Eyedea MMR SDK

Developer's Guide

Version 2.24

eyedea RECOGNITION

ADVANCED COMPUTER VISION SOLUTIONS

Contact:

Address:	Eyedea Recognition, s.r.o.
	Vyšehradská 320/49
	128 00, Prague 2
	Czech Republic
web:	www.eyedea.cz
email:	info@eyedea.cz

Table of Contents

1 Product Description

Eyedea MMR SDK is a cross-platform software library developed to provide "*road users*" recognition functionality. It defines an interface between the client's software and our state-of-the-art recognition solution. Eyedea MMR SDK allows a client to recognize a road user located in digital image. The recognition output contains the information about the road user's view and category, in case of vehicle also make, model, generation, variation and color. Additional information regarding the road user is presented in the form of tags.

Road user's *view* can be frontal or rear. Road users are divided into many *categories* like animal, pedestrian, bus, car, heavy truck, light truck, motorbike, van etc. The manufacturers of the vehicles are defined in the *make* output parameter – e.g. Toyota, Volkswagen, etc. The *model* parameter then distinguishes the bodywork of vehicles created by specific manufacturer – Avensis, Passat, etc. *Generation* specifies mark and year of first sale of a specific bodywork, e.g. Mk I (2015). *Variation* describes nuances in bodywork or trim level, e.g. Wagon, Sedan, Sportline etc.

Road user *tag* may label the road user, e.g. as ambulance or pickup.

1.1 Technical Details

Eyedea MMR SDK consists of two libraries – base MMR SDK library and the recognition module. Both are x86/x64/aarch64 libraries with C interface. The base MMR SDK library is the only entry point, the user never uses the recognition module directly.

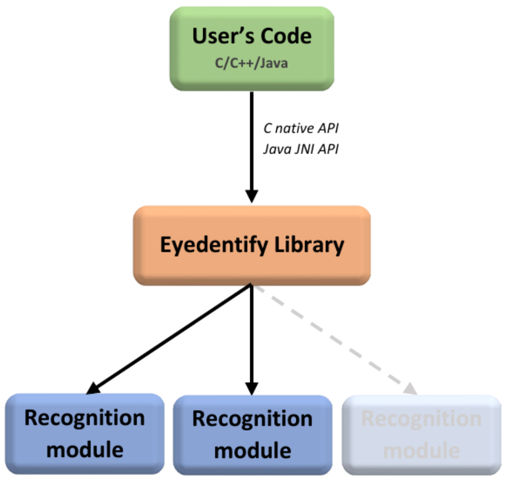
The recognition module is loaded and configured using the base MMR SDK library. The configuration parameters are loaded from one of the binary models, which are contained in the distribution.

The MMR SDK library provides following APIs:

- C native API
- C++ API
- Java JNI API
- C# API (Windows only)
- Python wrapper

Officially supported operating systems and platforms:

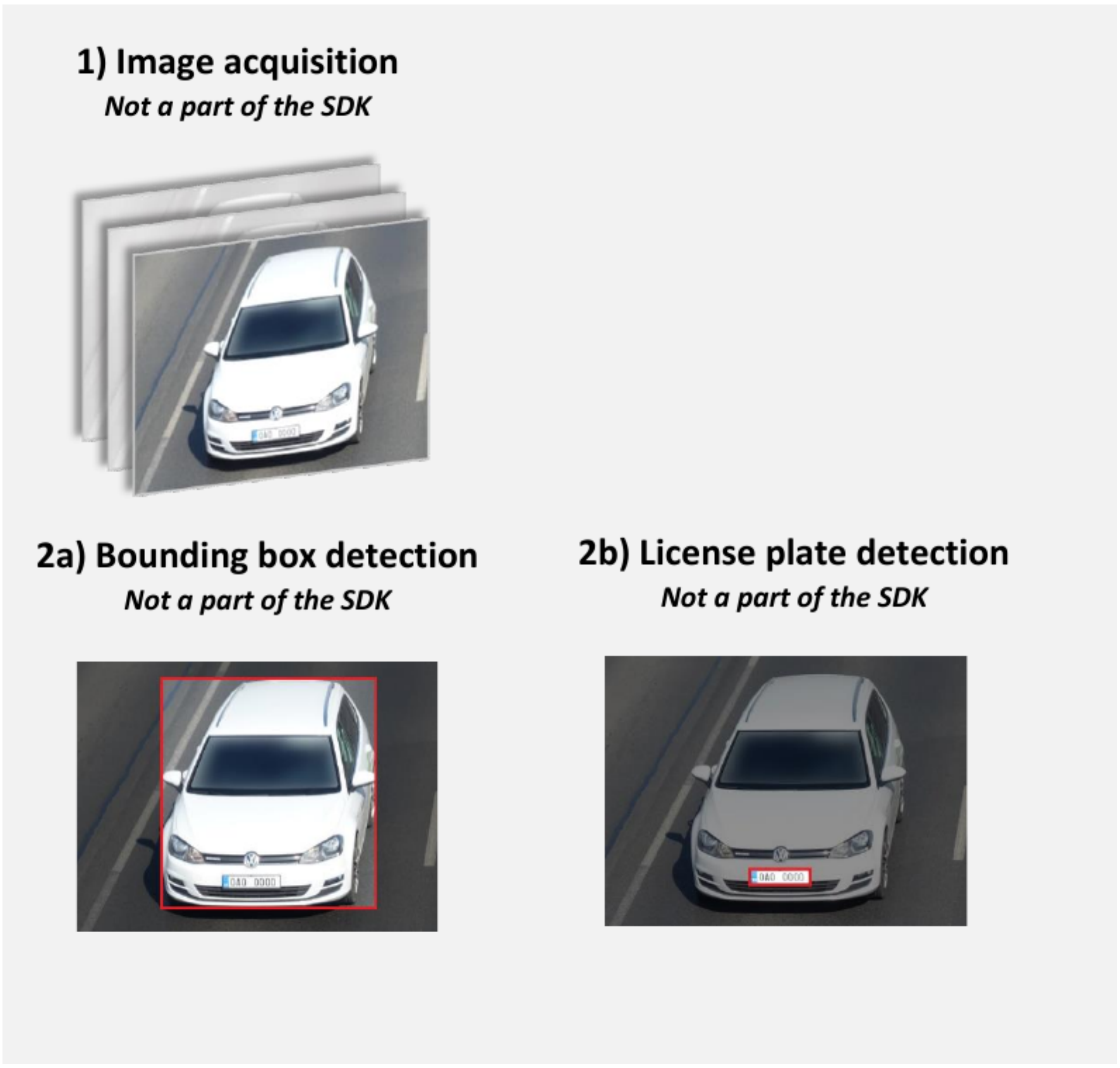
- Windows 7, 8, 8.1, 10 and 11
  32 and 64 bit (Visual Studio 2019)
- Ubuntu 18.04 and higher
  64 bit and aarch64



1.2 System Workflow

The workflow of the MMR system consists of image acquisition, road user detection by bounding box (in case of a vehicle a license plate detection can be used instead), input image cropping, road user descriptor computation and road user classification. The image acquisition and the bounding box and/or license plate detection are not part of this SDK and must be solved separately, either by client's library or using Eyedea LPM library.

The process starts with the input image cropping with respect to the bounding box or license plate detection. The image crop is done using the SDK, it crops and transforms the image in the way that only the road user of interest is contained in the cropped image. The crop is the input of the machine learning algorithm, which is contained in the SDK's recognition module. The output is a descriptor (real number vector) describing the input vehicle in a condensed form. The descriptor is then classified, where the output is the classification result – human readable output of the road user recognition.



3) Image cropping

Input image cropping around the road user of interest.

4) Descriptor computation

Descriptor computation from the image crop.

11101110011011011010..

5) Classification

Descriptor classification and getting the MMR output.

view:	frontal
category:	CAR
make:	VW
model:	Golf
generation:	Mk VII (2013,2017)
variation:	---
color:	WHITE
tag_ambulance	no

2 Installation Guide

Installation of the software licensing daemon is the first step to start using the MMR SDK. The library comes equipped with a standard third-party software licensing solution, Sentinel LDK by Thales. This chapter will guide the client through installation on Windows and Linux. In the process, the client will install a daemon service, Sentinel License Manager, that will automatically start upon system startup. The application enables execution of the encrypted MMR SDK binaries, and management of licenses using a web browser.

2.1 Pre-installation

Prior to the installation of the licensing software, all Sentinel Hardware Keys should be removed from the target computer based on the recommendation from Thales. Leaving it connected during the installation process might cause the Sentinel Hardware Key to not be properly recognized by the new installation of Sentinel License Manager.

Sentinel License Manager does not support read only filesystems (on Windows, the functionality is called *Enhanced Write Filter*).

2.2 Sentinel LDK Installation

2.2.1 Windows

Follow these steps to install Sentinel License Manager on a Windows machine:

- Start the command line “**cmd**” with **Administrator** privileges.
- Navigate to the **[MMR_SDK]/hasp/** directory.
- Execute “**dunst.bat**” to uninstall any previous versions of Sentinel License Manager.
- Execute “**dinst.bat**” to install Sentinel License Manager.

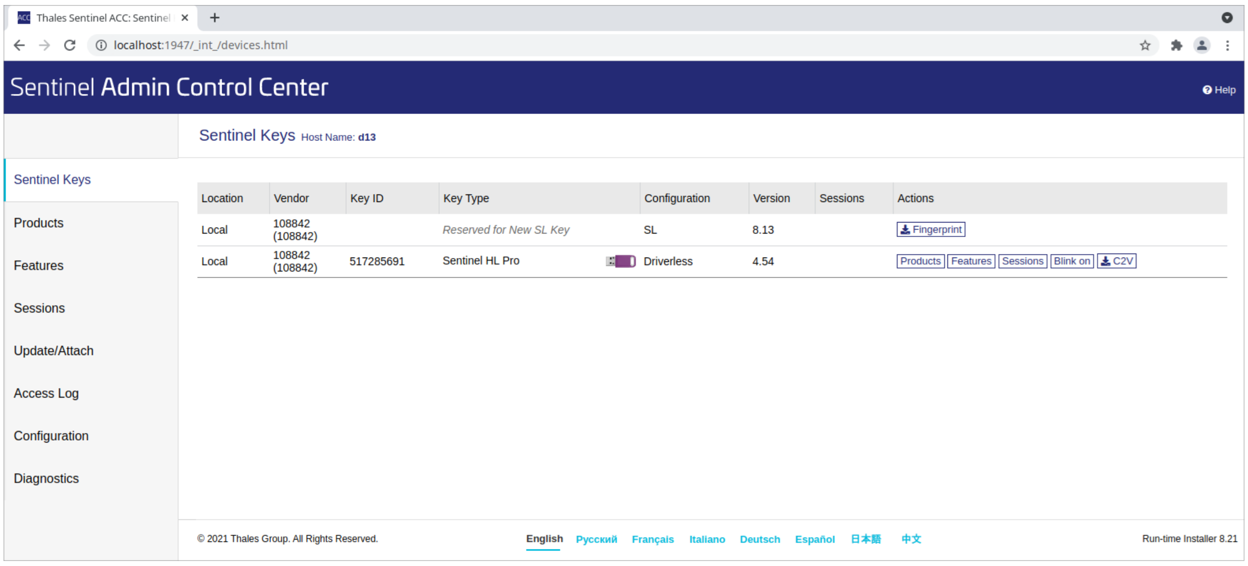
2.2.2 Linux

Follow these steps to install Sentinel License Manager on a Linux machine:

- Start the command line and navigate to the **[MMR_SDK]/hasp/** directory.
- On 64-bit Linux distributions, install the **32-bit** compatibility binaries.
  - On Ubuntu 18.04 and higher: Execute “**sudo apt-get install libc6:i386**”.
- Execute “**sudo ./dunst**” to uninstall any previous versions of Sentinel License Manager.
- Execute “**sudo ./dinst**” to install Sentinel License Manager.
  - Without compatibility binaries, error “*No such file or directory.*” might appear.

