Classification Level: Top Secret( ) Secret( ) Internal( ) Public(√)

RKNN API For RK356X User Guide

(Technology Department, Graphic Compute Platform Center)

Mark: [ ] Changing [√] Released	Version:	1.0.0
	Author:	HPC
	Completed Date:	30/Apr/2021
	Reviewer:	Vincent
	Reviewed Date:	30/Apr/2021

瑞芯微电子股份有限公司

Rockchip Electronics Co., Ltd.

Revision History

Version	Modifier	Date	Modify Description	Reviewer
v0.6.0	HPC	2/Mar/2021	Initial version	Vincent
v0.7.0	HPC	22/Apr/2021	Remove description of swapping input data channels	Vincent
v1.0.0	HPC	30/Apr/2021	Release version	Vincent

Table of Contents

1 Overview

RKNN SDK provides programming interfaces for RK3566/RK3568 chip platforms with NPU, which can help users deploy RKNN models exported using RKNN-Toolkit2 and accelerate the implementation of AI applications.

2 Supported hardware platforms

This document applies to the following hardware platforms:

RK3566, RK3568

Note: RK356X is used in some parts of the document to indicate RK3566/RK3568.

3 Instructions

3.1 RKNN SDK Development Process

Before using the RKNN SDK, users first need to use the RKNN-Toolkit2 tool to convert the user's model to the RKNN model.

After getting the RKNN model file, users can choose using C interface to develop the application. The following chapters will explain how to develop application based on the RKNN SDK on RK3566/RK3568 platform.

3.2 RKNN Linux Platform Development Instructions

3.2.1 RKNN API Library For Linux

For RK3566/ RK3568, the SDK library file is librknn_api.so under <sdk>/rknpu2/ directory.

3.2.2 Example Usage

The SDK provides MobileNet image classification, SSD object detection demos. These demos

provide reference for developer to develop applications based on the RKNN SDK. The demo code is located in the <sdk>/rknpu2/examples directory. Let's take rknn_mobilenet_demo as an example to explain how to get started quickly.

1) Compile Demo Source Code

```
cd examples/rknn_mobilenet_demo
# set GCC_COMPILER in build-linux.sh to the correct compiler path
./build-linux.sh
```

2) Deploy to the RK3566 or RK3568 device

```
adb push install/rknn_mobilenet_demo_Linux /userdata/
```

3) Run Demo

```
adb shell
cd /userdata/rknn_mobilenet_demo_Linux/
export LD_LIBRARY_PATH=./lib
./rknn_mobilenet_demo model/mobilenet_v1.rknn model/dog_224x224.jpg
```

3.3 RKNN Android platform development instructions

3.3.1 RKNN API Library For Android

There are two ways to call the RKNN API on the Android platform

1) The application directly links librknnrt.so

2) Application link to librknn_api_android.so implemented by HIDL on Android platform

For Android devices that need to pass the CTS/VTS test, you can use the RKNN API based on the Android platform HIDL implementation. If the device does not need to pass the CTS/VTS test, it is recommended to directly link and use librknnrt.so. The link for calling each interface is shorter, which can provide better performance.

The code for the RKNN API implemented using Android HIDL is located in the vendor/rockchip/hardware/interfaces/neuralnetworks directory of the RK356X Android system SDK.

When the Android system is compiled, some NPU-related libraries will be generated (for applications only need to link and use librknn_api_android.so), as shown below:

```
/system/lib/librknn_api_android.so
/system/lib/librknnhal_bridge.rockchip.so
/system/lib64/librknn_api_android.so
/system/lib64/librknnhal_bridge.rockchip.so
/vendor/lib64/rockchip.hardware.neuralnetworks@1.0.so
/vendor/lib64/rockchip.hardware.neuralnetworks@1.0-adapter-helper.so
/vendor/lib64/librknnrt.so
/vendor/lib64/hw/rockchip.hardware.neuralnetworks@1.0-impl.so
```

You can also use the following command to recompile and generate the above library separately.

```
mmm vendor/rockchip/hardware/interfaces/neuralnetworks/ -j8
```

For now, please make sure that the vendor/rockchip/hardware/interfaces/neuralnetworks git project directory includes the following commit:

```
commit 3df48976433d677edb3944132775450672c8d37c (HEAD)
Author: Huacong Yang <will.yang@rock-chips.com>
Date:   Thu Apr 29 15:35:25 2021 +0800

    neuralnetworks: update RKNN SDK 1.0

    Change-Id: I0c9f19461a4b356b0b0a81b307c4ad33bf61d4cf
    Signed-off-by: Huacong Yang <will.yang@rock-chips.com>
```

3.3.2 Example Usage

The SDK currently provides examples of MobileNet image classification and SSD target detection. The demo code is located in the <sdk>/rknpu2/examples directory. Users can use NDK to compile the demo executed in the Android command line. Let's take rknn_mobilenet_demo as an example to explain how to use this demo on the Android platform:

1) Compile Demo Source Code

```
cd examples/rknn_mobilenet_demo
#set ANDROID_NDK_PATH under build-android.sh to the correct NDK path
./build-android.sh
```

2) Deploy to the RK3566 or RK3568 device

```
adb push install/rknn_mobilenet_demo_Android /data/
```

3) Run Demo

```
adb shell
cd /data/rknn_mobilenet_demo_Andorid/
export LD_LIBRARY_PATH=./lib
./rknn_mobilenet_demo model/mobilenet_v1.rknn model/dog_224x224.jpg
```

3.4 RKNN C API

3.4.1 API process description

At present, there are two groups of APIs that can be used on RK356X, namely the general API interface and the API interface for the zero-copy process. The main difference between the two sets of APIs is that each time the general interface updates the frame data, the data allocated externally needs to be copied to the input memory of the NPU during runtime, while the interface of the zero-copy process creates a memory information structure.It is used by hardware unit such as NPU and RGA, which reduces the cost of copying memory. Users need to manage these memory information structures by themselves.

