UHD World Association
世界超高清视频产业联盟

Technical specification for interface of ultra-high definition video processing algorithm

（Version NO. 1.0）

Release Time
2023-08-30

UHD World Association (UWA)
T/UWA 020-2023

Contents

Introduction

With the increasing demand for efficient batch reproduction of UHD video content in the radio and television field, industry companies have successively introduced UHD video reproduction algorithm technology. These technologies provide a variety of technical solutions for UHD reproduction of standard definition and high-definition video. However, the current ultra-high definition reproduction scheme is the all-in-one mode or cloud service mode that the algorithm is bound with the hardware device, and can only use the overall scheme of one company. If the scheme of other companies want to be changed, the all-in-one machines and services must be purchased again, which increases unnecessary costs. In addition, as professional users continue to deepen their understanding of relevant technologies, they hope to incorporate the algorithms of various companies to optimize videos in different types of scenes and maximize the advantages of each algorithm. However, because the algorithm interfaces of different companies are inconsistent, it is unable to meet customers' actual demand for complementary advantages and seamless switching of algorithms of different companies.

To solve the above problems, this document describes the UHD video processing algorithm interface specification, including algorithm task interface and algorithm service interface. The standard is formulated to realize seamless switching between different algorithm schemes for one device and allow the integration of different algorithm modules through the unification of algorithm interfaces, the standardization of algorithm framework processes, and the platformization of algorithm services. In this way, users can flexibly choose algorithm solutions from different companies and are no longer limited to specific hardware devices or service providers. At the same time, through the standardized algorithm interface, professional users can more easily achieve algorithm compatibility and complementary advantages, thus improving the effect and quality of video processing. The formulation of this standard is of great significance for promoting the development and application of UHD video processing technology, helping to meet the demand for efficient batch reproduction in the UHD field, while reducing customer costs and improving user experience.

The issuing authority of this document draws attention that the following related patents may be used when declaring compliance with this document:

——A data and video processing method and device (Chinese patent application No. 202211528132.2);

The issuing authority of this document has no position on the authenticity, validity and scope of the patent.

The patent holder has promised the issuing authority of this document that he is willing to negotiate with any applicant on reasonable and non-discriminatory terms and conditions for patent licensing. The declaration of the patent holder has been filed with the issuing authority of this document. Relevant information can be obtained through the following contact information:

Contact: Su Jing
Mailing address: No. 9, Dize Road, Beijing Economic and Technological Development Zone
Postal code: 100176
E-mail: sujing@boe.com.cn
Tel: 13811947489
Website: https://www.boe.com.cn/

Please note that in addition to the above patents, some contents of this document may still involve patents. The issuing agency of this document does not assume the responsibility of identifying the patent.

Technical specification for interface of ultra-high definition video processing algorithm

1 Scope

This document describes the definition, composition and classification, data type, interface requirements, etc. of the UHD video processing algorithm interface.
This document is applicable to the access and application of UHD video processing algorithms, and can also be used to guide the system integration and development of UHD video processing systems, algorithm packages and algorithm services.

2 Normative references

The contents in the following documents, through normative references in the text, constitute indispensable provisions of this document. Among them, the dated references are only applicable to the version corresponding to that date; For undated references, the latest version (including all amendments) is applicable to this document.
GB/T 5271.15-2008 Information technology-Vocabulary-Part15：Programming languages

3 Terms, definitions and abbreviations

3.1 Terms and definitions

The following terms and definitions are applicable to this document.

3.1.1 Algorithm task

A collection of subprogram modules that are executed simultaneously in an interleaved manner on one processor or multiple processors.

3.1.2 Algorithm service

Process management of algorithm tasks.

3.1.3 Object

The set and data of these operations that store and maintain the effects of the indicated operations.
[Source: GB/T5271.15-2008, 2]

3.1.4 Class

An abstract reference data type, which is applicable to the objects referred to, and defines the internal structure and a set of operation templates for the instances of these objects, including member variables and member functions.
[Source: GB/T5271.15-2008, 2, modified]

3.1.5 Member variable

A component of a class, consisting of an identifier, a set of data attributes, one or more addresses, and multiple data values, occupies a fixed length of memory, including data types and names.

3.1.6 **Member function**

The component of a class, usually with formal parameters, returns to the subroutine at its start with the generated data value.

3.2 **Abbreviations**

The following abbreviations are applicable to this document.
AI: Artificial Intelligence
AVS: Audio Video Coding Standard
HDR: High Dynamic Range
GPU: Graphics Processing Unit
MPEG: Moving Picture Expert Group
WMV: Windows Media Video

4 **Interface composition and classification**

4.1 **Summary**

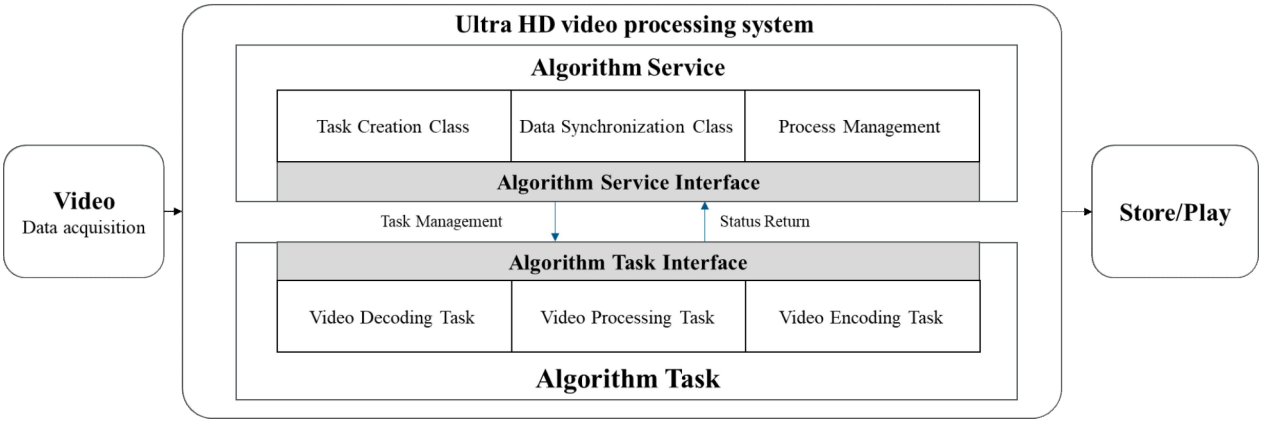

Figure 1 Interface composition diagram of UHD video processing system

Figure 1 shows the composition of the UHD video processing system interface, with the following functions:

a) Algorithm tasks include but are not limited to video decoding tasks, video processing tasks, and video encoding tasks;

—— The video decoding task decodes the input video file to obtain video media data;

—— Video processing tasks include but are not limited to video noise reduction, super resolution, video interpolation, HDR and other ultra HD video processing algorithm functions, which can be divided into AI algorithm tasks and other non AI traditional algorithm tasks;

—— The video encoding task is to encode the media data processed by the video processing task and output the video file.

The data transferred between algorithm tasks includes media data, metadata, etc. Metadata is used to record the name, version, parameters, manufacturer name, encoding and decoding parameters and other information of the UHD video processing algorithm.

b) The algorithm service schedules algorithm tasks concurrently. The specific functions include task creation, task scheduling, resource management, data synchronization, time synchronization, monitoring the return status of algorithm tasks and algorithm operation logs, and responding. The

data transferred between algorithm services includes configuration node and task process information.

c) Storage/playback: The video files output by the UHD video processing system can be stored in the local hard disk, or rendered and played on the terminal display device.

4.2 Interface classification

As shown in Figure 1, the UHD video processing algorithm interface is divided into algorithm task interface and algorithm service interface:

a) Algorithm task interface: the interface for calling algorithm task functions to realize specific algorithm functions;

b) Algorithm service interface: call the interface of algorithm task set to realize the process management of algorithm tasks.

