log Printing Format 1

Syntax

**#define HIAI_ENGINE_LOG(...) **

HIAI_ENGINE_LOG_IMPL(__FUNCTION__, __FILE__, __LINE__, ##__VA_ARGS__)

void HIAI_ENGINE_LOG_IMPL(const char* funcPointer, const char* filePath, int lineNumber,

hiai::Graph* graph, const uint32_t errorCode, const char* format, ...);

Parameter Description

Parameter	Description	Value Range
graph	Pointer to the graph object	-
errorCode	Error code	-
format	Log description	-
...	Variable parameter in **format**, which is added based on the log content	-

Calling Example

```
auto graph = Graph::Getinstance(1);
HIAI_ENGINE_LOG (graph.get(), HIAI_OK, "RUNNING OK");
```


5.5.2.2 Log Printing Format 2

Syntax

**#define HIAI_ENGINE_LOG(...) **

HIAI_ENGINE_LOG_IMPL(__FUNCTION__, __FILE__, __LINE__, ##__VA_ARGS__)

void HIAI_ENGINE_LOG_IMPL(const char* funcPointer, const char* filePath, int lineNumber,

hiai::Engine* engine, const uint32_t errorCode, const char* format, ...);

Parameter Description

Parameter	Description	Value Range
engine	Pointer to the engine object	-
errorCode	Error code	-
format	Log description	-
...	Variable parameter in **format**, which is added based on the log content	-

Calling Example

```
// This API is called in HIAI_IMPL_ENGINE_PROCESS.
HIAI_ENGINE_LOG(this, HIAI_OK, "RUNNING OK");
```

5.5.2.3 Log Printing Format 3

Syntax

**#define HIAI_ENGINE_LOG(...) **

HIAI_ENGINE_LOG_IMPL(__FUNCTION__, __FILE__, __LINE__, ##__VA_ARGS__)

void HIAI_ENGINE_LOG_IMPL(const char* funcPointer, const char* filePath, int lineNumber,

const uint32_t errorCode, const char* format, ...);

Parameter Description

Parameter	Description	Value Range
errorCode	Error code	-
format	Log description	-

Parameter	Description	Value Range
...	Variable parameter in **format**, which is added based on the log content	-

Calling Example

```
HIAI_ENGINE_LOG(HIAI_OK, "RUNNING OK");
```

5.5.2.4 Log Printing Format 4

Syntax

**#define HIAI_ENGINE_LOG(...) **

HIAI_ENGINE_LOG_IMPL(__FUNCTION__, __FILE__, __LINE__, ##__VA_ARGS__)

void HIAI_ENGINE_LOG_IMPL(const char* funcPointer, const char* filePath, int lineNumber,

const char* format, ...);

Parameter Description

Parameter	Description	Value Range
format	Log description	-
...	Variable parameter in **format**, which is added based on the log content	-

Calling Example

```
HIAI_ENGINE_LOG ("RUNNING OK");
```

5.5.2.5 Log Printing Format 5

Syntax

**#define HIAI_ENGINE_LOG(...) **

HIAI_ENGINE_LOG_IMPL(__FUNCTION__, __FILE__, __LINE__, ##__VA_ARGS__)

void HIAI_ENGINE_LOG_IMPL(const char* funcPointer, const char* filePath, int lineNumber,

const int32_t moudleID, hiai::Graph* graph, const uint32_t errorCode, const char* format, ...);

Parameter Description

Parameter	Description	Value Range
moudleID	Enumeration ID of the module name	-
graph	Pointer to the graph object	-
errorCode	Error code	-
format	Log description	-
...	Variable parameter in **format**, which is added based on the log content	-

Calling Example

```
auto graph = Graph::Getinstance(1);
HIAI_ENGINE_LOG(MODID_OTHER, graph.get(), HIAI_OK, "RUNNING OK");
```

5.5.2.6 Log Printing Format 6

Syntax

**#define HIAI_ENGINE_LOG(...) **

HIAI_ENGINE_LOG_IMPL(__FUNCTION__, __FILE__, __LINE__, ##__VA_ARGS__)

void HIAI_ENGINE_LOG_IMPL(const char* funcPointer, const char* filePath, int lineNumber,

const int32_t moudleID,hiai::Engine* engine, const uint32_t errorCode, const char* format, ...);

Parameter Description

Parameter	Description	Value Range
moudleID	Enumeration ID of the module name	-
engine	Pointer to the engine object	-
errorCode	Error code	-
format	Log description	-
...	Variable parameter in **format**, which is added based on the log content	-

Calling Example

```
// This API is called in HIAI_IMPL_ENGINE_PROCESS.
HIAI_ENGINE_LOG(MODID_OTHER, this, HIAI_OK, "RUNNING OK");
```

5.5.2.7 Log Printing Format 7

Syntax

**#define HIAI_ENGINE_LOG(...) **

HIAI_ENGINE_LOG_IMPL(__FUNCTION__, __FILE__, __LINE__, ##__VA_ARGS__)

void HIAI_ENGINE_LOG_IMPL(const char* funcPointer, const char* filePath, int lineNumber,

const int32_t moudleID, const char* format, ...);

Parameter Description

Parameter	Description	Value Range
moudleID	Enumeration ID of the module name	-
format	Log description	-
...	Variable parameter in **format**, which is added based on the log content	-

Calling Example

```
HIAI_ENGINE_LOG(MODID_OTHER, "RUNNING OK");
```

5.5.2.8 Log Printing Format 8

Syntax

**#define HIAI_ENGINE_LOG(...) **

HIAI_ENGINE_LOG_IMPL(__FUNCTION__, __FILE__, __LINE__, ##__VA_ARGS__)

void HIAI_ENGINE_LOG_IMPL(const char* funcPointer, const char* filePath, int lineNumber,

const int32_t moudleID,const uint32_t errorCode, const char* format, ...);

Parameter Description

Parameter	Description	Value Range
moudleID	Enumeration ID of the module name	-
errorCode	Error code	-
format	Log description	-
...	Variable parameter in **format**, which is added based on the log content	-

Calling Example

```
HIAI_ENGINE_LOG(MODID_OTHER, HIAI_OK, "RUNNING OK");
```

5.6 MultiTypeQueue API for Queue Management (C++ Language)

Supports storing multiple types of message queues. The queue management APIs are defined in **multitype_queue.h**.

5.6.1 MultiTypeQueue Constructor

MultiTypeQueue constructor

Syntax

MultiTypeQueue(uint32_t queNum, uint32_t maxQueLen = 1024, uint32_t durationMs = 0);

Parameter Description

Parameter	Description	Value Range
queNum	Number of queues	-
maxQueLen	Maximum length of each queue	-

Parameter	Description	Value Range
durationMs	Maximum interval between the time when a queue member enters a queue and the time when the member leaves the queue. If a member does not enter the queue (Pop) within **duration_ms** after entering the queue (Push), the member is automatically deleted.	-

Return Value

None

5.6.2 PushData

Inserts data into a specified queue.

Syntax

bool MultiTypeQueue::PushData(uint32_t qIndex, const std::shared_ptr<void>& dataPtr);

Parameter Description

Parameter	Description	Value Range
qIndex	Queue ID	0 to queNum – 1
dataPtr	Pointer to the queue function	-

Return Value

If the operation is successful, **true** is returned. Otherwise, **false** is returned (for example, when the queue is full).

5.6.3 FrontData

Reads the header data of a specified queue (not deleting it from the queue).

Syntax

bool MultiTypeQueue::FrontData(uint32_t qIndex, std::shared_ptr<void>& dataPtr);

Parameter Description

Parameter	Description	Value Range
qIndex	Queue ID	0 to queNum – 1
dataPtr	Pointer to the queue function	-

Return Value

If the operation is successful, **true** is returned. Otherwise, **false** is returned (for example, when the queue is empty).

5.6.4 PopData

Reads the header data of a specified queue (deleting it from the queue).

Syntax

bool MultiTypeQueue::PopData(uint32_t qIndex, std::shared_ptr<void>& dataPtr);

Parameter Description

Parameter	Description	Value Range
qIndex	Queue ID	0 to que_num – 1
dataPtr	Pointer to the queue function	-

Return Value

If the operation is successful, **true** is returned. Otherwise, **false** is returned (for example, when the queue is empty).

5.6.5 PopAllData

Reads the header data of all queues. The read operation is successful only when all queue headers have data, and the header data of the queues are deleted. Otherwise, a failure message is returned, but no data is deleted.

