rocDecode documentation

Release 1.7.0

Advanced Micro Devices, Inc.

Apr 02, 2026

CONTENTS

1 What is rocDecode? 3

2 rocDecode prerequisites 5

3 Installing rocDecode with the package installer 7
- 3.1 Basic installation without software decoding 7
- 3.2 Basic installation with software decoding 7
- 3.3 Complete installation 8

4 Building and installing rocDecode from source code 9

5 Video decoding pipeline 11

6 rocDecode surface data memory locations 13

7 rocDecode samples 15

8 Understanding the rocDecode videodecode sample 17

9 Using the rocdecdecode example 23

10 Using the rocDecode RocVideoDecoder 29

11 Using the rocDecode FFmpeg demultiplexer 31

12 Using the rocDecode bitstream reader APIs 33

13 The rocDecode core APIs 35
- 13.1 The rocDecode parser API 35
- 13.2 The rocDecode hardware decoder API 36
- 13.3 The rocDecode software decoder API 37

14 rocDecode logging control 39

15 rocDecode supported codecs and hardware capabilities 41

16 File List 43

17 Globals 45

18 Data Structures 47

19 License 49

rocDecode provides APIs, utilities, and samples that you can use to easily access the video decoding features of your media engines (VCNs). It also allows interoperability with other compute engines on the GPU using Video Acceleration API (VA-API)/HIP. To learn more, see *What is rocDecode?*

The rocDecode public repository is located at https://github.com/ROCm/rocDecode.

Install

- *rocDecode prerequisites*
- *Installing rocDecode with the package installer*
- *Building and installing rocDecode from source code*
- rocDecode Docker containers

Conceptual

- *Video decoding pipeline*
- *rocDecode surface memory locations*

How to

- *Understand the rocDecode videodecode.cpp sample*
- *Understand the rocDecode rocdecdecode.cpp sample*
- *Use the rocDecode RocVideoDecoder*
- *Use the rocDecode FFmpeg demultiplexer*
- *Use the rocDecode bitstream reader APIs*

Samples

- *rocDecode samples*

Reference

- *The rocDecode core APIs*
  - *The rocDecode parser API*
  - *The rocDecode hardware decoder API*
  - *The rocDecode software decoder API*
- *rocDecode logging levels*
- *rocDecode codec support and hardware capabilities*
- *API library*
- *Functions*
- *Data structures*

To contribute to the documentation, refer to Contributing to ROCm.

You can find licensing information on the Licensing page.

CHAPTER

ONE

WHAT IS ROCDECODE?

AMD GPUs contain one or more media engines (VCNs) that provide fully accelerated, hardware-based video decoding. Hardware decoders consume lower power than CPU-based decoders. Dedicated hardware decoders offload decoding tasks from the CPU, boosting overall decoding throughput. With proper power management, decoding on hardware decoders can lower the overall system power consumption and improve decoding performance.

Using the rocDecode API, you can decode compressed video streams while keeping the resulting YUV frames in video memory. With decoded frames in video memory, you can run video post-processing using ROCm HIP, thereby avoiding unnecessary data copies via the PCIe bus. You can post-process video frames using scaling or color conversion and augmentation kernels (on a GPU or host) in a format for GPU/CPU-accelerated inferencing and training.

In addition, you can use the rocDecode API to create multiple instances of video decoders based on the number of available VCNs on a GPU device. By configuring the decoder for a device, all available VCNs can be used seamlessly to decode a batch of video streams in parallel.

For more information, refer to the *Video decoding pipeline*.

CHAPTER

TWO

ROCDECODE PREREQUISITES

rocDecode requires ROCm running on GPUs based on the CDNA architecture.

ROCm must be installed using the AMDGPU installer with the `rocm` usecase:

```
sudo amdgpu-install --usecase=rocm
```

rocDecode has been tested on the following Linux environments:

- Ubuntu 22.04 and 24.04
- RHEL 8 and 9
- SLES 15 SP7

See Supported operating systems for the complete list of ROCm supported Linux environments.

The following prerequisites are installed by the package installer. If you are building and installing using the source code, use the rocDecode-setup.py to install these prerequisites.

Note

To use the rocDecode samples, the `rocdecode`, `rocdecode-dev`, `rocdecode-host`, and `rocdecode-test` packages need to be installed.

If you're installing using the rocDecode source code, the `rocDecode-setup.py` script must be run with `--developer` set to `ON`.

- Libva-amdgpu-dev, an AMD implementation for Video Acceleration API (VA-API)
- AMD VA Drivers
- CMake version 3.10 or later
- AMD Clang++ Version 18.0.0 or later
- pkg-config
- FFmpeg runtime and headers
- libstdc++-12-dev for installations on Ubuntu 22.04
- HIP, specifically the `hip-dev` package

CHAPTER THREE

INSTALLING ROCDECODE WITH THE PACKAGE INSTALLER

Three rocDecode packages are available:

- `rocdecode`: The rocDecode runtime package. This is the basic rocDecode package. It must always be installed.
- `rocdecode-host`: The rocDecode software decoding package. This package must be installed to use the FFmpeg software decoder.
- `rocdecode-dev`: The rocDecode development package. This package installs a full suite of libraries, header files, and samples. This package must be installed to use the rocDecode samples.
- `rocdecode-test`: A test package that provides a CTest to verify the installation. This package must be installed to use the rocDecode samples.

All the required prerequisites are installed when the package installation method is used.

3.1 Basic installation without software decoding

Use the following commands to install only the rocDecode runtime package:

Ubuntu

```
sudo apt install rocdecode
```

RHEL

```
sudo yum install rocdecode
```

SLES

```
sudo zypper install rocdecode
```

3.2 Basic installation with software decoding

Use the following commands to install the rocDecode runtime package and the host decoding package:

Ubuntu

```
sudo apt install rocdecode rocdecode-host
```

RHEL

```
sudo yum install rocdecode rocdecode-host
```

SLES

```
sudo zypper install rocdecode rocdecode-host
```

3.3 Complete installation

Use the following commands to install `rocdecode`, `rocdecode-host`, `rocdecode-dev`, and `rocdecode-test`:

Ubuntu

```
sudo apt install rocdecode rocdecode-host rocdecode-dev rocdecode-test
```

RHEL

```
sudo yum install rocdecode rocdecode-host rocdecode-devel rocdecode-test
```

SLES

```
sudo zypper install rocdecode rocdecode-host rocdecode-devel rocdecode-test
```

CHAPTER

FOUR

BUILDING AND INSTALLING ROCDECODE FROM SOURCE CODE

If you will be contributing to the rocDecode code base, or if you want to preview new features, build rocDecode from its source code.

If you will not be previewing features or contributing to the code base, use the *package installers* to install rocDecode.

Before building rocDecode, use rocDecode-setup.py to install all the required prerequisites:

```
python3 rocDecode-setup.py  [--rocm_path ROCM_INSTALLATION_PATH; default=/opt/rocm]
                            [--runtime {ON|OFF}; default=ON]
                            [--developer {ON|OFF}; default=OFF]
```

> **Note**
>
> Never run `rocDecode-setup.py` with `--runtime OFF`.
>
> `--developer ON` is required to use the code samples.

Build and install rocDecode using the following commands:

```
git clone https://github.com/ROCm/rocDecode.git
cd rocDecode
mkdir build && cd build
cmake ../
make -j8
sudo make install
```

After installation, the rocDecode libraries will be copied to `/opt/rocm/lib` and the rocDecode header files will be copied to `/opt/rocm/include/rocdecode`.