5 Algorithm interface requirements

5.1 Algorithm task interface

5.1.1 Overview of algorithm task interface

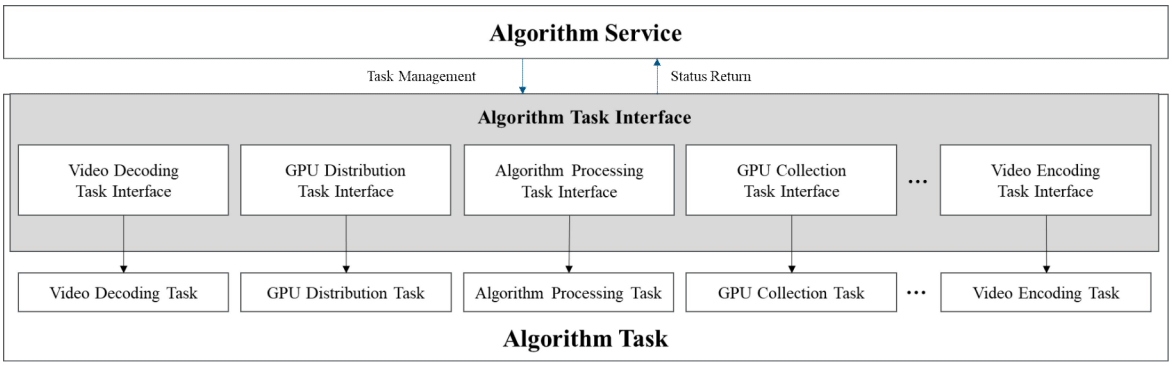

Figure 2 Schematic diagram of task interface of UHD video processing algorithm

Figure 2 is the schematic diagram of task interface of UHD video processing algorithm. The algorithm task interface includes video decoding task interface, GPU distribution task interface, video processing task interface, GPU collection task interface, and video coding task interface. The requirements are as follows:

a) Video decoding interface: obtain video decoding data by this interface;

b) GPU distribution task interface: distribute decoding frame data to GPU by his interface;

c) Video processing task interface: the algorithms process the video frame sequence by the interface, including but not limited to video noise reduction, super resolution, video interpolation, HDR and other video processing algorithm functions;

d) GPU collection task interface: sort the processed video frame data by this interface;

e) Video encoding task interface: output encoding data and files by this interface.

5.1.2 Algorithm task data type

5.1.2.1 Summary

Specifies the types of parameters transfered by the algorithm task interface.

5.1.2.2 Video stream encoding format type

The video stream encoding format type shall comply with the provisions in Table 1.

Table 1 Description of video stream encoding format type

Name	Type	Member List	Requirement
Video stream encoding format	Enumeration	MPEG1VIDEO	One of the necessary
		MPEG2VIDEO	
		MPEG4	
		RAWVIDEO	
		WMV1	
		WMV2	
		H264	
		AVS	
		VP5	
		VP6	
		VP8	
		H265	
		H266	
		VP7	
		AVS2	
		Scalable	

5.1.2.3 **Video stream pixel format type**

The video stream pixel format type shall comply with the provisions in Table 2.

Table 2 Description of video stream pixel format type

Name	Type	Member List	Requirement
Video stream pixel format	enumeration	YUV420P	One of the necessary
		RGB24	
		BGR24	
		YUV422P	
		YUV444P	
		GRAY8	
		PAL8	
		UYVY422	
		NV12	
		NV21	
		ARGB	
		RGBA	
		ABGR	
		BGRA	
		GRAY16BE	
		GRAY16LE	
		YUV440P	

		YUV420P10	
		YUV422P10	
		YUV444P10	
		P010	
		NV24	
		NV42	
		Scalable	

5.1.2.4 **Video stream gamut type**

The color gamut type of video stream shall comply with the provisions in Table 3.

Table 3 Description of Gamut Types

name	**type**	**Member List**	**requirement**
Video stream color gamut	enumeration	RGB	One of the necessary
		BT.709	
		FCC	
		BT.470 BG	
		SMPTE 170 M	
		SMPTE 240 M	
		YCOCG	
		BT.2020 NCL	
		BT.2020 CL	
		SMPTE 2085	
		Chroma derived NCL	
		Chroma derived CL	
		ICtCp	
		Scalable	

5.1.2.5 **HDR standard type**

The HDR standard type shall comply with the provisions in Table 4.

Table 4 HDR Standard Type Description

name	type	Metadata Type	Member List	requirement
HDR standard	enumeration	Static metadata	HDR10	One of the necessary
			HLG HDR	
			Scalable	
		Dynamic Metadata	HDR10+	
			Dolby Vision	
			HDR Vivid	
			Scalable	

5.1.2.6 **Video file encapsulation type**

The video file encapsulation type shall comply with the provisions in Table 5.

Table 5 Description of video file encapsulation type

name	type	Member List	requirement
Video file encapsulation	enumeration	MP4	One of the necessary
		MXF	
		AVI	
		MOV	
		Scalable	

5.1.2.7 **Identification type of video processing algorithm**

The identification type of video processing algorithm shall comply with the provisions in Table 6.

Table 6 Description of Identification Types of Video Processing Algorithms

name	type	Member List	requirement
Video processing algorithm	enumeration	noise reduction	One of the necessary
		Superscript	
		Interframe	
		HDR	
		Scalable	

5.1.2.8 **Metadata information type**

The metadata information type is in the form of string, which shall comply with the provisions in Table 7.

Table 7 Metadata Information Type Description

data type	Member List	requirement
Metadata information	Frame rate	necessary
	Video encoding mode	necessary
	Color space	necessary
	Bit depth	necessary
	Code rate	necessary
	Pixel format	necessary
	HDR standard name	necessary
	Video processing name	Optional
	Video processing version	Optional
	Name of manufacturer	Optional
	Processing algorithm name 1	Optional
	Processing parameter 1	Optional
	Processing algorithm name 2	Optional
	Processing parameter 2	Optional
	Scalable	-

5.1.2.9 **Algorithm task configuration information type**

The type of algorithm task configuration information is in the form of string, which shall comply with the

provisions in Table 8.

UHD video processing algorithm tasks include video decoding tasks, GPU distribution tasks, video processing tasks, GPU collection tasks, video coding tasks, etc. The video decoding task can decode the UHD video file, and the input file name needs to be configured; The GPU distribution task distributes the decoded video frame data, and the schedulable GPU resources need to be configured; The video processing task processes the distributed frame data and identifies the video processing algorithm to be configured; The frame data after video processing is collected orderly through GPU collection task; Finally, the video encoding task encodes and outputs the collected data. Encoder format, pixel format, packaging format, HDR standard type and output file name should be configured.

Table 8 Description of algorithm task configuration information

data type	explain	requirement
Algorithm task configuration information	Enter file name	necessary
	Schedulable GPU resource name	necessary
	Identification of video processing algorithm a	necessary
	Encoding format name	necessary
	Pixel format name	necessary
	Encapsulation format name	necessary
	HDR Standard Type Name	necessary
	Output file name	necessary
	Scalable	-
A See Table 6 for video processing algorithm.		

5.1.2.10 **Video decoding class**

The video decoding class implements the video decoding function, encapsulates the interface function of decoding video frame data, and obtains the interface function of original input video coding information, which shall comply with the provisions in Table 9.

Table 9 Description of main member functions of video decoding class

Class member function	**explain**	**requirement**
Open Video Function	The function parses the file information according to the path of the video file or the source of the video stream, and creates a decoding environment	necessary
Get video frame data function	Function to decode a frame of video frame data from a video file	necessary
Get video width function	Return video width	Optional
Get video height function	Return video high	Optional
Get video frame rate function	Return video frame rate	Optional
Get total video frames	Return the total number of video frames	Optional

function		
Get video encapsulation format function	Return the video encapsulation format information, the type is structure	Optional
Free resource function	Release the resources occupied by the class	necessary
Scalable	-	-

5.1.2.11 **Video coding class**

The video coding class implements the video coding function, encapsulates the interface functions for creating video files, video frame coding, etc., which should comply with the provisions of Table 10. The video file creation function needs to configure the encoding format and pixel format of the output video file, which should comply with the provisions of Table 11.

Table 10 Description of main member functions of video coding class

Class member function	explain	requirement
Create Video File Function	Create encoder and output video file according to the incoming parameters. See Table 11 for specific parameters	necessary
Set metadata information function	Set the metadata information required by the encoded video file. See 5.1.2.8 for specific parameters	Optional
Video frame encoding function	Encode a frame of video data, write it to the video file, and input it as the memory address of video frame data	necessary
Set Frame Rate Function	Set video encoder frame rate	necessary
Set rate function	Set video encoder code rate	necessary
Free resource function	Release the resources occupied by the class	necessary
Scalable	-	-

Table 11 Function parameter information of creating video file

Parameter name	Input/output type	explain	requirement
Video stream encoding format type	input parameter	See 5.1.2.2	necessary
Video stream pixel format type	input parameter	See 5.1.2.3	necessary
Video stream gamut type	input parameter	See 5.1.2.4	Optional
HDR standard type	input parameter	See 5.1.2.5	Optional
Video file encapsulation type	input parameter	See 5.1.2.6	necessary
Video file path	input parameter	Output video file path and file name	necessary
Video file width	input parameter	Width of output video frame	Optional
Video file high	input parameter	Height of output video frame	Optional
Scalable	-	-	-

5.1.3 **Requirements for algorithm task interface**

5.1.3.1 **Summary**

Unless otherwise specified, the return values of the algorithm task interface functions are: 0, if the call is successful, and error code if the call fails. See Appendix C for detailed error code types.