Syntax

template<typename T1>

bool MultiTypeQueue::PopAllData(std::shared_ptr<T1>& arg1);

template<typename T1, typename T2>

bool MultiTypeQueue::PopAllData(std::shared_ptr<T1>& arg1, std::shared_ptr<T2>& arg2);

.....

template<typename T1, typename T2, typename T3, typename T4, typename T5, typename T6, typename T7, typename T8, typename T9, typename T10, typename T11, typename T12, typename T13, typename T14, typename T15, typename T16>

bool MultiTypeQueue::PopAllData(std::shared_ptr<T1>& arg1,

std::shared_ptr<T2>& arg2, std::shared_ptr<T3>& arg3,

std::shared_ptr<T4>& arg4, std::shared_ptr<T5>& arg5,

std::shared_ptr<T6>& arg6, std::shared_ptr<T7>& arg7,

std::shared_ptr<T8>& arg8, std::shared_ptr<T9>& arg9,

std::shared_ptr<T16>& arg16)

Parameter Description

Parameter	Description	Value Range
arg1	Pointer to the queue function	-
...	Pointer to the queue function	-
arg16	Pointer to the queue function	-

Return Value

The read operation is successful and **true** is returned only when all queue headers have data. Otherwise, **false** is returned.

5.6.6 API Usage Example

For details, see **3.7 Calling Example**.

5.7 Event Registration API (C++ Language)

5.7.1 Graph::RegisterEventHandle

Subscribes to interested events. Currently, the host-device disconnection event is supported. When disconnection occurs, the user can receive this event and handle the disconnection logic (for example, waiting for stopping the main program) in the registered callback function. This API is defined in **graph.h**.

Syntax

HIAI_StatusT Graph::RegisterEventHandle(const HIAIEvent& event, const std::function<HIAI_StatusT(void)>&callBack)

Parameter Description

Parameter	Description	Value Range
event	Subscribed event	typedef enum { HIAI_DEVICE_DISCONNECT_EVENT // Disconnection message }HIAIEvent;
callBack	Event callback function	Callback function for user subscription

Return Value

If the operation is successful, **HIAI_OK** is returned. Otherwise, a failure message is returned.

5.8 Others (C++ Language)

5.8.1 DataRecvInterface::RecvData

Defines a subclass of class **DataRecvInterface** to initialize the object of the subclass. It works as the callback function for receiving data and transfers the API in **2.4.1 Graph::SetDataRecvFunctor** or **3.5 Engine::SetDataRecvFunctor**. In addition, the **RecvData** API needs to be overloaded and implemented in the subclass. This API is defined in **data_recv_interface.h**.

Syntax

virtual HIAI_StatusT DataRecvInterface::RecvData(const std::shared_ptr<void>& message)

Parameter Description

Parameter	Description	Value Range
message	Callback message	-

Return Value

For details about the returned error codes, see "Error Codes."

Error Codes

No.	Error Code	Description
1	HIAI_OK	The running is OK.

5.8.2 Graph::SetPublicKeyForSignatureEncryption

Set a public key which is used to verify the digital signature of the *.so file to be loaded by the Matrix on the device side. This API must be called before the creation of a graph. This API is defined in **graph.h**.

The **SHA256withRSA** algorithm is used for digital signature. This component provides only the digital signature verification function. You need to sign the required SO file based on this algorithm. The signature file must be stored in the same directory as the SO file on the host side. The signature file name is "SO file name+.signature". For example, if the SO file name is **libhosttodevice.so**, the signature file name is **libhosttodevice.so.signature**.

CAUTION

If **Graph::SetPublicKeyForSignatureEncryption** is not called or its return value is not **HIAI_OK**, the public key cannot be set in the Matrix, and the signature verification mechanism will not be enabled on the device side. As a result, the *.so file that may be tampered cannot be identified by using the digital signature.

Syntax

static HIAI_StatusT Graph::SetPublicKeyForSignatureEncryption(const std::string& publicKey)

Parameter Description

Parameter	Description	Value Range
publicKey	Public key required for digital signature verification	-

Return Value

For details about the returned error codes, see "Error Codes."

Error Codes

No.	Error Code	Description
1	HIAI_OK	The running is OK.

```
std::string publicKey =
"-----BEGIN RSA PUBLIC KEY-----\n"
"MIIBCAKCAQEAt+kdmu8CdViw2xHlh/JWXOl/0AitDgYGd+9RadULGZmj0tt/UQLv\n"
"EYPZaXC8E0a9e97kqfg/ZHinu04XG5RhXv2J0kwkhuBAeCCllefvrQy5OBRqdLYq\n"
"EZe8wUO0adSMDsLbPL52b+NdO9/zX+MUzvsBzitWJbiH96s4xCiEERX87/uQfr6F\n"
"lHLtrNtooeF2VmxOm3n0yzh4kcSouLgb/O0+v0l+fnR+CTNG95IvzDNOYLWD6ZkL\n"
"c7JyIYFZA0CNx9SlPhqbrfjOV5XSG3g3CW0TopUDfHyhAgZt5vACMpeDDx+89tg2\n"
"RfT4M9DH/qKzkLlOURvsMShRMD6/PwzsPwIBAw==\n"
"-----END RSA PUBLIC KEY-----";
auto ret = Graph::SetPublicKeyForSignatureEncryption(publicKey);
```


6 Python APIs

You are not advised to use the following APIs. If you need to use them, contact Huawei technical support.

Table 6-1 Basic process connection APIs

Class	API
Class Graph	Graph.__init__
	Graph.__del__
	Graph.get_id
	Graph.set_device_id
	Graph.get_graph
	Graph.destroy
	Graph.destroy_graph
	Graph.as_default
	Graph.create_graph
Class Engine	Engine.__init__
	Engine.update_config
	Engine.as_default
Class EngineConfig	EngineConfig.__init__
	EngineConfig.engine_id
	EngineConfig.engine_name
	EngineConfig.so_names
	EngineConfig.side
	EngineConfig.thread_num

Class	API
	EngineConfig.thread_priority
	EngineConfig.queue_size
	EngineConfig.ai_config
	EngineConfig.internel_so_names
	EngineConfig.wait_inputdata_max_time
	EngineConfig.hold_model_file_flag
Class GraphConfig	GraphConfig.__init__
	GraphConfig.graph_id
	GraphConfig.priority
	GraphConfig.device_id
Function APIs	get_default_graph
	get_default_engine
	crop
	resize
	inference

Table 6-2 Basic data types

Class	API
Class NNTensor	NNTensor. __init__
	NNTensor.height
	NNTensor.width
	NNTensor.channel
	NNTensor.batch_size
Class NNTensorList	NNTensorList.__init__
	NNTensorList.__getitem__
	NNTensorList.get_tensor_num
	NNTensorList.next
	NNTensorList.__iter__

Table 6-3 Data pre-processing

Class	API
Class Engine	Engine.crop
	Engine.resize
	Engine.padding
Class CropConfig	CropConfig.__init__
Class ResizeConfig	ResizeConfig.__init__

Table 6-4 Offline model inference

Class	API
Class Engine	Engine.inference
Class AIModelDescription	AIModelDescription.__init__
	AIModelDescription.name
	AIModelDescription.model_type
	AIModelDescription.version
	AIModelDescription.size
	AIModelDescription.path
Class AIConfig	AIConfig.__init__
Class AIConfigItem	AIConfigItem.__init__

Table 6-5 Data TX/RX

Class	API
Class Graph	Graph.proc

7 Appendix

7.1 APIs to Be Deleted in Later Versions

7.2 Matrix Data Types

7.3 Data Structures Registered in the Matrix

7.4 Example

7.5 Change History

7.1 APIs to Be Deleted in Later Versions

The APIs to be deleted in later versions are not called by users. The subsequent compatibility cannot be ensured after the APIs are called.

7.1.1 Low-Power APIs (C++ Language)

7.1.1.1 PowerState::SubscribePowerStateEvent

Registers the low-power callback function. This API is defined in **power_state.h**.

Syntax

HIAI_StatusT SubscribePowerStateEvent(PowerStateNotifyCallbackT notifyCallback)

Parameter Description

Parameter	Description	Value Range
notifyCallback	Callback function that the user expects to register	-

Return Value

For details about the returned error codes, see "Error Codes."

