
rocDecode documentation

Release 1.0.0

Advanced Micro Devices, Inc.

Sep 16, 2025

CONTENTS

| | | |
|-----------|---|-----------|
| 1 | What is rocDecode? | 3 |
| 2 | rocDecode prerequisites | 5 |
| 3 | Installing rocDecode with the package installer | 7 |
| 3.1 | Basic installation | 7 |
| 3.2 | Complete installation | 7 |
| 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 rocDecode RocVideoDecoder | 23 |
| 10 | Using the rocDecode FFmpeg demultiplexer | 25 |
| 11 | Using the rocDecode bitstream reader APIs | 27 |
| 12 | Using the rocDecode core APIs | 29 |
| 12.1 | API overview | 29 |
| 12.2 | Create a parser object | 29 |
| 12.3 | Parse video data | 30 |
| 12.4 | Query decode capabilities | 30 |
| 12.5 | Create a decoder | 31 |
| 12.6 | Decode the frame | 31 |
| 12.7 | Query the decoding status | 31 |
| 12.8 | Prepare the decoded frame for further processing | 31 |
| 12.9 | Reconfigure the decoder | 32 |
| 12.10 | Destroy the decoder | 32 |
| 12.11 | Destroy the parser | 32 |
| 13 | rocDecode supported codecs and hardware capabilities | 33 |
| 14 | rocDecode tested configurations | 35 |
| 15 | File List | 37 |

| | | |
|-----------|------------------------|-----------|
| 16 | Globals | 39 |
| 17 | Data Structures | 41 |
| 18 | License | 43 |

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*
- *Use the rocDecode RocVideoDecoder*
- *Use the rocDecode FFmpeg demultiplexer*
- *Use the rocDecode bitstream reader APIs*
- *Use the rocDecode core APIs*

Samples

- *rocDecode samples*

Reference

- *rocDecode codec support and hardware capabilities*
- *rocDecode tested configurations*
- *API library*
- *Functions*
- *Data structures*

To contribute to the documentation, refer to [Contributing to ROCm](#).

You can find licensing information on the [Licensing](#) page.

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](#).

ROCDECODE PREREQUISITES

rocDecode requires ROCm 6.1 or later running on [accelerators 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`, 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

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-dev`: The rocDecode development package. This package installs a full suite of libraries, header files, and samples. This package needs to be installed to use the rocDecode samples.
- `rocdecode-test`: A test package that provides a CTest to verify the installation. This package needs to be installed to use the rocDecode samples.

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

3.1 Basic installation

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 Complete installation

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

Ubuntu

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

RHEL

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

SLES

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

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`.

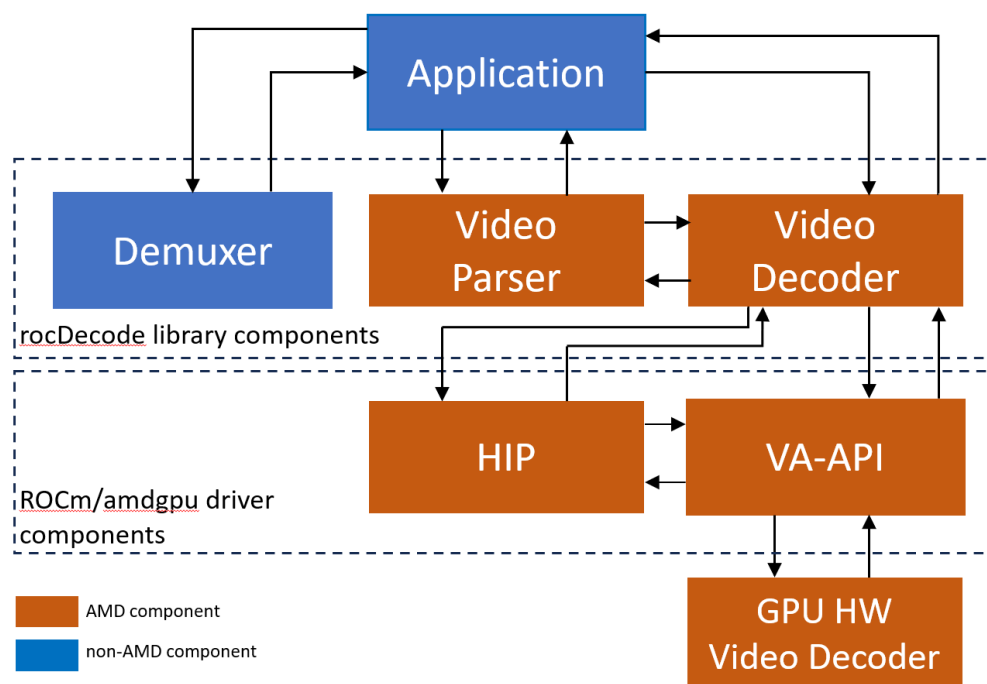
```
mkdir rocdecode-test && cd rocdecode-test
cmake /opt/rocm/share/rocdecode/test/
ctest -VV
```