5.1.3.2 **Video decoding task interface function**

The video decoding task decodes the input video file or video stream, and the interface function list shall comply with the provisions in Table 12.

Table 12 Video decoding task interface function list

Interface function	explain	requirement
Initialization function of video decoding task	Initialize the video decoding task and transfer the video decoding class to the video decoding task; See 5.1.2.10 for video decoding	necessary
Data processing function of video decoding task	The main body of the video decoding task, which obtains each frame of data from the video decoding class and transfers the data to other tasks	necessary
Release resource function of video decoding task	Release the resources occupied by the video decoding task and destroy class member variables	necessary
Get video encapsulation format information function	Get the encapsulation format of the input video	Optional
Get video frame rate function	Get the frame rate of the input video	Optional
Get total video frames function	Get the total frames of the input video	Optional
Scalable	-	-

5.1.3.3 **Video encoding task interface function**

The video encoding task encodes the video frame data, and the interface function list shall comply with the provisions in Table 13.

Table 13 List of Video Encoding Task Interface Functions

Interface function	explain	requirement
Initialization function of video encoding task	Initialize the video encoding task and transfer the video encoding class to the video encoding task; Video coding, see 5.1.2.11	necessary
Data processing function of video encoding task	The main body of the video encoding task, which realizes encoding a frame of video data	necessary
Release resource function of video encoding task	Release the resources occupied by the task	necessary
Scalable	-	-

5.1.3.4 **GPU distribution task interface function**

The GPU distribution task distributes the decoded video frame data to algorithm tasks on different GPUs in the frame number sequence. The interface function list shall comply with the provisions in Table 14.

Table 14 GPU distribution task interface function list

Interface function	explain	requirement
Initialization function of GPU distribution task	Initialize the GPU distribution task. the input parameter of the function is the available GPU resource information.	necessary
Data processing function of GPU distribution task	Distribute data to GPUs	necessary
Release resource function of GPU distribution task	Release Task Resources	necessary
Scalable	-	-

5.1.3.5 **GPU Collection Task Interface Functions**

The GPU assembly task collects the video frame data processed by multiple video processing tasks according to the frame number sequence, and the interface function list shall comply with the provisions in Table 15.

Table 15 List of GPU Set Task Interface Functions

Interface function	explain	requirement
Initialization function of GPU collection task	Initialize the GPU collection task. the input parameter of the function is the available GPU resource information.	necessary
data processing function of GPU collection task	Orderly collection GPU data	necessary
Release resource function of GPU collection task	Release Task Resources	necessary
Scalable	-	-

5.1.3.6 **Video processing task interface function**

The video processing task carries out algorithm processing on the video frame data, including but not limited to over division, noise reduction, video interpolation, HDR, and the list of interface functions shall comply with the provisions in Table 16.

Table 16 List of Video Processing Task Interface Functions

Interface function	explain	requirement
Initialization function of video processing task	Initialize the algorithm task. The input parameter of the function is the identification type of the video processing algorithm. See 5.1.2.7	necessary
Data processing function of video processing task	Call video processing algorithm for data processing	necessary
Release resource function of video processing task	Release Task Resources	necessary
Scalable	-	-

5.1.4 **Algorithm task interface configuration process**

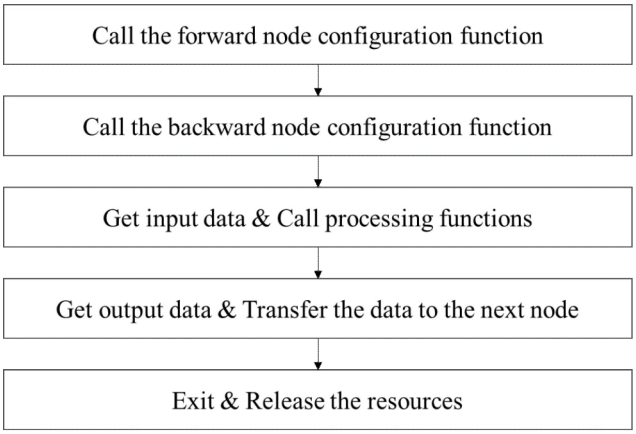

Figure 3 Algorithm Task Interface Configuration Process

Figure 3 shows the algorithm task interface configuration process. The requirements are as follows:

a) Call the forward node configuration function to configure the algorithm task and input the node information;

b) Call the backward node configuration function configuration algorithm task to output node information;

c) The data of the above node is readable, the input data is obtained, and the data processing function is called to process the data;

d) If the data of the next node can be written, transfer the data to the next node;

e) After the processing is completed, exit the algorithm task safely and release the resources.

5.2 Algorithm service interface

5.2.1 Overview of algorithm service interface

The algorithm service is responsible for managing algorithm tasks and scheduling resources in the UHD processing system, which includes three parts: task creation class, data synchronization class, and process management class.

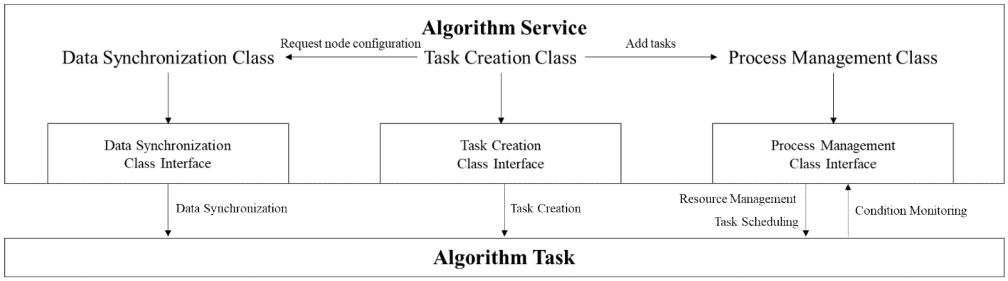

Figure 4 Schematic diagram of algorithm service interface

Figure 4 is the schematic diagram of the algorithm service interface. The algorithm service interface includes data synchronization class interface, task creation class interface, and process management class interface. These three class interfaces realize the flow management of algorithm tasks. The requirements are as follows:

a) Calling the data synchronization class interface can realize the data and time synchronization transmission of algorithm tasks;

b) Calling the task creation class interface can predefine the algorithm task rules;

c) Call the process management class interface to realize resource management, task scheduling and status monitoring.

5.2.2 Algorithm service data type

5.2.2.1 **summary**

Specifies the types of parameters passed by the algorithm service interface.

5.2.2.2 **Data node class**

The data node class encapsulates the interface functions for operating data, and the main members shall comply with the provisions in Table 17.

Table 17 Description of Main Members of Data Node Class

Member Type	name	explain	requirement
Member variable	Data link list	The specific contents of the data are stored and defined in the form of a lookup table	necessary
	Scalable	-	-
Member function	Initialization function	Create a data node class object based on the passed in parameters	necessary
	Get Data Function	The corresponding data according to the keyword can be searched. If the function returns NULL, it indicates failure. Otherwise, it indicates the memory address of the corresponding data	necessary
	Add Data Function	Data in the data linked list can be added	necessary
	Change Data Function	The corresponding data according to the keyword can be changed. If the function returns NULL, it means failure. Otherwise, it is the data memory address before the change	necessary
	Remove Data Function	The data in the data linked list can be removed according to the keyword. The function return indicates that NULL is a failure, otherwise it is the data memory address before removal	necessary
	Scalable	-	-

5.2.3 **Requirements for algorithm service interface**

5.2.3.1 **summary**

Unless otherwise specified, the status return values of the algorithm service interface functions are: call success, return 0; call failure, and return error code. See Appendix C for detailed error code types.

5.2.3.2 **Data synchronization class**

The data synchronization class implements the synchronous transmission mechanism of algorithm tasks, including data synchronization and time synchronization. The main members of the data synchronization class should comply with the provisions of Table 18.