Build and install the rocDecode test module. This module is required if you'll be using the rocDecode samples, and can only be installed if `rocDecode-setup.py` was run with `--developer ON`.

```
mkdir rocdecode-test && cd rocdecode-test
cmake /opt/rocm/share/rocdecode/test/
ctest -VV
```

Run `make test` to test your build. To run the test with the verbose option, run `make test ARGS="-VV"`.

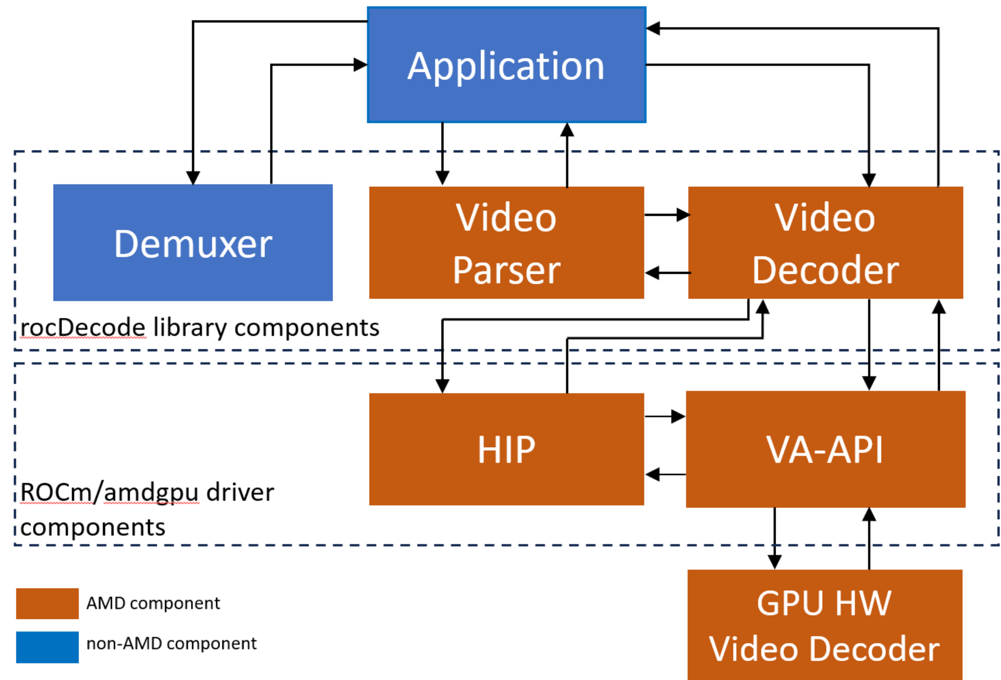
To create a package installer for rocDecode, run:

```
sudo make package
```

CHAPTER

FIVE

VIDEO DECODING PIPELINE



There are three main components in rocDecode:

- Demuxer: Our demuxer is based on FFmpeg, a leading multimedia framework. For more information, refer to the FFmpeg website.
- Video parser APIs
- Video decoder APIs

rocDecode follows this workflow:

1. The demuxer extracts a segment of video data and sends it to the video parser.
2. The video parser extracts crucial information, such as picture and slice parameters, and sends it to the decoder APIs.
3. The hardware receives the picture and slice parameters, then decodes a frame using Video Acceleration API (VA-API).
4. This process repeats in a loop until all frames have been decoded.

Steps in decoding video content for applications (available in the rocDecode Toolkit):

1. Demultiplex the content into elementary stream packets (FFmpeg)
2. Parse the demultiplexed packets into video frames for the decoder provided by rocDecode API.
3. Decode compressed video frames into YUV frames using rocDecode API.
4. Wait for the decoding to finish.
5. Get the decoded YUV frame from amd-gpu context to HIP (using VAAPI-HIP interoperability under ROCm).
6. Run HIP kernels in the mapped YUV frame. For example, format conversion, scaling, object detection, classification, and others.
7. Release the decoded frame.

Note

YUV is a color space that represents images using luminance (Y) for brightness and two chrominance components (U and V) for color information.

The preceding steps are demonstrated in the sample applications located in our GitHub repository directory.

CHAPTER

SIX

ROCDECODE SURFACE DATA MEMORY LOCATIONS

Surface data memory refers to the memory used by rocDecode for decoded frames and processing results. There are three locations where surface data memory can be stored: device memory, host memory, and internal memory.

Device memory refers to GPU memory. It's optimized for operations performed by the GPU, avoiding unnecessary memory transfers between the device and the host. It's used for standalone GPU processing and high-performance computing tasks where multiple operations are performed on the same data.

Host memory refers to CPU memory. It's suitable for when the memory needs to be accessed or manipulated by CPU-side applications or when data needs to be transferred between systems.

Internal memory refers to intermediate GPU memory that is shared between operators. It's optimized for operator chaining within GPU workflows. It keeps data localized on the GPU so it can be accessed by subsequent operations, reducing latency and improving throughput. For example, in image processing pipelines, the results of a resizing operator can directly feed into a filtering operator without needing to copy data to the host between each step. This optimization is especially useful for large datasets and real-time applications.

The `OutputSurfaceMemoryType_enum` enum type defines `OUT_SURFACE_MEM_DEV_COPIED`, `OUT_SURFACE_MEM_HOST_COPIED`, and `OUT_SURFACE_MEM_DEV_INTERNAL`, for the three different types of memory locations. `OUT_SURFACE_MEM_DEV_COPIED` indicates device, or GPU, memory. `OUT_SURFACE_MEM_HOST_COPIED` indicates host, or CPU, memory. And `OUT_SURFACE_MEM_DEV_INTERNAL` indicates intermediate GPU memory.

`OUT_SURFACE_MEM_DEV_COPIED` is not supported when the FFmpeg decoder is used.

A fourth enum, `OUT_SURFACE_MEM_NOT_MAPPED`, is used only for performance purposes. The decoded frames are not available when this memory type is used.

CHAPTER

SEVEN

ROCDECODE SAMPLES

rocDecode samples are available in the rocDecode GitHub repository.

You can find a walkthrough of the `videodecode.cpp` sample at *Understanding the videodecode.cpp sample*.

All rocDecode packages, `rocDecode`, `rocdecode-dev`, `rocdecode-host`, and `rocdecode-test`, must be installed to use the rocDecode samples.

If you're using a *package installer*, install `rocdecode`, `rocdecode-dev`, `rocdecode-host`, and `rocdecode-test`.

If you're building and installing rocDecode from its *source code*, `rocDecode-setup.py` needs to be run with `--developer` set to `ON`:

```
python3 rocDecode-setup.py --developer ON
```

The `rocDecode-test` package needs to be built and installed as well:

```
mkdir rocdecode-test && cd rocdecode-test
cmake /opt/rocm/share/rocdecode/test/
ctest -VV
```

CHAPTER

EIGHT

UNDERSTANDING THE ROCDECODE VIDEODECODE SAMPLE

The `videodecode.cpp` sample in the rocDecode GitHub repository `samples` folder demonstrates how to decode a video stream.

As with the other rocDecode samples, `videodecode.cpp` uses the utility classes in the rocDecode repository's `utils` folder.

rocDecode provides two ways to decode a video stream: using the rocDecode RocVideoDecoder on GPU or using the FFMpeg video decoder on CPU.

The `videodecode.cpp` sample lets the user choose which method to use through the `--backend` argument.