2.3 Verification of Installation

The software licensing daemon contains a web-based interface, which also allows the client to check the available licenses. To verify that the installation of Sentinel License Manager was successfully completed, the client should open a web browser at http://localhost:1947/_int_/devices.html. The web page will be displayed, as seen in the image below. The client must check that the trial licenses were installed properly, and that the MMR SDK works on the machine, before ordering a full license. If not, a problem may arise in the future when connecting the full license, resulting in a licensing failure and additional costs to relicense the software to another machine. The web page lists all available license keys. Under the “**Products**” link in the left pane all available products are listed.



Sentinel Keys Host Name: **d13**

Location	Vendor	Key ID	Key Type	Configuration	Version	Sessions	Actions
Local	108842 (108842)		*Reserved for New SL Key*	SL	8.13		Fingerprint
Local	108842 (108842)	517285691	Sentinel HL Pro	Driverless	4.54		Products Features Sessions Blink on C2V

© 2021 Thales Group. All Rights Reserved.

Run-time Installer 8.21

Sentinel License Manager screenshot.

2.4 Installation Failures

On Windows, antivirus application might break the installation of Sentinel License Manager. If the installation failed, the client should disable the antivirus application and rerun the installation of Sentinel License Manager. Even after successful installation, Sentinel License Manager might fail to show up in the web browser. This can be solved by adding

```
C:\Windows\system32\hasplms.exe
```

to the exception list of the antivirus. Port number **1947** must be also added to the exception list of the Windows firewall, and also to the antivirus exception list, if it uses its own firewall.

2.5 Managing Licenses

It is of the utmost importance that the client understands the licensing schemes used in the Thales Sentinel LDK software protection framework. Otherwise, unrepairable damage might be caused, leading to additional costs to recover the already purchased licensing keys. The topic of license management is fully covered in the chapter *MMR SDK Licensing*.

2.6 License Error Codes

Error codes are outputted to the error stream of the application (typically *stderr*) using MMR SDK. The user needs to check the error stream for error codes and fix the issues before deployment. The following error codes and messages are the most common ones:

- **H0007** – Sentinel HASP key not found. (No license for the MMR SDK on the PC.)
- **H0033** – Unable to access Sentinel HASP Runtime Environment. (No License Manager found.)
- **H0041** – Feature has expired. (The license on the PC has expired, consider renewal.)

The shared library of MMR SDK is encrypted for enhanced software protection. However, in case of failure, the application does not terminate, but crashes after a few calls to the library; this is a security measure against reverse engineering but may confuse the users. The client needs to make sure they monitor the error codes outputted by the error stream to distinguish between programming errors and licensing problems.

2.7 TensorRT

The MMR SDK can use TensorRT to run detection and OCR models, SDK package contains data files and command-line utility which can be used to generate TensorRT model for specific target device.

2.7.1 TensorRT MMR SDK Models

For devices with Nvidia GPUs, when TensorRT GPU mode is set, the classifiers cannot be prepared in advance and the folder

```
[MMR_SDK]/sdk/modules/edftrt-mmr/model
```

does not include prebuilt .dat files, but only their prototypes. Before running the software for the first time on a specific Nvidia GPU device type, the .dat files must be created using an utility called **edftrt_dat_encoder** which should be located in the **[MMR_SDK]/tools** directory. For example, if the client has 100 identical devices, they only need to follow this process once and then share the created .dat files among the devices.

To run the **edftrt_dat_encoder** utility, the client needs to make sure the relevant Nvidia TensorRT libraries are visible in the system, which can be checked using **ldd** utility as "**ldd edftrt_dat_encoder**". If not found, the Nvidia TensorRT need to be added to the library path using the following command like:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/lib/$(uname -m)-linux-gnu/
```

The **edftrt_dat_encoder** utility must be executed when there is no other process utilizing resources on the target device, otherwise the created .dat files will not give the best possible performance. By default, the generated .dat files use float32 (FP32) computation mode. Using float16 (FP16) computation mode evaluation speed can be improved, but the effect on accuracy needs to be verified. Use parameter "-h" with the **edftrt_dat_encoder** utility to see all options, run the utility without any options to use defaults. Conversion can take several minites depending on the specific device type. Warnings might appear during the generation which can be ignored.

2.7.2 Generating Device Specific Models

Here is an example of a command that can be used from inside the models directory:

```
./edftrt_dat_encoder -p=./ -w=2048 -q=FP16
```

The "-p" argument denotes the path in which the utility will look for model prototypes (file triples with extensions .dat.pre, .dat.net, .dat.post) to make optimized .dat files from, "-q" sets the quantization, and "-w" sets the workspace size - see the official NVIDIA TensorRT documentation (docs.nvidia.com/deeplearning/tensorrt/api/c_api/) for the function **IBuilderConfig::setMaxWorkspaceSize** for more information about this parameter.

2.7.3 Known Issues

As of Nvidia TensorRT 8.2, there are still documented known issues in Nvidia TensorRT library that can cause the generated .dat files to lose accuracy or completely misbehave. It is up to the customer to verify the newly created .dat files give expected performance, for example by comparing with the results of MMR SDK CPU version.

2.8 OpenGL Prerequisites

For Nvidia Jetson devices, we also provide MMR SDK with Tensorflow Lite backend, which utilizes OpenGL for GPU processing. To be able to use GPU, the Jetson SD card image must be installed with **nvidia-l4t-3d-core** package, described as "*NVIDIA GL EGL Package*". This package is installed during the default installation of Nvidia JetPack. When using a remote shell to connect to a device where the client wants to be using OpenGL GPU mode, X forwarding must be turned off.

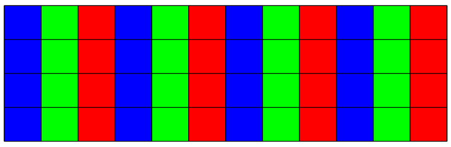
3 ERImage Application Interface

This part describes ERImage library used for image data storage and manipulation by MMR SDK. The *Image Format* section describes how image data is stored in the memory from a theoretical point of view, and the remaining parts cover the application interface used for image manipulation using the data structure *ERImage*. Description of all available *Enumerators*, *Structures* and *Functions* is included.

3.1 Image Format

Digital image data can be persisted in many different forms. Since it is the main input of the processing, it is very important to understand the form used for image storage and manipulation. Currently five color models are supported in the *ERImage* image structure. The first is the *BGR* color model, the second is the *Gray* color model, the third is the *YCbCr I420* color model, the fourth is the *BGRA* color model, and the fifth is the *YCbCr NV12* color model.

3.1.1 BGR



Three-channel model, which is derived from RGB, and is supported by the *ERImage* is *BGR* (B – blue, G – green, R – red). *BGR* (B – blue, G – green, R – red) is a three-channel model supported by *ERImage*; it is derived from RGB.

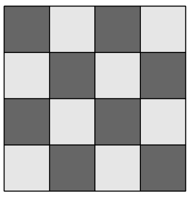
The model stores image using three values per pixel, where the first value is the blue component, the second value is the green component and the third is the red component. An image is saved row by row in a 1D array. The following formulas show how to access the pixel color components B, G and R in the 1D array *data* of the image with resolution *width × height* on coordinates *(x, y)*. Coordinates *x, y* and *data* array indices are 0-based.

$$B(x,y) = data(3 * (width * y + x) + 0)$$ B component at (x, y) coordinates

$$G(x,y) = data(3 * (width * y + x) + 1)$$ G component at (x, y) coordinates

$$R(x,y) = data(3 * (width * y + x) + 2)$$ R component at (x, y) coordinates

3.1.2 Gray

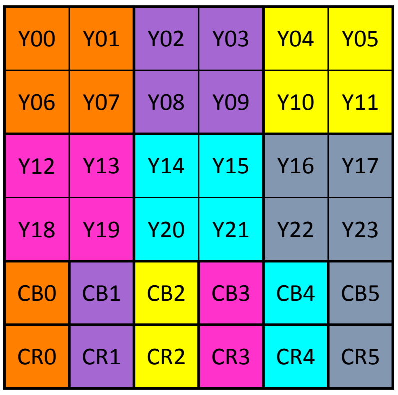


The one-channel model *Gray* is used for storing grayscale images, which are composed of luminance values (Y - luminance). The model stores images using one value per pixel, where the value is the luminance component. The image is saved row by row in a 1D array. The following formula shows how to access the pixel luminance component Y in the 1D array *data* of an image with resolution *width × height* at coordinates *(x, y)*. Coordinates *x, y* and *data* array indices are 0-based.

$$Y(x,y) = data(width * y + x)$$ Y component at (x, y) coordinates

3.1.3 YCbCr I420

The three-plane model *YCbCr I420* is used for storing color image, where the first plane contains luminance (Y component, image brightness), the second plane contains the blue-difference chroma component (Cb) and the third plane contains the red-difference chroma component (Cr). Cb and Cr planes have half the resolution of the Y image plane. Four neighboring Y values belongs to one Cb and one Cr value.



The image is saved per plane, where each plane is saved row by row in a 1D array. The following formulas show how to access the pixel color components Y, Cb and Cr in the 1D array *data* of an image with resolution *width* × *height* at coordinates *(x, y)*. Coordinates *x, y* and *data* array indices are 0-based. **All divisions in the formulas are integer divisions.**

$$Y(x,y) = data(width * y + x)$$ Y component at (x, y) coordinates

$$|Y| = width * height$$ Size of the Y image plane

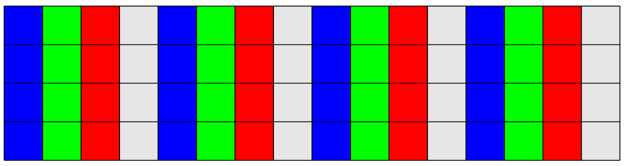
$$Cb(x,y) = data(|Y| + \frac{y}{2} * \frac{width}{2} + \frac{x}{2})$$ Cb component at (x, y) coordinates

$$|Cb| = |Cr| = \frac{width * height}{4}$$ Size of the Cb and Cr image plane

$$Cr(x,y) = data(|Y| + |Cb| + \frac{y}{2} * \frac{width}{2} + \frac{x}{2})$$ Cr component at (x, y) coordinate

3.1.4 BGRA

BGRA (B – blue, G – green, R – red, A – alpha) is a four-channel model supported by the *ERImage*; it is derived from *RGBA*. The model stores images using four values per pixel, where the first value is the blue component, the second value is the green component, the third is the red component and the fourth value is the alpha component (transparency). An image is saved row by row in a 1D array. Following formulas show how to access the pixel color components B, G, R and A in the 1D array *data* of an image with resolution *width* × *height* at coordinates *(x, y)*. Coordinates *x, y* and *data* array indices are 0-based.



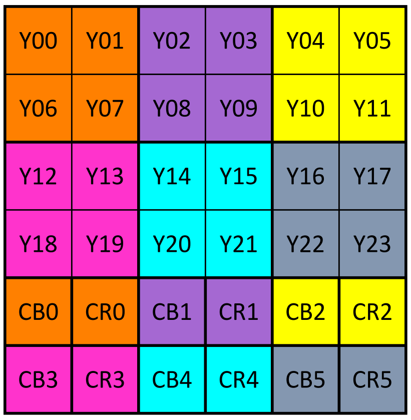
$$B(x,y) = data(4 * (width * y + x) + 0)$$ B component at (x, y) coordinates

$$G(x,y) = data(4 * (width * y + x) + 1)$$ G component at (x, y) coordinates

$$R(x,y) = data(4 * (width * y + x) + 2)$$ R component at (x, y) coordinates

$$A(x,y) = data(4 * (width * y + x) + 3)$$ A component at (x, y) coordinates

3.1.5 YCbCr NV12



The two-plane model *YCbCr NV12* is used for storing color images, where the first plane contains luminance (Y component, image brightness) and the second plane contains interleaved blue-difference chroma components (Cb) and red-difference chroma components (Cr). The Cb and Cr planes have half the height and the same width as the Y image plane (because there are two components). Four neighboring Y values belongs to one Cb and one Cr value.

The image is saved per plane, where each plane is saved row by row in a 1D array. The following formulas show how to access the pixel color components Y, Cb and Cr in the 1D array *data* of the image with resolution *width × height* at coordinates *(x, y)*. Coordinates *x, y* and *data* array indices are 0-based. **All divisions in the formulas are integer divisions**.

$$Y(x,y) = data(width * y + x)$$
Y component at (x, y) coordinates

$$|Y| = width * height$$
Size of the Y image plane

$$Cb(x,y) = data(|Y| + \frac{y}{2} * width + \frac{x}{2})$$
Cb component at (x, y) coordinates

$$Cr(x,y) = data(|Y| + \frac{y}{2} * width + \frac{x}{2} + 1)$$
Cr component at (x, y) coordinate

$$|CbCr| = width * \frac{height}{2}$$
Size of the CbCr image plane

3.2 Application Interface

3.2.1 Enumerators

This part defines the API enumerators which are related to the *ERImage* structure:

ERImageColorModel

ERImageColorModel is used to specify how the color channel values are saved in the image. More information about the supported color models is in the section *Image Format*.

- **ER_IMAGE_COLORMODEL_UNK = 0**
  Default value - Unknown color model.
- **ER_IMAGE_COLORMODEL_GRAY = 1**
  One-channel grayscale color model. Image luminance values are saved row by row.
- **ER_IMAGE_COLORMODEL_BGR = 2**
  Three-channel *BGR* color model. Three values per pixel, stored row by row.
- **ER_IMAGE_COLORMODEL_YCBCR420 = 3**
  Three-plane *YCbCr I420* color model. Luminance plane and two chroma planes are stored separately, each row by row.
- **ER_IMAGE_COLORMODEL_BGRA = 4**
  Four-channel *BGRA* color model. Four values per pixel, stored row by row.

- **ER_IMAGE_COLORMODEL_YCBCRNV12 = 5**
  Two-plane *YCbCr NV12* color model. Luminance plane and interleaved chroma plane are stored separately each row by row.

ERImageDataType

ERImageDataType specifies the data type used for storing values of the image.

- **ER_IMAGE_DATATYPE_UNK = 0**
  Default value – unknown data type.
- **ER_IMAGE_DATATYPE_UCHAR = 1**
  All image values are saved as unsigned char.
- **ER_IMAGE_DATATYPE_FLOAT = 2**
  All image values are saved as float.

3.2.2 Structures

This part defines the API structure *ERImage* used for digital image data manipulation:

ERImage