Table 18 Description of Main Members of Data Synchronization Class

Member Type	name	explain	requirement
Member variable	Writable semaphore	In a multithreaded environment, it is used to ensure that writable memory is not called concurrently. The type is semaphore. When the value is greater than 0, it indicates that data can be written. When the value is less than 0, it indicates that data cannot be written.	necessary
	Readable semaphore	In a multithreaded environment, it is used to ensure that the readable memory is not called concurrently. The type is semaphore. When the value is greater than 0, it means the data is readable. When the value is less than 0, it means the data is unreadable	necessary
	Data node class	See 5.2.2.2	necessary
	Scalable	-	-
Member function	Initialization function	Create the member variable of the class and initialize the assignment.	necessary
	Free resource function	Destroy and release the member variable of the synchronization class	necessary
	Scalable	-	-

Figure 5 illustrates the principle mechanism of the data synchronization class, as follows:

Data synchronization uses semaphores for synchronization control. When waiting for the semaphore, the

current thread will be blocked until the semaphore value is greater than 0 to continue execution; Releasing the semaphore means that the value of the semaphore will increase by 1.

Wait for the readable semaphore of memory segment 0 to be greater than 0, and read the data; Then the writable semaphore of memory segment 0 is released, and the data of the previous algorithm task can be written to memory segment 0 at the same time; The current algorithm task processes the data and waits for the writable semaphore of memory segment 1 to be greater than 0 to write data; Then release the readable semaphore of memory segment 1, and notify the next algorithm task to read data from memory segment 1.

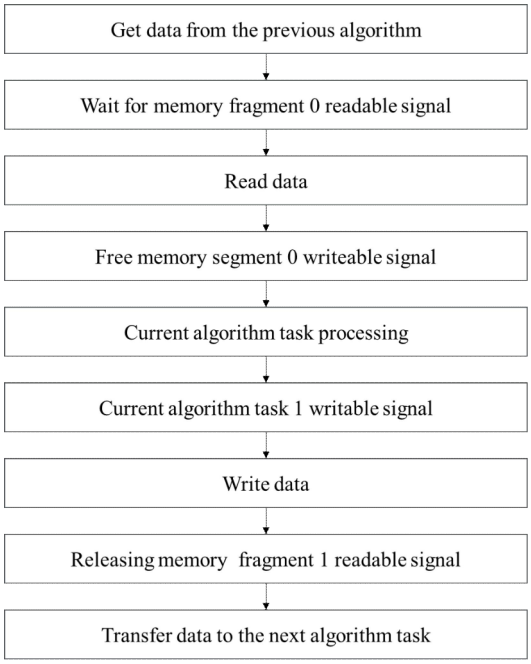

Figure 5 Principle mechanism of data synchronization class

5.2.3.3 **Task creation class**

The task creation class is a predefined rule for algorithm task creation. After algorithm tasks are created based on this rule, they can be added to algorithm services for process management. The main members of the task creation class should comply with the provisions of Table 19.

Table 19 Description of main members of task creation class

Member Type	name	explain	requirement
Member variable	Forward synchronization node	The type is data synchronization, see Table 18	necessary
	Backward synchronization node	The type is data synchronization, see Table 18	necessary
	Forward node data	The type is data node class, as shown in Table 17	necessary
	Backward node data	The type is data node class, as shown in Table 17	necessary
	Scalable		

Member function	Data processing function	Implement specific operations of algorithm tasks, and create functions that must be overloaded for algorithm tasks	necessary
	Resource release function	Release the resources of the algorithm task, and create functions that must be overloaded for the algorithm task	necessary
	Forward synchronization node configuration function	Set the forward synchronization node of the algorithm task as the function parameter type as the data synchronization type	necessary
	Backward synchronization node configuration function	Set the backward synchronization node of the algorithm task. The function parameter type is data synchronization type	necessary
	Data waiting function	Wait for the data of the last algorithm task. The function parameter type is data node class	necessary
	Data transfer function	Pass the data to the next algorithm task. The function parameter type is data node class	necessary
	Scalable	-	-

5.2.3.4 **Process management**

The process management class realizes the process management of algorithm tasks, which can allocate and manage resources for algorithm tasks, schedule algorithm tasks, and detect the status of algorithm tasks. The main members of the process management class should comply with the provisions in Table 20.

Table 20 Description of Main Members of Process Management

Member Type	name	explain	requirement
Member variable	Algorithm task storage table	Class member variable, which stores the linked list of algorithm tasks in the video processing list	necessary
	Scalable	-	-
Member function	Initialization function	Initialize the resources occupied by the video processing flow management framework class	necessary
	Add Task Function	The algorithm task is added to the video processing list, and the function parameter is the pointer to the algorithm task	necessary
	Algorithm task thread function	Create thread function of algorithm task	necessary
	Start Task Function	Create thread and start algorithm task	necessary
	Close Task Function	Close the algorithm task in the video processing list, and release the resources occupied by the algorithm task	necessary
	Algorithm task storage table	Class member variable, which stores the linked list of algorithm tasks in the video processing list	necessary
	Scalable	-	-

5.2.4 **Algorithm service interface configuration process**

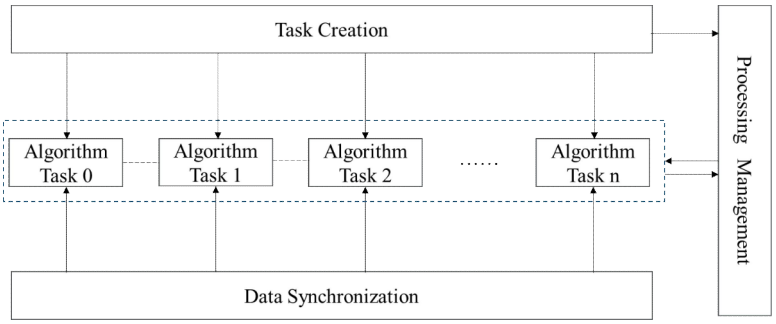

Figure 6 Reference execution flow of algorithm service interface

Figure 6 shows the reference implementation process of the algorithm service interface, which is specified as follows:

a) Create algorithm tasks, that is, algorithm task 0, algorithm task 1, algorithm task 2... algorithm task n in the figure;

b) Add algorithm tasks to the process management class for management, create threads for algorithm tasks, and allocate computing resources;

c) Create transfer objects between algorithm tasks, and call forward/backward node functions to configure algorithm task nodes;

d) When the output node data of the process management class monitoring algorithm task is empty, close the algorithm task and release resources;

e) If the incoming data is empty, the algorithm can be safely exit and the resources can be released.

5.3 **Supplementary interface description**

The interface is supplemented as follows:

a) The interface function is released in the form of dynamic link library and open to users;

b) Windows system, Linux system and MacOS system should be provided, and different operating systems should be compiled into different dynamic link libraries;

c) Memory management is implemented in specific algorithm tasks;

d) Appendix A is the data type and interface specification of audio processing task;

e) Appendix B is the function prototype description and c++language call examples;

f) Appendix C is the detailed description of the error code returned by the function;

g) Appendix D is a detailed description of the dependent library version.

Appendix A (Informative) Audio processing task data type and interface specification

A.1 Audio processing task data type

A.1.1 Audio encoding format type

The audio coding format type shall comply with Table A.1.

Table A.1 Description of Audio Coding Format Type

name	type	Enumerate member list
Audio encoding format	enumeration	AAC
		PCM
		Scalable

A.1.2 Audio decoding class

The audio decoding class encapsulates the interface function for acquiring audio frame data, which shall comply with the provisions in Table A.2.

Table A.2 Description of main member functions of audio decoding class

Class member function	explain
Open Audio Function	Create audio decoding environment
Get audio frame data function	Decode one frame of audio data
Free resource function	Release the resources occupied by the class

A.1.3 Audio encoding

The audio encoding class encapsulates the interface function for encoding audio frame data, which shall comply with the provisions in Table A.3.

A. 3 Description table of main member functions of audio encoding class

Class member function	explain
Create audio function	Create an audio coding environment according to the incoming parameters. See Table A.1 for specific parameters
Audio frame encoding function	Encode one frame of audio data
Free resource function	Release the resources occupied by the class

A.2 Audio processing task interface

A.2.1 summary

Unless otherwise specified, the return values of the audio processing task interface function are: call success, return 0, call failure, return error code. See Appendix C for detailed error code types.

A.2.2 Audio decoding task class interface

The audio decoding task decodes the input audio stream, and the interface function list shall comply with the

provisions in Table A.4.

Table A.4 List of Audio Decoding Task Interface Functions

Interface function	explain
Audio decoding task initialization function	Initialize the audio decoding task and transfer the audio decoding class to the audio decoding task; See Table A.2 for audio decoding
Audio decoding task data processing function	The main body of the audio decoding task is used to obtain each frame of data from the audio decoding class and transfer the data to other tasks
Audio decoding task release resource function	Release resources occupied by audio decoding task and destroy class member variables

A.2.3 **Audio coding task class interface**

The audio coding task encodes the audio data, and the interface function list shall comply with the provisions in Table A.5.