`videodecode.cpp` takes the following arguments:

Argument	Description	Note
-i	Input file path	Required. The path to the input video stream.
-o	Output file path	Optional. The file to which to write the decoded frames, including those that remain in the decoded frame buffer pool when the RocVideoDecoder is being reconfigured.
-d	GPU device ID	Optional. Set it to 0 for the first device, 1 for the second device, 2 for the third device, and so on for each subsequent device. Set to 0 by default.
-backend	The backend to use for decoding	Optional. Set it to 0 to use RocVideoDecode on GPU, 1 to use the FFMpeg decoder on CPU, or 2 to use the FFMpeg decoder with no multithreading on CPU. Uses RocVideoDecode on GPU by default.
-f	Number of frames to decode	Optional. Decodes the entire stream by default.
-z	Force zero latency	Optional. When set to `true` forces decoded frames to be flushed out for display immediately. `false` by default.
-disp_de	Display delay	Optional. The number of frames to decode before displaying the results. Set to 1 by default.
-sei	Extract Supplemental Enhancement Information (SEI)	Optional. Set to `true` to extract SEI. `false` by default.
-md5	Generate MD5 message digest	Optional. Set to `true` to generate the MD5 message digest for the decoded YUV image sequence. `false` by default.
-md5_che	Compare the generated MD5 with a provided MD5 string	Optional. When a file containing an MD5 string is passed to this argument, the MD5 message is compared to the string in the file.
-crop	Crop rectangle	Optional. Takes four integers defining the crop rectangle to use with the output. This argument is ignored when using interopped decoded frame. See the documentation for the Rect struct for more information. There is no cropping by default.
-m	The output surface memory type	Optional. The memory type where the surface data, such as the decoded frames, resides. Set this to 0 for intermediate GPU memory, to 1 for GPU memory, and to 2 for CPU memory. See *Surface data memory locations* for more information. Uses intermediate GPU memory by default.
-seek_cr	Seek criteria and seek starting point	Optional. Set to 1 and the frame number to start demultiplexing from that specific frame. Set to 2 and the timestamp to start demultiplexing from that specific timestamp. The seek criteria and starting point must be comma-separated (`,`). Demultiplexing begins at the first frame by default.
-seek_mo	Seek mode	Optional. Set to 0 to seek to the previous keyframe. Set to 1 to seek to the exact frame. Seeks to previous keyframe by default.
-no_ffmp	Don't use the FFMpeg demultiplexer	Optional. Set to `true` to use the RocDecode bitstream reader to obtain picture data. The bitstream reader can only be used with an elementary stream. The FFmpeg demultiplexer is used by default.

Because the `videodecode.cpp` example can use the RocDecode RocVideoDecoder, the FFMpeg decoder, the FFmpeg demultiplexer (demuxer), or the RocDecode bitstream reader, it imports the `roc_video_dec.h`, `video_demuxer.h`, and `ffmpeg_video_dec.h` header files. These headers contain the convenience classes and functions for decoding and demultiplexing video.

The FFMpeg demuxer is used to demultiplex the input stream unless the `-no_ffmpeg_demux` argument was set to `true`.

```
VideoDemuxer *demuxer;
```

(continues on next page)

(continued from previous page)

```
demuxer = new VideoDemuxer(input_file_path.c_str());
```

The GetCodecId and GetBitDepth functions are used to obtain the video stream's codec and bit depth. The AVCodec2RocDecVideoCodec utility function converts the codec returned from the demuxer to its corresponding rocDecVideoCodec_enum value.

```
rocdec_codec_id = AVCodec2RocDecVideoCodec(demuxer->GetCodecID());
bit_depth = demuxer->GetBitDepth();
```

The codec ID and bit depth are used to instantiate the video decoder. If the GPU backend was selected, the RocVideoDecoder is instantiated:

```
RocVideoDecoder *viddec;
viddec = new RocVideoDecoder(device_id, mem_type, rocdec_codec_id, b_force_zero_latency,␣
↪p_crop_rect, b_extract_sei_messages, disp_delay);
```

For more information about the rocDecode RocVideoDecoder, see *Using the rocDecode RocVideoDecoder*.

If the CPU backend was selected, the FFMpeg decoder is instantiated:

```
viddec = new FFMpegVideoDecoder(device_id, mem_type, rocdec_codec_id, b_force_zero_
↪latency, p_crop_rect, b_extract_sei_messages, disp_delay);
```

The decoder instance is reused when there is a change to the video resolution without a change in the codec. When the video stream resolution changes, the decoder is reconfigured for the new resolution and the pool of frame buffers that the decoder maintains is deleted.

The ReconfigParams_t struct is used to store information on how to handle the frames that remain in the buffers at the time of reconfiguration. A callback, a user-defined flush mode, and a user-defined struct are passed to ReconfigParams_t. The reconfiguration parameters are then passed to the decoder using SetReconfigParams.

The reconfiguration structs are defined in common.h in the rocDecode samples. Three possibilities for the remaining frames in the decoded frame buffer pool are provided:

- RECONFIG_FLUSH_MODE_NONE: delete the frames along with the buffers.
- RECONFIG_FLUSH_MODE_DUMP_TO_FILE: write the frames to the specified output file before deleting the buffers.
- RECONFIG_FLUSH_MODE_CALCULATE_MD5: calculate the MD5 of the frames before deleting the buffers.

```
typedef enum ReconfigFlushMode_enum {
  RECONFIG_FLUSH_MODE_NONE = 0x0,                    /**<  Just flush to get the frame␣
↪count */
  RECONFIG_FLUSH_MODE_DUMP_TO_FILE = 0x1,            /**<  The remaining frames will␣
↪be dumped to file in this mode */
  RECONFIG_FLUSH_MODE_CALCULATE_MD5 = (0x1 << 1),    /**<  Calculate the MD5 of the␣
↪flushed frames */
} ReconfigFlushMode;

typedef struct ReconfigDumpFileStruct_t {
  bool b_dump_frames_to_file;
  std::string output_file_name;
  void *md5_generator_handle;
} ReconfigDumpFileStruct;
```

If the -o output file path argument was set, the remaining frames in the decoded frame buffer pool will be written to the output file upon reconfiguration. If the -md5 argument was set to true, the MD5 of the frames in the decoded frame

buffer pool will be calculated before they're flushed or written to file. If neither option was selected, the frames in the decoded frame buffer pool will be deleted along with the buffers without being saved or processed.

```
reconfig_params.p_fn_reconfigure_flush = ReconfigureFlushCallback;
reconfig_user_struct.b_dump_frames_to_file = dump_output_frames;
reconfig_user_struct.output_file_name = output_file_path;
reconfig_params.reconfig_flush_mode = RECONFIG_FLUSH_MODE_NONE;
if (dump_output_frames) {
    reconfig_params.reconfig_flush_mode |= RECONFIG_FLUSH_MODE_DUMP_TO_FILE;
}
if (b_generate_md5) {
    reconfig_params.reconfig_flush_mode |= RECONFIG_FLUSH_MODE_CALCULATE_MD5;
}
reconfig_params.p_reconfig_user_struct = &reconfig_user_struct;
```

The reconfiguration parameters need to be defined prior to entering the decoding loop.

In the decode loop, the video stream is demultiplexed before being decoded.

The demuxer will demultiplex frames sequentially starting at the beginning of the stream unless `-seek_criteria` was set to either 1 or 2.

If the `-seek_criteria` argument was set to 1 and `-seek_mode` was set to 1, the demuxer will start demultiplexing the video at the frame provided.