For the general API interface, initialize the *rknn_input* structure at first, the frame data is contained in the structure, use the *rknn_inputs_set* function to set the input of model. In the end of the inference, use the *rknn_outputs_get* function to get the inference output and perform post-processing. The frame data must be updated before inference. The general API call process is shown in Figure 3-1, and the process in yellow font indicates user behavior.

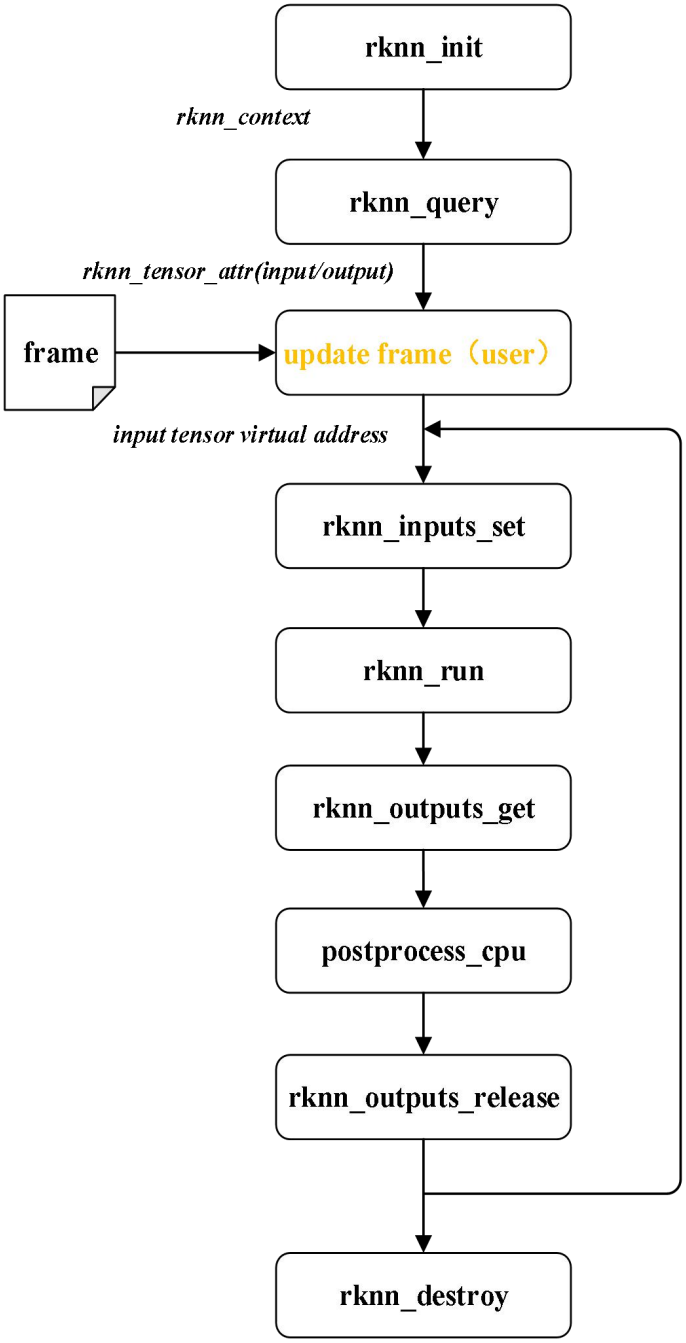

Figure 3-1 General API call process

For the zero-copy API interface, after allocating memory, use the memory information to initialize the *rknn_tensor_mem* structure, create and set the structure before inference, then read the memory information in the structure after inference. According to whether the user needs allocate the model's part memory (input/output/weights/intermediate result) and the differences of memory representation (file descriptor/physical address, etc.), there are the following three typical zero-copy calling processes, as shown in Figure 3-2, Figure 3-3 and Figure 3-4, the red font indicates the

interface and data structure specially added for zero copy, and the italic indicates the data structure passed between interface calls.

1) Input/output memory is allocated by runtime

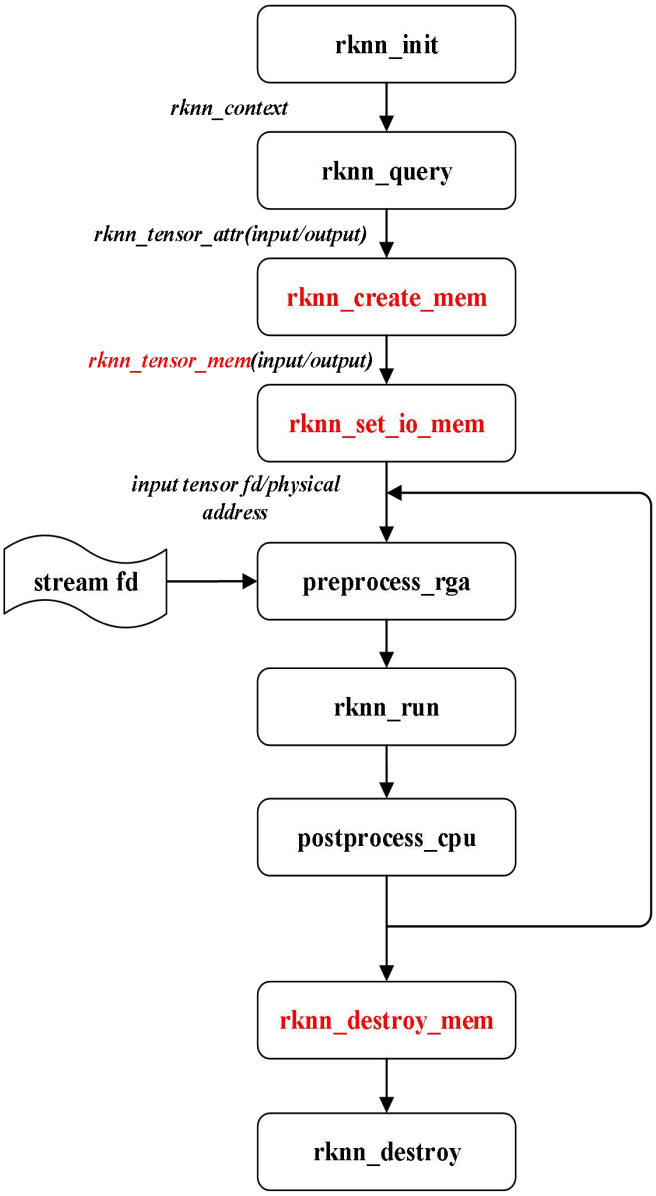

图 3-2 Zero-copy API interface call process (input/output allocated internally)

As shown in Figure 3-2, the input/output memory information structure created by the rknn_create_mem interface contains the file descriptor and physical addresses. The RGA interface uses the memory information allocated by the NPU during runtime, preprocess_rga represents the RGA interface, and stream_fd represents the input source buffer represented by fd in RGA, postprocess_cpu represents the CPU implementation of post-processing.

