Rockchip Linux Rockit Development Guide

File identification: RK-KF-YF-532

Release version: V0.7.0

Date: 2020-09-15

File Confidentiality: □ Top Secret □ Secret □ Internal Information ■ Public

Rockchip Microelectronics Co., Ltd.

Rockchip Electronics Co., Ltd.

Address: No. 18, Area A, Software Park, Tongpan Road, Fuzhou City, Fujian Province

Website: www.rock-chips.com

Customer Service Tel: +86-4007-700-590

Customer Service Fax: +86-591-83951833

Customer Service Email: fae@rock-chips.com

Preface

Rockit is positioned as a general media pipeline, plug-in common media components, and build a flexible application pipeline in a modular way. Developer
With the help of Rockit, rich media applications can be developed.

Overview

This article mainly describes Rockit media development reference. Mainly introduce the application development interface of Rockit; the topological relationship of the media pipeline; the media insert The types and parameters of the software; the development of custom plug-ins, etc.

product version

Chip name	Kernel version
RV1126/RV1109	Linux 4.19

Audience

This document (this guide) is mainly applicable to the following engineers:

Technical Support Engineer

Software Development Engineer

Revision record

version number	Author	Modified date	Modify the description
V0.6.1	Cheng Ming Chuan	2020-09-11	Rockit features and interfaces
V0.6.2	Xu Liming	2020-09-11	How to develop rockit plugins and applications
V0.7.0	Cheng Ming Chuan	2020-09-15	Improve rockit application development interface

table of Contents

1. **rockit** properties

The rockit framework has the following characteristics:

- Stable operation interface abstraction.
- Stable media interface abstraction. Convert the platform media interface to a universal media interface.
- Stable plugin abstraction.
- Support general plug-in management (TaskGraph). Plug-in assembly, data transfer and control, etc.
- Support the development of multiple media applications.

Media plugins currently supported by the rockit framework:

The current stable plug-ins are shown in the following table, and the macro definition of the plug-in name is shown in RTNodeCommon.h.

Special note : There are many examples of plug-in parameter configuration, please refer to the configuration in the SDK

File name: ${rv1126_sdk}/external/rockit/sdk/conf/aicamera_rockx.json

File name: ${rv1126_sdk}/external/rockit/sdk/conf/aicamera_stasteria.json

Types of	Plug-in name	Plug-in role	Input type	Output type
Device	rkisp	RK-ISP	N/A	multi(NV12 etc)
Device	alsa_capture	RK-alsa	N/A	PCM
Device	alsa_playback	RK-alsa	PCM	N/A
Codec	rkmpp_dec	RK-MPP encoding Device	NV12	H264/H265/MJPEG
		RK-MPP decoding		

Codec	rkmpp_enc	Device	H264/H265/MJPEG	NV12
Filter	alg_3a / alg_anr	RK-Audio3A-Old	PCM	PCM
Filter	skv_aec / skv_agc / skv_bf	RK-Audio3A-Skv	PCM	PCM
Filter	resample	Audio-Resample	PCM	PCM
Filter	rkrga	RK-RGA 2D plus speed	multi	multi
Filter	rockx	RK-NPU AI plus speed	multi (preferably NV12)	RTResult
Filter	st_asteria	ST Asteria AI accelerate	multi (preferably NV12)	STResult
Filter	rkeptz	EPTZ composite plug-in	multi (preferably NV12)	RTResult

Page 5

2. Development interface of **rockit** application

2.1 The interface of the plug-in **(RTTaskNode)**

Scope of application: software engineers develop plug-ins
Best practice: Only plug-ins that are not supported by the framework need to be developed. Try not to develop plug-ins by yourself.
Related file: RTTaskNode.h