If the `-seek_criteria` argument was set to 1 and `-seek_mode` wasn't set or was set to 0, the demuxer will start demultiplexing the video at the first keyframe before the frame provided.

If the `-seek_criteria` argument was set to 2 the demuxer will start demultiplexing the video at the timestamp provided.

The seek criteria is defined by the `SeekCriteriaEnum` enum and the seek mode is defined by the `SeekModeEnum` enum. Both the `SeekCriteriaEnum` and the `SeekModeEnum` are defined in `video_demuxer.h`.

From `videodecode.cpp`:

```
VideoSeekContext video_seek_ctx;
[...]
do {
  [...]
  if (seek_criteria == 1 && first_frame) {
    // use VideoSeekContext class to seek to given frame number
    video_seek_ctx.seek_frame_ = seek_to_frame;
    video_seek_ctx.seek_crit_ = SEEK_CRITERIA_FRAME_NUM;
    video_seek_ctx.seek_mode_ = (seek_mode ? SEEK_MODE_EXACT_FRAME : SEEK_MODE_PREV_KEY_
↪FRAME);
    demuxer->Seek(video_seek_ctx, &pvideo, &n_video_bytes);
    pts = video_seek_ctx.out_frame_pts_;
    std::cout << "info: Number of frames that were decoded during seek - " << video_seek_
↪ctx.num_frames_decoded_ << std::endl;
    first_frame = false;
  } else if (seek_criteria == 2 && first_frame) {
    // use VideoSeekContext class to seek to given timestamp
    video_seek_ctx.seek_frame_ = seek_to_frame;
    video_seek_ctx.seek_crit_ = SEEK_CRITERIA_TIME_STAMP;
    video_seek_ctx.seek_mode_ = (seek_mode ? SEEK_MODE_EXACT_FRAME : SEEK_MODE_PREV_KEY_
↪FRAME);
```

(continues on next page)

(continued from previous page)

```
    demuxer->Seek(video_seek_ctx, &pvideo, &n_video_bytes);
    pts = video_seek_ctx.out_frame_pts_;
    std::cout << "info: Duration of frame found after seek - " << video_seek_ctx.out_
↪frame_duration_ << " ms" << std::endl;
    first_frame = false;
  } else {
    demuxer->Demux(&pvideo, &n_video_bytes, &pts);
}
```

The video can now be decoded using the `DecodeFrame` function.

If the `-md5` argument was set to `true`, MD5 is calculated for the file. If an output file path was provided, the decoded frames will be written to file.

The frame is released with `ReleaseFrame` once processing is complete.

```
n_frame_returned = viddec->DecodeFrame(pvideo, n_video_bytes, pkg_flags, pts, &decoded_
↪pics);

[...]

for (int i = 0; i < n_frame_returned; i++) {
  pframe = viddec->GetFrame(&pts);
  if (b_generate_md5) {
    md5_generator->UpdateMd5ForFrame(pframe, surf_info);
  }
  if (dump_output_frames && mem_type != OUT_SURFACE_MEM_NOT_MAPPED) {
    viddec->SaveFrameToFile(output_file_path, pframe, surf_info);
  }

viddec->ReleaseFrame(pts);
```

The demuxer is deleted once decoding is done.

```
delete demuxer;
```

CHAPTER NINE

USING THE ROCDECDECODE EXAMPLE

rocDecode provides four core APIs exposed in the header files in the `api/rocdecode/` directory:

The rocDecode parser API, exposed in `rocparser.h`.
The hardware decoder API, exposed in `rocdecode.h`.
The software decoder API, exposed in `rocdecode_host.h`.
The bitstream reader API, exposed in `roc_bitstream_reader.h`.

The `rocdecdecode` sample demonstrates how to use the rocDecode core APIs in an application. It shows how to use the parser and both the hardware and software decoders. For information on how to use the bitstream reader API, see *Using the rocDecode bitstream reader API*.

The sample decodes raw elementary video frame files as input and produces individually decoded frames in YUV format as output. The input can be one individual frame file or multiple frames from one or more video files. The individual frame files must be numbered in ascending order of frames.

`rocdecdecode.cpp` takes the following arguments:

Argument	Description	Note
`-i`	Path to the input video frame file or to frame folder.	Required.
`-o`	Output path. Saves the decoded YUV frames to this folder.	Optional. Decoded frames aren't saved by default.
`-d`	GPU device ID. Set it to 0 for the first device, 1 for the second device, 2 for the third device, and so on for each subsequent device.	Optional. Set to 0 by default.
`-b`	Backend. Set it to 0 to use the hardware decoder on the GPU or to 1 to use the software decoder on the CPU.	Optional. Set to 0 by default.
`-c`	Codec. Set to 0 for HEVC, 1 for H264, 2 for AV1, 4 for VP9, 5 for VP8, or 6 for MJPEG.	Optional. Set to 0 by default.
`-n`	Number of iterations for performance evaluation.	Optional. Set to 1 by default.
`-m`	The output surface memory type. The memory type where the surface data, such as the decoded frames, resides. Set this to 0 for intermediate GPU memory, to 1 for GPU memory, and to 2 for CPU memory. See *Surface data memory locations* for more information.	Optional. Set to 0 by default.

The `DecoderInfo` struct defined in the sample is used to store user-supplied parameters as well as the decoder and parser handles.

The memory type and the type of decoder is set by the specified backend. If the GPU (device) backend is selected, both a parser and a hardware decoder are created. If the CPU (host) backend is selected, only a software decoder is created:

```
DecoderInfo dec_info;
[...]
int main(int argc, char** argv) {
  [...]
  dec_info.rocdec_codec_id = CodecTypeToRocDecVideoCodec(codec_type);
  dec_info.dec_device_id = device_id;
  dec_info.mem_type = (!backend) ? OUT_SURFACE_MEM_DEV_INTERNAL : OUT_SURFACE_MEM_HOST;
  init();
  if (backend == DECODER_BACKEND_DEVICE) {
    create_parser(dec_info);
    create_decoder(dec_info);
  } else {
    create_decoder_host(dec_info);
  }
[...]
}
```

All applications need to register the `pfn_sequence_callback` and `pfn_display_picture` callbacks. Applications that use the parser must also register the `pfn_decode_picture` callback.

When the GPU backend is selected, these callbacks are registered in the `create_parser()` function. `create_parser` also creates the parser using `rocDecCreateVideoParser()`:

```
void create_parser(DecoderInfo& dec_info) {
  RocdecParserParams params = {};
  params.codec_type = dec_info.rocdec_codec_id;
  params.max_num_decode_surfaces = 6;
  params.max_display_delay = 1;
  params.user_data = &dec_info;
  params.pfn_sequence_callback = handle_video_sequence;
  params.pfn_decode_picture = handle_picture_decode;
  params.pfn_display_picture = handle_picture_display;
  CHECK(rocDecCreateVideoParser(&dec_info.parser, &params));
}
```

The `create_decoder()` function sets the decoder parameters and passes them to `rocDecCreateDecoder()` to create the hardware decoder:

```
void create_decoder(DecoderInfo& dec_info) {
  RocDecoderCreateInfo create_info = {};
  create_info.codec_type = dec_info.rocdec_codec_id;     // user specified codec_type␣
↪for raw files
  [...]
  CHECK(rocDecCreateDecoder(&dec_info.decoder, &create_info));
}
```