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
```


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 VA-API-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.

i 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.

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.

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

If you're using a [package installer](#), install `rocdecode`, `rocdecode-dev`, 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
```


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 <code>true</code> forces decoded frames to be flushed out for display immediately. <code>false</code> 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 <code>true</code> to extract SEI. <code>false</code> by default. |
| -md5 | Generate MD5 message digest | Optional. Set to <code>true</code> to generate the MD5 message digest for the decoded YUV image sequence. <code>false</code> 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 <code>Rect struct</code> 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 <i>Surface data memory locations</i> 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_mc | 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 <code>true</code> 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;
```


USING THE ROCDECODE ROCVIDEODECODER

rocDecode provides two ways of decoding a video stream: using the rocDecode RocVideoDecoder on GPU or using the FFmpeg decoder on 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 |
|---------------------------|---|--|
| <code>device_id</code> | <code>int</code> 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 |
| <code>out_mem_type</code> | <code>OutputSurfaceMemoryType</code> The memory type where the surface data, such as the decoded frames, resides. 0: <code>OUT_SURFACE_MEM_DEV_INTERNAL</code> . The surface data is stored internally on memory shared by the GPU and CPU. 1: <code>OUT_SURFACE_MEM_DEV_COPIED</code> . The surface data resides on the GPU. 2: <code>OUT_SURFACE_MEM_HOST_COPIED</code> . The surface data resides on the CPU. See <i>Surface data memory locations</i> for more information. | 0, <code>OUT_SURFACE_MEM_DEV_IN</code> |
| <code>codec</code> | <code>rocDecVideoCodec</code> The video file's codec ID converted to <code>rocDecVideoCodec</code> using <code>AVCodec2RocDecVideoCodec</code> . | No default, a value must be provided |
| <code>force_zero_l</code> | <code>bool</code> Set to true to flush decoded frames for immediate display. | false |
| <code>p_crop_rect</code> | <code>const Rect *</code> The rectangle to use for cropping. | No cropping |
| <code>extract_user</code> | <code>bool</code> Set to true to extract Supplemental Enhancement Information (SEI) from the video stream. | false, no SEI will be extracted |
| <code>disp_delay</code> | <code>uint32_t</code> Delay the display by this number of frames. | 0, no delay in displaying the frames |
| <code>max_width</code> | <code>int</code> Max width. | 0 |
| <code>max_height</code> | <code>int</code> Max height. | 0 |
| <code>clk_rate</code> | <code>uint32_t</code> 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`.

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);
}
[...]
```

Delete the demuxer once demultiplexing is complete.

```
delete demuxer;
```

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.

USING THE ROCDECODE CORE APIS

rocDecode core APIs are available in the `api` folder of the rocDecode GitHub repository.

Note

The rocDecode samples use the utility classes in the `utils` folder instead of the core APIs. For information about using the utility classes, see *Using rocDecode with the FFmpeg decoder* and *Using rocDecode with the bitstream decoder*.

12.1 API overview

All rocDecode APIs are exposed in the header files `rocdecode.h` and `rocparser.h`. You can find these files in the `api` folder in the rocDecode repository.

The samples use the `RocVideoDecoder` user class provided in `roc_video_dec.h` in the `utils` folder of the rocDecode repository.

A video parser (defined in `rocparser.h`) is needed to extract and decode headers from the bitstream in order to organize the data into a structured format for the hardware decoder. The parser is critical in video decoding, as it controls the decoding and display of a bitstream's individual frames and fields.