The base class of the plug-in is RTTaskNode. After encapsulating the functional module (such as ISP/RGA/MPP/Rockx) into the plug-in (RTTaskNode), the general TaskGraph can flexibly call these plug-ins and build a variety of application scenarios according to configuration files. In plug-in development, we need to understand Under the interface.

```
// Input parameters: context: node context, used to store various parameters required for node initialization.
// Output parameters: RT_RET: RT_OK means successful execution, others means failure. After failure, the node cannot be created successfully.
// Function: complete the initialization of the node.
virtual RT_RET RTTaskNode::open (RTTaskNodeContext * context);

// Input parameters: context: node context, used to store various parameters and input/output data required for node processing.
// Output parameters: RT_RET: RT_OK means successful execution, others means failure.
// Function: complete node data processing. RT_RET: RT_OK means successful execution, others means failure.
virtual RT_RET RTTaskNode::process (RTTaskNodeContext * context);

// Input parameters: context: node context, used to store various parameters required for node deinitialization
// Output parameters: RT_RET: RT_OK means successful execution, others means failure.
// Function: complete node de-initialization
virtual RT_RET RTTaskNode::close (RTTaskNodeContext * context);

// Input parameters: meta: user configuration parameters
// Output parameters: RT_RET: RT_OK means successful execution, others means failure.
// Function: user configures node parameters
virtual RT_RET RTTaskNode::invoke (RtMetaData * meta);
```

The classes related to RTTaskNode include: InputStreamManager, OutputStreamManager, InputStreamHandler, OutputStreamHandler and RTTaskNodeContext.

2.2 Task Map **(TaskGraph)** Interface

Scope of application: software engineer development and application
Best practice: Use the TaskGraph interface to develop applications and avoid using the plug-in layer interface to manage plug-ins and data flows by yourself.
Related file: RTTaskGraph.h

The base class of task graph is RTTaskGraph. The main functions of the task graph (RTTaskGraph) are as follows:

Manage the configuration file and initialization of the plug-in system.

Manage the data flow of the plug-in system, including: data input, data output and data transfer.
Manage the control flow of the plug-in system, including: init/prepare/start/stop, etc.
Manage the task execution of the plug-in system.

The principle of the operating mechanism of the plug-in is as follows: Scheduler extracts atomic tasks from TaskNode and submits the tasks to SchedulerQueue for execution
The executor (Executor) removes the atomic task from the SchedulerQueue and dispatches it to a specific executor (such as: ThreadPool). The execution of plug-in tasks can be
It can be regarded as a black box; don't care about the running process, just pay attention to the input and output of RTTaskGraph.

Page 6

The task graph (RTTaskGraph) is a class for application development. In application development, we need to understand the following interfaces. .

```
// Input parameters: configFile: json configuration file
// Output parameters: RT_RET: RT_OK means successful execution, others means failure.
// Function: Generate TaskGraph according to user configuration
RT_RET RTTaskGraph::autoBuild ( const char* configFile);

// Input parameters: cmd: see enumeration definition; params: additional parameters of cmd.
// Output parameters: RT_RET: RT_OK means successful execution, others means failure.
// Function: Configure node parameters according to user configuration
RT_RET RTTaskGraph::invoke (INT32 cmd, RtMetaData * params);

// Input parameter: NONE
// Output parameters: RT_RET: RT_OK means successful execution, others means failure.
// Function function: wait for the end of the run
RT_RET RTTaskGraph::waitUntilDone ();

// Input parameter: NONE
// Output parameters: RT_RET: RT_OK means successful execution, others means failure.
// Function: Waiting for the observer's output.
RT_RET RTTaskGraphImpl::waitForObservedOutput ();

// Input parameters: streamName, the name of the stream; streamId, the ID number of the stream; streamCallback, the callback of the stream.
// Output parameter: NONE
// Function: Observe the specified output stream
RT_RET observeOutputStream ( const std::string & streamName,
                             INT32 streamId,
                             std::function < RT_RET(RTMediaBuffer * ) > streamCallback);

// Input parameters: streamId, the ID number of the stream.
// Output parameters: RT_RET: RT_OK means successful execution, others means failure.
// Function: Observe the specified output stream
RT_RET cancelObserveOutputStream (INT32 streamId);
```

The classes related to RTTaskGraph include: TaskNode, Scheduler, SchedulerQueue, Executor, ThreadPool. Above this
These classes are internal classes and don't care about developing rockit applications.