The `create_decoder_host()` function performs the same actions as `create_decoder()`, but uses `rocDecCreateDecoderHost()` to create a software decoder. Because the parser isn't used with the software decoder, and because the software decoder uses different function calls, the callbacks for the software decoder are registered in `create_decoder_host()`:

```
void create_decoder_host(DecoderInfo& dec_info) {
  RocDecoderHostCreateInfo create_info = {};
  create_info.codec_type = dec_info.rocdec_codec_id;
```

(continues on next page)

(continued from previous page)

```
  [...]
  create_info.pfn_sequence_callback = handle_video_sequence_host;
  create_info.pfn_display_picture = handle_picture_display_host;
  CHECK(rocDecCreateDecoderHost(&dec_info.decoder, &create_info));
  dec_info.backend = DECODER_BACKEND_HOST;
}
```

After the decoder and parser have been created, `decode_frames` is called.

```
int main(int argc, char** argv) {
  [...]
  dec_info.dump_decoded_frames = dump_output_frames;
  auto input_frames = read_frames(input_file_names);
  auto start = std::chrono::high_resolution_clock::now();
  for (int i = 0; i < num_iterations; i++) {
    decode_frames(dec_info, input_frames);
  }
  [...]
}
```

`decode_frames` calls `rocDecParseVideoData()` or `rocDecDecodeFrameHost()`, depending on the backend, to parse and decode the frames:

```
void decode_frames(DecoderInfo& dec_info, const std::vector<std::vector<uint8_t>>&␣
↪frames) {
  // gpu backend using VCN
  if (dec_info.backend == DECODER_BACKEND_DEVICE) {
      for (int i=0; i < static_cast<int>(frames.size()); ++i) {
          RocdecSourceDataPacket packet = {};
          packet.payload_size = frames[i].size();
          packet.payload = frames[i].data();
          if (i == static_cast<int>(frames.size() - 1)) {
              packet.flags = ROCDEC_PKT_ENDOFPICTURE;     // mark end_of_picture flag␣
↪for last frame
          }
          CHECK(rocDecParseVideoData(dec_info.parser, &packet));
      }
  } else if (dec_info.backend == DECODER_BACKEND_HOST) {
      for (int i=0; i < static_cast<int>(frames.size()); ++i) {
          RocdecPicParamsHost pic_params = {};
          pic_params.bitstream_data_len = frames[i].size();
          pic_params.bitstream_data = frames[i].data();
          if (i == static_cast<int>(frames.size() - 1)) {
              pic_params.flags = ROCDEC_PKT_ENDOFPICTURE;     // mark end_of_picture␣
↪flag for last frame
          }
          CHECK(rocDecDecodeFrameHost(dec_info.decoder, &pic_params));
      }
  }
}
```

The registered callbacks are triggered during the calls to `rocDecParseVideoData()` and `rocDecDecodeFrameHost()`.

pfn_decode_picture is triggered when a new frame is ready to be decoded, pfn_sequence_callback is triggered when a new sequence header is encountered, and pfn_display_picture is triggered when a frame has finished being decoded.

pfn_decode_picture needs to call rocDecDecodeFrame() or rocDecodeFrameHost(), depending on the specified backend, to decode a frame.

In rocdecdecode.cpp, pfn_decode_picture calls handle_picture_decode() or handle_picture_decode_host(), depending on the specified backend:

```
int ROCDECAPI handle_picture_decode(void* user_data, RocdecPicParams* params) {
  DecoderInfo *p_dec_info = static_cast<DecoderInfo *>(user_data);
  CHECK(rocDecDecodeFrame(p_dec_info->decoder, params));
  return 1;
}
```

pfn_sequence_callback is triggered when a format change occurs or when a new sequence header is encountered. When this happens, the decoder is reconfigured to handle the new sequence or format.

pfn_sequence_callback needs to call rocDecReconfigureDecoder() or rocDecReconfigureDecoderHost() depending on the backend, to reconfigure the decoder.

In the rocdecdecode.cpp sample, pfn_sequence_callback calls handle_video_sequence() or handle_video_sequence_host(), depending on the specified backend:

```
int ROCDECAPI handle_video_sequence(void* user_data, RocdecVideoFormat* format) {
  DecoderInfo *p_dec_info = static_cast<DecoderInfo *>(user_data);
  [...]
  RocdecReconfigureDecoderInfo reconfig_params = {};
  reconfig_params.width = format->coded_width;
  reconfig_params.height = format->coded_height;
  reconfig_params.bit_depth_minus_8 = bitdepth_minus_8;
  reconfig_params.num_decode_surfaces = format->min_num_decode_surfaces;
  reconfig_params.target_width = target_width;
  reconfig_params.target_height = target_height;
  reconfig_params.display_rect.left = format->display_area.left;
  reconfig_params.display_rect.right = format->display_area.right;
  reconfig_params.display_rect.top = format->display_area.top;
  reconfig_params.display_rect.bottom = format->display_area.bottom;
  CHECK(rocDecReconfigureDecoder(p_dec_info->decoder, &reconfig_params));
  [...]
  return 1;
}
```

pfn_display_picture is triggered when a frame has been decoded. It needs to call rocDecGetVideoFrame() or rocDecGetVideoFrameHost(), depending on the specified backend.

rocDecGetVideoFrame() and rocDecGetVideoFrameHost() map the video ID of the decoded frame to HIP. Calls to both these functions block until the frame is decoded and the memory mapping is complete. They return the HIP device pointer or the host memory pointer, depending on the backend specified, as well as information about the *output surface*.

pfn_display_picture calls handle_picture_display() or handle_handle_picture_display_host(), depending on the specified backend, and saves the frames to file if the rocdecdecode was run with the -o option:

From the rocdecdecode.cpp sample:

```
int ROCDECAPI handle_picture_display(void* user_data, RocdecParserDispInfo* disp_info) {
  DecoderInfo *p_dec_info = static_cast<DecoderInfo *>(user_data);
  RocdecProcParams params = {};
  params.progressive_frame = disp_info->progressive_frame;
  params.top_field_first = disp_info->top_field_first;
  void* dev_mem_ptr[3] = { 0 };
  uint32_t pitch[3] = { 0 };
  CHECK(rocDecGetVideoFrame(p_dec_info->decoder, disp_info->picture_index, dev_mem_ptr,
→pitch, &params));
    if (p_dec_info->dump_decoded_frames) {
      save_frame_to_file(p_dec_info, dev_mem_ptr, pitch);
    }
  return 1;
}
```

Once decoding is complete, `rocDecDestroyVideoParser()` needs to be called to destroy the parser, and either `rocDecDestroyDecoderHost()` or `rocDecDestroyDecoder()` needs to be called to destroy the decoder.

CHAPTER

TEN

USING THE ROCDECODE ROCVIDEODECODER

rocDecode provides two methods fpr decoding a video stream: using the rocDecode RocVideoDecoder on the GPU or using the FFmpeg decoder on the CPU.

This topic covers how to decode a video stream using the RocVideoDecoder class in `roc_video_dec.h`. The RocVideoDecode class provides high-level calls to the core APIs in the `api` folder of the rocDecode GitHub repository. For information about the core APIs, see *Using the rocDecode core APIs*.

The RocVideoDecoder takes a demultiplexed coded picture as input. The picture can be demultiplexed from a video stream using the *FFmpeg demultiplexer*.

To use the rocDecode video decoder, import the `roc_video_dec.h` header file and instantiate `RocVideoDecoder`.