Table A.5 List of Audio Coding Task Interface Functions

Interface function	explain
Initialization function of audio coding task	Initialize the audio coding task and pass the audio coding class to the audio coding task; Audio coding, see A.3
Audio Coding Task Data Processing Function	It is the main body of the audio coding task, realizing the encoding of a frame of audio data
Audio coding task resource release function	Release Task Resources

附 录 B

Appendix B (Informative) Prototype of data type and interface function

B.1 Algorithm service interface function

B.1.1 Data synchronization class

B.1.1.1 Function declaration

```
Class UnitSynArg{
Public:
        Init()// Initialization function
Int release()// Resource release function
Sem_ T readable_ Sem// Readable semaphore
Sem_ T writeable_ Sem// Writable semaphore
Passing Para * passing_ Para// Data node class
};
```

B.1.1.2 Data node class

B.1.1.2.1 Function declaration

```
Class PassingPara{
Public:
        Std:: map<const char *, void *, cmp_ Str>para_ Dict// Data link list
Int init (PassingPara * Passing_para)// Initialization function
Void * get_ Item (const char *)// Get Data Function
Int add_ Item (const char *, void *)// Add Data Function
Void * change_ Item (const char *, void *)// Change Data Function
Void * remove_ Item (const char *)// Remove Data Function
};
```

B.1.1.2.2 Initialization function

Function prototype:

Int PassingPara:: init (PassingPara * Passing_para);

See Table B.1 for function call parameters.

Table B.1 Initialization Function Call Parameters

parameter	data type	Input/output type	Parameter Description
Passing_ Para	PassingPara*	input parameter	Data node object to copy

B.1.1.2.3 Get Data Function

Function prototype:

Void * PassingPara:: get_ Item (const char * key);

See Table B.2 for function call parameters.

Table B.2 Get Data Function Call Parameters

parameter	data type	Input/output type	Parameter Description
Key	Const char*	input parameter	Keyword for data lookup

B.1.1.2.4 Add Data Function

Function prototype:

Int PassingPara:: add_ Item (const char * key, void * val);

See Table B.3 for adding data function call parameters.

Table B.3 Adding Data Function Call Parameters

parameter	data type	Input/output type	Parameter Description
Key	Const char*	input parameter	Keyword for data lookup
Val	Void*	input parameter	Address pointer of memory data

B.1.1.2.5 Change Data Function

Function prototype:

Void * PassingPara:: change_ Item (const char * key, void * val);

See Table B.4 for function call parameters.

Table B.4 Changing Data Function Call Parameters

parameter	data type	Input/output type	Parameter Description
Key	Const char*	input parameter	Keyword for data lookup
Val	Void*	input parameter	Address pointer of memory data

B.1.1.2.6 Delete Data Function

Function prototype:

Void * PassingPara:: remove_ Item (const char * key);

See Table B.5 for function call parameters.

Table B.5 Delete Data Function Call Parameters

parameter	data type	Input/output type	Parameter Description
Key	Const char*	input parameter	Keyword for data lookup

B.1.2 Process management

B.1.2.1 Function declaration

```
Class VideoProcessFramework{
Public:
Int init()// Initialization function
Int add_ Task (FrameProcessUnit * frame_process_unit)// Add Task Function
Int start_ Tasks ()// Start Task Function
Int close_ Tasks ()// Close Task Function
Private:
Static void * process_ Thread (void * arg)// Algorithm task thread function
Std:: vector<FrameProcessUnit *>process_ List// Algorithm task storage list
```

};

B.1.2.2 Add Task Function

Function prototype:

Nt VideoProcessFramework:: add_ Task (FrameProcessUnit * frame_process_unit);

See Table B.6 for function call parameters

Table B.6 Adding Task Function Call Parameters

parameter	data type	Input/output type	Parameter Description
Frame_ Process_ Unit	FrameProcessUnit*	input parameter	Pointer to algorithm task unit

B.1.3 Task creation class

B.1.3.1 Function declaration

```
Class FrameProcessUnit{
Public:
Int pre_ Node (std:: vector<UnitSynArg *>in_nodes)// Forward node configuration function
Int next_ Node (std:: vector<UnitSynArg *>out_nodes)// Backward node configuration function
Int wait (int index, PassingPara * * para_temp)// Data waiting function
Int post (int index, PassingPara * para_temp)// Data transfer function
Virtual int process() {}// Data processing function
Virtual int close() {}// Resource release function
Private:
Std:: vector<UnitSynArg *>syn_ In// Forward synchronization node
Std:: vector<PassingPara *>para_ In// Read forward data transfer
Std:: vector<UnitSynArg *>syn_ Out// Backward synchronization node
Std:: vector<PassingPara *>para_ Out// Read backward data transfer
};
```

B.1.3.2 Forward node configuration function

Function prototype:

Int FrameProcessUnit:: pre_ Node (std:: vector<UnitSynArg *>in_nodes);

See Table B.7 for function call parameters.

Table B.7 Forward Node Configuration Function Call Parameters

parameter	data type	Input/output type	Parameter Description
In_ Nodes	Std:: vector<UnitSynArg *>	input parameter	Set the forward node of the algorithm task

B.1.3.3 Backward node configuration function

Function prototype:

Int FrameProcessUnit:: next_ Node (std:: vector<UnitSynArg *>out_nodes);

See Table B.8 for function call parameters.

Table B.8 Configuration Function Call Parameters for Backward Nodes

parameter	data type	Input/output type	Parameter Description
In_ Nodes	Std:: vector<UnitSynArg *>	input parameter	Set the backward node of the algorithm task

B. 1. 3. 4 Data waiting function

Function prototype:

Int FrameProcessUnit:: wait (int index, PassingPara * * para_temp);

See Table B.9 for function call parameters.

Table B.9 Data Waiting Function Call Parameters

parameter	data type	Input/output type	Parameter Description
Index	Int	input parameter	Index number of forward data node of algorithm task
Para_ Temp	PassingPara**	Input/output parameters	Pointer of algorithm task forward data node pointer

B. 1. 3. 5 Data transfer function

Function prototype:

Int FrameProcessUnit:: post (int index, PassingPara * para_temp);

See Table B.10 for function call parameters.

Table B.10 Data Transfer Function Call Parameters

parameter	data type	Input/output type	Parameter Description
Index	Int	input parameter	Index number of algorithm task backward data node
Para_ Temp	PassingPara**	Input/output parameters	Pointer to data node pointer after algorithm task

B. 2 Algorithm task interface function

B. 2. 1 Audio and video decoding class

B. 2. 1. 1 Function declaration

```
Class VideoReader{
Public:
Int open_ Input (const char * filename);
Int read_ Frame (void *&data_out);
Int getwidth();
Int gethight();
Void get_ Fps (int&den, int&num);
Void get_ Nb_ Frames (int&nbframes);
AVFormatContext * get_ Ifmt_ Ctx();
Int close();
Private:
```

```
//Video context information
AVFormatContext * ifmt_ Ctx=NULL;
//Decoded frame
AVFrame * frame=NULL;
//Media Streaming
AVStream * stream=NULL;
//Number of video media stream
Int video_ Stream_ Index=-1// Add the index of the video stream
Int audio_ Stream_ Index=-1// Add the index of the audio stream
//Encoding Context
AVCodecContext * codec_ Ctx=NULL;
//Frame format conversion context
SwsContext * swsctx=NULL;
//Frame conversion output decoding frame
AVFrame * out_ Frame=NULL;
//Output pixel format
AVPixelFormat outfixfmt;
//Video width
Int width=0;
//Video high
Int height=0;
//Width of output image
Int output_ Width=0;
//Height of output image
Int output_ Height=0;
//Bytes per pixel corresponding to video pixel format
Int pixbyte=0;
};
```

B. 2. 1. 2 Open input video function

Function prototype:

Int VideoReader:: open_ Input (const char * filename);

See Table B.11 for function call parameters.

Table B.11 Open Input Video Function Call Parameters

parameter	data type	Input/output type	Parameter Description
Filename	Const char*	input	Video file path/video stream

B. 2. 1. 3 Get video frame data function

Function prototype:

Int VideoReader:: read_ Frame (void *&data_out);

See Table B.12 for the function call parameters to obtain video frame data.