2) Input/output memory is allocated externally

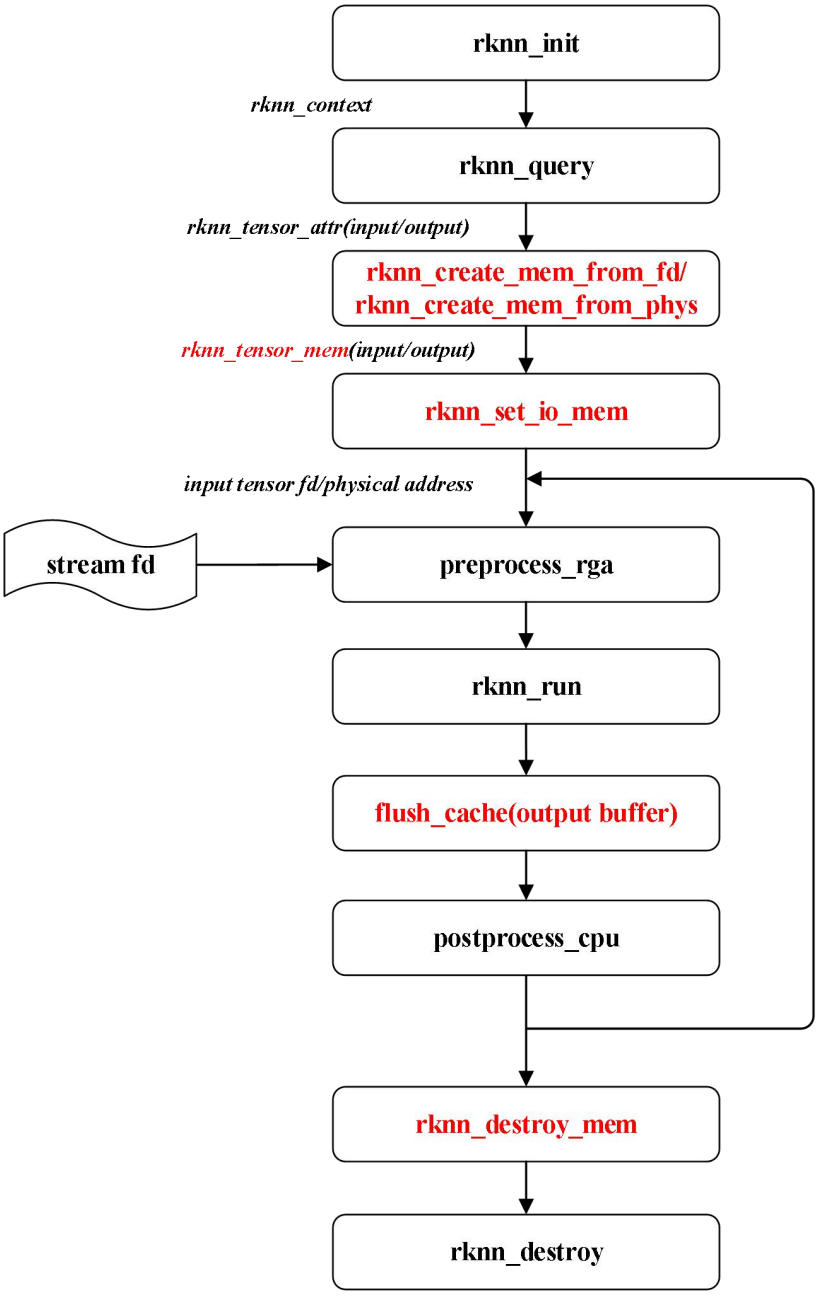

图 3-3 Zero-copy API interface call process (input/output allocated externally)

As shown in Figure 3-3, flush_cache indicates that the user needs to call the interface associated with the memory type to flush the output cache.