The `RocVideoDecoder` constructor takes the following parameters:

Parameter	Description	Default
`device_id`	`int` The GPU device ID. Set it to 0 for the first device, 1 for the second device, 2 for the third device, and so on for each subsequent device.	0
`out_mem_type`	`OutputSurfaceMemoryType` The memory type where the surface data, such as the decoded frames, resides. 0: `OUT_SURFACE_MEM_DEV_INTERNAL`. The surface data is stored internally on memory shared by the GPU and CPU. 1: `OUT_SURFACE_MEM_DEV_COPIED`. The surface data resides on the GPU. 2: `OUT_SURFACE_MEM_HOST_COPIED`. The surface data resides on the CPU. See *Surface data memory locations* for more information.	0, OUT_SURFACE_MEM_DEV_IN
`codec`	`rocDecVideoCodec` The video file's codec ID converted to `rocDecVideoCodec` using `AVCodec2RocDecVideoCodec`.	No default, a value must be provided
`force_zero_l`	`bool` Set to `true` to flush decoded frames for immediate display.	`false`
`p_crop_rect`	`const Rect *` The rectangle to use for cropping.	No cropping
`extract_user`	`bool` Set to `true` to extract Supplemental Enhancement Information (SEI) from the video stream.	`false`, no SEI will be extracted
`disp_delay`	`uint32_t` Delay the display by this number of frames.	0, no delay in displaying the frames
`max_width`	`int` Max width.	0
`max_height`	`int` Max height.	0
`clk_rate`	`uint32_t` Clock rate.	1000

For example, from `videodecode.cpp`:

```
RocVideoDecoder viddec(device_id, mem_type, rocdec_codec_id, b_force_zero_latency, p_
↪crop_rect, b_extract_sei_messages, disp_delay);
```

RocVideoDecoder will create a parser and a decoder, and initialize HIP on the device.

The same decoder instance is reused when there's a change to the video resolution without a change in the codec.

The decoder maintains a pool of frame buffers for decoded images that haven't yet been displayed or processed. When the video stream resolution changes, the existing frame buffers in the buffer pool are deleted. The decoder is then reconfigured for the new resolution and new buffers are created.

To prevent the remaining frames in the buffers from being deleted along with the buffers, a callback function can be defined to consume the remaining frames.

The ReconfigParams_t struct stores information on how to handle the reconfiguration. A callback, a user-defined flush mode, and a user-defined struct are passed to ReconfigParams_t. The reconfiguration parameters are then passed to the decoder using SetReconfigParams.

The reconfiguration parameters need to be defined prior to entering the decoding loop. For example, the reconfiguration structs are defined in common.h in the rocDecode samples and then used in videodecode.cpp

```
typedef enum ReconfigFlushMode_enum {
    RECONFIG_FLUSH_MODE_NONE = 0x0,                   /**<  Just flush to get the
↪frame count */
    RECONFIG_FLUSH_MODE_DUMP_TO_FILE = 0x1,           /**<  The remaining frames will
↪be dumped to file in this mode */
    RECONFIG_FLUSH_MODE_CALCULATE_MD5 = (0x1 << 1),   /**<  Calculate the MD5 of the
↪flushed frames */
} ReconfigFlushMode;

typedef struct ReconfigDumpFileStruct_t {
    bool b_dump_frames_to_file;
    std::string output_file_name;
    void *md5_generator_handle;
} ReconfigDumpFileStruct;

reconfig_params.p_fn_reconfigure_flush = ReconfigureFlushCallback;
reconfig_user_struct.b_dump_frames_to_file = dump_output_frames;
reconfig_user_struct.output_file_name = output_file_path;
reconfig_params.reconfig_flush_mode = RECONFIG_FLUSH_MODE_NONE;
if (dump_output_frames) {
    reconfig_params.reconfig_flush_mode |= RECONFIG_FLUSH_MODE_DUMP_TO_FILE;
}
if (b_generate_md5) {
    reconfig_params.reconfig_flush_mode |= RECONFIG_FLUSH_MODE_CALCULATE_MD5;
}
reconfig_params.p_reconfig_user_struct = &reconfig_user_struct;
viddec.SetReconfigParams(&reconfig_params);
```

In the decode loop, the demultiplexed coded picture is passed to DecodeFrame. Once the frame is decoded and processed, it is released with ReleaseFrame.

CHAPTER ELEVEN

USING THE ROCDECODE FFMPEG DEMULTIPLEXER

The rocDecode FFmpeg demultiplexer (demuxer) extracts coded picture data from digital media files.

To use the rocDecode FFmpeg demuxer , import the `video_demuxer.h` header file.

```
#include "video_demuxer.h"
```

Instantiate a `VideoDemuxer` with the path to the video file. The `GetCodecId` and `GetBitDepth` functions can be used to obtain the video stream's codec ID and bit depth. The `AVCodec2RocDecVideoCodec` utility function converts the codec ID returned from the demuxer to its corresponding `rocDecVideoCodec_enum` value.

```
VideoDemuxer *demuxer;
demuxer = new VideoDemuxer(input_file_path.c_str());
rocdec_codec_id = AVCodec2RocDecVideoCodec(demuxer->GetCodecID());
bit_depth = demuxer->GetBitDepth();
```

Call `Demux` to extract frame data from the stream:

```
demuxer->Demux(&pvideo, &n_video_bytes, &pts);
```

The demuxer will demultiplex frames sequentially starting at the beginning of the stream. To start the demultiplexing and decoding process from a different frame, create a seek context that specifies a seek criteria and a seek mode.

The seek criteria describes whether the demuxer needs to seek to a specific frame or seek to a specific timestamp. The seek mode indicates whether the demuxer should seek to the exact frame or to the previous keyframe.

The seek criteria is defined by the `SeekCriteriaEnum` enum and the seek mode is defined by the `SeekModeEnum` enum. Both the `SeekCriteriaEnum` and the `SeekModeEnum` are defined in `video_demuxer.h`.

Set the seek criteria to `SEEK_CRITERIA_FRAME_NUM` to seek to a frame or to `SEEK_CRITERIA_TIME_STAMP` to seek to a timestamp. Set the seek mode to `SEEK_MODE_EXACT_FRAME` to seek to the exact frame or to `SEEK_MODE_PREV_KEY_FRAME` to seek to the previous keyframe.

From `videodecode.cpp`:

```
VideoSeekContext video_seek_ctx;
[...]
do {
  [...]
  if (seek_criteria == 1 && first_frame) {
    // use VideoSeekContext class to seek to given frame number
    video_seek_ctx.seek_frame_ = seek_to_frame;
    video_seek_ctx.seek_crit_ = SEEK_CRITERIA_FRAME_NUM;
    video_seek_ctx.seek_mode_ = (seek_mode ? SEEK_MODE_EXACT_FRAME : SEEK_MODE_PREV_KEY_
↪FRAME);
```

(continues on next page)

(continued from previous page)

```
    demuxer->Seek(video_seek_ctx, &pvideo, &n_video_bytes);
    pts = video_seek_ctx.out_frame_pts_;
    std::cout << "info: Number of frames that were decoded during seek - " << video_seek_
↪ctx.num_frames_decoded_ << std::endl;
    first_frame = false;
  } else if (seek_criteria == 2 && first_frame) {
    // use VideoSeekContext class to seek to given timestamp
    video_seek_ctx.seek_frame_ = seek_to_frame;
    video_seek_ctx.seek_crit_ = SEEK_CRITERIA_TIME_STAMP;
    video_seek_ctx.seek_mode_ = (seek_mode ? SEEK_MODE_EXACT_FRAME : SEEK_MODE_PREV_KEY_
↪FRAME);
    demuxer->Seek(video_seek_ctx, &pvideo, &n_video_bytes);
    pts = video_seek_ctx.out_frame_pts_;
    std::cout << "info: Duration of frame found after seek - " << video_seek_ctx.out_
↪frame_duration_ << " ms" << std::endl;
    first_frame = false;
  } else {
    demuxer->Demux(&pvideo, &n_video_bytes, &pts);
  }
  [...]
} while (n_video_bytes);
```

Delete the demuxer once demultiplexing is complete.

```
delete demuxer;
```

CHAPTER

TWELVE

USING THE ROCDECODE BITSTREAM READER APIS

The rocDecode bitstream reader APIs are a simplified set of APIs that provide a way to use and test the decoder without relying on FFMpeg. The bitstream reader APIs can be used to extract and parse coded picture data from an elementary video stream for the decoder to consume.