The parser object in `rocparser.h` has three main APIs:

- `rocDecCreateVideoParser()`
- `rocDecParseVideoData()`
- `rocDecDestroyVideoParser()`

12.2 Create a parser object

The `rocDecCreateVideoParser()` API creates a video parser object for the codec that you specify. The API takes `max_num_decode_surfaces`, which determines the Decoded Picture Buffer (DPB) size for decoding. When creating a parser object, the application must register certain callback functions with the driver, which is called from the parser during decode.

- `pfn_sequence_callback` is called when the parser encounters a new sequence header. The parser informs you of the minimum number of surfaces needed by the parser's DPB to successfully decode the bitstream. In addition, the caller can set additional parameters, like `max_display_delay`, to control frame decoding and display.
- The `pfn_decode_picture` callback function is triggered when a picture is set for decoding.
- The `pfn_display_picture` callback function is triggered when a frame in display order is ready to be consumed by the caller.

- The `pfn_get_sei_msg` callback function is triggered when your Supplementation Enhancement Information (SEI) message is parsed and returned to the caller.

12.3 Parse video data

Elementary stream video packets extracted from the de-multiplexer are fed into the parser using the `rocDecParseVideoData()` API.

During this call, the parser triggers the callbacks as it encounters a new sequence header, receives compressed frame/field data ready to be decoded, or when it's ready to display a frame. If any of the callbacks return a failure, it is propagated back to the application so the decoding can be ended gracefully.

12.4 Query decode capabilities

The `rocDecGetDecoderCaps()` API allows you to query the capabilities of the underlying hardware video decoder. Decoder capabilities usually include supported codecs, maximum resolution, and bit depth.

The following pseudo-code illustrates the use of this API. The application handles the error appropriately for non-supported decoder capabilities.

```
RocdecDecodeCaps decode_caps;
memset(&decode_caps, 0, sizeof(decode_caps));
decode_caps.codec_type = p_video_format->codec;
decode_caps.chroma_format = p_video_format->chroma_format;
decode_caps.bit_depth_minus_8 = p_video_format->bit_depth_luma_minus8;

ROCDEC_API_CALL(rocDecGetDecoderCaps(&decode_caps));

if(!decode_caps.is_supported) {
    ROCDEC_THROW("Rocdec:: Codec not supported on this GPU: ", ROCDEC_NOT_SUPPORTED);
    return 0;
}

if ((p_video_format->coded_width > decode_caps.max_width) ||
    (p_video_format->coded_height > decode_caps.max_height)) {

    std::ostringstream errorString;
    errorString << std::endl
                << "Resolution          : " << p_video_format->coded_width << "x" << p_
↪video_format->coded_height << std::endl
                << "Max Supported (wxh) : " << decode_caps.max_width << "x" << decode_
↪caps.max_height << std::endl
                << "Resolution not supported on this GPU ";

    const std::string cErr = errorString.str();
    ROCDEC_THROW(cErr, ROCDEC_NOT_SUPPORTED);
    return 0;
}
```

12.5 Create a decoder

`rocDecCreateDecoder()` creates an instance of the hardware video decoder object and provides you with a handle upon successful creation. Refer to the `RocDecoderCreateInfo` structure for information about the parameters passed for creating the decoder. For example, `RocDecoderCreateInfo::codec_type` represents the codec type of the video. The decoder handle returned by `rocDecCreateDecoder()` needs to be retained for the entire decode session because the handle is passed along with the other decoding APIs. In addition, you can inform display or crop dimensions along with this API.

12.6 Decode the frame

After de-multiplexing and parsing, you can decode bitstream data containing a frame/field using hardware.

Use the `rocDecDecodeFrame()` API to submit a new frame for hardware decoding. Underneath the driver, the Video Acceleration API (VA-API) is used to submit compressed picture data to the driver. The parser extracts all the necessary information from the bitstream and fills the `RocdecPicParams` structure that's appropriate for the codec. The high-level `RocVideoDecoder` class connects the parser and decoder used for all sample applications.

The `rocDecDecodeFrame()` call takes the decoder handle and the pointer to the `RocdecPicParams` structure and initiates the video decoding using VA-API.

12.7 Query the decoding status

After submitting a frame for decoding, you can call `rocDecGetDecodeStatus()` to query the decoding status for a given frame. A structure pointer, `RocdecDecodeStatus*`, is filled and returned.

The API inputs are:

- `decoder_handle`: A `RocDecoder` handler, `rocDecDecoderHandle`.
- `pic_idx`: An `int` value for the `picIdx` for which you want a status in order to index of the picture.
- `decode_status`: A pointer to `RocdecDecodeStatus` as a return value.