2.3 Control operation of **RTTaskGraph**

After RTTaskGraph reads the plug-in configuration file, it will automatically build the Pipeline and then work automatically. It can be done through a few interfaces
Cooperative pipeline processing of complex data streams. Application developers only need to pay attention to plug-in configuration and RTTaskGraph to develop complex applications.

```
RTTaskGraph * graph = new RTTaskGraph();
graph -> autoBuild( "your_graph.json" );
// Prepare the pipeline, including: plug-in preparation, link input and output, etc.
graph -> invoke(GRAPH_CMD_PREPARE, NULL);
// Start the pipeline, open up the entire data flow
graph -> invoke(GRAPH_CMD_START, NULL);
// Wait for the end of the pipeline
graph -> waitUntilDone();
// stop the pipeline
graph -> invoke(GRAPH_CMD_STOP, NULL);

// Set the parameters of pipleline
```

Page 7

```
// Configure the parameters of the framework: GRAPH_CMD_PRIVATE_CMD
// Configure the parameters of the node: GRAPH_CMD_TASK_NODE_PRIVATE_CMD,)
```

```
RtMetaData params;
params.setInt32(kKeyTaskNodeId, nodeId);
params.setCString(kKeyPipeInvokeCmd, "qp-control" );
params.setInt32( "qp_init" , 24 );
params.setInt32( "qp_step" , 8 );
params.setInt32( "qp_min" , 4 );
params.setInt32( "qp_max" , 24 );
params.setInt32( "min_i_qp" , 4 );
params.setInt32( "max_i_qp" , 24 );
graph -> invoke(GRAPH_CMD_TASK_NODE_PRIVATE_CMD, & params);
```

2.4 Interface for monitoring **TaskGraph** data output

Application developers need to pay attention to the input and output of RTTaskGraph. Input/data source plugin for RTTaskGraph (NodeIsp/NodeDemuxer) is self-driven input (lower-level consumption data), just pay attention to the configuration of this type of plug-in. RTTaskGraph Sink plug-ins are self-driven output (consumption data at this level), just pay attention to the configuration of such plug-ins. Filter plugin for RTTaskGraph (AIVisionFilter) generally do intermediate processing, this type of plug-in is used as output, and the application developer needs to take away the plug-in data (AIVisionData) Do additional post-processing.

```
// Customized OBSERVER post-processing functions are generally used for the application program and ROCKIT output data docking.
RT_RET YOUR_OBSERVER_FUNC (RTMediaBuffer * buffer) {
        RTRknnAnalysisResults * nnResult = NULL;
        buffer -> getMetaData() -> findPointer(ROCKX_OUT_RESULT,
                                                reinterpret_cast < RT_PTR *> ( & nnResult));
        if (nnResult -> counter > 0 ) {
#ifdef HAVE_ROCKX
                INT32 rawX = nnResult -> results[ 0 ].face_info.object.box.left;
                INT32 rawY = nnResult -> results[ 0 ].face_info.object.box.top;
#endif
        }
        buffer -> release();
        return RT_OK;
}

RTTaskGraph * graph = new RTTaskGraph();
graph -> autoBuild( "your_ai_vision.json" );
// Prepare the pipeline, including: plug-in preparation, link input and output, etc.
graph -> invoke(GRAPH_CMD_PREPARE, NULL);
// Observe the output of the pipeline
graph -> observeOutputStream( "ai_rockx" , ${stream_id} << 16 , YOUR_OBSERVER_FUNC);
// Start the pipeline, open up the entire data flow
graph -> invoke(GRAPH_CMD_START, NULL);
// Other operations...
```

2.5 Processing flow when the key data flow changes

2.5.1 Resolution changes

@TODO

2.5.2 Data format changes

@TODO

2.6 Description of core plug-in parameters

Special note : There are many examples of plug-in parameter configuration, please refer to the configuration in the SDK

File name: ${rv1126_sdk}/external/rockit/sdk/conf/aicamera_rockx.json

File name: ${rv1126_sdk}/external/rockit/sdk/conf/aicamera_stasteria.json

Plug-in **RKISP**

Refer to the configuration file in the above SDK.

Plug-in **RKRGA**

Refer to the configuration file in the above SDK.

Plug-in **RKROCKX**

Refer to the configuration file in the above SDK.