Table B.12 Get Video Frame Data Function Call Parameters

parameter	data type	Input/output type	Parameter Description
Data_ Out	Void*&	I/O	Data after video/audio decoding

B. 2. 2 Audio and video coding class

B. 2. 2. 1 Function declaration

```
Class VideoWriter{
Public:
    Typedef struct metadataParam{
    Char * key;
    Metadata_ ID value;
      }MetadataParam// Metadata Structure
      //Set metadata information function
      Int set_ Metadata_ Param (metadataParam * metadata_param);
        //创建视频文件函数
Int open_ Video (const char * filename, int width, int height,
                        Filefmt_ ID file_ Id=0,
                        Codec_ ID codec_ Id=0,
                        Pixfmt_ ID pixfmt_ Id=0,
                        Hdr_ ID hdr_ Id=0,
                        ACodec_ ID acodec_ Id=0,
                        AVFormatContext * ifmt_ Ctx=NULL);
Int write_ Videoframe (uint8_t * frame)// Video frame encoding function
Int write_ Audiopack (AVPacket * avpkt)// Audio frame encoding function
Int flush()// Brush Frame Function
Int set_ Fps (int den, int num)// Set Frame Rate Function
Int set_ BitRate (int rate)// Set rate function
Int close()// Free resource function
Private:
        Std:: vector<metadataParam *>metadata_ Param_ List// List for storing metadata information
//Output File Context
AVFormatContext * pFormatCtx=NULL;
//Input File Context
AVFormatContext * ifmt_ Ctx=NULL;
//Output file encapsulation format
AVOutputFormat * fmt=NULL;
//Output video stream
AVStream * video_ St=NULL;
AVStream * video_ St_ In=NULL;
AVStream * audio_ St_ In=NULL;
AVStream * audio_ St_ Out=NULL;
Int video_ Stream_ Index=-1;
Int audio_ Stream_ Index=-1;
```

```
//Encoding Context
AVCodecContext * pCodecCtx=NULL;
AVCodecContext * pAudioDeCodecCtx=NULL;
AVCodecContext * pAudioEnCodecCtx=NULL;
//Encoder
AVCodec * pCodec=NULL;
AVCodec * pAudioDeCodec=NULL;
AVCodec * pAudioEnCodec=NULL;
//Coded frame
AVPacket * packet=NULL;
//Original data frame
AVFrame * pFrame=NULL;
//Frame format conversion context
SwsContext * swsctx=NULL;
//Encoder name
Const char * codec_ Name=NULL;
//Code ID
AVCodecID av_ Codec_ Id;
AVCodecID av_ Audio_ Codec_ Id;
//Pixel format
AVPixelFormat config_ Pixfmt;
//Video metadata
AVDictionary * dict=0;
//Write lock
Pthread_ Mutex_ T write_ Lock;
//Frame count
Long int frame_ Cnt=0;
//Width of video
Int width=0;
//Video height
Int height=0;
//Frame rate
Int fps_ Den=1;
Int fps_ Num=25;
//Code rate
Long int bit_ Rate=100000000;
//Hdr mode switch
Int hdr_ On=0;
//Create Output Video File Function
Int open_ Output_ File (const char * filename, int widht, int height);
//Audio resampling encoding correlation ffmpeg function
SwrContext * audio_ Resampler_ Context=NULL;
AVAudioFifo * audio_ Fifo=NULL;
Int init_ Fifo (AVAudioFifo * * fifo, AVCodecContext * output_codc_context);
```

```
Int convert_ Samples (const uint8_t * * input_data,
Uint8_ T * * converted_ Data, const int frame_ Size,
SwrContext * example_ Context);
Int init_ Resampler (AVCodecContext * input_codc_context,
AVCodecContext * output_ Codec_ Context,
SwrContext * * example_ Context);
Init_ Converted_ Samples (uint8_t * * * converted_input_samples,
AVCodecContext * output_ Codec_ Context,
Int frame_ Size);
Add_ Samples_ To_ Fifo (AVAudioFifo * fifo,
Uint8_ T * * converted_ Input_ Samples,
Const int frame_ Size);
};
```

B.2.2.2 Create Video File Function

Function prototype:

```
Int VideoWriter:: open_ Video (const char * filename, int width, int height,
                               Filefmt_ ID file_ Id=0,
                               Codec_ ID codec_ Id=0,
                               Pixfmt_ ID pixfmt_ Id=0,
                               Hdr_ ID hdr_ Id=0,
                               ACodec_ ID acodec_ Id=0,
                               AVFormatContext * ifmt_ Ctx=NULL);
```

See Table B.13 for function call parameters.

Table B.13 Function Call Parameters for Creating Video Files

parameter	data type	Input/output type	Parameter Description
Filename	Const char*	input	Path and file name of output video file
Width	Int	input	Width of output video file
Height	Int	input	Height of output video file
File_ Id	Filefmt_ ID	input	The package format of the output video file is mp4 by default
Codec_ Id	Codec_ ID	input	The type of video encoder. The default is video H264
Pixfmt	Pixfmt_ ID	input	Pixel format of video coding frame, default to YUV420P
Hdr_ Id	Hdr_ ID	input	Video file HDR standard type, no HDR by default
Acodec_ Id	ACodec_ ID	input	Type of audio encoder, AAC by default
Ifmt_ Ctx	AVFormatContext*	input	The encapsulation environment of the source

			video file, which can selectively assign values to the encapsulation environment corresponding to the output video file

B. 2. 2. 3 Video frame encoding function

Function prototype:

Int VideoWriter:: write_ Videoframe (uint8_t * frame);

See Table B.14 for function call parameters.

Table B.14 Video Frame Encoding Function Calling Parameters

parameter	data type	Input/output type	Parameter Description
Frame	Uint8_ T*	input	Video frame data pointer

B. 2. 2. 4 Audio frame encoding function

Function prototype:

Int VideoWriter:: write_ Audiopacket (AVPacket * avpkt);

See Table B.15 for function call parameters.

Table B.15 Audio Frame Coding Function Calling Parameters

parameter	data type	Input/output type	Parameter Description
Avpkt	AVPacket*	input	Audio packet

B. 2. 2. 5 Set Frame Rate Function

Function prototype:

Int VideoWriter:: set_ Fps (int den, int num);

See Table B.16 for function call parameters.

Table B.16 Setting Frame Rate Function Call Parameters

parameter	data type	Input/output type	Parameter Description
Den	Int	input	Time based molecule
Num	Int	input	Denominator of time base

B. 2. 2. 6 Set rate function

Function prototype:

Int VideoWriter:: set_ BitRate (long int bit_rate);

See Table B.17 for function call parameters.

Table B.17 Setting Bit Rate Function Call Parameters

parameter	data type	Input/output type	Parameter Description
Bit_ Rate	Long int	input	Bit rate of output video file

B. 2. 2. 7 Set metadata information function

Function prototype:

Int set_ Metadata_ Param (metadataParam * metadata_param);

See Table B.18 for function call parameters.

Table B.18 Setting Metadata Information Function Call Parameters

parameter	data type	Input/output type	Parameter Description
Metadata_ Param	MetadataParam*	input	Pointer to metadata information structure

B. 2. 3 Audio and video decoding task

B. 2. 3. 1 Function declaration

```
Class VideoReaderUnit: public FrameProcessUnit{
Public:
Int init (const char * input_path);
AVFormatContext * get_ Ifmt_ Ctx();
Int process() override;
Int close() override;
Int get_ Fps (int&den, int&num);
Int get_ Nb_ Frames (int&nbframes);
VideoReader video_ Reader;
};
```

B. 2. 3. 2 Initialization function

Function prototype:

Int VideoReaderUnit:: init (const char * input_path);

See Table B.19 for function call parameters.

Table B.19 Initialization Function Call Parameters

parameter	data type	Input/output type	Parameter Description
Input_ Path	Const char*	input	Address and file name of video file

B. 2. 3. 3 Get video frame rate function

Function prototype:

Int VideoReaderUnit:: get_ Fps (int&den, int&num);

See Table B.20 for function call parameters.

Table B.20 Get Video Frame Rate Function Call Parameters

parameter	data type	Input/output type	Parameter Description
Den	Int	I/O	Time based molecule
Num	Int	I/O	Denominator of time base

B. 2. 3. 4 Get total video frames function

Function prototype:

Int VideoReaderUnit:: get_ Nb_ Frames (int&nbframes);

See Table B.21 for function call parameters.

Table B.21 Function Call Parameters for Obtaining Total Video Frames

parameter	data type	Input/output type	Parameter Description
Nbframes	Int&	I/O	Total frames of video files

B. 2. 4 Video encoding task

B. 2. 4. 1 Function declaration

```
Class VideoWriterImgUnit: public FrameProcessUnit{
    Public:
Int init (VideoWriter * video_writer)// Initialization function
Int process() override// Data processing function
Int close() override// Resource release function
Private:
VideoWriter * video_ Writer// Audio and video coding class
};
```

B. 2. 4. 2 Initialization function

Function prototype:

Int VideoWriterImgUnit:: init (VideoWriter * video_writer);

See Table B.22 for function call parameters.