3) Input/output/weights/internal tensor memory is allocated externally

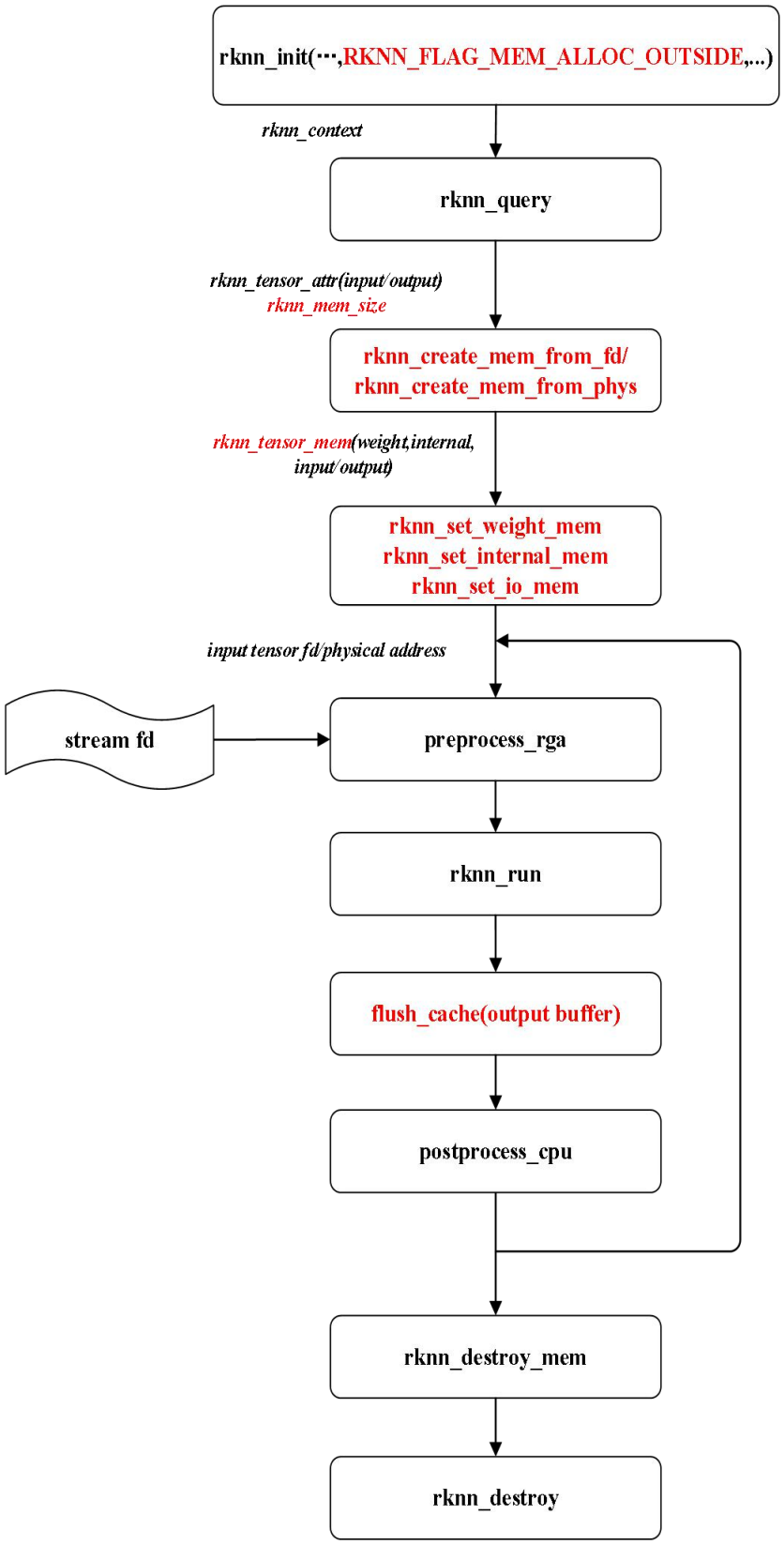

图 3-4 Zero-copy API interface call process (input/output/weights/internal tensor allocated externally)

3.4.1.1 API internal processing flow

When inferring the RKNN model, the original data has to go through three major processes: input processing, NPU running model, and output processing. Currently, according to different model input formats and quantization methods, there are two processing flows exist inside the general API interface as followed.

1) int8 quantized model and the number of input channels is 1 or 3 or 4

The processing flow of the original data is shown in Figure 3-5. Assuming that the input is a 3-channel model, the user must ensure that the color sequence of the R, G, and B channels is consistent with the training model. Normalization, quantization, and model inference are all running on the NPU, the output data layout format and dequantization process run on the CPU.

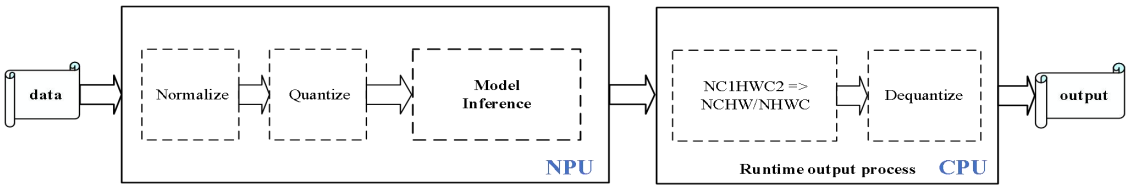

Figure 3-5 Optimized data processing flow

2) int8 quantized model which number of channels is 2 or >= 4 or non-quantized model

The flow of data processing is shown in Figure 3-6. The normalization, quantification, data layout format conversion, and dequantization of data all running on the CPU, and the inference of the model runs on the NPU. In this scenario, the processing efficiency of the input data flow will be lower than the optimized input data processing flow in Figure 3-5.

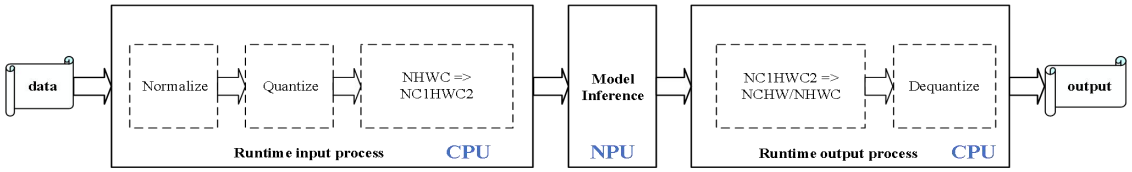

Figure 3-6 Ordinary data processing flow

For the zero-copy API, there is only one processing flow for the internal process of the API, as shown in Figure 3-5. Conditions for zero copy are as follow:

1. The number of input channels is 1 or 3 or 4.

2. The input width is 8 pixel aligned.

3. Int8 asymmetric quantized model.

3.4.1.2 Quantification and dequantization

The quantization method, quantization data type and quantization parameters used in quantization and dequantization can be queried through the rknn_query interface.

Currently, the NPU of RK3566/RK3568 only supports asymmetric quantization, and does not support dynamic fixed-point quantization. The combination of data type and quantization method includes:

- int8 (asymmetric quantization)
- int16 (asymmetric quantization, not yet implemented)
- float16

Normally, the normalized data is stored in 32-bit floating point data. For conversion of 32-bit floating point data to 16-bit floating point data, please refer to the IEEE-754 standard. Assuming that the normalized 32-bit floating point data is D, the following describes the quantization process:

1) float32 to int8(asymmetric quantization)

Assuming that the asymmetric quantization parameter of the input tensor is S_q, ZP, the data quantization process is expressed as the following formula:

$$D_q = round(clamp(D/S_q + ZP, -128, 127))$$

In the above formula, clamp means to limit the value to a certain range. round means rounding processing.