Plugin **RKMPP**

Refer to the configuration file in the above SDK.

Plug-in **UAC** related

Refer to the document: "Rockchip Linux UAC App Development Guide" in the linux SDK.

3. How to develop **rockit** plugin

The following uses a demo to introduce the structure of the graph configuration

```
// Create an external node, the external node needs to inherit RTTaskNode
// The basic interface needs to complete open/process/close
class RTRockitDemoNode : public RTTaskNode {
  public :
        RTRockitDemoNode() {}
        virtual ~RTRockitDemoNode() {}

        virtual RT_RET open(RTTaskNodeContext * context) { return RT_OK;}
        virtual RT_RET process(RTTaskNodeContext * context);
        virtual RT_RET close(RTTaskNodeContext * context) { return RT_OK;}
};

// Used for node creation, the function pointer will be stored in RTNodeStub.mCreateObj
```

Page 9

```
static RTTaskNode * createRockitDemoNode () {
        return new RTRockitDemoNode();
}

// Node processing function, the input data is processed, and then output to the lower node
RT_RET RTRockitDemoNode::process (RTTaskNodeContext * context) {
        RTMediaBuffer * inputBuffer = RT_NULL;
        RTMediaBuffer * outputBuffer = RT_NULL;

        // Determine whether the input is empty
        if (context -> inputIsEmpty()) {
                return RT_ERR_BAD;
        }

        // Take out the input buffer
        inputBuffer = context -> dequeInputBuffer();
        // Take out an unused output Buffer
        outputBuffer = context -> dequeOutpuBuffer(RT_TRUE, inputBuffer -> getLength());
        if (RT_NULL == outputBuffer) {
                inputBuffer -> release();
                return RT_ERR_BAD;
        }
        // Copy the input data to the output (demo is the simplest copy processing)
        rt_memcpy (outputBuffer -> the getData (), INPUTBUFFER -> the getData (), INPUTBUFFER -
> getLength());
        // Set the range of the output buffer
        outputBuffer -> setRange( 0 , inputBuffer -> getLength());
        // mark EOS
        if (inputBuffer -> isEOS()) {
                outputBuffer -> getMetaData() -> setInt32(kKeyFrameEOS, 1 );
        }
        // Input Buffer is used, call release
```

```
        inputBuffer -> release();
        // Bring out the output buffer and complete the processing flow
        context -> queueOutputBuffer(outputBuffer);

        return RT_OK;
}

//Node information stub, used to complete node registration
RTNodeStub node_stub_rockit_demo {
        // Node uid, the unique identifier of the node (0~1000)
        .mUid                   = kStubRockitDemo,
        // Node name, mainly used for node search and creation
        // corp_role_name, the name is guaranteed to be unique
        .mName                  = "rockit_demo" ,
        // version number
        .mVersion = "v1.0" ,
        // Node creation method; Change to macro definition.
        .mCreateObj = createRockitDemoNode,
        .mCapsSrc = { "video/x-raw" , RT_PAD_SRC, {RT_NULL, RT_NULL} },
        .mCapsSink = { "video/x-raw" , RT_PAD_SINK, {RT_NULL, RT_NULL} },
};

// Detect duplicate uuid and report an error
RT_NODE_FACTORY_REGISTER_STUB(node_stub_rockit_demo);
```


3.1 Node registration

A stub is required for node registration. The basic information of the stub is as follows:

```
//Node information stub, used to complete node registration
RTNodeStub node_stub_rockit_demo {
        // Node uid, the unique identifier of the node
        .mUid                   = kStubRockitDemo,
        // Node name, mainly used for node search and creation
        .mName                  = "rockit_demo" ,
        // version number
        .mVersion = "v1.0" ,
        // Node creation method
        .mCreateObj = createRockitDemoNode,
        .mCapsSrc = { "video/x-raw" , RT_PAD_SRC, {RT_NULL, RT_NULL} },
        .mCapsSink = { "video/x-raw" , RT_PAD_SINK, {RT_NULL, RT_NULL} },
};
```

Call the following function globally at the place where the stub is defined to complete the registration

```
RT_NODE_FACTORY_REGISTER_STUB(node_stub_rockit_demo);
```