Table B.22 Initialization Function Call Parameters

parameter	data type	Input/output type	Parameter Description
Video_ Writer	VideoWriter*	input	Audio and video coding class pointer

B. 2. 5 Audio encoding task

B. 2. 5. 1 Function declaration

```
Class VideoWriterPktUnit: public FrameProcessUnit{
Public:
Int init (VideoWriter * video_writer)// Initialization function
Int process() override// Data processing function
Int close() override// Resource release function
Private:
VideoWriter * video_ Writer// Audio and video coding class
};
```

B. 2. 5. 2 Initialization function

Function prototype:

Int VideoWriterPktUnit:: init (VideoWriter * video_writer);

See Table B.23 for function call parameters.

Table B.23 Initialization Function Call Parameters

parameter	data type	Input/output type	Parameter Description

Video_ Writer	VideoWriter*	input	Audio and video coding class pointer

B. 2. 6 GPU distribution task

B. 2. 6. 1 Function declaration

```
Class OneToManyUnit: public FrameProcessUnit{
    Public:
    Int init (vector<int>vecdeviceid)// Initialization function
    Int process() override// Data processing function
    Int close() override// Resource release function
    Private:
    Int numGPU=0// Total GPUs
    Int cnt=0// count
};
```

B. 2. 6. 2 Initialization function

Function prototype:

Int OneToManyUnit:: init (vector<int>vecDeviceID);

See Table B.24 for function call parameters.

Table B.24 Initialization Function Call Parameters

parameter	data type	Input/output type	Parameter Description
VecDeviceID	Vector<int>	input	ID for storing GPU resources in the container

B. 2. 7 GPU Collection Task

B. 2. 7. 1 Function declaration

```
Class ManyToOneUnit: public FrameProcessUnit{
    Public:
    Int init (vector<int>vecdeviceid)// Initialization function
    Int process() override// Data processing function
    Int close() override// Resource release function
    Private:
    Int numGPU=0// Total GPUs
    Int cnt=0// count
};
```

B. 2. 7. 2 Initialization function

Function prototype:

Init (vector<int>vecdeviceid);

See Table B.25 for function call parameters.

Table B.25 Initialization Function Call Parameters

parameter	data type	Input/output type	Parameter Description
VecDeviceID	Vector<int>	input	ID for storing GPU resources in the container

B.2.8 AI model processing task

B.2.8.1 Function declaration

```
Class modelProcessUnit: public FrameProcessUnit{
    Public:
    Int init (eumMODEL_ID modelID, int gpu_id)// Initialization function
    Int process() override// Data processing function
    Int close() override// Resource release function
    Private:
    Int getModelConfig (eumMODEL_ID modelID, modelConfigInfo&thisConfig)// Get model configuration information function
    MyconfigInfo model_ Config [MODEL_NUM]// Model Configuration Information Table
    Trtfilter * model=NULL// Model tensor inference engine pointer
};
```

B.2.8.2 Initialization function

Function prototype:

Int modelProcessUnit:: init (eumMODEL_ID modelID, int gpu_id);

See Table B.26 for function call parameters.

Table B.26 Initialization Function Call Parameters

parameter	data type	Input/output type	Parameter Description
ModelID	EumMODEL_ ID	input	Model enumeration number
Gpu_ Num	Int	input	ID number of GPU

B.3 Call example

The call example is as follows:

```
Int main (int argc, char * argv []){
Int gpu_ Num=0// Number of GPUs
CudaSts=cudaGetDeviceCount (&gpu_num)
CudaGetDeviceCount (&gpu_num);
    Char * pInputVideo="./test_video/2022-08-09-172550. mov"// Enter the video path and file name
Char * pOutVideo="./out_video/"// Output video path
    Long int nBitRate=88888888// Output video bit rate
    //输出视频编码信息格式
Codec_ ID codec_ Id=H264;
Pixfmt_ ID pixfmt_ Id=YUV420;
        Vector<eumMODEL_ ID>vecTrtModelID// AI algorithm model container
VecTrtModelID.push_ Back (SR_1080)// Add super score model
```

```
VecTrtModelID.push_ Back (HDR_2160)// Add HDR model
Int nw=1920// Enter the width of the video decoding target
Int nh=1080// Enter the height of the video decoding target
Int nc=3// Enter the number of video channels
Int nscale=2// Overshoot model tension ratio
Int ishdr=1// Whether it is hdr mode
Gpu_ Num=vecdeviceid. size()// Number of gpus
//Initialization
Int nw_ Sr=nw * nscale// Output video width
Int nh_ Sr=nh * nscale// Output video high
VideoProcessFramework video_ Process_ Framework// Create video pipeline class
    VideoReader * video_ Reader=new VideoReader (pInputVideo, nw, nh, true)// Create audio and video decoding class
VideoReaderUnit video_ Reader_ Unit// Create video decoding task
SdkSts=video_ Reader_ Unit. init (video_reader);
VideoWriter * video_ Writer=new VideoWriter()// Create audio and video coding class
Int den=0, num=0;
Int nbframes;
Video_ Reader_ Unit.get_ Fps (den, num);
Video_ Reader_ Unit.get_ Nb_ Frames (nbframes);
Video_ Writer ->set_ Fps (den, num);
Video_ Writer ->set_ BitRate (nBitRate);
SdkSts=video_ Writer ->open_ Video (pOutVideo, nw_sr, nh_sr,
Codec_ Id, pixfmt_ Id, video_ Reader_ Unit.get_ Ifmt_ Ctx());
VideoWriterImgUnit video_ Writer_ Img_ Unit// Create video encoding task
Video_ Writer_ Img_ Unit. init (video_writer, iscopyfrc, frc, ishdr);
Std:: vector<UnitSynArg *>node_ VideoWriteUnit_ In;
VideoWriterPktUnit video_ Writer_ Pkt_ Unit// Create audio encoding task
Video_ Writer_ Pkt_ Unit. init (video_writer);
//Video and audio data are output after the video file is decoded, so there are two synchronization nodes
//Package the two synchronization nodes
    UnitSynArg * synArg_ VideoReadUint_ Img_ Out=new UnitSynArg()// Video data node
    UnitSynArg * synArg_ VideoReadUint_ Pkt_ Out=new UnitSynArg()// Audio Data Node
Std:: vector<UnitSynArg *>videoReadUint_ Node_ Out={synArg_videoReadUint_img_out,
synArg_videoReadUint_pkt_out};
Video_ Reader_ Unit.next_ Node (videoReadUint_node_out)// Configure the backward node of video decoding task
FrameProcessUnit * video_ Reader_ Process_ Unit=&video_ Reader_ Unit//
Video_ Process_ Framework.add_ Task (video_reader_process_unit)// Add the video decoding task to the video processing pipeline
Std:: vector<UnitSynArg *>node_ VideoPktwWriter_ In={synArg_videoReadUint_pkt_out};
//Configure the entry for writing audio
Video_ Writer_ Pkt_ Unit.pre_ Node (node_videoPktwWriter_in);
//Add audio coding task to video processing pipeline
```

```
FrameProcessUnit * video_ Writer_ Pkt_ Process_ Unit=&video_ Writer_ Pkt_ Unit;
Video_ Process_ Framework.add_ Task (video_writer_pkt_process_unit);
Int model_ Nums=vecTrtModelID. size();
    //分发器
    OneToManyUnit one_ To_ Many;
    One_ To_ Many. init (vecdeviceid);
    Std:: vector<UnitSynArg *>node_ OneToManyUint_ In={synArg_videoReadUint_img_out}// The entrance of the distributor is video_ Video data outlet of reader class
    One_ To_ Many.pr_ Node (node_oneToManyUint_in);
    Std:: vector<UnitSynArg *>node_ OneToManyUint_ Out//
    //Many_ To_ One integrator
    ManyToOneUnit many_ To_ One;
    Many_ To_ One. init (vecdeviceid);
    Std:: vector<UnitSynArg *>node_ ManyToOneUint_ In//
    Std:: vector<UnitSynArg *>node_ ManyToOneUint_ Out//
    //根据 GPU 数量对 AI 算法模型任务进行初始化
TensorrtProcessUnit tensor_ Model [gpu_num];
   For (int i=0; i<gpu_num; i++)
    {
Tensorrt_ Model [i]. init (vecTrtModelID, vecdeviceid, i, ishdr);
    }
//Cycle each model to process the model
For (int i=0; i<gpu_num; i++)
{
//One entry and one exit node for each model
UnitSynArg * synArg_ TrtModelUnit_ In=new UnitSynArg()//
UnitSynArg * synArg_ TrtModelUnit_ Out=new UnitSynArg();
Node_ OneToManyUint_ Out.push_ Back (synArg_trtModelUnit_in)// The outlet of the distributor is the inlet of multiple gpus and trt model units
Node_ ManyToOneUint_ In.push_ Back (synArg_trtModelUnit_out)// The entrance of the aggregator is the exit of multiple gpus and trt model units
//Configure the entry node of AI algorithm model processing task
Std:: vector<UnitSynArg *>node_ TrtModelUint_ In={synArg_trtModelUnit_in};
Tensorrt_ Model [i]. pr e_ Node (node_trtModelUint_in);
//Configure the exit node of AI algorithm model processing task
Std:: vector<UnitSynArg *>node_ TrtModelUint_ Out={synArg_trtModelUnit_out};
Tensorrt_ Model [i]. next_ Node (node_trtModelUint_out);
//Add AI algorithm model processing task to the video processing pipeline
FrameProcessUnit * tensor_ Model_ Process_ Unit=&(tensor_model [i]);
Video_ Process_ Framework.add_ Task (tensor_model_process_unit);
}
    //将分发任务加入视频处理流水线中
    One_ To_ Many.next_ Node (node_oneToManyUint_out);
    FrameProcessUnit * one2many_ Unit=&one_ To_ Many;
```

```
        Video_ Process_ Framework.add_ Task (one2many_unit);
        //The exit of AI model task is many_ To_ Entrance of one integrator
        Many_ To_ One.pre_ Node (node_ManyToOneUint_in);
        //配置模型出口节点
        UnitSynArg * synArg_ MangToOneUint_ Out=new UnitSynArg;
        Syn_ List.push_ Back (synArg_mangToOneUint_Out);
        Node_ ManyToOneUint_ Out.push_ Back (synArg_mangToOneUint_Out);
        Many_ To_ One.next_ Node (node_ManyToOneUint_out);
        //模型集合器加入视频处理流水线中
        FrameProcessUnit * many2one_ Unit=&many_ To_ One;
        Video_ Process_ Framework.add_ Task (many2one_unit);
        Node_ VideoWriteUnit_ In.push_ Back (synArg_mangToOneUint_Out);
//Configure the entry node of the video encoding task
Video_ Writer_ Img_ Unit.pre_ Node (node_videoWriteUnit_in);
FrameProcessUnit * video_ Writer_ Img_ Process_ Unit=&video_ Writer_ Img_ Unit;
//The video coding task is added to the video processing pipeline
Video_ Process_ Framework.add_ Task (video_writer_img_process_unit);
//Start the task and wait until all tasks are finished.
Video_ Process_ Framework.start_ Tasks ();
Video_ Process_ Framework.close_ Tasks ();

}
```