Error Codes

No.	Error Code	Description
1	HIAI_OK	The running is OK.
2	HIAI_POWER_STATE_CALLBACK_ERROR	A null pointer is returned when the app callback function is called.

7.1.1.2 PowerState::UnsubscribePowerStateEvent

Deregisters the previously registered low-power callback function. This API is defined in **power_state.h**.

Syntax

HIAI_StatusT UnsubscribePowerStateEvent(PowerStateNotifyCallbackT notifyCallback)

Parameter Description

Parameter	Description	Value Range
notifyCallback	Callback function that the user expects to deregister	-

Return Value

For details about the returned error codes, see "Error Codes."

Error Codes

No.	Error Code	Description
1	HIAI_OK	The running is OK.
2	HIAI_POWER_STATE_CALLBACK_ERROR	A null pointer is returned when the app callback function is called.

7.1.2 Data Types

7.1.2.1 Enumeration: PowerState::DEVICE_POWER_STATE

```
enum DEVICE_POWER_STATE {
    DEVICE_POWER_STATE_S0 = 0,       // The system enters the S0 state, the device is available, and the app is woken up.
    DEVICE_POWER_STATE_SUSPEND,      // The system is about to enter the S3/S4 state, and the app releases resources such as channels and memory.
    DEVICE_POWER_STATE_D0,           // The NPU device enters the D0 state, the device is available, and the app is woken up. (No processing is required for the Matrix services.)
    DEVICE_POWER_STATE_D3,           // The NPU device is about to enter the D3 state, and the app stops communicating with the device. (No processing is required for the Matrix services.)
    DEVICE_POWER_STATE_ENABLE,       // The NPU device driver is installed or enabled, and the NPU is available.
    DEVICE_POWER_STATE_DISABLE,      // The NPU device driver is uninstalled or disabled, and the NPU is unavailable.
     DEVICE_POWER_STATE_MAX          // Indicates the maximum number of states. It does not indicate the actual state.
};
```

7.1.2.2 PowerState::PowerStateNotifyCallbackT

```
typedef HIAI_StatusT (*PowerStateNotifyCallbackT)(uint32_t, DEVICE_POWER_STATE);
```

7.2 Matrix Data Types

7.2.1 Protobuf Data Types

For details about how to use Protobuf, see **https://developers.google.com/protocol-buffers/docs/reference/proto3-spec**.

The Protobuf data types used by Matrix are as follows:

ai_types.proto

```
syntax = "proto3";IAITensor

package hiai;

// Tensor parameter description
message AITensorParaDescription
{
    string name = 1; // Parameter name
    string type = 2; // Parameter type
    string value = 3;  // Parameter value
    string desc = 4; // Parameter description
    repeated AITensorParaDescription sub_paras = 5; // Sub-parameter list
};

// Tensor description
message AITensorDescription
{
    string name = 1; // Tensor name
    string type = 2; // Tensor type
    repeated string compatible_type = 3; // Specifies all types of parent classes that are compatible.
   repeated AITensorParaDescription paras = 4; // Parameter list
};

// Tensor description list
message AITensorDescriptionList
{
    repeated AITensorDescription tensor_descs = 1; // Tensor list
}

// Common configuration items
// If sub_items exists, the current node is the parent node and the value is invalid.
```



```
message AIConfigItem
{
    string name = 1; // Configuration item name
    string value = 2;  // Configuration item value. When this parameter is set to the path of a model file, it contains the file name, supporting only digits, letters, underscores (_), and dots (.). You can set this parameter to the path of a single model file, for example, ./test_data/model/resnet18.om. You can also compress the model files into a tar package and set the parameter to the path of the tar package, for example, ./test_data/model/resnet18.tar. If there are multiple AIConfigItem, the files with the same name but in different formats cannot be in the same directory when you set the path of the tar package, for example, ./test_data/model/test and ./test_data/model/test.tar. The APIs for creating graphs include Graph::CreateGraph or HIAI_CreateGraph.
    repeated AIConfigItem sub_items = 3; // Configuration subitem
};

// Configuration parameter for running an NN node or API
message AIConfig
{
    repeated AIConfigItem items = 1; // Configuration item list
};

// Model description
message AIModelDescription
{
     string name = 1; // Model name
    int32 type = 2; // Model type. Currently, only the DAVINCI_OFFLINE_MODEL type is supported. The value is 0.
    string version = 3; // Model version
    int32 size = 4; // Model size
    string path = 5; // Model path
    string key = 8; // Model key
    bytes data = 100; // Model data
    repeated AITensorDescription inputs = 6; // Description of input tensors
    repeated AITensorDescription outputs = 7; // Description of output tensors
};

// NNNode description
message AINNNodeDescription
{
    string name = 1; // NN node name
    string desc = 2; // NN node description
   bool isPreAllocateOutputMem = 3; // Whether to pre-allocate the output memory
   AIConfig config = 4; // Configuration parameter
   repeated AIModelDescription model_list = 5;  // List of models required by an NN node
    repeated AITensorDescription inputs = 6; // Description of input tensors
    repeated AITensorDescription outputs = 7; // Description of output tensors
    bool need_verify = 8; // Whether to verify the tensor match during connection
    repeated string ignored_check_aitensor = 9; // Specifies the list of tensors for which the mechanism for verifying whether they match the tensors of the inputs is ignored during connection.
};

// NNNode description list
message AINNNodeDescriptionList
{
    repeated AINNNodeDescription nnnode_descs = 1; // NN node list
}

// API description
message AIAPIDescription
{
    string name = 1; // API name
    string desc = 2; // API description
    bool isPreAllocateOutputMem = 3; // Whether to pre-allocate the output memory
    AIConfig config = 4; // Configuration parameter
    repeated AITensorDescription inputs = 5; // Description of input tensors
    repeated AITensorDescription outputs = 6; // Description of output tensors
    bool need_verify = 7; // Whether to verify the tensor match during connection
    repeated string ignored_check_aitensor = 8; // Specifies the list of tensors for which the mechanism for verifying whether they match the tensors of the inputs is ignored during connection.
};
```



```
// API description list
message AIAPIDescriptionList
{
    repeated AIAPIDescription api_descs = 1; // API list
}

// AI operation description
message AIOPDescription
{
    enum OP_Type
    {
        AI_API = 0;
        AI_NNNODE = 1;
    }

    OP_Type type = 1;
    AINNNodeDescription nnnode_desc = 2;
    AIAPIDescription api_desc = 3;
};

// AI operation description list
message AIOPDescriptionList
{
   repeated AIOPDescription op_descs = 1; // AI operation list
}

// IDE passes parameters to Matrix APIs.

message NodeDesc
{
   string name=1; // IAINNNode or ALG_API name
   AIConfig config=2; // Initialization parameter required by IAINNNode or ALG_API
   repeated AIModelDescription mode_desc=3; // Initialization parameter required by IAINNNode
}

message EngineDesc
{
    enum RunSide
    {
        DEVICE=0;
        HOST=1;
    }
    enum EngineType
    {
        NORMAL=0;
        SOURCE=1;
        DEST=2;
    }
    uint32 id =1; // Engine ID (node)
    EngineType type=2; //
    string name =3; // Engine node name
    repeated string so_name=4; // The list of all DLL .so file names that need to be copied is loaded first
according to the configuration sequence. If the loading fails, the system tries to load the list of files
according to another loading sequence.
    RunSide side=5; // Run on the host side or device side.
    int32 priority=6; // Node priority
    uint32 instance_cnt=7; // Number of instances (equivalent to the number of threads)
    repeated uint32 next_node_id=8; // Next node list
    bool user_input_cb=9; // IDE can be ignored.
    bool user_output_cb=10; // IDE can be ignored.
    repeated NodeDesc oper=11; // Matrix node list
}

message GraphInitDesc
{
    int32 priority=1; // Priority of the entire process for a graph
   //Runside side = 2; // Assume that the graph is deployed on the host and cannot be configured.
}
```



```
message GeneralFileBuffer
{
    bytes raw_data = 1;
    string file_name = 2;
}
```