3.2 The key function of the node

```
// Input parameters: context: node context, used to store various parameters required for node initialization.
// Output parameters: RT_RET: RT_OK means successful execution, others means failure. After failure, the node cannot be created successfully.
// Function: complete the initialization of the node.
RT_RET open (RTTaskNodeContext * context);

// Input parameters: context: node context, used to store various parameters and input/output data required for node processing.
// Output parameters: RT_RET: RT_OK means successful execution, others means failure.
// Function: complete node data processing. RT_RET: RT_OK means successful execution, others means failure.
RT_RET process (RTTaskNodeContext * context);

// Input parameters: context: node context, used to store various parameters required for node deinitialization
// Output parameters: RT_RET: RT_OK means successful execution, others means failure.
// Function: complete node de-initialization
RT_RET close (RTTaskNodeContext * context);
```

3.3 Context function description

RTTaskNodeContext stores the information needed for node initialization, processing, and de-initialization.

```
// Store node parameters. For example, coding nodes, it is possible to store width, height, bitrate and other information
RtMetaData * options ();

// Input parameter: streamType: Input the packet type. The default is none, which determines which input packet queue to fetch data from in the case of multiple inputs
// Output parameter: RTMediaBuffer: input package RTMediaBuffer.
// Function: Get a piece of input package
RTMediaBuffer * dequeInputBuffer (std::string streamType = "none" );
```

Page 11

```
// Input parameters:
// block: Whether to block. The default is blocking, waiting for an idle packet when blocking, and immediately returning to an idle output packet.
// size: The requested packet size, a buffer larger than size will be returned.
// streamType: output packet type. The default is none, which determines which output pool to take out the free buffer from in the case of multiple outputs
// Output parameters: RTMediaBuffer, idle output package RTMediaBuffer.
// Function: Get a free output package
RTMediaBuffer * dequeOutputBuffer (
                                    RT_BOOL block = RT_TRUE,
                                    UINT32 size = 0 ,
                                    std::string streamType = "none" );

// Input parameters:
// packet: output data generated by the process.
// streamType: output packet type. The default is none. In the case of multiple outputs, it determines which lower node the input flows to.
// Output parameters: RT_RET: RT_OK means success, others means failure.
// Function: store a piece of output packet, which will flow this piece of output packet to lower nodes
RT_RET queueOutputBuffer (RTMediaBuffer * packet, std::string streamType = "none" );

// Input parameters: streamType input type. The default is none, in the case of multiple inputs, determine which input queue is empty
// Output parameters: RT_BOOL, RT_TRUE is empty, RT_FALSE is not empty.
// Function function: Determine whether the input is empty
RT_BOOL inputIsEmpty (std::string streamType = "none" );

// Input parameters: streamType: output type. The default is none, in the case of multiple outputs, determine which output queue is empty
// Output parameters: RT_BOOL: RT_TRUE is empty, RT_FALSE is not empty.
// Function: Determine whether the output is empty
RT_BOOL outputIsEmpty (std::string streamType = "none" );
```

4. How to develop a **rockit** application

Currently we support automatic and manual construction of applications based on the ROCKIT framework. Automatically building a rockit application means using JSON configuration File, automatically build plug-ins and automatically build PIPELINE. Manually building a rockit application means that the developer manually builds the plug-in and builds it manually PIPELINE. Automatically build rockit applications only need to understand the configuration items and a few interfaces to develop applications, it is recommended to use automatic construction Develop applications in the way of rockit applications. To develop an application based on the ROCKIT framework, the main steps include:

- Evaluate whether ROCKIT's existing plug-ins meet the needs of the application and whether new plug-ins need to be developed or extended.
- Confirm the plug-in parameters, connection relationship and control relationship, and then configure the TaskGraph plug-in configuration according to the application requirements.
- The application loads the JSON configuration file to complete the data flow/control flow docking between the application and ROCKIT.
- Test verification of application business logic and application stability.