2) float32 to int16(asymmetric quantization)

Assuming that the asymmetric quantization parameter of the input tensor is S_q, ZP, the data quantization process is expressed as the following formula:

$$D_q = round(clamp(D / S_q + ZP, -32768, 32767))$$

The dequantization process is the inverse process of quantization, and the dequantization formula can be deduced according to the above quantization formula, which will not be repeated here.

3.4.2 API Reference

3.4.2.1 rknn_init

The rknn_init initialization function will create the *rknn_context* object, load the RKNN model, and perform specific initialization behaviors according to the flag and *rknn_init_extend* structure.

API	rknn_init
Description	Initialize rknn
Parameters	*rknn_context *context*: The pointer of *rknn_context* object. After the function is called, the context object will be assigned.
	*void *model*: Binary data for the RKNN model or the path of RKNN model.
	uint32_t size: When model is binary data, it indicates the size of the model. When model is a path, it is set to 0.
	uint32_t flag: A specific initialization flag.
	rknn_init_extend: The extended information during specific initialization. It is not used currently, just pass in NULL.
Return	int: Error code (See RKNN Error Code).

Sample Code:

```
rknn_context ctx;
int ret = rknn_init(&ctx, model_data, model_data_size, 0, NULL);
```

3.4.2.2 rknn_destroy

The rknn_destroy function will release the *rknn_context* object and its associated resources.

API	rknn_destroy
Description	Destroy the rknn_context object and its related resources
Parameters	*rknn_context context*: The rknn_context object to be destroyed.
Return	int: Error code (See RKNN Error Code).

Sample Code:

```
int ret = rknn_destroy (ctx);
```

3.4.2.3 rknn_query

The rknn_query function can query and obtain model input and output information, layer-by-layer running time, total model inference time, SDK version, memory usage information, user-defined strings and other information.

API	rknn_query
Description	Query the information about the model and the SDK.
Parameters	*rknn_context context*: The object of rknn_contex.
	rknn_query_cmd cmd: Query command.
	void info*: The structure object that stores the result of the query.
	uint32_t size: The size of the info structure object.
Return	int: Error code (See RKNN Error Code).

Currently, the SDK supports the following query commands:

Query command	Return result structure	Function
RKNN_QUERY_IN_OUT_NUM	rknn_input_output_num	Query the number of input and output Tensor.
RKNN_QUERY_INPUT_ATTR	rknn_tensor_attr	Query input Tensor attribute.
RKNN_QUERY_OUTPUT_ATTR	rknn_tensor_attr	Query output Tensor attribute.
RKNN_QUERY_PERF_DETAIL	rknn_perf_detail	Query the running time of each layer of the network.
RKNN_QUERY_PERF_RUN	rknn_perf_run	Query the total time of inference (excluding setting input/output) in microseconds.
RKNN_QUERY_SDK_VERSION	rknn_sdk_version	Query the SDK version.
RKNN_QUERY_MEM_SIZE	rknn_mem_size	Query the memory size allocated to the weights and internal tensors in the network.
RKNN_QUERY_CUSTOM_STRING	rknn_custom_string	Query the user-defined strings in the RKNN model.

Next we will explain each query command in detail.

1） Query the SDK version

The *RKNN_QUERY_SDK_VERSION* command can be used to query the version information of the RKNN SDK. You need to create the *rknn_sdk_version* structure object first.

Sample Code:

```
rknn_sdk_version version;
ret = rknn_query(ctx, RKNN_QUERY_SDK_VERSION, &version,
                 sizeof(rknn_sdk_version));
printf("sdk api version: %s\n", version.api_version);
printf("driver version: %s\n", version.drv_version);
```

2） Query the number of input and output Tensor

The *RKNN_QUERY_IN_OUT_NUM* command can be used to query the number of model

input and output Tensor. You need to create the *rknn_input_output_num* structure object first.

Sample Code:

```
rknn_input_output_num io_num;
ret = rknn_query(ctx, RKNN_QUERY_IN_OUT_NUM, &io_num,
                 sizeof(io_num));
printf("model input num: %d, output num: %d\n", io_num.n_input,
                 io_num.n_output);
```

3） Query input Tensor attribute

The *RKNN_QUERY_INPUT_ATTR* command can be used to query the attribute of the model input Tensor. You need to create the *rknn_tensor_attr* structure object first.

Sample Code:

```
rknn_tensor_attr input_attrs[io_num.n_input];
memset(input_attrs, 0, sizeof(input_attrs));
for (int i = 0; i < io_num.n_input; i++) {
    input_attrs[i].index = i;
    ret = rknn_query(ctx, RKNN_QUERY_INPUT_ATTR, &(input_attrs[i]),
                     sizeof(rknn_tensor_attr));
}
```

4） Query output Tensor attribute

The *RKNN_QUERY_OUTPUT_ATTR* command can be used to query the attribute of the model output Tensor. You need to create the *rknn_tensor_attr* structure object first.

Sample Code:

```
rknn_tensor_attr output_attrs[io_num.n_output];
memset(output_attrs, 0, sizeof(output_attrs));
for (int i = 0; i < io_num.n_output; i++) {
    output_attrs[i].index = i;
    ret = rknn_query(ctx, RKNN_QUERY_OUTPUT_ATTR, &(output_attrs[i]),
                     sizeof(rknn_tensor_attr));
}
```

5） Query layer-by-layer inference time of model

After the rknn_run interface is called, the *RKNN_QUERY_PERF_DETAIL* command can be used to query the layer-by-layer inference time in microseconds. The premise of using this command is that the flag parameter of the rknn_init interface needs to include the

RKNN_FLAG_COLLECT_PERF_MASK flag.