Please note that this API makes a non-blocking call and returns the status of the frame associated with `nPicIdx` at the time of the call, without waiting for the decoding to complete. The `decode_status->decode_status` can be either `rocDecodeStatus_Success`, indicating that the decoding has been completed, or `rocDecodeStatus_InProgress`, which means that the decoding is still in progress.

12.8 Prepare the decoded frame for further processing

The decoded frames can be used for further postprocessing using `rocDecGetVideoFrame()`. The successful completion of `rocDecGetVideoFrame()` indicates that the decoding process is complete and the device memory pointer is inter-opped into the ROCm HIP address space in order to further process the decoded frame in device memory. Please note that the `rocDecGetVideoFrame()` API is a blocking call. If the video frame associated with the `pic_idx` is not ready, the call will wait for the decoding to complete before mapping the video frame for use in HIP. The caller gets the necessary information on the output surface, such as YUV format, dimensions, and pitch from this call. In the high-level `RocVideoDecoder` class, we provide four different surface type modes for the mapped surface, as specified in `OutputSurfaceMemoryType`.

```
typedef enum OutputSurfaceMemoryType_enum {
    OUT_SURFACE_MEM_DEV_INTERNAL = 0,          /**< Internal interopped decoded surface_
↪memory **/
    OUT_SURFACE_MEM_DEV_COPIED = 1,          /**< decoded output will be copied to a_
↪separate device memory **/
```

(continues on next page)

(continued from previous page)

```

    OUT_SURFACE_MEM_HOST_COPIED = 2          /**< decoded output will be copied to a
↪ separate host memory */
    OUT_SURFACE_MEM_NOT_MAPPED = 3         /**< decoded output is not available
↪ (interop won't be used): useful for decode only performance app*/
} OutputSurfaceMemoryType;

```

If the mapped surface type is `OUT_SURFACE_MEM_DEV_INTERNAL`, the direct pointer to the decoded surface is provided. You need to call `ReleaseFrame()` (`RocVideoDecoder` class). 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. After that, it's immediately unmapped for reuse by the `RocVideoDecoder` class.

Refer to the `RocVideoDecoder` class and [samples](#) for details on how to use these APIs.

12.9 Reconfigure the decoder

You can call `rocDecReconfigureDecoder()` to reuse a single decoder for multiple clips or when the video resolution changes during the decode. The API currently supports resolution changes, resize parameter changes, and target area parameter changes for the same codec without destroying an ongoing decoder instance. This can improve performance and reduce overall latency.

The API inputs are:

- `decoder_handle`: A `RocDecoder` handler, `rocDecDecoderHandle`.
- `reconfig_params`: You need to specify the parameters for the changes in `RocdecReconfigureDecoderInfo`. The width and height used for reconfiguration cannot exceed the values set for `max_width` and `max_height`, defined in `RocDecoderCreateInfo`. If you need to change these values, you have to destroy and recreate the session.

Note

You need to call `rocDecReconfigureDecoder()` during `RocdecParserParams::pfn_sequence_callback`.

12.10 Destroy the decoder

You need to call the `rocDecDestroyDecoder()` to destroy the session and free up resources.

The API input is:

- `decoder_handle`: A `RocDecoder` handler, `rocDecDecoderHandle`.

The API returns a `RocdecDecodeStatus` value.

12.11 Destroy the parser

You need to call `rocDecDestroyVideoParser()` to destroy the parser object and free up all allocated resources at the end of video decoding.

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 |

ROCDECODE TESTED CONFIGURATIONS

rocDecode has been tested on the following operating system, software, and library versions:

- Linux:
 - Ubuntu 20.04, 22.04
 - RHEL 8, 9
 - SLES 15-SP5
- ROCm:
 - rocm-core 6.2.0.60200-66
 - amdgpu-core 1:6.2.60200-2009582
- libva-dev 2.7.0-2, 2.14.0-1
- mesa-amdgpu-va-drivers 1:24.2.0.60200-2009582
- FFmpeg 4.2.7, 4.4.2-0

CHAPTER
FIFTEEN

FILE LIST

CHAPTER
SIXTEEN

GLOBALS

DATA STRUCTURES

LICENSE

MIT License

Copyright © 2023 - 2025 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.