For application demo, see ${SDK_ROOT}/external/rockit

4.1 Automatically build **rockit** applications **(** recommended **)**

Currently we support automatic and manual construction of applications based on the ROCKIT framework. Automatically building a rockit application means using JSON configuration File, automatically build plug-ins and automatically build PIPELINE. Automatically build rockit application interface is more friendly, it is recommended for application engineers Ways to develop applications based on the ROCKIT framework.

Page 12

4.1.1 Diagram configuration file example

```
{
    // Configure a first-level directory, pipe_0 is a picture, currently it is not allowed to configure multiple pictures in a configuration, so here is usually
As "pipe_0"
    "pipe_0": {
        // Configure the secondary directory, configure information for the node
        "node_0": {
            // Configure node general information, currently only node name
            "node_opts": {
                "node_name": "rkisp"
            },
            // Configure node context information, such as data source, output buffer type, number, size, etc.
            "node_opts_extra": {
                "node_source_uri": "/dev/media1" ,
                "node_buff_type": 0 ,
                "node_buff_count": 4 ,
                "node_buff_size": 460800
            },
            // Configure general information about the data stream, such as type, name, etc.
            "stream_opts": {
                "stream_output": "image:nv12_0" ,
                "stream_fmt_out": "image:nv12"
            },
            // Configure the personalized information of the node, such as here will be configured in the rkisp node
            "stream_opts_extra": {
                "opt_entity_name": "rkispp_scale1" ,
                "opt_width": 640 ,
                "opt_height": 480 ,
                "opt_vir_width": 640 ,
                "opt_vir_height": 480 ,
                "opt_buf_type": 1 ,
                "opt_mem_type": 4 ,
                "opt_use_libv4l2": 1 ,
                "opt_colorspace": 0
            }
        },
        "node_1": {
            "node_opts": {
                "node_name": "rkmpp_enc"
            },
            "node_opts_extra": {
                "node_buff_type": 0 ,
                "node_buff_count": 4 ,
                "node_buff_size": 460800
            },
            "stream_opts": {
                "stream_input": "image:nv12_0" ,
                "stream_output": "image:h264_0" ,
                "stream_fmt_in": "image:nv12" ,
                "stream_fmt_out": "image:h264"
            },
            "stream_opts_extra": {
                "opt_width": 640 ,
                "opt_height": 480 ,
                "opt_vir_width": 680 ,
```

Page 13

```
                "opt_vir_height": 480 ,
                "opt_bitrate": 1000000 ,
                "opt_codec_type": 6 ,
                "opt_frame_rate": 30 ,
                "opt_profile": 100 ,
                "opt_level": 52 ,
                "opt_gop"                    : 30 ,
                "opt_qp_init": 24 ,
                "opt_qp_step": 4 ,
                "opt_qp_min": 12 ,
                "opt_qp_max": 48
            }
        },
        "node_2": {
            "node_opts": {
```

```
            "node_name": "fwrite"
        },
        "node_opts_extra": {
            "node_source_uri": "/data/test.yuv" ,
            "node_buff_type": 1 ,
            "node_buff_count": 0
        },
        "stream_opts": {
            "stream_input": "image:h264_0" ,
            "stream_fmt_in": "image:h264"
        }
    }
}
```

4.1.2 Picture configuration parameter list

Only general information is listed. For general information of non-graphs and nodes, please inquire inside the node code by yourself. Only indicates the configuration parameter information of the curre
But the future version is not limited to this. The macro definition is named as the macro definition in RTNodeCommon.h

parameter name	Macro definition name	Features	Yes no for must want letter interest	Remarks
node_name	OPT_NODE_NAME	node name	Yes	The system will find it by the node name Corresponding to the node, complete the construction.
node_source_uri	OPT_NODE_SOURCE_URI	data source	no	This value will be used inside the node to hit Open files, devices, etc. to read and write letters interest.
node_buff_type	OPT_NODE_BUFFER_TYPE	Output buffer Types of	no	When it is 0, it means that the buffer is in the Partial allocation; when it is 1, it means buffer Distributed by the node outside, the node only Provide RTMediaBuffer junction Structure.
node_buff_count	OPT_NODE_BUFFER_COUNT	Output buffer Number	Yes	0 means no allocation
node_buff_size	OPT_NODE_BUFFER_SIZE	Output buffer size	no	
				Used for link node data flow, need