Sample Code:

```
rknn_context ctx;
int ret = rknn_init(&ctx, model_data, model_data_size,
                    RKNN_FLAG_COLLECT_PERF_MASK, NULL);
...
ret = rknn_run(ctx,NULL);
...
rknn_perf_detail perf_detail;
ret = rknn_query(ctx, RKNN_QUERY_PERF_DETAIL, &perf_detail,
                    sizeof(perf_detail));
```

6） Query total inference time of model

After the rknn_run interface is called, the *RKNN_QUERY_PERF_RUN* command can be used to query the inference time of the model (not including setting input/output) in microseconds.

Sample Code:

```
rknn_context ctx;
int ret = rknn_init(&ctx, model_data, model_data_size, 0, NULL);
...
ret = rknn_run(ctx,NULL);
...
rknn_perf_run perf_run;
ret = rknn_query(ctx, RKNN_QUERY_PERF_RUN, &perf_run,
                    sizeof(perf_run));
```

7） Query the memory allocation of the model

After the rknn_init interface is called, when the user needs to allocate some parts of network memory, the *RKNN_QUERY_MEM_SIZE* command can be used to query the weights of the model and the internal memory (excluding input and output) in the network. The premise of using this command is that the flag parameter of the rknn_init interface needs to include the RKNN_FLAG_MEM_ALLOC_OUTSIDE flag.

Sample Code:

```
rknn_context ctx;
int ret = rknn_init(&ctx, model_data, model_data_size,
                    RKNN_FLAG_MEM_ALLOC_OUTSIDE , NULL);
rknn_mem_size mem_size;
ret = rknn_query(ctx, RKNN_QUERY_MEM_SIZE, &mem_size,
                    sizeof(mem_size));
```

8） Query User-defined string in the model

After the rknn_init interface is called, if the user has added custom strings when generating the RKNN model, the *RKNN_QUERY_CUSTOM_STRING* command can be used to get user-defined strings.

Sample Code:

```
rknn_context ctx;
int ret = rknn_init(&ctx, model_data, model_data_size, 0, NULL);
rknn_custom_string custom_string;
ret = rknn_query(ctx, RKNN_QUERY_CUSTOM_STRING, &custom_string,
                    sizeof(custom_string));
```

3.4.2.4 rknn_inputs_set

The input data of the model can be set by the rknn_inputs_set function. This function can support multiple inputs, each one is a *rknn_input* structure object. The user needs to set these object field before passing in *rknn_inputs_set* function.

API	rknn_inputs_set
Description	Set the model input data.
Parameter	*rknn_context context*: The object of rknn_contex.
	uint32_t n_inputs: Number of inputs.
	rknn_input inputs[]: Array of rknn_input.
Return	int: Error code (See RKNN Error Code).

Sample Code:

```
rknn_input inputs[1];
memset(inputs, 0, sizeof(inputs));
inputs[0].index = 0;
inputs[0].type = RKNN_TENSOR_UINT8;
inputs[0].size = img_width*img_height*img_channels;
inputs[0].fmt = RKNN_TENSOR_NHWC;
inputs[0].buf = in_data;
inputs[0].pass_through = 0;

ret = rknn_inputs_set(ctx, 1, inputs);
```

3.4.2.5 rknn_run

The rknn_run function will perform a model inference. The input data need to be set by the *rknn_inputs_set* function or zero-copy interface before *rknn_run is* called.

API	rknn_run
Description	Perform a model inference.
Parameter	*rknn_context context*: The object of rknn_contex.
	rknn_run_extend extend*: Reserved for extension, it is not used currently, just pass in NULL.
Return	int: Error code (See RKNN Error Code).

Sample Code:

```
ret = rknn_run(ctx, NULL);
```

3.4.2.6 rknn_wait

This interface is used for non-blocking mode inference and has not been implemented yet.

3.4.2.7 rknn_outputs_get

The rknn_outputs_get function can get the output data of the model. This function can get multiple output data. Each of these outputs is a *rknn_output* structure object, which needs to be

created and set in turn before the function is called.

There are two ways to store buffers for output data:

1) The user allocate and release buffers themselves. At this time, the *rknn_output.is_prealloc* needs to be set to 1, and the *rknn_output.buf* points to users' allocated buffer;

2) The other is allocated by SDK. At this time, the *rknn_output .is_prealloc* needs to be set to 0. After the function is executed, *rknn_output.buf* will be created and store the output data.

API	rknn_outputs_get
Description	Get model inference output data.
Parameter	*rknn_context context*: The object of rknn_context.
	uint32_t n_outputs: Number of output.
	rknn_output outputs[]: Array of rknn_output.
	rknn_run_extend extend*: Reserved for extension, currently not used, you can pass NULL.
Return	int: Error code (See RKNN Error Code).

Sample Code:

```
rknn_output outputs[io_num.n_output];
memset(outputs, 0, sizeof(outputs));
for (int i = 0; i < io_num.n_output; i++) {
    outputs[i].index = i;
    outputs[i].is_prealloc = 0;
    outputs[i].want_float = 1;
}
ret = rknn_outputs_get(ctx, io_num.n_output, outputs, NULL);
```

3.4.2.8 rknn_outputs_release

The rknn_outputs_release function will release the relevant resources of the *rknn_output* object.

API	rknn_outputs_release
Description	Release the rknn_output object
Parameter	*rknn_context context*: rknn_context object
	uint32_t n_outputs: Number of output.
	rknn_output outputs[]: The array of rknn_output to be released.
Return	int: Error code (See RKNN Error Code).

Sample Code:

```
ret = rknn_outputs_release(ctx, io_num.n_output, outputs);
```

3.4.2.9 rknn_create_mem_from_mb_blk

It has not been implemented Currently .

3.4.2.10 rknn_create_mem_from_phys

When the user wants to allocate and manage memory by himself which can be used directly by the NPU, the rknn_create_mem_from_phys function a *rknn_tensor_mem* structure and return its pointer. This function passes in the physical address, logical address and size, and the information related to the external memory will be assigned to the *rknn_tensor_mem* structure.

API	rknn_create_mem_from_phys
Description	Create rknn_tensor_mem structure and allocate memory through physical address
Parameter	*rknn_context context*: rknn_context object.
	uint32_t phys_addr: The physical address of buffer.
	*void *virt_addr:* The virtual address of buffer.
	uint32_t size: The size of buffer.
Return	*rknn_tensor_mem*:* The tensor memory information structure pointer.