graph_config.proto

```
syntax = "proto3";
import "ai_types.proto";
package hiai;

message DumpDef
{
    bytes model_name = 1; // Model name
    bytes is_dump_all = 2; // Whether to enable the dump function. This parameter is used to dump information about operators at all layers in the model.
    repeated bytes layer = 3;    // Determines the layers on which the operators need to be dumped in the model.
}
message OAMConfigDef
{
    repeated DumpDef dump_list = 1;
    bytes dump_path = 2;    // Path for storing dump files
}
message EngineConfig
{
    enum RunSide
    {
        DEVICE=0; // Run on the device side.
        HOST=1;     // Run on the host side.
    }

    uint32 id =1; // Engine ID (node)
    string engine_name =2; // Engine node name
    repeated string so_name=3; // The list of all DLL .so file names that need to be copied is loaded first according to the configuration sequence. If the loading fails, the system tries to load the list of files according to another loading sequence.
    RunSide side=4; // Run on the host side or device side.
    uint32 thread_num = 5;// Number of threads. If multi-channel decoding is used, you are advised to set this parameter to 1. If the value of "thread_num" is greater than 1, the decoding sequence between threads cannot be ensured.
    uint32 thread_priority = 6;// Thread priority
    uint32 queue_size = 7;// Queue size
    AIConfig ai_config = 8; // Aiconfig configuration file
    repeated AIModelDescription ai_model = 9; //AIModelDescription
    repeated string internal_so_name=10; // List of all DLL .so files that do not need to be copied
    uint32 wait_inputdata_max_time = 12; // Maximum timeout for waiting to receive the next data (in ms)
   uint32 holdModelFileFlag = 13; // Whether to retain the model file of this engine. 0: Not reserved; Other values: Reserved.
    OAMConfigDef oam_config = 14; // OAM configuration
    bool is_repeat_timeout_flag = 15; // Whether to perform timeout processing repeatedly when no data is received by an engine. 0: no; 1: yes. For example, if there is a large amount of multi-channel data to be analyzed, you are advised to set this parameter to 1. This parameter is used together with the wait_inputdata_max_time parameter. If "is_repeat_timeout_flag" is set to 0, the wait_inputdata_max_time parameter is used to set the timeout duration for a single time. If "is_repeat_timeout_flag" is set to 1, the wait_inputdata_max_time parameter is used to set the timeout duration of the entire period.
}
message ConnectConfig
{
    uint32 src_engine_id=1; // Engine ID of the transmit end
    uint32 src_port_id = 2; // Port ID of the transmit end
    uint32 target_graph_id=3; // Graph ID of the receive end
    uint32 target_engine_id=4; // Port ID of the receive end
    uint32 target_port_id=5; // Port ID of the receive end
     bool receive_memory_without_dvpp=6; // The default value is 0, indicating that the RX memory of the target engines running on the device side must meet the 4 GB address space limit. If this parameter is set to 1, the 4 GB address space limit does not need to be met. You can set the receive_memory_without_dvpp parameter (under the "connects" property) to 1 in the graph configuration file for all target engines
```

```
running on the device side. You can also set the receive_memory_without_dvpp parameter under the
"connects" property to 1 because the target engines running on the host side are not affected even if the
parameter is set. In this way, the Matrix RX memory pool is not limited by the 4 GB address space, which
improves the memory utilization. In the current DVPP, the input memory for the VPC, JPEGE, JPEGD, and
PNGD must meet the 4 GB address space requirements, while that for the VDEC and VENC can either or
not meet the 4 GB address space requirements.
}
message GraphConfig
{
    uint32 graph_id = 1;  //Graph ID
    int32 priority = 2; // Priority
    string device_id = 3;// Device ID, for example, "0"
   repeated EngineConfig engines = 3; // All engines. You are advised to configure multiple engines for multi-
channel decoding. One engine corresponds to one thread. If one engine corresponds to multiple threads,
the decoding sequence cannot be ensured.
    repeated ConnectConfig connects = 4; // Connection mode
}
message GraphConfigList
{
    repeated GraphConfig graphs = 1; // Configuration parameter for a single graph
}

message ProfileConfig
{
    string matrix_profiling = 1; //  Whether to enable profiling for Matrix. The value "on" indicates that the
function is enabled.
    string ome_profiling = 2; // Whether to enable the performance statistics function of OME. The value
"on" indicates that the function is enabled.
   string cce_profiling =3;  // Whether to enable the performance statistics function of CCE. The value "on"
indicates that the function is enabled.
    string runtime_profiling = 4;  // Whether to enable the performance statistics function of Runtime. The
value "on" indicates that the function is enabled.
    string PROFILER_TARGET = 5; //  Parameter for transparent transmission profiling
    string PROFILER_JOBCTX = 6;
    string src_path = 7;
    string dest_path = 8;
    string runtime_config = 9;
    string RTS_PATH = 10;
    string profiler_jobctx_path = 11;
    string profiler_target_path = 12;
}

message GraphUpdateConfig
{
    GraphConfig updataGraphConfig = 1;
    repeated uint32 del_engine_ids = 2;
    repeated ConnectConfig del_connects = 3;
}
```

7.2.2 Matrix Custom Data Types

- **HIAI_StatusT**: Specifies the data type of the result code as **uint32_t**.
- **EnginePortID**: Specifies the port ID in the Matrix.

```
struct EnginePortID {
    uint32_t graph_id;
    uint32_t engine_id;
    uint32_t port_id;
};
```