Note

The bitstream reader APIs can only be used with elementary video streams and IVF container files.

The `videodecoderaw.cpp` sample demonstrates how to use the bitstream reader APIs, including how to create a bitstream reader and use it to extract picture data and pass it to the decoder:

```
RocdecBitstreamReader bs_reader = nullptr;
rocDecVideoCodec rocdec_codec_id;
int bit_depth;
if (rocDecCreateBitstreamReader(&bs_reader, input_file_path.c_str()) != ROCDEC_SUCCESS) {
    std::cerr << "Failed to create the bitstream reader." << std::endl;
    return 1;
}
[...]
# Decode loop:
do {
    if (rocDecGetBitstreamPicData(bs_reader, &pvideo, &n_video_bytes, &pts) != ROCDEC_
↪SUCCESS) {
        std::cerr << "Failed to get picture data." << std::endl;
        return 1;
    }
    [...]
    n_frame_returned = viddec.DecodeFrame(pvideo, n_video_bytes, pkg_flags, pts, &
↪decoded_pics);
}
```

The `videodecoderaw.cpp` example also demonstrates how to use the bitstream reader APIs to obtain the bit depth and codec of a stream:

```
if (rocDecGetBitstreamCodecType(bs_reader, &rocdec_codec_id) != ROCDEC_SUCCESS) {
    std::cerr << "Failed to get stream codec type." << std::endl;
     return 1;
}
[...]
if (rocDecGetBitstreamBitDepth(bs_reader, &bit_depth) != ROCDEC_SUCCESS) {
```

(continues on next page)

(continued from previous page)

```
    std::cerr << "Failed to get stream bit depth." << std::endl;
    return 1;
}
```

Note

`rocDecDestroyBitstreamReader` must always be called to destroy the bitstream reader once processing is complete.

CHAPTER

THIRTEEN

THE ROCDECODE CORE APIS

The rocDecode core APIs are intended for users who want to have full control of the decoding pipeline and interact with the core components instead of the utility classes. The *Using the rocDecode videodecode sample* provides an introduction to using the utility classes.

The rocDecode core APIs are exposed in header files in the `api/rocdecode` folder of the rocDecode GitHub repository.

The rocDecode parser API is exposed in `api/rocdecode/rocparser.h`. It contains functions that create and destroy the parser, as well as functions that parse the bitstream.

The hardware decoder API is exposed in `api/rocDecode/rocdecode.h`. It contains functions that create, control, and destroy the decoder, as well as functions that decode the parsed frames on the GPU.

The software decoder API is exposed in `api/rocDecode/rocdecode_host.h`. It contains the same functionality as `rocdecode.h`, but all the operations are run on the host rather than the GPU.

The bitstream reader API is exposed in `api/rocDecode/roc_bitstream_reader.h`. It provides an alternative to the FFMpeg demuxer and contains a simple stream file parser that can read elementary files and IVF container files.

13.1 The rocDecode parser API

The rocDecode parser API, exposed in `rocparser.h`, is used to decode bitstreams and organize them in a structured format that can be consumed by the hardware decoder.

The parser parameters are stored in the `RocdecParserParams` struct and passed to `rocDecCreateVideoParser()` to create a new parser. `rocDecCreateVideoParser()` returns a handle to the parser. For example:

```
RocdecParserParams params = {};
params.codec_type = rocdec_codec_id;
params.max_num_decode_surfaces = 6;
params.max_display_delay = 1;
params.user_data = &dec_info;
rocDecCreateVideoParser(&parser_handle, &params);
```

Elementary stream video packets extracted from the demultiplexer (demuxer) are passed to the parser using the `RocdecSourceDataPacket` struct. Packet information in `RocdecSourceDataPacket` is passed to `rocDecParseVideoData()`. For example:

```
RocdecSourceDataPacket packet = {};
packet.payload_size = frames[i].size();
packet.payload = frames[i].data();
rocDecParseVideoData(parser_handle, &packet);
```

Three callbacks must be registered when the parser is used: `pfn_decode_picture`, `pfn_sequence_callback`, and `pfn_display_picture`. These callbacks are triggered in the `rocDecParseVideoData()` call.

`pfn_decode_picture` is triggered when a picture is ready for decoding. Its implementation must call `rocDecDecodeFrame()` from the hardware decoder API.

`pfn_sequence_callback` is triggered when a new sequence header is encountered or when there's a format change. Its implementation handles reconfiguring the decoder to handle the new frame format. Its implementation must call `rocDecReconfigureDecoder()` from the hardware decoder API.

`pfn_display_picture` is triggered when a frame has been decoded. Its implementation must call `rocDecGetVideoFrame()` from the hardware decoder API.

A fourth callback, `pfn_get_sei_msg`, is optional. `pfn_get_sei_msg` is triggered when a Supplementation Enhancement Information (SEI) message is parsed and returned to the caller.

If any of the callbacks return an error, the error is propagated back to the application.

Once the stream is fully decoded, `rocDecDestroyVideoParser()` must be called to destroy the parser object and free all allocated resources.

13.2 The rocDecode hardware decoder API

The rocDecode hardware decoder API exposed in `api/rocDecode/rocdecode.h` is used to decode frames that were parsed by *the rocDecode parser*.

Parsing parameters are stored in the `RocDecoderCreateInfo` struct and passed to `rocDecCreateDecoder()` to create a new decoder. `rocDecCreateDecoder()` returns a handle to the decoder. For example:

```
RocDecoderCreateInfo create_info = {};
create_info.codec_type = dec_info.rocdec_codec_id;     // user specified codec_type for raw files
create_info.max_width = DEFAULT_WIDTH;
create_info.max_height = DEFAULT_HEIGHT;
create_info.width = DEFAULT_WIDTH;
create_info.height = DEFAULT_HEIGHT;
create_info.num_decode_surfaces = 6;
create_info.num_output_surfaces = 1;
rocDecCreateDecoder(&decoder_handle, &create_info);
```

`rocDecGetDecoderCaps()` queries the capabilities of the underlying hardware video decoder. Decoder capabilities usually include supported codecs, maximum resolution, and bit depth.

`rocDecDecodeFrame()` is used to submit frames for hardware decoding. This function must be called when the `pfn_decode_picture` callback is triggered in the `rocDecParseVideoData()` call. See *The rocDecode parser API* for details about this call.

`rocDecDecodeFrame()` takes the decoder handle and the pointer to the `RocdecPicParams()` struct and initiates the video decoding using VA-API. `RocdecPicParams` is populated with the decoded frame information.

The `pfn_sequence_callback` callback is triggered when a format change occurs or when a new sequence header is encountered. The implementation of `pfn_sequence_callback` must call `rocDecReconfigureDecoder()` to reconfigure the decoder to handle the new sequence or format. See *The rocDecode parser API* for details about this callback.