附 录 C

Appendix C (Informative) Detailed description of error code returned by the function

C. 1 Return error code enumeration

The error codes returned by the algorithm service and algorithm task interface functions are listed in the form of enumeration. The details are as follows.

```
Enum mmsdkErrorSts
{
//AVfile
MmsdkErrorAVOpenInputFileFailed=-1000,
MmsdkErrorAVOpenOutputFileFailed,
MmsdkErrorAVInputFileReadFinshed,
MmsdkErrorAVerrorEOF,
MmsdkErrorAVFileHeaderWriteFailed,
MmsdkErrorAVFileTrailerWriteFailed,

//Reader
MmsdkErrorAVVideofillArrayFailed,
//Fmttxt
MmsdkErrorAVfmtWrong,
//DecoderTxt.encoderTxt
MmsdkErrorAVAudioCodecTxtParameterCopyFailed,
MmsdkErrorAVAudioCodecTxtAllocateFailed,
MmsdkErrorAVVideoCodecTxtParameterCopyFailed,
MmsdkErrorAVVideoCodecTxtAllocateFailed,
//Decoder
MmsdkErrorAVAudioDecoderOpenFailed,
MmsdkErrorAVAudioDecoderFindFailed,
MmsdkErrorAVVideoDecoderOpenFailed,
MmsdkErrorAVVideoDecoderFindFailed,
//Encoder
MmsdkErrorAVAudioEncoderFindFailed,
MmsdkErrorAVAudioEncoderOpenFailed,
MmsdkErrorAVVideoEncoderFindFailed,
MmsdkErrorAVVideoEncoderOpenFailed,
//Stream
MmsdkErrorAVAudioStreamCreateFailed,
MmsdkErrorAVAudioStreamInfoFailed,
MmsdkErrorAVAudioStreamAllocateFailed,
MmsdkErrorAVVideoStreamCreateFailed,
MmsdkErrorAVVideoStreamInfoFailed,
MmsdkErrorAVVideoStreamAllocateFailed,
```

```
//Swr
MmsdkErrorAVAudioSwrCtxCalloFailed,
MmsdkErrorAVAudioSwrCtxAllocFailed,
MmsdkErrorAVAudioSwrCtxInitFailed,
MmsdkErrorAVAudioSamplesSwrConvertFailed,
MmsdkErrorAVAudioSwrConvertedSamplesPointerCalloFailed,
MmsdkErrorAVAudioSwrConvertedSamplesAllocateFailed,
MmsdkErrorAVVideoSwscaleFailed,
//Packet
MmsdkErrorAVAudioPacketReadFailed,
MmsdkErrorAVAudioPacketSendFailed,
MmsdkErrorAVVideoPacketReadFailed,
MmsdkErrorAVVideoPacketReceiveFailed,
MmsdkErrorAVAudioPacketReceiveFailed,
MmsdkErrorAVVideoPacketSendFailed,
MmsdkErrorAVAudioAVPacketRescaleTsFailed,
MmsdkErrorAVVideoAVPacketRescaleTsFailed,
MmsdkErrorAVVideoPacketInterLeavedWriteFailed,
MmsdkErrorAVAudioPacketInterLeavedWriteFailed,
//Frame
MmsdkErrorAVAudioReceiveFrameFailed,
MmsdkErrorAVVideoReceiveFrameFailed,
MmsdkErrorAVAudioFrameSendFailed,
MmsdkErrorAVVideoFrameSendFailed,
MmsdkErrorVideoFrameSwsScaleFailed,
MmsdkErrorAVAudioFrameAllocateFailed,
MmsdkErrorAVVideoFrameAllocateFailed,
MmsdkErrorAVAudioFrameGetBufferFailed,
MmsdkErrorAVVideoPtsWorng,
//Audiofifo
MmsdkErrorAVFifoAllocateFailed,
MmsdkErrorAVFifoReAllocateFailed,
MmsdkErrorAVFifoReadFailed,
MmsdkErrorAVFifoWriteFailed,
//Flush
MmsdkErrorAVFrameFlushFailed,
MmsdkErrorAVPacketFlushFailed,
//CUDA
MmsdkErrorCUDADeviceFailed=-800,
MmsdkErrorCUDAMallocFailed,
MmsdkErrorCUDAMemcpyDeviceToHostFailed,
MmsdkErrorCUDAMemcpyHostToDeviceFailed,
MmsdkErrorCUDAMemcpyDeviceToDeviceFailed,
//Filter Unit
```

```
MmsdkErrorUnitWaitWorng=-700,
MmsdkErrorUnitPostWorng,
MmsdkErrorParaAdditemWorng,
MmsdkErrorUnitVideoReadExit,
MmsdkErrorUnitImgWriterExit,
MmsdkErrorUnitAudioWriterExit,
MmsdkErrorUnitTrtModelExit,
MmsdkErrorUnitOneToMannyExit,
MmsdkErrorUnitMannyToOneExit,
MmsdkErrorUnitHostToDeviceExit,
MmsdkErrorUnitDeviceToHostExit,
MmsdkSuccess=0
};
```

C.1 Error code description function

Function prototype:

Char * mmsdkErrorToStr (mmsdkErrorSts threadError);

See Table C.1 for function call parameters.

Table C.1 Error Code Description Function Call Parameters

parameter	data type	Input/output type	Parameter Description
ThreadErr	MmsdkErrorSts	input	Function returns error code enumeration number

附 录 D

Appendix D (Informative) Detailed description of dependent database version

D.1 Dependency Library Version Description

See Table D.1 for the detailed description of the third-party dependency database version

Table D.1 Version Description of the Third Party Dependency Database

Dependent library name	Version No
Ffmpeg	4.3.2
Cuda	11.2
Cudnn	8.0