7.3 Data Structures Registered in the Matrix

The data structures transferred in the Matrix need to be registered first. The following data structures have been registered and can be directly used.

```
namespace hiai {
   // message name  <<<<<<==================>>>>>>> message type
   // name:"BatchInfo"                        type:BatchInfo
   // name:"FrameInfo"                        type:FrameInfo
   // name:"IMAGEFORMAT"                          type:IMAGEFORMAT
   // name:"Angle"                           type:Angle
   // name:"Point2D"                          type:Point2D
   // name:"Point3D"                          type:Point3D
   // name:"ROICube"                          type:ROICube
   // name:"RLECode"                          type:RLECode
   // name:"IDTreeNode"                        type:IDTreeNode
   // name:"IDTreePara"                       type:IDTreePara
   // name:"BatchIDTreePara"                     type:BatchIDTreePara
   // name:"RetrievalResult"                  type:RetrievalResult
   // name:"RetrievalResultTopN"                 type:RetrievalResultTopN
   // name:"RetrievalResultPara"                 type:RetrievalResultPara
   // name:"RawDataBuffer"                       type:RawDataBuffer
   // name:"BatchRawDataBuffer"                     type:BatchRawDataBuffer
   // name:"string"                          type:string


   // name:"vector_uint8_t"                    type:vector<uint8_t>
   // name:"vector_float"                     type:vector<float>


   // name:"ImageData_uint8_t"                   type:ImageData<uint8_t>
   // name:"ImageData_float"                    type:ImageData<float>


   // name:"ImagePara_uint8_t"                   type:ImagePara<uint8_t>
   // name:"ImagePara_float"                    type:ImagePara<float>


   // name:"BatchImagePara_uint8_t"                type:BatchImagePara<uint8_t>
   // name:"BatchImagePara_float"                 type:BatchImagePara<float>


   // name:"Line_uint8_t"                     type:Line<uint8_t>
   // name:"Line_float"                      type:Line<float>


   // name:"Rectangle_uint8_t"                   type:Rectangle<uint8_t>
   // name:"Rectangle_float"                    type:Rectangle<float>


   // name:"Polygon_uint8_t"                    type:Polygon<uint8_t>
   // name:"Polygon_float"                     type:Polygon<float>


   // name:"MaskMatrix_uint8_t"                   type:MaskMatrix<uint8_t>
   // name:"MaskMatrix_float"                    type:MaskMatrix<float>


   // name:"ObjectLocation_uint8_t_uint8_t"        type:ObjectLocation<uint8_t, uint8_t>
   // name:"ObjectLocation_float_float"           type:ObjectLocation<float, uint8_t>


   // name:"DetectedObjectPara_uint8_t_uint8_t"    type:DetectedObjectPara<uint8_t, uint8_t>
   // name:"DetectedObjectPara_float_float"        type:DetectedObjectPara<float, float>


   // name:"BatchDetectedObjectPara_uint8_t_uint8_t"
type:BatchDetectedObjectPara<uint8_t,uint8_t>
   // name:"BatchDetectedObjectPara_float_float"                 type:BatchDetectedObjectPara<float,float>
   // name:"BatchDetectedObjectPara_Rectangle_Point2D_int32_t"
type:BatchDetectedObjectPara<Rectangle<Point2D>, int32_t>
   // name:"BatchDetectedObjectPara_Rectangle_Point2D_float"
type:BatchDetectedObjectPara<Rectangle<Point2D>, float>
```



```
// name:"RegionImage_uint8_t"                type:RegionImage<uint8_t>
// name:"RegionImage_float"                  type:RegionImage<float>

// name:"RegionImagePara_uint8_t"            type:RegionImagePara<uint8_t>
// name:"RegionImagePara_float"              type:RegionImagePara<float>

// name:"BatchRegionImagePara_uint8_t"       type:BatchRegionImagePara<uint8_t>
// name:"BatchRegionImagePara_float"         type:BatchRegionImagePara<float>

// name:"Attribute_uint8_t"                  type:Attribute<uint8_t>
// name:"Attribute_float"                    type:Attribute<float>

// name:"AttributeTopN_uint8_t"              type:AttributeTopN<uint8_t>
// name:"AttributeTopN_float"                type:AttributeTopN<float>

// name:"AttributeVec_uint8_t"               type:AttributeVec<uint8_t>
// name:"AttributeVec_float"                 type:AttributeVec<float>

// name:"AttributeResultPara_uint8_t"        type: AttributeVec<uint8_t>
// name:"AttributeResultPara_float"          type:AttributeVec<float>

// name:"BatchAttributeResultPara_uint8_t"   type:AttributeVec<uint8_t>
// name:"BatchAttributeResultPara_float"     type:AttributeVec<float>

// name:"Feature_uint8_t"                    type:AttributeVec<uint8_t>
// name:"Feature_float"                      type:AttributeVec<float>

// name:"FeatureList_uint8_t"                type:FeatureList<uint8_t>
// name:"FeatureList_float"                  type:FeatureList<float>

// name:"FeatureVec_uint8_t"                 type:FeatureVec<uint8_t>
// name:"FeatureVec_float"                   type:FeatureVec<float>

// name:"FeatureResultPara_uint8_t"          type:FeatureResultPara<uint8_t>
// name:"FeatureResultPara_float"            type:FeatureResultPara<float>

// name:"BatchFeatureResultPara_uint8_t"     type:BatchFeatureResultPara<uint8_t>
// name:"BatchFeatureResultPara_float"       type:BatchFeatureResultPara<float>

// name:"FeatureRecord_uint8_t"              type:FeatureRecord<uint8_t>
// name:"FeatureRecord_float"                type:FeatureRecord<float>

// name:"RetrievalSet_uint8_t_uint8_t"       type:RetrievalSet<uint8_t, uint8_t>
// name:"RetrievalSet_float_float"           type:RetrievalSet<float, float>

// name:"EvaluationResult_uint8_t"           type:EvaluationResult<uint8_t>
// name:"EvaluationResult_float"             type:EvaluationResult<float>

// name:"EvaluationResultVec_uint8_t"        type:EvaluationResultVec<uint8_t>
// name:"EvaluationResultVec_float"          type:EvaluationResultVec<float>
```



```
// name:"EvaluationResultPara_uint8_t"          type:EvaluationResultPara<uint8_t>
// name:"EvaluationResultPara_float"            type:EvaluationResultPara<float>


// name:"BatchEvaluationResultPara_uint8_t"     type:BatchEvaluationResultPara<uint8_t>
// name:"BatchEvaluationResultPara_float"       type:BatchEvaluationResultPara<float>


// name:"Classification_uint8_t"               type:Classification<uint8_t>
// name:"Classification_float"                 type:Classification<float>


// name:"ClassificationTopN_uint8_t"           type:ClassificationTopN<uint8_t>
// name:"ClassificationTopN_float"             type:ClassificationTopN<float>


// name:"ClassificationVec_uint8_t"            type:ClassificationVec<uint8_t>
// name:"ClassificationVec_float"              type:ClassificationVec<float>


// name:"ClassificationResultPara_uint8_t"     type:ClassificationResultPara<uint8_t>
// name:"ClassificationResultPara_float"       type:ClassificationResultPara<float>


// name:"BatchClassificationResultPara_uint8_t" type:BatchClassificationResultPara<uint8_t>
// name:"BatchClassificationResultPara_float"   type:BatchClassificationResultPara<float>


////////////////////////////////////////////////////
/////////////// batch information //////////////////
////////////////////////////////////////////////////
struct BatchInfo {
    bool is_first = false;              //  Whether it is the first batch
    bool is_last = false;               // Whether it is the last batch
    uint32_t batch_size = 0;            // Actual size of the current batch
    uint32_t max_batch_size = 0;        // Preset size of a batch (maximum capacity)
    uint32_t batch_ID = 0;              // Current batch ID
    uint32_t channel_ID = 0;            // ID of the channel that processes the current batch
    uint32_t processor_stream_ID = 0;   // ID of the processor computing stream


    std::vector<uint32_t> frame_ID;     // ID of the image frame in the batch
    std::vector<uint32_t> source_ID;    // ID of the image source in the batch


    std::vector<uint64_t> timestamp;    //  Image timestamp in the batch
};


template<class Archive>
void serialize(Archive& ar, BatchInfo& data) {
    ar(data.is_first, data.is_last, data.batch_size,
       data.max_batch_size, data.batch_ID, data.channel_ID,
       data.processor_stream_ID, data.frame_ID, data.source_ID,
       data.timestamp);
}


////////////////////////////////////////////////////
/////////////// frame information //////////////////
////////////////////////////////////////////////////
struct FrameInfo {
    bool is_first = false;              //  Whether it is the first frame
    bool is_last = false;               // Whether it is the last frame
    uint32_t channel_ID = 0;            // ID of the channel that processes the current frame
    uint32_t processor_stream_ID = 0;   // ID of the processor computing stream
    uint32_t frame_ID = 0;              // Image frame ID
    uint32_t source_ID = 0;             // Image source ID
    uint64_t timestamp = 0;             // Image timestamp
```

```
};


template<class Archive>
void serialize(Archive& ar, FrameInfo& data) {
    ar(data.is_first, data.is_last, data.channel_ID,
        data.processor_stream_ID, data.frame_ID, data.source_ID,
        data.timestamp);
}


//////////////////////////////////////////////////////
//////////////////  1. image tensor ///////////////////
//////////////////////////////////////////////////////


// Image format
enum IMAGEFORMAT {
    RGB565,          //  Red 15:11, Green 10:5, Blue 4:0
    BGR565,          //  Blue 15:11, Green 10:5, Red 4:0
    RGB888,          //  Red 24:16, Green 15:8, Blue 7:0
    BGR888,          //  Blue 24:16, Green 15:8, Red 7:0