Sample Code:

```
//suppose we have got buffer information as input_phys, input_virt and size
rknn_tensor_mem* input_mems [1];
input_mems[0] = rknn_create_mem_from_phys(ctx, input_phys, input_virt,
                                          size);
```

3.4.2.11 rknn_create_mem_from_fd

When the user wants to allocate and manage memory by himself which can be used directly by the NPU, the rknn_create_mem_from_fd function a *rknn_tensor_mem* structure and return its pointer. This function passes in the file descriptor, logical address and size, and the information related to the external memory will be assigned to the *rknn_tensor_mem* structure.

API	rknn_create_mem_from_fd
Description	Create rknn_tensor_mem structure and allocate memory through file descriptor
Parameter	*rknn_context context*: rknn_context object.
	int32_t fd: The file descriptor of buffer.
	*void *virt_addr*: The virtual address of buffer, which indicates the beginning of fd.
	uint32_t size: The size of buffer.
	int32_t offset: The offset of buffer.
Return	rknn_tensor_mem*: The tensor memory information structure pointer.

Sample Code:

```
//suppose we have got buffer information as input_fd, input_virt and size
rknn_tensor_mem* input_mems [1];
input_mems[0] = rknn_create_mem_from_fd(ctx, input_fd, input_virt, size,0);
```

3.4.2.12 rknn_create_mem

When the user wants to allocate memory which can be used directly in the NPU, the rknn_create_mem function can create a *rknn_tensor_mem* structure and get its pointer. This function passes in the memory size and initializes the *rknn_tensor_mem* structure at runtime.

API	rknn_create_mem
Description	Create rknn_tensor_mem structure internally and allocate memory at runtime
Parameter	*rknn_context context*: rknn_context object.
	uint32_t size: The size of buffer.
Return	*rknn_tensor_mem**: The tensor memory information structure pointer.

Sample Code:

```
//suppose we have got buffer size
rknn_tensor_mem* input_mems [1];
input_mems[0] = rknn_create_mem(ctx, size);
```

3.4.2.13 rknn_destroy_mem

The rknn_destroy_mem function destroys the rknn_tensor_mem structure, and the memory allocated by the user needs to be released manually.

API	rknn_destroy_mem
Description	Destroy rknn_tensor_mem structure
Parameter	*rknn_context context*: rknn_context object.
	rknn_tensor_mem:* The tensor memory information structure pointer.
Return	int: Error code (See RKNN Error Code).

Sample Code:

```
rknn_tensor_mem* input_mems [1];
int ret = rknn_destroy_mem(ctx, input_mems[0]);
```

3.4.2.14 rknn_set_weight_mem

If the user has allocated memory for the network weights, after initializing the corresponding *rknn_tensor_mem* structure, the NPU can use the memory through the rknn_set_weight_mem function. This function must be called before calling rknn_run.

API	rknn_set_weight_mem
Description	Set up the rknn_tensor_mem structure containing weights memory information
Parameter	*rknn_context context*: rknn_context object.
	rknn_tensor_mem:* The tensor memory information structure pointer.
Return	int: Error code (See RKNN Error Code).

Sample Code:

```
rknn_tensor_mem* weight_mems [1];
int ret = rknn_set_weight_mem(ctx, weight_mems[0]);
```

3.4.2.15 rknn_set_internal_mem

If the user has allocated memory for the internal tensor in network, after initializing the corresponding rknn_tensor_mem structure, the NPU can use the memory through the rknn_set_internal_mem function. This function must be called before calling rknn_run.

API	rknn_set_internal_mem
Description	Set up the rknn_tensor_mem structure containing internal tensor memory information in network
Parameter	*rknn_context context*: rknn_context object.
	rknn_tensor_mem:* The tensor memory information structure pointer.
Return	int: Error code (See RKNN Error Code).

Sample Code:

```
rknn_tensor_mem* internal_tensor_mems [1];
int ret = rknn_set_internal_mem(ctx, internal_tensor_mems[0]);
```

3.4.2.16 rknn_set_io_mem

If the user has allocated memory for the input/output tensor in network, after initializing the corresponding *rknn_tensor_mem* structure, the NPU can use the memory through the

rknn_set_io_mem function. This function must be called before calling rknn_run.

API	rknn_set_io_mem
Description	Set up the rknn_tensor_mem structure containing input/output tensor memory information in network
Parameter	*rknn_context context*: rknn_context object.
	rknn_tensor_mem:* The tensor memory information structure pointer.
Return	int: Error code (See RKNN Error Code).

Sample Code:

```
rknn_tensor_mem* io_tensor_mems [1];
int ret = rknn_set_io_mem(ctx, io_tensor_mems[0]);
```

3.4.3 RKNN Data Structcture Definition

3.4.3.1 rknn_sdk_version

The structure *rknn_sdk_version* is used to indicate the version information of the RKNN SDK. The following table shows the definition:

Field	Type	Meaning
api_version	char[]	SDK API Version information.
drv_version	char[]	Driver version information.

3.4.3.2 rknn_input_output_num

The structure *rknn_input_output_num* represents the number of input and output Tensor， The following table shows the definition:

Field	Type	Meaning
n_input	uint32_t	The number of input tensor
n_output	uint32_t	The number of output tensor

3.4.3.3 rknn_tensor_attr

The structure *rknn_tensor_attr* represents the attribute of the model's Tensor. The following table shows the definition:

Field	Type	Meaning
index	uint32_t	Indicates the index position of the input and output Tensor.
n_dims	uint32_t	The number of Tensor dimensions.
dims	uint32_t[]	Values for each dimension.
name	char[]	Tensor name.
n_elems	uint32_t	The number of Tensor data elements.
size	uint32_t	The memory size of Tensor data.
fmt	rknn_tensor_format	The format of Tensor dimension, has the following format: **RKNN_TENSOR_NCHW** **RKNN_TENSOR_NHWC**
type	rknn_tensor_type	Tensor data type, has the following data types: **RKNN_TENSOR_FLOAT32** **RKNN_TENSOR_FLOAT16** **RKNN_TENSOR_INT8** **RKNN_TENSOR_UINT8** **RKNN_TENSOR_INT16**
qnt_type	rknn_tensor_qnt_type	Tensor Quantization Type, has the following types of quantization: **RKNN_TENSOR_QNT_NONE:** Not quantified; **RKNN_TENSOR_QNT_DFP**: Dynamic fixed

		point quantization; **RKNN_TENSOR_QNT_AFFINE_ASYMMETRIC**: Asymmetric quantification.
fl	int8_t	RKNN_TENSOR_QNT_DFP quantization parameter
zp	uint32_t	RKNN_TENSOR_QNT_AFFINE_ASYMMETRIC quantization parameter.
scale	float	RKNN_TENSOR_QNT_AFFINE_ASYMMETRIC quantization parameter.
stride	uint32_t	The number of pixels that actually store one line of image data, which is equal to the number of valid data pixels in one line + the number of invalid pixels that are filled in for the hardware to quickly jump to the next line.
size_with_stride	uint32_t	The actual byte size of image data (including the byte size of filled invalid pixels).
pass_through	uint8_t	0 means unconverted data, 1 means converted data, conversion includes normalization and quantization.

3.4.3.4 rknn_perf_detail

The structure rknn_perf_detail represents the performance details of the model. The definition of the structure is shown in the following table:

Field	Type	Meaning
perf_data	char*	The performance details include the running time of each layer of the network.
data_len	uint64_t	The data length of perf_data

3.4.3.5 rknn_perf_run

The structure rknn_perf_run represents the total inference time of the model. The definition of the structure is shown in the following table:

Field	Type	Meaning
run_duration	int64_t	The total inference time of the network (not including setting input/output) in microseconds.

3.4.3.6 rknn_mem_size

The structure *rknn_mem_size* represents the memory allocation when the model is initialized. The definition of the structure is shown in the following table:

Field	Type	Meaning
total_weight_size	uint32_t	The memory size allocated for weights of the network.
total_internal_size	uint32_t	The memory size allocated for internal tensor of the network.

3.4.3.7 rknn_tensor_mem

The structure *rknn_tensor_mem* represents the tensor memory information. The definition of the structure is shown in the following table:

Field	Type	Meaning
virt_addr	void*	The virtual address of tensor.
phys_addr	uint64_t	The physical address of tensor.
fd	int32_t	The file descriptor of tensor.
offset	int32_t	The offset of fd and virtual address.
size	uint32_t	The actual size of tensor.
flags	uint32_t	Reserved flag.
priv_data	void*	private data.

3.4.3.8 rknn_input

The structure *rknn_input* represents a data input to the model, used as a parameter to the rknn_inputs_set function. The following table shows the definition:

Field	Type	Meaning
index	uint32_t	The index position of this input.
buf	void*	The pointer of the input data buffer.
size	uint32_t	The memory size of the input data buffer.
pass_through	uint8_t	When set to 1, buf will be directly set to the input node of the model without any pre-processing.
type	rknn_tensor_type	The type of input data.
fmt	rknn_tensor_format	The format of input data.

3.4.3.9 rknn_output

The structure *rknn_output* represents a data output of the model, used as a parameter to the rknn_outputs_get function. The following table shows the definition:

Field	Type	Meaning
want_float	uint8_t	Indicates if the output data needs to be converted to float type.
is_prealloc	uint8_t	Indicates whether the Buffer that stores the output data is pre-allocated.
index	uint32_t	The index position of this output.
buf	void*	The pointer which point to the output data buffer.
size	uint32_t	Output data buffer size in byte.

3.4.3.10 rknn_init_extend

The structure *rknn_init_extend* represents the extended information when the runtime is initialized. **It is not supported currently**. The definition of the structure is shown in the following table:

Field	Type	Meaning
core_id	int32_t	The specific index of NPU core.
reserved	uint8_t[128]	The reserved data.

3.4.3.11 rknn_run_extend

The structure *rknn_run_extend* represents the extended information during model inference. **It is not supported currently**. The definition of the structure is shown in the following table:

Field	Type	Meaning
frame_id	uint64_t	The index of frame of current inference.
non_block	int32_t	0 means blocking mode, 1 means non-blocking mode, non-blocking means that the rknn_run call returns immediately.
timeout_ms	int32_t	The timeout of inference in milliseconds.

3.4.3.12 rknn_output_extend

The structure *rknn_output_extend* means to obtain the extended information of the output. **It is not supported currently**. The definition of the structure is shown in the following table:

Field	Type	Meaning
frame_id	int32_t	The frame index of the output result.

3.4.3.13 rknn_custom_string

The structure rknn_custom_string represents the custom string set by the user when converting the RKNN model. The definition of the structure is shown in the following table:

Field	Type	Meaning
string	char[]	User-defined string.

3.4.4 RKNN Error Code

The return code of the RKNN API function is defined as shown in the following table.

Error Code	Message
RKNN_SUCC（0）	Execution is successful
RKNN_ERR_FAIL (-1)	Execution error
RKNN_ERR_TIMEOUT (-2)	Execution timeout
RKNN_ERR_DEVICE_UNAVAILABLE (-3)	NPU device is unavailable
RKNN_ERR_MALLOC_FAIL (-4)	Memory allocation is failed
RKNN_ERR_PARAM_INVALID (-5)	Parameter error
RKNN_ERR_MODEL_INVALID (-6)	RKNN model is invalid
RKNN_ERR_CTX_INVALID (-7)	rknn_context is invalid
RKNN_ERR_INPUT_INVALID (-8)	rknn_input object is invalid
RKNN_ERR_OUTPUT_INVALID (-9)	rknn_output object is invalid
RKNN_ERR_DEVICE_UNMATCH (-10)	Version does not match
RKNN_ERR_INCOMPATILE_OPTIMIZATION_LEVEL_VERSION (-12)	This RKNN model use optimization level mode, but not compatible with current driver.
RKNN_ERR_TARGET_PLATFORM_UNMATCH (-13)	This RKNN model don’t compatible with current platform.