stream_input	OPT_STREAM_INPUT_NAME	data Flow Become a name	no	Corresponding to the upper-level node stream_output. If in section There are multiple inputs inside the point, Please use stream_input_ + Index method to distinguish, such as stream_input_0.
stream_output	OPT_STREAM_OUTPUT_NAME	data Flow famous	no	Used for link node data flow, need Corresponding to the lower node stream_input. If in section There are multiple outputs inside the point, Please use stream_output_ + Index method to distinguish, such as stream_output_0.
stream_fmt_in	OPT_STREAM_FMT_IN	data Flow Into the class type	no	Define the data input type, this class The type is used to judge the number inside the node The type of data stream. Multi-input Situation needs to be paired with stream_input Should be, such as stream_fmt_in_0.

Page 15

parameter name	Macro definition name	Features	Yes no for must want letter interest	Remarks
stream_fmt_out	OPT_STREAM_FMT_OUT	data Flow Out of class type	no	Define the data output type, this class The type is used to judge the number inside the node The type of data stream. Multi-input Situation needs to be paired with stream_output Should, such as stream_fmt_out_0.

4.1.3 Automatically build application examples

Below we describe how to use the above configuration file to complete the automatic construction of the graph

```
#define RT_GRAPH_TRANSCODING_FILE
"/data/file_h264_rkmpp_dec_rkmpp_h265_enc_write.json"

RT_RET unit_test_graph_transcoding (INT32 index, INT32 total) {
        ( void )index;
        ( void )total;

        // Create a blank image
        RTTaskGraph * transcodingGraph = new RTTaskGraph( "UVCGraphTranscodingTest" );
        // Complete automated construction through configuration files
        transcodingGraph -> autoBuild(RT_GRAPH_TRANSCODING_FILE);
        // complete the preparation of the picture
        transcodingGraph -> invoke(GRAPH_CMD_PREPARE, NULL);
        // Startup diagram
        transcodingGraph -> invoke(GRAPH_CMD_START, NULL);
        // Wait for the graph to complete the work
        transcodingGraph -> waitUntilDone();
        // Destroy the image
        rt_safe_delete(transcodingGraph);

        return RT_OK;
}
```

4.2 Manually build a **rockit** application **(** not recommended **)**

Currently we support automatic and manual construction of applications based on the ROCKIT framework. Manually building a rockit application means that the developer manually builds Build a plug-in and manually build PIPELINE. Automatically build rockit applications only need to understand the configuration items and a few interfaces to develop applications, push It is recommended to use the method of automatically building rockit applications to develop applications.

Page 16

4.2.1 Manually build a **rockit** example

```
// Create a blank image
RTTaskGraph * demoGraph = new RTTaskGraph( "rockit_demo" );
// Complete node creation through configuration information
RT_NODE_CONFIG_STRING_APPEND(nodeConfig, XXX, XXX);
RT_NODE_CONFIG_STRING_APPEND(streamConfig, XXX, XXX);
RTTaskNode * freadNode = demoGraph -> createNode(nodeConfig, streamConfig);
XXX
.....
// Link node data flow
demoGraph -> addNodeLink( 3 , freadNode, demoNode, fwriteNode);
// Picture resource preparation
demoGraph -> invoke(GRAPH_CMD_PREPARE, NULL);
// The graph starts to work
demoGraph -> invoke(GRAPH_CMD_START, NULL);
// Wait for the completion of all work in the picture
demoGraph -> waitUntilDone();
// Destroy the image
rt_safe_delete(demoGraph);
```