    BGRA8888,        //  Blue 31:24, Green 23:16, Red 15:8, Alpha 7:0
    ARGB8888,        //  Alpha 31:24, Red 23:16, Green 15:8, Blue 7:0
    RGBX8888,
    XRGB8888,
    YUV420Planar,    //  I420
    YVU420Planar,    //  YV12
    YUV420SP,        //  NV12
    YVU420SP,        //  NV21
    YUV420Packed,    //  YUV420 Interleaved
    YVU420Packed,    //  YVU420 Interleaved
    YUV422Planar,    //  Three arrays Y,U,V.
    YVU422Planar,
    YUYVPacked,      //  422 packed per payload in planar slices
    YVYUPacked,      //  422 packed
    UYVYPacket,      //  422 packed
    VYUYPacket,      //  422 packed
    YUV422SP,        //  422 packed
    YVU422SP,
    YUV444Interleaved,  //  Each pixel contains equal parts YUV
    Y8,
    Y16,
    RAW
};


enum OBJECTTYPE {
    OT_VEHICLE,
    OT_HUMAN,
    OT_NON_MOTOR,
    OT_FACE,
    OT_FACE_BODY,
    OT_PLATE
};


template<class T>   // T: uint8_t  float
struct ImageData {
    IMAGEFORMAT format;   // Image format


    uint32_t width = 0;        // Image width
    uint32_t height = 0;       // Image height
    uint32_t channel = 0;      // Number of image channels
    uint32_t depth = 0;        // Bit depth
    uint32_t height_step = 0;  // Alignment height
    uint32_t width_step = 0;  // Alignment width
    uint32_t size = 0;         // Data size (in bytes)
```



```
    std::shared_ptr<T> data;   // Data pointer
};


template<class Archive, class T>
void serialize(Archive& ar, ImageData<T>& data) {
    ar(data.format, data.width, data.height, data.channel,
        data.depth, data.height_step, data.width_step, data.size);
    if (data.size > 0 && data.data.get() == nullptr) {
        data.data.reset(new(std::nothrow) T[data.size]);
    }
    ar(cereal::binary_data(data.data.get(), data.size * sizeof(T)));
}


// Parameter
template<class T>
struct ImagePara {
    FrameInfo f_info;          // Frame information
    ImageData<T> img;          // Image
};


template<class Archive, class T>
void serialize(Archive& ar, ImagePara<T>& data) {
    ar(data.f_info, data.img);
}


// Parameter
template<class T>
struct BatchImagePara {
    BatchInfo b_info;                  // Batch information
    std::vector<ImageData<T> > v_img;   // Image in the batch
};


template<class Archive, class T>
void serialize(Archive& ar, BatchImagePara<T>& data) {
    ar(data.b_info, data.v_img);
}




// Angle
typedef struct {
    float x;
    float y;
    float z;
}Angle;


template<class Archive>
void serialize(Archive& ar, Angle& data) {
    ar(data.x, data.y, data.z);
}


// 2D point
struct Point2D {
    int32_t x;
    int32_t y;
};


template<class Archive>
void serialize(Archive& ar, Point2D& data) {
    ar(data.x, data.y);
```



```
}


//////////////////////////////////////////////////////
/////////////// ROI information ///////////////////
//////////////////////////////////////////////////////


struct RoiPolygon {
    uint32_t         uiPtNum;       // Number of polygon vertices
    std::vector<Point2D> astPts;    // Multiple retrieval results (TopN)
};


template<class Archive>
void serialize(Archive &ar, RoiPolygon &data) {
    ar(data.uiPtNum, data.astPts);
}


struct ArgsRoiPolygon {
    RoiPolygon stRoiPolygon; // Configuration item ROI
    uint32_t iMinFace; // Minimum face size of 26 x 26
    uint32_t iMaxFace; // Maximum face size of 300 x 300
    uint32_t imgWidth; // Video frame image width
    uint32_t imgHeight; // Video frame image height
};


template<class Archive>
void serialize(Archive &ar, ArgsRoiPolygon &data) {
    ar(data.stRoiPolygon, data.iMinFace, data.iMaxFace, data.imgWidth, data.imgHeight, data.iMinFace);
}
// 3D point
struct Point3D {
    int32_t x;
    int32_t y;
    int32_t z;
};


template<class Archive>
void serialize(Archive& ar, Point3D& data) {
    ar(data.x, data.y, data.z);
}


// Lines in the 2D or 3D space
template<class T>   // T: Point2D  Point3D
struct Line {
    T start;
    T end;
};


template<class Archive, class T>
void serialize(Archive& ar, Line<T>& data) {
    ar(data.start, data.end);
}


// Rectangular plane in the 2D or 3D space
template<class T>  // T: Point2D  Point3D
struct Rectangle {
    T anchor_lt;
    T anchor_rb;
};
```



```
template<class Archive, class T>
void serialize(Archive& ar, Rectangle<T>& data) {
    ar(data.anchor_lt, data.anchor_rb);
}


// Polygon plane in the 2D or 3D space
template<class T>  // T: Point2D  Point3D
struct Polygon {
    std::vector<T> anchors;
};


template<class Archive, class T>
void serialize(Archive& ar, Polygon<T>& data) {
    ar(data.anchors);
}


// 3D cube
struct ROICube {
    Point3D anchor;
    uint32_t length;
    uint32_t width;
    uint32_t height;
};


template<class Archive>
void serialize(Archive& ar, ROICube& data) {
    ar(data.anchor, data.length, data.width, data.height);
}


// Mask matrix
template<class T>    // T: int32_t float etc.
struct MaskMatrix {
    uint32_t cols;
    uint32_t rows;
    std::shared_ptr<T> data;
};


template<class Archive, class T>
void serialize(Archive& ar, MaskMatrix<T>& data) {
    ar(data.cols, data.rows);
    if (data.cols*data.rows > 0 && data.data.get() == nullptr) {
        data.data.reset(new(std::nothrow) T[data.cols*data.rows]);
    }
    ar(cereal::binary_data(data.data.get(),
        data.cols * data.rows * sizeof(T)));
}


// RLE coding
struct RLECode {
    uint32_t len;
    uint32_t cols;
    uint32_t rows;
    std::shared_ptr<uint32_t> data;
};


template<class Archive>
void serialize(Archive& ar, RLECode& data) {
    ar(data.len, data.cols, data.rows);
    if (data.len > 0 && data.data.get() == nullptr) {
        data.data.reset(new(std::nothrow) uint32_t[data.len]);
    }
```



```
    ar(cereal::binary_data(data.data.get(),
        data.len * sizeof(uint32_t)));
}


template<class T1, class T2>    // T1: Point2D Rectangle RLECode etc.
struct ObjectLocation {          // T2: int8_t float etc.
    std::vector<uint32_t> v_obj_id;
    std::vector<T1> range;           // Target scope description
    std::vector<Angle> angle;        // Angle information
    std::vector<T2> confidence;      // Confidence
    std::vector<uint32_t> label;     // Target type tag
};


template<class Archive, class T1, class T2>
void serialize(Archive& ar, ObjectLocation<T1, T2>& data) {
    ar(data.v_obj_id, data.range, data.angle,
       data.confidence, data.label);
}


// Parameter
template<class T1, class T2>
struct DetectedObjectPara {
    FrameInfo f_info;                   // Frame information
    ObjectLocation<T1, T2> location;    // Multiple target positions
};


template<class Archive, class T1, class T2>
void serialize(Archive& ar, DetectedObjectPara<T1, T2>& data) {
    ar(data.f_info, data.location);
}


// Parameter
template<class T1, class T2>
struct BatchDetectedObjectPara {
    // Batch information
    BatchInfo b_info;
   // Multiple target positions corresponding to each frame of image
    std::vector<ObjectLocation<T1, T2> > v_location;
};


template<class Archive, class T1, class T2>
void serialize(Archive& ar, BatchDetectedObjectPara<T1, T2>& data) {
    ar(data.b_info, data.v_location);
}







//////////////////////////////////////////////////////
/////////////// 3. region image tensor ///////////////
//////////////////////////////////////////////////////


template<class T>
struct RegionImage {
    std::vector<ImageData<T> > region;
    std::vector<uint32_t> v_obj_id;
};


template<class Archive, class T>
```

```
void serialize(Archive& ar, RegionImage<T>& data) {
    ar(data.v_obj_id, data.region);
}


// Parameter
template<class T>
struct RegionImagePara {
    FrameInfo f_info;             // Frame information
    RegionImage<T> region;        // Small image corresponding to each frame
};


template<class Archive, class T>
void serialize(Archive& ar, RegionImagePara<T>& data) {
    ar(data.f_info, data.region);
}


// Parameter
template<class T>
struct BatchRegionImagePara {
    // Batch information
    BatchInfo b_info;
   // Each frame has several region images, which are referenced by a pointer pointing to the image array.
    std::vector<RegionImage<T>> v_region;
};


template<class Archive, class T>
void serialize(Archive& ar, BatchRegionImagePara<T>& data) {
    ar(data.b_info, data.v_region);
}




// Recursive ID tree node
struct IDTreeNode {
    std::vector<uint32_t> nodeID;      // Node ID
    std::vector<bool> is_leaf;         // Whether it is a leaf node
    std::vector<IDTreeNode> node;     // Pointer to the subnode structure
};


template <class Archive>
void serialize(Archive& ar, IDTreeNode& data) {
    ar(data.nodeID, data.is_leaf, data.node);
}


// Parameter
struct IDTreePara {
    FrameInfo f_info;    // Frame information
    IDTreeNode tree;     // ID tree in the image
};


template <class Archive>
void serialize(Archive& ar, IDTreePara& data) {
    ar(data.f_info, data.tree);
}


// Parameter
struct BatchIDTreePara {
    BatchInfo b_info;                  // Batch information
    std::vector<IDTreeNode> v_tree;     // ID tree in each frame of image
};
```



```
template <class Archive>
void serialize(Archive& ar, BatchIDTreePara& data) {
    ar(data.b_info, data.v_tree);
}



template<class T>
struct Attribute {
    std::string attr_value;
    T score;
};


template <class Archive, class T>
void serialize(Archive& ar, Attribute<T>& data) {
    ar(data.attr_value, data.score);
}


template<class T>
struct AttributeTopN {
    uint32_t obj_ID;
    std::map<std::string, std::vector<Attribute<T>> > attr_map;
};


template <class Archive, class T>
void serialize(Archive& ar, AttributeTopN<T>& data) {