`rocDecGetDecodeStatus()` can be called to query the decoding status of a frame. The result of the query is either `rocDecodeStatus_Success`, if decoding is complete, or `rocDecodeStatus_InProgress`, if decoding is still in progress.

The `pfn_display_picture` callback is triggered when a frame has been decoded. The decoded frame can then be further processed in device memory. The implementation for this callback must call `rocDecGetVideoFrame()` to obtain the decoded frame's HIP device pointer.

`rocDecGetVideoFrame()` provides a way to access the decoded frame in HIP. This is a blocking call that only returns once frame decoding and memory mapping is complete. It returns the HIP device pointer as well as information about the *output surface type*.

If the output surface type is `OUT_SURFACE_MEM_DEV_INTERNAL`, meaning intermediate GPU memory, the direct pointer to the decoded surface is provided. If the requested surface type is `OUT_SURFACE_MEM_DEV_COPIED` or `OUT_SURFACE_MEM_HOST_COPIED`, the internal decoded frame is copied to another buffer, either in device memory or host memory.

Once decoding is complete, `rocDecDestroyVideoParser()` and `rocDecDestroyDecoder()` must be called to destroy the parser and the decoding session, and free resources.

13.3 The rocDecode software decoder API

The rocDecode software decoder API exposed in `api/rocDecode/rocdecode_host.h` is used to decode frames that have been demultiplexed (demuxed) by *the FFmpeg demuxer*.

Decoding parameters are stored in the `RocDecoderHostCreateInfo` struct and passed to `rocDecCreateDecoderHost()` to create a new software decoder. `rocDecCreateDecoderHost()` returns a handle to the decoder. For example:

```
RocDecoderHostCreateInfo create_info = {};
create_info.codec_type = rocdec_codec_id;
create_info.num_decode_threads = 0;     // default
create_info.max_width = DEFAULT_WIDTH;
create_info.max_height = DEFAULT_HEIGHT;
create_info.chroma_format = rocDecVideoChromaFormat_420;
create_info.output_format = rocDecVideoSurfaceFormat_P016;
create_info.bit_depth_minus_8 = 2;
create_info.num_output_surfaces = 1;
create_info.user_data = &dec_info;
rocDecCreateDecoderHost(&dec_info.decoder, &create_info);
```

`rocDecGetDecoderCapsHost()` queries the capabilities of the underlying software video decoder. Decoder capabilities usually include supported codecs, maximum resolution, and bit depth.

`rocDecDecodeFrameHost()` is used to submit frames for software decoding. `rocDecDecodeFrameHost()` takes the decoder handle and the pointer to the `RocdecPicParamsHost` struct and initiates the video decoding. `RocdecPicParamsHost` is populated with the decoded frame information.

The `pfn_sequence_callback` callback is triggered when a format change occurs or when a new sequence header is encountered. This callback must be registered in any application that uses the software decoder and its implementation must call `rocDecReconfigureDecoderHost()` to reconfigure the decoder to handle the new sequence or format.

`rocDecGetDecodeStatusHost()` can be called to query the decoding status of a frame. The result of the query is either `rocDecodeStatus_Success`, if decoding is complete, or `rocDecodeStatus_InProgress`, if decoding is still in progress.

The `pfn_display_picture` callback is triggered when a frame has been decoded. This callback must be registered by any application that uses the software decoder and its implementation must call `rocDecGetVideoFrameHost()`. `rocDecGetVideoFrameHost()` returns the decoded frame's host memory pointer. The decoded frame can then be further processed using this pointer.

`rocDecGetVideoFrameHost()` provides a way to access the decoded frame in host memory. This is a blocking call that only returns once both frame decoding and memory mapping are done. It returns the host memory pointer as well as information about the *output surface type*.

If the output surface type is OUT_SURFACE_MEM_DEV_INTERNAL, meaning intermediate GPU memory, the direct pointer to the decoded surface is provided. If the requested surface type is OUT_SURFACE_MEM_DEV_COPIED or OUT_SURFACE_MEM_HOST_COPIED, the internal decoded frame is copied to another buffer, either in device memory or host memory.

Once decoding is complete, rocDecDestroyDecoderHost() must be called to destroy the decoder and free resources.

CHAPTER

FOURTEEN

ROCDECODE LOGGING CONTROL

rocDecode core components can be configured to output different levels of log messages during decoding.

The log level can be changed by either setting the log level through the `ROCDEC_LOG_LEVEL` environment variable, or by calling the `RocDecLogger::SetLogLevel()` function in `commons.h`.

The logging levels are:

0: Critical (Default level)
1: Error
2: Warning
3: Info
4: Debug

The log level defines the maximum severity of log messages to output. For example, to output warning and error messages as well as critical messages, `ROCDEC_LOG_LEVEL` would need to be set to 2:

```
ROCDEC_LOG_LEVEL = 2
```

or

```
SetLogLevel(2);
```

CHAPTER

FIFTEEN

ROCDECODE SUPPORTED CODECS AND HARDWARE CAPABILITIES

rocDecode supports the following codecs:

- H.265 (HEVC): 8 bit and 10 bit
- H.264 (AVC): 8 bit
- AV1: 8 bit and 10 bit
- VP9: 8 bit and 10 bit

The following table shows the codec support and capabilities of the VCN for each supported GPU architecture:

GPU Architecture	VCN Generation	Number of VCNs	H.265	Max width, Max height - H.265	H.264	Max width, Max height - H.264	AV1	Max width, Max height - AV1	VP9	Max width, Max height - VP9
gfx908 - MI1xx	VCN 2.5.0	2	Yes	7680, 4320	Yes	4096, 2160	No	N/A, N/A	Yes	7680, 4320
gfx90a - MI2xx	VCN 2.6.0	2	Yes	7680, 4320	Yes	4096, 2160	No	N/A, N/A	Yes	7680, 4320
gfx942 - MI3xx	VCN 4.0	3/4	Yes	7680, 4320	Yes	4096, 2176	Yes	8192, 4352	Yes	7680, 4320
gfx1030, gfx1031, gfx1032 - Navi2x	VCN 3.x	2	Yes	7680, 4320	Yes	4096, 2176	Yes	8192, 4352	Yes	7680, 4320
gfx1100, gfx1102 - Navi3x	VCN 4.0	2	Yes	7680, 4320	Yes	4096, 2176	Yes	8192, 4352	Yes	7680, 4320
gfx1101 - Navi3x	VCN 4.0	1	Yes	7680, 4320	Yes	4096, 2176	Yes	8192, 4352	Yes	7680, 4320

CHAPTER

SIXTEEN

FILE LIST

CHAPTER

SEVENTEEN

GLOBALS

CHAPTER EIGHTEEN

DATA STRUCTURES

CHAPTER NINETEEN

LICENSE

MIT License

Copyright © 2023 - 2026 Advanced Micro Devices, Inc. All rights reserved

NOTICE REGARDING STANDARDS

AMD does not provide a license or sublicense to any Intellectual Property Rights relating to any standards, including but not limited to any audio and/or video codec technologies such as MPEG-2, MPEG-4; AVC/H.264; HEVC/H.265; AAC decode/FFMPEG; AAC encode/FFMPEG; VC-1; and MP3 (collectively, the "Media Technologies"). For clarity, you will pay any royalties due for such third party technologies, which may include the Media Technologies that are owed as a result of AMD providing the Software to you.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.