The detailed steps are described below

4.2.2 Example of manually creating a plug-in

The following uses an example to introduce how to manually create a node. For the use of node parameters, please refer to "4.2 Automatically Building Rockit Applications use".

```
// Define node configuration and data flow configuration
std::string nodeConfig;
std::string streamConfig;
// RT_NODE_CONFIG_STRING_APPEND adds configuration information
RT_NODE_CONFIG_STRING_APPEND (nodeConfig, OPT_NODE_NAME,
  NODE_NAME_FREAD);
RT_NODE_CONFIG_STRING_APPEND (nodeConfig, OPT_NODE_SOURCE_URI,
  "/data/test.h264" );
RT_NODE_CONFIG_NUMBER_APPEND (nodeConfig, OPT_NODE_BUFFER_TYPE, 0 );
RT_NODE_CONFIG_NUMBER_APPEND (nodeConfig, OPT_NODE_BUFFER_COUNT, 4 );
RT_NODE_CONFIG_NUMBER_APPEND (nodeConfig, OPT_NODE_BUFFER_SIZE, 1024 *
1024 );
// RT_NODE_CONFIG_NUMBER_LAST_APPEND adds the last configuration information
RT_NODE_CONFIG_NUMBER_LAST_APPEND(nodeConfig, OPT_FILE_READ_SIZE, 1024 *
1024 );
RT_NODE_CONFIG_STRING_APPEND (streamConfig, OPT_STREAM_OUTPUT_NAME,
"fread_out" );
RT_NODE_CONFIG_STRING_LAST_APPEND(streamConfig, OPT_STREAM_FMT_OUT,
"image:h264" );
// Create node
RTTaskNode * freadNode = demoGraph -> createNode(nodeConfig, streamConfig);
```

In the above creation method, **OPT_NODE_NAME** is the node name, which is used to find the corresponding node from the **node factory** . You must Only correct to complete the creation.

Page 17

4.2.3 Example of manually linking plugins

There are two methods for linking nodes. The two methods are essentially the same, but the parameters are different.

```
// Input parameters:
// srcNode: The upstream node of the link, which provides the output in the link.
// dstNode: The downstream node of the link, which provides input in the link.
// Output parameter: RT_RET, return RT_OK if the link is successful, and return the return value of the specific reason if it fails.
// Function: complete the link of the node. After the link is completed, the output data completed by the upstream node process will flow to the input team of the downstream node Column.
//                 It should be noted that information such as node input and output types need to be matched to complete the link.
RT_RET linkNode (RTTaskNode * srcNode, RTTaskNode * dstNode);

// Input parameters:
// srcNodeId: The upstream node ID of the link.
// dstNodeId: The downstream node ID of the link.
// Output parameter: RT_RET, return RT_OK if the link is successful, and return the return value of the specific reason if it fails.
// Function: complete the link of the node. After the link is completed, the output data completed by the upstream node process will flow to the input team of the downstream node Column.
//                 It should be noted that information such as node input and output types need to be matched to complete the link.
RT_RET linkNode (INT32 srcNodeId, INT32 dstNodeId);
```

There are two ways to unlink a node. The two methods are essentially the same, but the parameters are different.

```
// Input parameters:
// srcNode: The upstream node of the link, which provides the output in the link.
// dstNode: The downstream node of the link, which provides input in the link.
// Output parameter: RT_RET, return RT_OK if the link is successful, and return the return value of the specific reason if it fails.
// Function: cancel the link relationship between plug-ins.
RT_RET unlinkNode (RTTaskNode * srcNode, RTTaskNode * dstNode);

// Input parameters:
// srcNodeId: The upstream node ID of the link.
// dstNodeId: The downstream node ID of the link.
// Output parameter: RT_RET, return RT_OK if the link is successful, and return the return value of the specific reason if it fails.
// Function: cancel the link relationship between plug-ins.
RT_RET unlinkNode (INT32 srcNodeId, INT32 dstNodeId);
```

5. How to extend existing applications

5.1 Extended **UVC (** Added **Video AI** plug-in **)**

Clarify UVC functional requirements

Evaluate which media plug-ins can be reused and which media plug-ins need to be developed.

Clarify the topology of the UVC pipeline plug-in and define the configuration file.

Added Video AI plug-in, implemented in the way of custom media plug-ins.

Page 18

5.2 Extend **UAC (** Add **Audio 3A** plug-in **)**

Clarify UVC functional requirements

Evaluate which media plug-ins can be reused and which media plug-ins need to be developed.

For the audio 3A plug-in of UAC, please refer to Rockchip Linux UAC related documents.

Clarify the topology of the UVC pipeline plug-in and define the configuration file.

Added Audio 3A plug-in, implemented in the way of custom media plug-ins.