    ar(data.obj_ID, data.attr_map);
}


template<class T>
struct AttributeVec {
    std::vector<AttributeTopN<T>> obj_attr;
};


template <class Archive, class T>
void serialize(Archive& ar, AttributeVec<T>& data) {
    ar(data.obj_attr);
}


// ²ÎÊý
template<class T>
struct AttributeResultPara {
    FrameInfo f_info;
    AttributeVec<T> attr;
};


template <class Archive, class T>
void serialize(Archive& ar, AttributeResultPara<T>& data) {
    ar(data.f_info, data.attr);
}



// ²ÎÊý
template<class T>
struct BatchAttributeResultPara {
    BatchInfo b_info;
    std::vector<AttributeVec<T>> v_attr;
```

```
};


template <class Archive, class T>
void serialize(Archive& ar, BatchAttributeResultPara<T>& data) {
    ar(data.b_info, data.v_attr);
}


template<class T>
struct Feature {
    int32_t len;                // Feature vector length
    std::shared_ptr<T> feature;      // Pointer to the feature vector
};


template<class Archive, class T>
void serialize(Archive& ar, Feature<T>& data) {
    ar(data.len);
    if (data.len > 0 && data.feature.get() == nullptr) {
        data.feature.reset(new(std::nothrow) T[data.len]);
    }
    ar(cereal::binary_data(data.feature.get(), data.len * sizeof(T)));
}




template<class T>
struct FeatureList {
    uint32_t obj_ID;                // Target ID
    std::vector<Feature<T>> feature_list;    // Multiple features of the target
};


template<class Archive, class T>
void serialize(Archive& ar, FeatureList<T>& data) {
    ar(data.obj_ID, data.feature_list);
}


template<class T>
struct FeatureVec {
    std::vector<FeatureList<T>> obj_feature;   // Attributes of multiple targets
};


template<class Archive, class T>
void serialize(Archive& ar, FeatureVec<T>& data) {
    ar(data.obj_feature);
}


// Parameter
template<class T>
struct FeatureResultPara {
    FrameInfo f_info;         // Frame information
    FeatureVec<T> feature;     // Features of multiple targets in an image
};


template<class Archive, class T>
void serialize(Archive& ar, FeatureResultPara<T>& data) {
    ar(data.f_info, data.feature);
}


// Parameter
template<class T>
```



```
struct BatchFeatureResultPara {
   // Batch information
    BatchInfo b_info;
    // Features of multiple targets corresponding to each frame of image
    std::vector<FeatureVec<T>> v_feature;
};


template<class Archive, class T>
void serialize(Archive& ar, BatchFeatureResultPara<T>& data) {
    ar(data.b_info, data.v_feature);
}




template<class T>
struct FeatureRecord {
    uint32_t ID;           // Feature map ID
    uint32_t len;          // Feature vector length
    std::shared_ptr<T> feature; // Pointer to the feature vector
};


template<class Archive, class T>
void serialize(Archive& ar, FeatureRecord<T>& data) {
    ar(data.ID, data.len);
    if (data.len > 0 && data.feature.get() == nullptr) {
        data.feature.reset(new(std::nothrow) T[data.len]);
    }
    ar(cereal::binary_data(data.feature.get(), data.len * sizeof(T)));
}


template<class T1, class T2>
struct RetrievalSet {
    uint32_t TopN;       // TopN result setting
    T1 threshold;        // Similarity threshold
    std::vector<FeatureRecord<T2>> record;    // All records in the feature set
};


template<class Archive, class T1, class T2>
void serialize(Archive& ar, RetrievalSet<T1, T2>& data) {
    ar(data.TopN, data.threshold, data.record);
}


// Parameter
template<class T1, class T2>
struct RetrievalSetPara {
    RetrievalSet<T1, T2> set;   // Parameter and feature set
};


template<class Archive, class T1, class T2>
void serialize(Archive& ar, RetrievalSetPara<T1, T2>& data) {
    ar(data.set);
}


struct RetrievalResult {
    uint32_t ID;        // Feature map ID
    float similarity;  // Similarity
};


template<class Archive>
void serialize(Archive& ar, RetrievalResult& data) {
```

```
    ar(data.ID, data.similarity);
}


struct RetrievalResultTopN {
    std::vector<RetrievalResult> result_list;   // Multiple retrieval results (TopN)
};


template<class Archive>
void serialize(Archive& ar, RetrievalResultTopN& data) {
    ar(data.result_list);
}


// Parameter
struct RetrievalResultPara {
    FrameInfo f_info;               // Frame information
    RetrievalResultTopN result;     // Retrieval result
};


template<class Archive>
void serialize(Archive& ar, RetrievalResultPara& data) {
    ar(data.f_info, data.result);
}


// Parameter
struct BatchRetrievalResultPara {
    // Batch information
    BatchInfo b_info;
    // Retrieval result corresponding to each frame of image
    std::vector<RetrievalResultTopN> v_result;
};


template<class Archive>
void serialize(Archive& ar, BatchRetrievalResultPara& data) {
    ar(data.b_info, data.v_result);
}


template<class T>
struct EvaluationResult {
    uint32_t obj_ID;            // Target ID
    std::string description;    // Description
    T score;                    // Evaluation score
};


template <class Archive, class T>
void serialize(Archive& ar, EvaluationResult<T>& data) {
    ar(data.obj_ID, data.description, data.score);
}


template<class T>
struct EvaluationResultVec {
    std::vector<EvaluationResult<T>> result;    // Evaluation results of multiple targets
};


template <class Archive, class T>
void serialize(Archive& ar, EvaluationResultVec<T>& data) {
    ar(data.result);
}
```



```
// Parameter
template<class T>
struct EvaluationResultPara {
    FrameInfo f_info;                // Frame information
    EvaluationResultVec<T> result;   // Evaluation result
};


template <class Archive, class T>
void serialize(Archive& ar, EvaluationResultPara<T>& data) {
    ar(data.f_info, data.result);
}


// Parameter
template<class T>
struct BatchEvaluationResultPara {
    // Batch information
    BatchInfo b_info;
    // Evaluation result of each frame of image
    std::vector<EvaluationResultVec<T>> v_result;
};


template <class Archive, class T>
void serialize(Archive& ar, BatchEvaluationResultPara<T>& data) {
    ar(data.b_info, data.v_result);
}




template<class T>
struct Classification {
    std::string class_value;   // Type value
    T score;                   // Confidence
};


template<class Archive, class T>
void serialize(Archive& ar, Classification<T>& data) {
    ar(data.class_value, data.score);
}


template<class T>
struct ClassificationTopN {
   uint32_t obj_ID;                            // Target ID
    std::vector<Classification<T>> class_result;   // Classification result (TopN)
};


template<class Archive, class T>
void serialize(Archive& ar, ClassificationTopN<T>& data) {
    ar(data.obj_ID, data.class_result);
}


template<class T>
struct ClassificationVec {
    std::vector<ClassificationTopN<T>> class_list;   // Classification of multiple targets
};


template<class Archive, class T>
void serialize(Archive& ar, ClassificationVec<T>& data) {
    ar(data.class_list);
}
```



```
// Parameter
template<class T>
struct ClassificationResultPara {
    FrameInfo f_info;                       // Frame information
    ClassificationVec<T> classification;       // Classification of multiple targets
};


template<class Archive, class T>
void serialize(Archive& ar, ClassificationResultPara<T>& data) {
    ar(data.f_info, data.classification);
}


// Parameter
template<class T>
struct BatchClassificationResultPara {
    // Batch information
    BatchInfo b_info;
   // Classification of multiple targets corresponding to each frame of image
    std::vector<ClassificationVec<T>> v_class;
};



template<class Archive, class T>
void serialize(Archive& ar, BatchClassificationResultPara<T>& data) {
    ar(data.b_info, data.v_class);
}


struct RawDataBuffer {
    uint32_t len_of_byte;  // size length
    std::shared_ptr<uint8_t> data; // One buffer, which can be converted to a user-defined type
};


template<class Archive>
void serialize(Archive& ar, RawDataBuffer& data) {
    ar(data.len_of_byte);
    if (data.len_of_byte > 0 && data.data.get() == nullptr) {
        data.data.reset(new(std::nothrow) uint8_t[data.len_of_byte]);
    }
    ar(cereal::binary_data(data.data.get(), data.len_of_byte *
        sizeof(uint8_t)));
}


// common raw databuffer struct
struct BatchRawDataBuffer {
    BatchInfo b_info;  // batch info
    std::vector<RawDataBuffer> v_info;
};


template<class Archive>
void serialize(Archive& ar, BatchRawDataBuffer& data) {
    ar(data.b_info, data.v_info);
}



template<typename T> const char* TypeName(void);


#define REGISTER_TYPE_DEFINITION(type) \
```

```
    template<> inline const char* TypeName<type>(void) { return #type; }


    REGISTER_TYPE_DEFINITION(int8_t);
    REGISTER_TYPE_DEFINITION(uint8_t);
    REGISTER_TYPE_DEFINITION(int16_t);
    REGISTER_TYPE_DEFINITION(uint16_t);
    REGISTER_TYPE_DEFINITION(int32_t);
    REGISTER_TYPE_DEFINITION(uint32_t);
    REGISTER_TYPE_DEFINITION(int64_t);
    REGISTER_TYPE_DEFINITION(uint64_t);
    REGISTER_TYPE_DEFINITION(float);
    REGISTER_TYPE_DEFINITION(double);
    REGISTER_TYPE_DEFINITION(Point2D);
    REGISTER_TYPE_DEFINITION(Point3D);
    REGISTER_TYPE_DEFINITION(Line<Point2D>);
    REGISTER_TYPE_DEFINITION(Line<Point3D>);
    REGISTER_TYPE_DEFINITION(Rectangle<Point2D>);
    REGISTER_TYPE_DEFINITION(Rectangle<Point3D>);
    REGISTER_TYPE_DEFINITION(Polygon<Point2D>);
    REGISTER_TYPE_DEFINITION(Polygon<Point3D>);
    REGISTER_TYPE_DEFINITION(ROICube);
    REGISTER_TYPE_DEFINITION(MaskMatrix<int8_t>);
    REGISTER_TYPE_DEFINITION(MaskMatrix<int32_t>);
    REGISTER_TYPE_DEFINITION(MaskMatrix<float>);
    REGISTER_TYPE_DEFINITION(RLECode);
    REGISTER_TYPE_DEFINITION(RoiPolygon);
    REGISTER_TYPE_DEFINITION(ArgsRoiPolygon);
}
```

7.4 Example

7.4.1 Orchestration Configuration Example

The configuration file for creating a graph (**graph.prototxt**) is in proto format. In the following example, two graphs are created at the same time, multiple engines are created for each graph, and the mapping is configured.

NOTICE

To transfer a file, the following conditions must be met. Otherwise, the system transfers a string or number by default.

- In the Linux operating system, the file must contain a relative path or an absolute path, for example, **/home/1.txt** or **../test/2.txt**. Note: The string may be mistakenly considered as a file if it contains the "\" or "/" symbol. Therefore, do not use the symbols.
- In Windows, the file must contain a relative path or an absolute path, for example: **c:\1.txt** or **..\test\2.txt**. Note: The string may be mistakenly considered as a file if it contains the "\" or "/" symbol. Therefore, do not use the symbols.

```
graphs {
  graph_id: 100
  device_id: "0"
  priority: 1
  engines {
    id: 1000
    engine_name: "SrcEngine"
```



```
    side: HOST
    thread_num: 1
  }
  engines {
    id: 1001
    engine_name: "HelloWorldEngine"
    so_name: "./libhelloworld.so"
    side: DEVICE
    thread_num: 1
  }
  engines {
    id: 1002
    engine_name: "DestEngine"
    side: HOST
    thread_num: 1
  }
  connects {
    src_engine_id: 1000
    src_port_id: 0
    target_engine_id: 1001
    target_port_id: 0
  }
  connects {
    src_engine_id: 1001
    src_port_id: 0
    target_engine_id: 1002
    target_port_id: 0
  }
}
graphs {
  graph_id: 200
  device_id: "1"
  priority: 1
  engines {
    id: 1000
    engine_name: "SrcEngine"
    side: HOST
    thread_num: 1
  }
  engines {
    id: 1001
    engine_name: "HelloWorldEngine"
    internal_so_name: "/lib64/libhelloworld.so"
    side: DEVICE
    thread_num: 1
  }
  engines {
    id: 1002
    engine_name: "DestEngine"
    side: HOST
    thread_num: 1
  }
  connects {
    src_engine_id: 1000
    src_port_id: 0
    target_engine_id: 1001
    target_port_id: 0
  }
  connects {
    src_engine_id: 1001
    src_port_id: 0
    target_engine_id: 1002
    target_port_id: 0
  }
}
```


7.4.2 Example of Data Transmission Based on Optimized Performance

(1) When the performance optimization solution is used to send data, the data must be manually serialized and deserialized.
// Note: The serialization function is used at the transmit end, while the deserialization function is used at the receive end. Therefore, the registration function must be registered at both the receive and transmit ends.

```
   Data structure
typedef struct
{
    uint32_t frameId;
    uint8_t  bufferId;
    uint8_t* image_data;
    uint8_t  image_size;
}TEST_STR;

// Serialize the TEST_STR structure. This function needs to be registered only by calling the registration function. The parameters are described as follows:
// Input: inputPtr, pointer to the TEST_STR structure
// Output: ctrlStr, control information address
        imageData, data information pointer
         imageLen, data information size
void GetTestStrSearPtr(void* inputPtr, std::string& ctrlStr, uint8_t*& imageData, uint32_t& imageLen)
{
   // Obtain the structure buffer.
    TEST_STR* test_str = (TEST_STR*)inputPtr;
    ctrlStr = std::string((char*)inputPtr, sizeof(TEST_STR));

   // Obtain data information and assign a value to the data pointer of the structure.
    imageData = (uint8_t*)test_str->image_data;
    imageLen = test_str->image_size;
}

// Deserialization structure. The returned data is the structure buffer and data block buffer.
// Input: ctrlPtr, control information address
         ctrlLen, control information size
         imageData, data information pointer
         imageLen, data information size
// Output: std::shared_ptr<void>, smart pointer to the structure
std::shared_ptr<void> GetTestStrDearPtr(const char* ctrlPtr, const uint32_t& ctrlLen,const uint8_t* imageData, const uint32_t& imageLen)
{
   // Obtain the structure.
    TEST_STR* test_str = (TEST_STR*)ctrlPtr;

   // Obtain the transferred large memory data.
   // Note: For large memory data, assign a value to the smart pointer and register the deleter. If the smart pointer is not used or the deleter is not registered, manually invoke hiai::Graph::ReleaseDataBuffer(void* ptr) when releasing the memory.
    std::shared_ptr<TEST_STR<uint8_t>> shared_data = std::make_shared<TEST_STR<uint8_t>>();
    shared_data->frameId = test_str->frame_ID;
    shared_data->bufferId= test_str->bufferId;
    shared_data->image_size = imageLen;
    shared_data->image_data.reset(imageData, hiai::Graph::ReleaseDataBuffer);

   // The smart pointer is returned to the engine on the device.
    return std::static_pointer_cast<void>(shared_data);
}
// Register the serialiation and deserialization functions.
HIAI_REGISTER_SERIALIZE_FUNC("TEST_STR", TEST_STR, GetTestStrSearPtr, GetTestStrDearPtr);
```

(2) When sending data, you must use a registered data type together with HIAI_DMalloc which is used to allocate data memory to improve the performance.
Note: During data transfer from the host to the device, you are advised to use HIAI_DMalloc, which can greatly improve the transmission efficiency. The HIAI_DMalloc interface supports the size of 0 to (256 MB – 96 bytes). If the data exceeds this range, use the malloc interface to allocate memory.


```
  // Use the Dmalloc interface to allocate the data memory. 10000 indicates the delay (in milliseconds), that is, if the memory is insufficient, wait for 10000 milliseconds.
   HIAI_StatusT get_ret =  HIAIMemory::HIAI_DMalloc(width*align_height*3/2,(void*&)align_buffer, 10000);

  // Send data. After this interface is called, the HIAI_Dfree interface does not need to be called. The dealy 10000 indicates the delay.
   graph->SendData(engine_id_0, "TEST_STR", std::static_pointer_cast<void>(align_buffer), 10000);
```

7.5 Change History

Release Date	Description
2020-05-30	This issue is the first official release.