```
typedef struct {
    ERImageColorModel   color_model;
    ERImageDataType     data_type;
    unsigned int        width;
    unsigned int        height;
    unsigned int        num_channels;
    unsigned int        depth;
    unsigned int        step;
    unsigned int        size;
    unsigned int        data_size;
    unsigned char*      data;
    unsigned char**     row_data;
    unsigned char       data_allocated;
} ERImage;
```

ERImage represents the digital image data in a special structure designed to work with the MMR SDK. The structure contains the color model and the data type in the *ERImageColorModel*, and the *ERImageDataType* enumerators, together with the parameters defining the size of the image and the underlying data. Image data is saved in the data field row by row as a contiguous 1D array. For more information see the section *Image Format*.

- **color_model**
  Image data color model represented by the enumerator *ERImageColorModel*.
- **data_type**
  Image date type represented by the enumerator *ERImageDataType*.
- **width**
  Width of the image in pixels.
- **height**
  Height of the image in pixels.
- **num_channels**
  Number of image channels. Zero for YCbCr color models.
- **depth**
  Size of one image pixel in bytes. Zero for YCbCr color models.
- **step**
  Number of bytes between each two beginnings of the row in the data array.

- **size**
  Size of the image in bytes.
- **data_size**
  Size of the allocated data in the structure.
- **data**
  Array containing the image data.
- **Row_data**
  Array containing pointers to the data array. Each element points to the beginning of the specific image row in the data array.
- **data_allocated**
  Value containing the flag whether the data field was allocated within the structure or on the user's side. (0 – allocated by user, 1 – allocated within the structure)

3.2.3 Functions

This part defines the API functions which are designed to work with the *ERImage* structure:

- **Allocation**
  erImageAllocate, *erImageAllocateBlank*, *erImageAllocateAndWrap* and *erImageCopy*
- **Properties**
  erImageGetDataTypeSize, *erImageGetColorModelNumChannels*, *erImageGetPixelDepth* and *erVersion*
- **IO Operations**
  erImageRead and *erImageWrite*
- **Freeing**
  erImageFree

These functions are defined in the **er_image.h** file.

erImageAllocate

Allocates an *ERImage* structure.

Specification:

```
int erImageAllocate(ERImage* image, unsigned int width, unsigned int height,
                    ERImageColorModel color_model, ERImageDataType data_type);
```

Input:

- **image**
  Pointer to the *ERImage* structure instance to allocate.
- **width**
  Width of the image to allocate.
- **height**
  Height of the image to allocate.
- **color_model**
  Color model of the image to allocate (see *ERImageColorModel*).
- **data_type**
  Data type of the image to allocate (see *ERImageDataType*).

Returns:

- **0** – Image successfully allocated.
- **other** – Error during image allocation.

Description:
The function *erImageAllocate*() is used for ERImage structure data allocation. The input of the function is the pointer to the *ERImage* structure instance, the width and height of the image to allocate, and the color model and the data type specification.

Example:
```
ERImage* image = new ERImage();
// Allocate grayscale (1 channel) image with resolution 800x600 and 1 byte per channel
int res = erImageAllocate(image, 800, 600, ER_IMAGE_COLORMODEL_GRAY, ER_IMAGE_DATATYPE_UCHAR);
```

erImageAllocateBlank

Allocates an *ERImage* structure without allocating the internal data arrays.

Specification:
```
int erImageAllocateBlank(ERImage* image, unsigned int width, unsigned int height,
                         ERImageColorModel color_model, ERImageDataType data_type);
```

Input:
- **image**
  Pointer to the *ERImage* structure instance to allocate.
- **width**
  Width of the image to allocate.
- **height**
  Height of the image to allocate.
- **color_model**
  Color model of the image to allocate (see *ERImageColorModel*).
- **data_type**
  Data type of the image to allocate (see *ERImageDataType*).

Returns:
- **0** – Image successfully allocated.
- **other** – Error during image allocation.

Description:
The function *erImageAllocateBlank*() is used for *ERImage* structure properties allocation, but without the internal data array allocation. The input of the function is the pointer to the *ERImage* structure instance, the width and height of the image to allocate, and the color model and the data type specification.

Example:
```
ERImage* image = new ERImage();
// Allocate blank BGR (3 channel) image with resolution 640x480 and 1 float per channel
int res = erImageAllocateBlank(image,640,480, ER_IMAGE_COLORMODEL_BGR, ER_IMAGE_DATATYPE_FLOAT);
// image->data == NULL, image->row_data == NULL and image->data_size == 0
```

IMPORTANT: Only the fields with image properties are allocated. Image data field is NULL, row_data is NULL and field data_size is 0 after a successful function call.

erImageAllocateAndWrap

Allocates an *ERImage* structure and wraps it over the supplied image data.

Specification:

```
int erImageAllocateAndWrap(ERImage* image, unsigned int width, unsigned int height,
                           ERImageColorModel color_model, ERImageDataType data_type,
                           unsigned char* data, unsigned int step);
```

Input:

- **image**
  Pointer to the *ERImage* structure instance to allocate.
- **width**
  Width of the image to allocate.
- **height**
  Height of the image to allocate.
- **color_model**
  Color model of the image to allocate (see *ERImageColorModel*).
- **data_type**
  Data type of the image to allocate (see *ERImageDataType*).
- **data**
  Image data to wrap.
- **step**
  Definition of the input data image row step. (length of one image row in bytes in the input data)

Returns:

- **0** – Image successfully allocated.
- **other** – Error during image allocation.

Description:

The function *erImageAllocateAndWrap*() is used for *ERImage* structure data allocation and wrapping of the supplied image data. The input of the function is the pointer to the *ERImage* structure instance, the width and height of the image to allocate, the color model and the data type specification, the pointer to the image data to wrap, and the step value which defines the size of the row in bytes.

Example:

```
unsigned char* data; // Image data to wrap
ERImage* image = new ERImage();
// Allocate grayscale (1 channel) image with resolution 800x600 and 1 byte per channel
// and wrap it over the image data supplied in the unsigned char* data array.
int res = erImageAllocateAndWrap(image, 800, 600, ER_IMAGE_COLORMODEL_GRAY,
                                 ER_IMAGE_DATATYPE_UCHAR, data, 800);
```

erImageCopy

Performs a deep copy of the *ERImage* structure instance.

Specification:

```
int erImageCopy(const ERImage* image, ERImage* image_copy);
```

Input:

- **image**
  Pointer to the *ERImage* structure instance to copy.
- **image_copy**
  Pointer to the *ERImage* structure to copy the data into.

Returns:

- **0** – Image successfully copied.
- **other** – Error during image copying.

Description:
The function *erImageCopy*() is used for *ERImage* data copying to another instance of an *ERImage* structure. The input is the pointer to the *ERImage* structure instance to copy and the output is the pointer to the *ERImage* structure instance to copy the data into.

IMPORTANT: The allocation of image_copy is done within the function before the data copying.

Example:

```
ERImage* image;                      // Image with source data
ERImage* image_copy = new ERImage(); // Destination image to copy the data into
// Deep copy of the image
int res = erImageCopy(image, image_copy);
```

erImageGetDataTypeSize

Returns the size of the specific *ERImageDataType* in bytes.

Specification:

```
unsigned int erImageGetDataTypeSize(ERImageDataType data_type);
```

Input:

- **data_type**
  ERImageDataType to get the size of.

Returns:

- **data type size** – Size of one channel image element in bytes.
- **0** – Unknown *ERImageDataType* used.

Description:
The function *erImageGetDataTypeSize*() is used to get the size in bytes of the specific *ERImageDataType* when used for image allocation. The input is the *ERImageDataType* value. The output is the value which represents the number of bytes needed for storing one channel value of one pixel when a specific *ERImageDataType* is used.

Example:

```
unsigned int sizeUC = erImageGetDataTypeSize(ER_IMAGE_DATATYPE_UCHAR);
// sizeUC == sizeof(unsigned char)

unsigned int sizeF  = erImageGetDataTypeSize(ER_IMAGE_DATATYPE_FLOAT);
// sizeF  == sizeof(float)
```

erImageGetColorModelNumChannels

Returns the number of channels of the provided *ERImageColorModel* value.

Specification:

```
unsigned int erImageGetColorModelNumChannels(ERImageColorModel color_model);
```

Input:

- **color_model**
  ERImageColorModel to get the number of channels.

Returns:

- **number of channels** – Number of channels of the supplied color model.
- **0** – Unknown or *YCbCr* *ERImageColorModel* used.

Description:
The function *erImageGetColorModelNumChannels*() is used to get the number of channels of the specific *ERImageColorModel*. The input is the *ERImageColorModel* value. The output is the value which represents the number color model channels used when storing the image with specific *ERImageColorModel*.

IMPORTANT: For the ER_IMAGE_COLORMODEL_YCBCR* color model, zero is returned.

Example:

```
unsigned int numChannelsGRAY   = erImageGetColorModelNumChannels}(ER_IMAGE_COLORMODEL_GRAY);
// numChannelsGRAY    == 1

unsigned int numChannelsBGR    = erImageGetColorModelNumChannels}(ER_IMAGE_COLORMODEL_BGR);
// numChannelsBGR     == 3

unsigned int numPlanesYCBCR420 = erImageGetColorModelNumChannels}(ER_IMAGE_COLORMODEL_YCBCR420);
// numPlanesYCBCR420  == 0
```

erImageGetPixelDepth

Returns the size of a pixel in bytes for the supplied *ERImageColorModel* and *ERImageDataType*.

Specification:

```
unsigned int erImageGetPixelDepth(ERImageColorModel color_model, ERImageDataType data_type);
```

Input:

- **color_model**
  Input *ERImageColorModel* for pixel depth computation.
- **data_type**
  Input *ERImageDataType* for pixel depth computation.

Returns:

- **depth of the pixel** – Number of bytes needed to store one pixel using the specified color model and data type.
- **0** – Unknown *ERImageColorModel* and/or *ERImageDataType* used.

Description:
The function *erImageGetPixelDepth*() is used to get the size of one pixel in bytes for the combination of *ERImageColorModel* and *ERImageDataType*. The input is the *ERImageColorModel* and *ERImageDataType* values. The output is the value which represents the size of one pixel in bytes when storing an image with the supplied *ERImageColorModel* and *ERImageDataType*.

IMPORTANT: For the ER_IMAGE_COLORMODEL_YCBCR* color model, zero is returned.

Example:

```
unsigned int dUCGray = erImageGetPixelDepth(ER_IMAGE_COLORMODEL_GRAY, ER_IMAGE_DATATYPE_UCHAR);
// dUCGray == 1

unsigned int dFBGR   = erImageGetPixelDepth(ER_IMAGE_COLORMODEL_BGR, ER_IMAGE_DATATYPE_FLOAT);
// dFBGR   == 3*sizeof(float)
```

erVersion

Returns the version of the *ERImage* structure and all related image utilities.

Specification:

```
const char* erVersion(void);
```

Returns:

- version of the *ERImage* – String containing the version of the *ERImage*.

Description:

The function *erVersion*() is used to get the version of the *ERImage* structure and all related image utilities. The function returns a string which contains the version number.

Example:

```
const char* version = erVersion();
std::cout << "ERImage␣version:␣" << version << std::endl;
```

erImageRead

Reads the image from a file, decodes it, and loads it into the supplied *ERImage* structure instance.

Specification:

```
int erImageRead(ERImage* image, const char* filename);
```

Input:

- **image**
  Pointer to the *ERImage* structure instance to load the image into.
- **filename**
  String containing the path to the image file to read.

Returns:

- **0** – Image successfully read.
- **other** – Error during image reading.

Description:

The function *erImageRead*() is used to read and decode the image from the given file and load it into the supplied *ERImage* structure instance. The input is the pointer to the *ERImage* instance and the string containing the path to the image file to open.

Supported image formats:

JPEG files	*.jpeg, *.jpg, *.jpe
JPEG 2000 files	*.jp2
Portable Network Graphics	*.png
Windows bitmaps	*.bmp, *.dib
TIFF files	*.tiff, *.tif
Portable image format	*.pbm, *.pgm, *.ppm *.pxm, *.pnm

Example:

```
char* filename = "./image.jpg";          // Image file path to read
ERImage* image = new ERImage();          //hrefi Initialize the ERImage
int res = erImageRead(image, filename); // Read the image
```

erImageWrite

Encodes and writes the image from the *ERImage* structure to a file.

Specification:

```
int erImageWrite(const ERImage* image, const char* filename);
```

Input:

- **image**
  Pointer to the *ERImage* structure instance containing the image to write.
- **filename**
  String containing the path to the image file to write.

Returns:

- **0** – Image successfully written.
- **other** – Error during image writing.

Description:

The function *erImageWrite*() is used to encode and write the image to the given file from the *ERImage* structure instance. The input is the pointer to the *ERImage* instance and the string containing the path to the image file to write. Output image format is automatically selected from the filename extension with respect to the table of supported formats in the *erImageRead* chapter.

Example:

```
char* filename = "./image.jpg";           // Image file path to write
ERImage* image;                           // ERImage containing the image to write
int res = erImageWrite(image, filename); // Write the image
```

erImageFree

Frees the whole *ERImage* structure instance.

Specification:

```
void erImageFree(ERImage* image);
```

Input:

- **image**
  Pointer to the *ERImage* structure instance to delete.

Description:

The function *erImageFree*() is used to free the image data arrays contained in the *ERImage* structure instance and also to set all the property fields to 0. The input is the pointer to the *ERImage* instance you wish to free.

IMPORTANT: The function DOES NOT delete the ERImage instance pointer because the user creates the pointer.

Example:

```
erImageAllocate(image, 800, 600, ER_IMAGE_COLORMODEL_GRAY, ER_IMAGE_DATATYPE_UCHAR);
// ...
erImageFree(image); // every field in the image structure is freed and set to NULL or 0
```

4 ER Types

The purpose of the **er_types.h** header file is to define common C data structures and enumerations used in Eyedea Recognition's libraries for computer vision and image processing.

4.1 Enumerators

ERComputationMode

ERComputationMode defines the different modes of computation that can be used. The mode is used in the SDK instance initialization to specify the mode in which the SDK instance computation will operate on.

- **ER_COMPUTATION_MODE_GPU = 0**
  CPU is exclusively used for all computations inside the SDK instance.
- **ER_COMPUTATION_MODE_GPU = 1**
  The most expensive computations are processed on the computer's GPU (graphics processing unit), which gives significant processing speed-up. This functionality is available in the GPU version of the SDK only. For NVIDIA Jetson, OpenGL is used, for all other release packs OpenCL capable GPU is required.
- **ER_COMPUTATION_MODE_TPU = 2**
  Reserved enumeration for computation on specialized hardware (Tensor Processor Unit)

4.2 Structures

ERPoint2i

```
typedef struct {
        int x;
        int y;
} ERPoint2i;
```

ERPoint2i defines 2D coordinates of a pixel in an image specified by its index cordinates x and y.

ERPoint2f (ERPoint)

```
typedef struct {
        float x;
        float y;
} ERPoint2f;
```

ERPoint2f defines 2D floating point coordinate.

ERRoI

```
typedef struct {
        int x;
        int y;
        int width;
        int height;
} ERRoI;
```

This structure defines a region of interest (ROI) within an image. The ROI is defined by top-left corner coordinates and its size. It is possible define ROI starting outside of the image using negative values of x,y or define "full" width, height with respect to the image by negative values. The *ERRoI* has following fields:

- **x**
  the x-coordinate (index) of the top left pixel of the ROI.
- **y**
  the y-coordinate (index) of the top left pixel of the ROI
- **width**
  the width of the ROI. A negative value means the ROI should cover the full width of the image.
- **height**
  the height of the ROI. A negative value means the ROI should cover the full height of the image.

ERRotatedRect

```
typedef struct {
        float x;
        float y;
        float width;
        float height;
        float angle;
} ERRotatedRect;
```

This structure defines a rotated rectangle in a 2D coordinate system by its mass center, width, height and angle of rotation. The *ERRotatedRect* has following fields:

- **x**
  the x-coordinate (index) of the mass center.
- **y**
  the y-coordinate (index) of the mass center.
- **width**
  the width of the rectangle
- **height**
  the height of the rectangle
- **angle**
  the clockwise rotation angle (in degrees) of the rectangle. When the angle is 0, 90, 180, 270 etc., the rectangle becomes an up-right rectangle.

4.3 Functions

erRotatedRectToPoints

Helper function to convert *ERRotatedRect* structure to *ERPoint2f*[4] array. The function returns corner points of the rotated rectangle in clockwise direction starting with top-left corner.

Specification:

```
        int erRotatedRectToPoints(const ERRotatedRect* rect, ERPoint2f (*points)[4]);
```

Inputs:

- rectPointer to the *ERRotatedRect* structure
- pointsAllocated array of *ERPoint2f* structure with at least 4 elements.

Returns:

- **0** – On success.
- **other** – Error during conversion.

5 SDK Application Interface

This chapter describes all the parts of the SDK's public application interface for C/C++ programming language including defined *Structures* and all available *Functions*. It gives developer detailed overview of the SDK and helps to orientate during SDK integration.

ERImage structure and image manipulation functions are described in section *ERImage*. Examples of MMR SDK usage are in *Examples* section.

5.1 Structures

Document section Structures covers all the information about structures used in the SDK's public application interface. *EdfInitConfig* is used during the SDK initialization, *EdfDescriptor* stores the unique description of the specified object, *EdfPoints*, *EdfValues* and *EdfCropParams* are used during input image preparation, *EdfClassifyResultValue* and *EdfClassifyResult* store the results of the classification process, *EdfCropImageConfig* specifies the configuration of the input image cropping, *EdfComputeDescConfig* specifies the configuration of the descriptor computation and *EdfClassifyConfig* specifies the configuration of the descriptor classification.

EdfInitConfig

```
typedef struct {
    const char*         module_path;
    const char*         model_file;
    ERComputationMode   computation_mode;
    int                 gpu_device_id;
    int                 num_threads;
} EdfInitConfig;
```

EdfInitConfig represents the configuration parameters set used during SDK module initialization. The structure contains following fields:

- **module_path**
  NULL terminated string with the path to the module.
  Example: "*C:/folder/Eyedea-MMR-2.24/sdk/modules/edftf2lite/*"
- **model_file**
  NULL terminated string with model filename.
  Example: "*MMR_VCMMCT_PREC_2024Q4.dat*" or "*MMRBOX_VCMMGVCT_FAST_2024Q4.dat*".
- **computation_mode**
  Selected computation mode from *ERComputationMode* enumerator..
- **gpu_device_id**
  Zero based integer representing GPU device identifier. Use "*tools/gpu-devices*" to list available GPUs and their IDs.
  (used only when computation_mode == ER_COMPUTATION_MODE_GPU)
- **num_threads**
  Sets the number of threads used by the module's backend. The number of threads is used in all computation modes, for GPU and TPU mode, only affects CPU code blocks.
  Set to **-1** to use *std::thread::hardware_concurrency* value (number of virtual CPU cores).
  Set to **>1** to use *min(num_threads, std::thread::hardware_concurrency)*.
  Others (**<=1** except **-1**) initializes backend in a single thread (default).

EdfDescriptor

```
typedef struct {
        unsigned int   version;
        unsigned int   size;
        unsigned char* data;
} EdfDescriptor;
```

EdfDescriptor contains the data which describes the recognized object. The data is called the descriptor, which is generated as the output of the machine learning algorithms. The advantage of this approach is that the descriptor is usually smaller than the input image and it is easier and more efficient to compare the descriptors than the input images. The structure contains following fields:

- **version**
  Version of the binary model used for the descriptor computation.
- **size**
  Size of the descriptor data in bytes.
- **data**
  The descriptor data.

EdfPoints

```
typedef struct {
        int     length;
        double* rows;
        double* cols;
} EdfPoints;
```

EdfPoints contains the 2D points defined by the row (y-axis) and column (x-axis) coordinates. The structure contains following fields:

- **length**
  Number of the contained 2D points.
- **rows**
  Pointer to the array containing the row (y-axis) coordinates of the 2D points.
- **cols**
  Pointer to the array containing the column (x-axis) coordinates of the 2D points.

EdfValues

```
typedef struct {
        int     length;
        double* values;
} EdfValues;
```

EdfValues contains real value array. The structure contains following fields:

- **length**
  Number of contained real values.
- **values**
  Pointer to the array containing the real values.

EdfCropParams

```
typedef struct {
        EdfPoints points;
        EdfValues values;
} EdfCropParams;
```

EdfCropParams contains the parameters used for image cropping with the function *edfCropImage*. The contained parameters are internally stored in the *EdfPoints* and *EdfValues* structures. To fill this structure, use macros defined in **edf_type_mmr.h**. See subsection *Cropping the Input Image* of *Examples* section for more info. The structure contains following fields:

- **points**
  Instance of the *EdfPoints* structure.
- **values**
  Instance of the *EdfValues* structure.

EdfClassifyResultValue

```
typedef struct {
        char*        task_name;
        unsigned int task_name_length;
        char*        class_name;
        unsigned int class_name_length;
        int          class_id;
        float        score;
} EdfClassifyResultValue;
```

EdfClassifyResultValue represents the classification result value. The result value contains the name of the task (classifier name) and name of the class with the ID and the highest score. The structure contains following fields:

- **task_name**
  NULL terminated string containing the name of the classified task.
- **task_name_length**
  Length of the *task_name* string without the terminating NULL character.
- **class_name**
  NULL terminated string containing the name of the resulting class.
- **class_name_length**
  Length of the *class_name* string without the terminating NULL character.
- **class_id**
  Identifier of the resulting class.
- **score**
  Classification score of the resulting class.

EdfClassifyResult

```
typedef struct {
        unsigned int            num_values;
        EdfClassifyResultValue* values;
} EdfClassifyResult;
```

EdfClassifyResult contains the array with the *EdfClassifyResultValue* values as the result of the classification using the function *edfClassify*. The structure contains following fields:

- **num_values**
  Number of the contained *EdfClassifyResultValue* values.
- **values**
  Array containing the *EdfClassifyResultValue* values.

EdfCropImageConfig

```
typedef struct {
        int          full_crop;
        int          color_normalization;
        int          use_antialiasing;
        unsigned int antialiasing_kernel_size;
        float        antialiasing_sigma;
} EdfCropImageConfig;
```

EdfCropImageConfig is used for image cropping configuration in the function *edfCropImage*. Use this configuration only if you know what you are doing. The structure contains following fields:

- **full_crop**
  Specifies whether the image is returned with the boundary. (*Default: 0*)
  Set to **1** to create the full crop (with the border) – FOR DEBUGGING ONLY.
  Set to **0** to create default crop.
  Set to **-1** to create standard crop – FOR DEBUGGING ONLY.
- **color_normalization**
  Specifies whether the color normalization is applied. (*Default: 1*)
  Set to **1** to use color normalization – FOR DEBUGGING ONLY.
  Set to **0** to use color normalization default setting.
  Set to **-1** to not use color normalization – FOR DEBUGGING ONLY.
- **use_antialiasing**
  Specifies whether the antialiasing procedure is applied during the image transformation.
  Set to **1** to use antialiasing during image transformation – FOR DEBUGGING ONLY.
  Set to **0** to use antialiasing default setting.
  Set to **-1** to not use antialiasing during image transformation – FOR DEBUGGING ONLY.
  (Default antialiasing setting is loaded from the binary model and is specified for each task.)

  IMPORTANT: The antialiasing option can significantly improve recognition results, especially in the cases when the scale change during the cropping is high and aliasing occurs in the cropped image. Cropping time can be significantly higher with antialiasing option enabled. It is caused by image filtering which is computationally expensive. The computation time depends on the scale change during the cropping and on the size of the image area to be cropped.
- **antialiasing_kernel_size**
  The size of the convolution kernel used during antialiasing.
  Set to **0** to use the default convolution kernel size (computed from transformation scale).
  Used only in combination with *use_antialiasing == 1*. (*Default: 0*)
  FOR DEBUGGING ONLY.
- **antialiasing_sigma**
  The sigma parameter of the Gaussian distribution in the antialiasing convolution kernel.
  Set to **0.0f** to use the default sigma size (computed from kernel size).
  Used only in combination with *use_antialiasing == 1*. (*Default: 0.0f*)
  FOR DEBUGGING ONLY.

EdfComputeDescConfig

```
typedef struct {
       unsigned int batch_size;
} EdfComputeDescConfig;
```

EdfComputeDescConfig configures the descriptor computation in the function *edfComputeDesc*. The structure contains following fields:

- **batch_size**
  The size of the descriptors batch to compute.

  IMPORTANT: The advantage of the batch descriptor computation is the speed of the processing where the batch processing can be faster than sequential on some system configurations *(especially when using GPU computation mode)*.
  To use the batch descriptor computation in the function *edfComputeDesc*(), the input **image crops must be stored in the memory consecutively** *(like in an array or a vector)*. In the same way **must be initialized the memory for storing the computed descriptors.**

  IMPORTANT: Setting batch_size is not allowed in SDK 2.24 version.
  Set to **0** to disable batch processing.
  ~~Set to **1-N** to set the size of the batch (value 1 has the same effect as 0).~~

Example:

```
std::vector<EdfImage> imageCrops = { imageCrop1, imageCrop2, imageCrop3, imageCrop4 };
std::vector<EdfDescriptor> descriptors;
descriptors.resize(imageCrops.size());
EdfComputeDescConfig compDescConfig;
compDescConfig.batch_size = (unsigned int)imageCrops.size();
edfAPI.edfComputeDesc(&imageCrops[0], module_state, &descriptors[0], &compDescConfig);
```

EdfClassifyConfig

```
typedef struct {
       int use_dependency_rules;
} EdfClassifyConfig;
```

EdfClassifyConfig is used for descriptor classification configuration in the function *edfClassify*. Use this configuration only if you know what you are doing. The structure contains following field:

- **use_dependency_rules**
  Specifies whether the dependency rules between the classifiers are applied. (*Default: 0*)
  When the dependency rules are not applied, *task_name* field in returned classification output structure *EdfClassifyResultValue* contains string with *_NODEP* suffix.
  Set to **0** to get classification results with dependency rules applied.
  Set to **1** to get classification results both with and without dependency rules applied.
  Set to **-1** to get classification results without dependency rules applied.
  FOR DEBUGGING ONLY.

5.2 Functions

This chapter contains the definition of the MMR SDK library functions which are present in the public API. The chapter is divided into five parts. First describes the main API functions, the other refers to the functions designed for the manipulation with the API public data structure *EdfCropParams*.

IMPORTANT: The MMR SDK library **is not thread safe**. Do not call the library instance from multiple threads but initialize an instance of the library for each thread separately.

5.2.1 Main API

This part defines the API functions which are designed to control the main part of the Eyedentify recognition library. The functions are: *edfInitEyedentify*, *edfFreeEyedentify*, *edfCropImage*, *edfComputeDesc*, *edfCompareDescs*, *edfClassify*, *edfModelVersion*, *edfAllocDesc*, *edfFreeDesc*, *edfFreeCropImage*, *edfFreeClassifyResult*. These functions are defined in the **edf.h** file.

edfInitEyedentify

Initializes the Eyedentify module using supplied parameters.

Specification:

```
int edfInitEyedentify(const EdfInitConfig* init_config, void** module_state)
```

Inputs:

- **init_config**
  EdfInitConfig structure containing the parameters needed for initialization.

Outputs:

- **module_state**
  Pointer to the successfully initialized MMR SDK module instance.

Returns:

- **0** – Initialization was successful
- **other** – errno value or -1 for other errors.

Description:

The function *edfInitEyedentify*() is used for initialization of the MMR SDK module. The initialization is required to be able to use the library for recognition tasks. The input of the function call is the structure *EdfInitConfig* and the output is the error code and the pointer to the initialized module.

IMPORTANT: It is not possible to mix CPU and GPU computation mode in one process. Only one computation mode can be initialized within the process at a time.

IMPORTANT: The initialization of Eyedentify module must be done in the same thread which will run the module. Only one initialization function must be executed at a time in a single process, which requires mutexing. No other SDK function can be executed while MMR SDK module initialization is in progress in any thread.

GPU version only: To initialize the GPU version of the library, the ID of the GPU hardware must be supplied. The ID of the GPU is zero-based number. To get the list of the installed devices on your system, command line utility "*[EyedentifySDK]/tools/gpu-devices*" can be used. If only one GPU is installed ID 0 is used for initialization. Do not forget you can also have an integrated GPU on your system and always verify whether the correct GPU is used using e.g., a Task Manager, nvidia-smi, or similar GPU utilization measurement tool.

Example:

```
void* module_state = NULL;
EdfInitConfig initConfig;
initConfig.module_path      = EDF_MODULE_PATH;
initConfig.model_file       = MODEL_NAME;
initConfig.computation_mode = ER_COMPUTATION_MODE_CPU;
initConfig.gpu_device_id    = 0;
initConfig.num_threads      = 1;
int initResult = edfInitEyedentify(&initConfig, &module_state);
if (initResult != 0) {
       // Handle errors
}
```

edfFreeEyedentify

Frees initialized Eyedentify module.

Specification:

```
void edfFreeEyedentify(void** module_state)
```

Inputs:

- **module_state**
  Pointer to the pointer to the initialized MMR SDK module instance.

Description:
The function *edfFreeEyedentify*() is used for freeing the Eyedentify module. When the module is not needed anymore, for example at the end of the program, all underlying structures must be deallocated. The input of the function call is the pointer *module_state*, which was created using *edfInitEyedentify* function during module initialization.

IMPORTANT: Always free the module when it is not needed anymore otherwise your program will have memory leaks. Freeing the module in Sentinel LDK protected version frees the license key allocation.

Example:

```
void* module_state = NULL;
edfInitEyedentify(&initConfig, &module_state);
// ...
edfFreeEyedentify(&module_state);
```

edfCropImage

Prepares/crops the input image for processing.

Specification:

```
int edfCropImage(const ERImage* image_in, EdfCropParams* params, void* module_state,
                 ERImage* cropped_image, EdfCropImageConfig* config)
```

Inputs:

- **image_in**
  Input image stored in the *ERImage* structure.

- **params**
  Pointer to the cropping parameters stored in the *EdfCropParams* structure.
- **module_state**
  Pointer to the initialized MMR sDK module instance.
- **config**
  Cropping configuration parameters stored in the *EdfCropImageConfig* structure.
  For default configuration set to **NULL**.

Outputs:

- **cropped_image**
  Pointer to the successfully cropped and transformed image in the *ERImage*.

Returns:

- **0** – Cropping was successful.
- **other** – errno value or -1 for other errors.

Description:

The function *edfCropImage*() crops and transforms the input image according to the input cropping parameters, MMR SDK module and binary model requirements. It is used before the function *edfComputeDesc* to prepare the input image for the descriptor computation. The input of the function call is the pointer to the input image structure *ERImage*, void pointer *module_state*, cropping parameters defined using *EdfCropParams* structure and the cropping configuration in the *EdfCropImageConfig* structure. Specification of the configuration is optional, for default configuration use NULL pointer. The output is the error code and the pointer to the *ERImage* structure containing cropped image. Cropping parameters defined in the *EdfCropParams* structure are dependent on the MMR SDK module and recognition task. They allow to define 2D points and double values. For more detailed information see the *Examples* chapter and the example source codes included in the SDK package. The function *edfCropImage*() must be called for specific module.

Example:

```
// License plate-based crop
EdfCropParams cropParams;
edfCropParamsAllocate(1, 2, &cropParams);
cropParams.points.cols[0]   = 123.4;
cropParams.points.rows[0]   =  56.7;
cropParams.values.values[0] =   8.9;
cropParams.values.values[1] =   0.0;

ERImage cropImage;
int cropResult = edfCropImage(&image, &cropParams, module_state, &cropImage, NULL);
if (cropResult != 0) {
       // Handle errors
}
```

Example:

```
// Bounding box-based crop
EdfCropParams cropParams;
edfCropParamsAllocate(2, 0, &cropParams);
cropParams.points.cols[0]   = 123.4;
cropParams.points.rows[0]   =  56.7;
cropParams.points.cols[1]   = 453.8;
cropParams.points.rows[1]   = 230.5;

ERImage cropImage;
int cropResult = edfCropImage(&image, &cropParams, module_state, &cropImage, NULL);
if (cropResult != 0) {
       // Handle errors
}
```

edfComputeDesc

Computes descriptor from input image cropped by *edfCropImage* function.

Specification:

```
int edfComputeDesc(const ERImage* img, const void* module_state,
                   EdfDescriptor* descriptor, EdfComputeDescConfig* config)
```

Inputs:

- **img**
  Input image cropped by *edfCropImage* function.
- **module_state**
  Pointer to the initialized MMR SDK module instance.
- **config**
  Descriptor computation configuration in *EdfComputeDescConfig* structure.
  For default configuration set to **NULL**.

Outputs:

- **descriptor**
  Descriptor raw data and metadata saved in the *EdfDescriptor* structure.

Returns:

- **0** – Computation was successful.
- **other** – errno value or -1 for other errors.

Description:

The function *edfComputeDesc*() is used for machine learning method descriptor computation. The descriptor contains information describing input image in the condensed form, which is designed for efficient classification and matching. The input of the function call is the pointer to the cropped input image structure *ERImage* created with *edfCropImage*, void pointer *module_state*, which was created using *edfInitEyedentify* function during module initialization and the computation configuration in the *EdfComputeDescConfig* structure. Specification of the configuration is optional, for default configuration use NULL pointer. The output is the error code and the pointer to the *EdfDescriptor* structure.

IMPORTANT: Descriptor computation is the most time and resource consuming process which can last from tens to hundreds of milliseconds on standard x86 computer, depending on the recognition task complexity.

Example:

```
edfCropImage(&image, &cropParams, module_state, &cropImage, NULL);
// ...
EdfDescriptor descriptor;
int computeDescResult = edfComputeDesc(cropImage, module_state, &descriptor, NULL);
if (computeDescResult != 0) {
        // Handle errors
}
```

edfCompareDescs

Compares two descriptors and returns similarity score. Typically not used in standard scenarios.

Specification:

```
int edfCompareDescs(const EdfDescriptor* desc_A, const EdfDescriptor* desc_B,
                    void const* module_state, float* score)
```

Inputs:

- **desc_A**
  First descriptor for comparison stored in the *EdfDescriptor* structure.
- **desc_B**
  Second descriptor for comparison stored in the *EdfDescriptor* structure.
- **module_state**
  Pointer to the initialized MMR SDK module instance.

Outputs:

- **score**
  Pointer to the float where similarity score is saved on success.

Returns:

• **0**	–	Comparison was successful.
• **other**	–	errno value or -1 for other errors.

Description:
The function *edfCompareDescs*() compares two descriptors generated by *edfComputeDesc* with the same MMR SDK module and same binary model. Comparison function is not defined generally but is dependent on selected recognition task. The result of the comparison is a similarity score. For the similarity score the following rule is valid: the higher the score is the more similar two descriptors are. The similarity score minimal and maximal possible values are dependent on the MMR SDK module, binary model and on the recognition task, therefore cannot be defined in general. Typically they would be in range 0.0-1.0, although numerical precision can cause the value to be outside of this range. The input of the function call are the pointers to the descriptors to compare and the void pointer *module_state*. The output is the error code and the pointer to the similarity score stored as float.

IMPORTANT: The function *edfCompareDescs*() requires the data in the *EdfDescriptor* structure to be stored in the data field which points to the aligned memory, because SSE instructions are used for computation speedup. To ensure that the memory is aligned, use *edfComputeDesc* or *edfAllocDesc* functions for descriptor allocation.

Example:

```
edfComputeDesc(cropImageA, module_state, &descA, NULL);
edfComputeDesc(cropImageB, module_state, &descB, NULL);
// ...
float score = 0.0f;
int compareResult = edfCompareDescs(&descA, &descB, module_state, &score);
if (compareResult != 0) {
        // Handle errors
}
```

edfClassify

Classifies descriptor to the classes defined in the binary model.

Specification:

```
int edfClassify(const EdfDescriptor* desc, void* module_state,
                EdfClassifyResult** classify_result, EdfClassifyConfig* config)
```

Inputs:

- **desc**
  Descriptor for classification stored in the *EdfDescriptor* structure.
- **module_state**
  Pointer to the initialized MMR SDK module instance.

- **config**
  Descriptor classification configuration in *EdfClassifyConfig* structure.
  For default configuration set to **NULL**.

Outputs:

- **classify_result**
  Double pointer to the *EdfClassifyResult*, where classification result is saved on success.

Returns:

- **0** – Classification was successful.
- **other** – errno value or -1 for other errors.

Description:
The function *edfClassify*() classifies the descriptor to the classes defined in the binary model for the specific recognition task. The result of the classification is the name and the ID of one or more classes to which the object specified by the descriptor belongs. The input of the function call is the pointer to the descriptor to classify and the void pointer *module_state* and the classification configuration in the *EdfClassifyConfig* structure. Specification of the configuration is optional, for default configuration use NULL pointer. The output is the error code and the double pointer to the classification result stored in the *EdfClassifyResult* structure.

Example:

```
edfComputeDesc(cropImage, module_state, & descriptor, NULL);
// ...
EdfClassifyResult* classify_result = NULL;
int resultCode = edfClassify(&descriptor, module_state, &classify_result, NULL);
if (resultCode == 0) {
       // Print the results to the console
       for (unsigned int i = 0; i < classify_result->num_values; i++) {
              char* task_name  = classify_result->values[i].task_name;
              char* class_name = classify_result->values[i].class_name;
              int   class_id   = classify_result->values[i].class_id;
              float score      = classify_result->values[i].score;
              std::cout << task_name << ":␣" << class_name << "(" << class_id << ")" <<
   std::endl;
              std::cout << "Score:␣" << score << std::endl;
}
} else {
       // Handle errors
}
```

edfModelVersion

Returns the version of the loaded binary model (.DAT file).

Specification:

```
unsigned int edfModelVersion(const void* module_state)
```

Inputs:

- **module_state**
  Pointer to the initialized MMR SDK module instance.

Returns:
Function returns the model version on success, otherwise **0** is returned.

Description:
The function *edfModelVersion*() is used for getting the version of the loaded binary model. The version is the date of the binary model in the format "**YYYYMMDD**". The input of the function call is the void pointer

module_state. The output is the version number or **0** on an error.

Example:

```
void* module_state = NULL;
edfInitEyedentify(&initConfig, &module_state);
// ...
int modelVersion = edfModelVersion(module_state);
if (modelVersion == 0) {
        // Handle errors
}
```

edfAllocDesc

Allocates the *EdfDescriptor* structure.

Specification:

```
void edfAllocDesc(EdfDescriptor* desc, unsigned int size, unsigned int version)
```

Inputs:

- **desc**
  Pointer to the initialized *EdfDescriptor* structure instance.
- **size**
  Size of the descriptor data to be allocated in bytes.
- **version**
  Version of the binary model used for descriptor computation (see *edfModelVersion*).

Description:
The function *edfAllocDesc*() is used for allocating descriptor structure *EdfDescriptor*. Function is used in the cases when the descriptor data are loaded from the external storage, because functions *edfCompareDescs* and *edfClassify* requires aligned memory for the descriptor. The input of the function call is the pointer to the *EdfDescriptor*, size of the data in bytes and the version number.

Example:

```
char* desc_data; // Data loaded from external storage
EdfDescriptor* descriptor = new EdfDescriptor();
edfAllocDesc(descriptor, desc_size, desc_version);
memcpy(descriptor->data, desc_data, desc_size);
```

edfFreeDesc

Frees the descriptor *EdfDescriptor* structure.

Specification:

```
void edfFreeDesc(EdfDescriptor* desc)
```

Inputs:

- **desc**
  Pointer to the initialized *EdfDescriptor* structure instance.

Description:
The function *edfFreeDesc*() is used for freeing the allocated descriptor structure *EdfDescriptor*. This special function is required to correctly free the descriptor's aligned memory. The input of the function call is the pointer to the allocated *EdfDescriptor*.

IMPORTANT: The function DOES NOT delete the *EdfDescriptor* pointer because the user creates the pointer.

Example:

```
EdfDescriptor* descriptor = new EdfDescriptor();
edfAllocDesc(descriptor, desc_size, desc_version);
// ...
edfFreeDesc(descriptor);
delete descriptor;
```

edfFreeCropImage

Frees the *ERImage* structure from the *edfCropImage* function.

Specification:

Example:

```
void edfFreeCropImage(void* module_state, ERImage* cropped_image)
```

Inputs:

- **module_state**
  Pointer to the initialized MMR SDK module instance.
- **cropped_image**
  Pointer to the *ERImage* structure instance created using *edfCropImage* function.

Description:

The function *edfFreeCropImage*() is used for freeing the allocated image structure *ERImage* created using function *edfCropImage*. The input of the function is the pointer to the module which cropped the image and the pointer to the allocated *ERImage*.

IMPORTANT: The function DOES NOT delete the *ERImage* structure pointer because the user creates the pointer.

Example:

```
ERImage cropImage;
edfCropImage(&image, &cropParams, module_state, &cropImage, NULL);
// ...
edfFreeCropImage(module_state, &cropImage);
```

edfFreeClassifyResult

Frees the *EdfClassifyResult* structure from the *edfClassify* function.

Specification:

```
int edfFreeClassifyResult(EdfClassifyResult** classify_result, void* module_state)
```

Inputs:

- **classify_result**
  Double pointer to the *EdfClassifyResult* structure instance created using *edfClassify* function.
- **module_state**
  Pointer to the initialized MMR SDK module instance.

Returns:

- **0** – Freeing was successful.
- **other** – errno value or -1 for other errors.

Description:

The function *edfFreeClassifyResult*() is used for freeing the allocated structure *EdfClassifyResult* created using function *edfClassify*. The input of the function is the double pointer to the allocated *EdfClassifyResult* and the void pointer *module_state*.

IMPORTANT: The function DOES delete the *EdfClassifyResult* pointer because the pointer is created by the *edfClassify* function.

Example:

```
EdfClassifyResult* classify_result = NULL;
edfClassify(&descriptor, module_state, &classify_result, NULL);
// ...
edfFreeClassifyResult(&classify_result, module_state);
```

5.2.2 EdfCropParams

This part defines the API functions which are designed to work with the *EdfCropParams* structure:

- Allocation: *edfCropParamsAllocate*
- Wrapping: *edfCropParamsWrap*
- Freeing: *edfCropParamsFree*

These functions are defined in the **edf_utils.h** file.

edfCropParamsAllocate

Allocates *EdfCropParams* structure content.

Specification:

```
void edfCropParamsAllocate(int size_points, int size_values, EdfCropParams* params);
```

Inputs:

- **size_points**
  Number of points to allocate.
- **size_values**
  Number of values to allocate.
- **params**
  Pointer to the *EdfCropParams* structure instance.

Description:

The function *edfCropParamsAllocate*() is used to allocate the cropping parameters contained in the *EdfCropParams* structure instance. The input is the size of the points and values to allocate and the pointer to the *EdfCropParams* structure instance. The required setup is described in *Examples* and in **edf_type_mmr.h** header file.

Example:

```
EdfCropParams* params = new EdfCropParams();
edfCropParamsAllocate(1, 2, params);
params->points->rows[0]    = 12.3;
params->points->cols[0]    = 45.6;
params->values->values[0] = 12.3;
params->values->values[1] =  0.0;
```

edfCropParamsWrap

Wraps *EdfCropParams* structure over the data arrays.

Specification:

```
void edfCropParamsWrap(int size_points, double* rows, double* cols,
                       int size_values, double* values_data, EdfCropParams* params);
```

Inputs:

- **size_points**
  Number of points to wrap.
- **rows**
  Pointer to the row (y-axis) point data to wrap.
- **cols**
  Pointer to the column (x-axis) point data to wrap.
- **size_values**
  Number of values to wrap.
- **values_data**
  Pointer to the values data to wrap.
- **params**
  Pointer to the *EdfCropParams* structure instance.

Description:
The function *edfCropParamsWrap*() is used to wrap the *EdfCropParams* structure instance over the supplied data arrays. The input is the size of the points, the pointers to the 2D points data arrays to wrap, the size of the values, the pointers to the values data arrays to wrap and the pointer to the *EdfCropParams* structure instance.

Example:

```
double* rows        = {12.3, 45.6};
double* cols        = {78.9, 10.0};
double* values_data = {12.3, 45.6};

EdfCropParams* params = new EdfCropParams();
edfCropParamsWrap(2, rows, cols, 2, values_data, params);
```

edfCropParamsFree

Frees the *EdfCropParams* structure content.

Specification:

```
void edfCropParamsFree(EdfCropParams* params);
```

Inputs:

- **params**
  Pointer to the *EdfCropParams* structure instance.

Description:
The function *edfCropParamsFree*() is used to free the 2D points and values data arrays contained in the *EdfCropParams* structure instance. The input is the pointer to the *EdfCropParams* instance.

IMPORTANT: The function DOES NOT delete the *EdfCropParams* instance pointer because the user creates the pointer.

Example:

```
EdfCropParams* params = new EdfCropParams();
edfCropParamsAllocate(2, 0, params);
// ...
edfCropParamsFree(params);
delete params;
```

6 MMR Results

This Chapter will explain the output of *edfClassify* function, stored in *EdfClassifyResult* and values returned inside its internal *EdfClassifyResultValue* array.

In MMR SDK, the results can vary based on software license. For example, your license might be limited to return road user's category and make only, so you will not receive results regarding model, generation, and variation. The license is enforced via usage of specific binary module. Technical Sheet will give you full details on binary modules and the classification tasks they support.

6.1 EdfClassifyResult and EdfClassifyResultValue

This structure holds an array of *EdfClassifyResultValue* with size *num_values*. You need to iterate the array to get results for each task. A task is a classification query, in MMR possible tasks and their respective classifier names are view, category, make, model, generation, variation, color and multiple tags inside tag_*. The task is returned in *task_name* variable of each *EdfClassifyResultValue*. You need to iterate the *EdfClassifyResult* and look for the result you want to process.

Each binary module has an accompanying file with information regarding the possible values it can produce. All the files are present in the same directory as the binary modules themselves, **[EyedeaMMR]/sdk/modules/edftf2lite/model/** or **[EyedeaMMR]/sdk/modules/edftrt/model/** depending on the backend used. The accompanying files are present in the form of CSV in the *.csv file and also in a JSON format in *.info file, where the star symbol represent the filename of the *.dat file in use. The values produced consist of a string with name of the class, stored in *class_name* and the unique ID of the class, stored in *class_id*. Users are encouraged to use the unique IDs as stable identifiers, as the names can sometimes change between releases. The IDs can also change in a situation where a mistake has been fixed or a new vehicle is available in the market with the same visual appearance as another vehicle already present, however this situation is very rare and typically occurs only in generation and variation tasks.

When a classification result would violate its predecessor, the result is not returned and the value is empty. For example if the system would classify vehicle as "*Ford Golf*", the MMR SDK will return "*Ford*" only, discarding "*Golf*".

6.2 Task View

Returns road user's positional information, i.e. whether the road user is seen from the front, the side or the rear.

6.3 Task View8

Returns road user's positional information with increased specificity to 8 classes. Compared to the *view* task, this task is more detailed and can be used to distinguish between more specific views of the road user. Only available in case of using bounding box detection as input. The classes are defined as follows:

View8 Class	View8 Short Name	View Base Class	Description
frontal exact	F.	frontal	Frontal view, does not allow to see sides of the vehicle clearly. Typically -10° to 10°.
frontal+left	FL	frontal	Frontal view, allows to also see left side of the vehicle. Typically -10° to -80°.
frontal+right	FR	frontal	Frontal view, allows to also see right side of the vehicle. Typically 10° to 80°.
left	.L	side	Left side view, does not allow to see front or rear of the vehicle. Typically -80° to -100°.
right	.R	side	Right side view, does not allow to see front or rear of the vehicle. Typically 80° to 100°.
rear exact	R.	rear	Rear view, does not allow to see sides of the vehicle clearly. Typically 170° to 190°.
rear+left	RL	rear	Rear view, allows to also see left side of the vehicle. Typically 100° to 170°.
rear+right	RR	rear	Rear view, allows to also see right side of the vehicle. Typically 190° to 260°.

6.4 Task Category

Category results are defined by considering United Nations Resolution R.E.3 (Consolidated Resolution on the Construction of Vehicles), also other road users are returned as category value, for example PEDESTRIAN class. The transposition is as defined in the following table:

MMR SDK Name	MMR SDK ID	R.E.3 Class	Description
ANIMAL	20	—	Animals or animals with riders.
BUS	1	M3	Buses.
CAR	2	M1	Personal cars, incl. pickups.
CYCLE	11	—	Bicycles.
HVT	4	N3	Trucks exceeding 12t.
KICKBIKE	21	—	Kickbikes and kick scooters, including e-scooters.
LGT	5	N2	Trucks over 3.5t but not exceeding 12t.
LUGGAGE	23	—	Luggage, to identify pedestrians with luggage.
MTB	6	L*	Motorbikes, Tricycles, Quadbikes.
PEDESTRIAN	16	—	Pedestrians.
SPECIAL	14	T, R, S	Industrial machines, agricultural tractors.
TO12	10	O1 or O2	Small trailers.
TO34	13	O3 or O4	Large trailers, >3.5t.
TROLLEY	22	—	Trolleys and prams. To identify pedestrians pushing trolleys.
TRUCK	12	O3, O4 or N2, N3	Helper class for LGT, HVT and TO34, when these cannot be distinguished. Trucks or large trailers from rear view (LP) or rear exact view (BOX).
VAN	9	N1, M2	Vehicles for the carriage of goods under 3.5t or minibus (typically same visual appearance).

For non-vehicles and non-motorized vehicles, category is based on appearance. This applies to ANIMAL, CYCLE, KICKBIKE, LUGGAGE, TROLLEY. For motorized vehicles, the category is defined based on vehicle model. That means that when a vehicle model can appear in two categories, the MMR SDK will return only the most probable category. There are basically two cases, either a CAR vs VAN case, when manufacturer makes the same vehicle available as personal car and minivan (we classify this typically as CAR), and case number two, where manufacturers create the same model of van as under 3.5t and over 3.5t (we classify this typically as VAN). It is not possible to recognize which one of the maximum masses the vehicle has from image only. Thus, we report the most probable class, based on internal weight data. You can verify for each vehicle model which category is expected to be returned in the binary module accompanying file.

Not all categories are available from all views, please see Technical Sheet for details.

6.5 Task Make

For vehicles only. This task represents classification of the name of the brand. Brand name can be returned as the name used in marketing or its shortcut instead of the full legal name. Please see the binary module accompanying file for the list of possible car maker names. For example, instead of “Mazda Motor Corporation” we return “Mazda”, and instead of “Volkswagen Aktiensgesellschaft” we return “VW”.

Make recognition is not supported for ANIMAL, CYCLE, KICKBIKE, LUGGAGE, MTB, PEDESTRIAN, TO12, TO34, TROLLEY AND TRUCK. Also from side views, only category is recognized.

There exist such vehicles which were produced by several car makers with only minor changes to the exterior, or no changes at all. Some vehicles only differ by brand logo, which is often missing in the rear of the vehicle. In that case, the MMR SDK returns only the most probable vehicle make, but the error rate might be high. There are two typical cases, the first being a VAN created by a consortium for example Fiat Ducato/Peugeot Boxer/Citroen Jumper, the other typical case is rebranding on a specific market, for example Opel/Vauxhall/Holden. In the latter case, we often recognize which brand is the correct one by brand logo only, but it requires very good image quality and resolution.

6.6 Task Model

For vehicles only. A car model is a unique name given to a vehicle within car maker. Car model name example is "*Golf*" for "*VW*" make and CAR category. Please see the binary module accompanying file for the list of possible car model names.

When a vehicle model is not uniquely recognizable in the image, we use tilde sign to denote this as for example in "*Golf* $\sim$ *Jetta*". During Golf Mk IV and Mk V era, Jetta had the same rear appearance as Golf, so we present the user with both possibilities. There can also be cases where more than two possibilities exist, in that case all possibilities are separated by tildes. The nonspecific recognition is typically a product of incomplete visual information, because two or more models can be similar from frontal or rear view but will most of the time differ from the opposite view. It is then necessary to recognize the vehicle from both frontal and rear view at the same time using two cameras to get to the specific class information without tilde. The tilde is present in 5% of model classification classes, but tilde classification is only returned if the specific generation and view has non unique model appearance. In our example, Golf Mk VII will be returned without tilde as "*Golf*". A problem can arise when using SDK on a specific market, where the car maker decided to market the model with a different name. We return EU names of models. If the model is not available on EU market, then we use domestic model name. To handle this, the client using MMR SDK must map the returned class ID corresponding to the violating class name to a localized class name and discard the MMR SDK *class_name* return value. The mapping can be easily done by looping a set of ID to name translation rules or using a hash map of ID to name translations.

For example, "*Mitsubishi ASX* " is called "*Mitsubishi Outlander Sport*" in the USA. One needs then to set up a rule such that if model class ID returned by MMR SDK is equal to 3787, then replace the class name with string "*Outlander Sport*" while using the MMR SDK in the USA. Another example could be "*Ford Kuga*" where Mk III and Mk IV have been sold in US and Middle East as "*Ford Escape*". You see that sometimes the name has changed only for some of the generations of the vehicle, so it is not possible to remap all generations of a model. In that case, the mapping must be performed based on generation class ID instead.

A list of car makers' brands supporting model recognition is available in the Technical Sheet. We cover >99% brands of typical traffic in North America, EU, Middle East, East Asia (not China and Japan) and Australia and >95% in South America, North and South Africa with models. In exceptional cases only frontal model recognition is available. Models are not available for vehicles without make recognition support.

Special cases are bus brands and their models, which might only appear in a single city in the whole world. This makes it very difficult to integrate. We are always available to include additional model support into the MMR SDK based on customer demands.

6.7 Task Generation

For vehicles only. Car makers group models into so-called generations. Generation is commonly a new external visual design referred to with mark numbers and first model year, for example Mk VI (2019). Other car makers can use code names instead of marks. Our notation is the mark value, or the code name, sometimes expanded with additional information, ended by comma separated list of model years in parenthesis, for example "*Ford Explorer Mk VI (2019)*". In some countries, there is a requirement to proceed with registration process when changing any part of the vehicle, for example interior volume knob. Then the car maker would present the vehicle with a new model year, that cannot be distinguished visually from the exterior from the previous model year. As explained, we only return the model year of the first registration when there is no visual exterior change. In our example, that means that all "*Ford Explorer*" vehicles of 2019, 2020 and 2021 are recognized as "*Mk VI (2019)*". Another example is when there are changes to only half of the vehicle. Typically, car makers only update the front part of the vehicle during facelift, keeping the rear unchanged. That means a grouped generation class must be used, as in the rear generation of "*Focus Mk III (2012,2015)*". This means that it cannot be distinguished whether the vehicle is of first model year 2012 or first model year 2015 from rear view. It can be easily distinguished from frontal view as "*Focus Mk III (2012)*" and "*Focus Mk III (2015)*". There are also rare cases where the rear is updated but the front remains unchanged.

Using generation recognition in practice is very difficult due to the nature of incomplete generation naming convention by the car manufacturers. At present, we are unable to release images of vehicles corresponding to the classes to help with the mapping, due to the licenses of the image files we process. We are aggregating license conforming image files to release a catalog of vehicles in a future release. Also, in various regions it is possible that the model year is shifted by one year. Please see the binary module accompanying file for the list of possible car generation names and expect a necessity to adapt the names using name localization. There are several thousands of car generations available for recognition.

6.8 Task Variation

For vehicles only. Variation is a visual difference in a generation. Variation is typically used to distinguish among trim levels or body types. For example, we can distinguish trim levels "*Mercedes E-Class W213 (2016) AMG*" and "*Mercedes E-Class W213 (2016) Estate*". We can also distinguish body types as in "*Mercedes GLC-Class Mk I (2015,2020) Coupe*" and "*Mercedes GLC-Class Mk I (2015,2020) SUV*". The naming convention in case of variations is vague and is now left for the client to interpret.

Variation only exists if we deemed the visual difference is high enough. Otherwise, all trims or all body types will be grouped, and no variation will be available.

6.9 Task Color

For selected categories only (BUS, CAR, HVT, LGT, SPECIAL, TO12, TO34, TRUCK, VAN). Returns vehicle dominant color. Only usable during day, do not use on night images without external artificial light. Color of plastic pieces like plastic bumper or plastic front mask is ignored. When classifying rear TRUCK, if there is a flatbed truck with container, the color of the container is dominant. A large area is used for color classification, make sure the whole vehicle fits inside the image. The best results are achieved when camera color calibration is performed.

6.10 Task Tag

There are various tasks, each task name starting with prefix "*tag_*". A tag task may typically recognize whether a vehicle belongs to certain group, e.g. caravan, ambulance, pickup, animal transport, etc. The system splits the recognition into frontal and rear view, which means some tag tasks can only be recognized from a certain view. In other cases, some subclasses of a tag tasks can be recognized from both views, but some only from one of the views. The following table lists all recognizable tags, the availability per view and notes regarding recognition. The tag tasks require using the score to limit the number of mistakes. The thresholds need to be set based on required precision and recall. A typical threshold could be 0.9. Tag type can be a yes/no, meaning the return value is either yes or no, a numeric value or a subclass value.

In the current version, we recommend using tags only on day images and turning off tags on night images.

License plate based tags are described in the following table:

Tag task name	Tag type	Frontal	Rear	Notes
tag_ambulance	yes/no	yes	yes	Ambulance or a medic.
tag_animal_transport	yes/no	no	yes	Vehicles are carrying livestock. Racing horses transport is not recognized.
tag_caravan	yes/no	yes*	yes*	The vehicle is a caravan. We distinguish 4 types of caravans. Camper van – this is not recognized, because from outside it looks like a van, the difference is only in the inside. Camper trailer – this is recognized from rear view. Camper semi-integrated – from frontal it looks like a classic van, recognized only from rear view. Camper integrated – from both views it can be uniquely distinguished from a van, recognized from frontal and rear view.
tag_fire_brigade	yes/no	yes	yes	Fire brigade.
tag_law_enforcement	yes/no	yes	yes	Police, Douane, Sheriff, Highway Patrol, etc.
tag_pickup	yes/no	no	yes	Vehicle is a pickup.
tag_push_bumper	yes/no	yes	no	Vehicle's frontal side is obstructed by a push bumper (typical for Australia). This significantly reduces MMR recognition accuracy, so in some cases it may be better to remove vehicles with push bumper from MMR recognition.
tag_rear_mount	yes/no	no	yes	The vehicle's rear side is obstructed by a mounted device, typically a bike or a bike holder. This significantly reduces MMR recognition accuracy, so in some cases it may be better to remove vehicles with rear mount from MMR recognition.
tag_towed	yes/no	yes	yes	The vehicle is towed by another vehicle and so is not moving on its own.
tag_wood_truck	yes/no	no	yes	Vehicle is carrying wood (tree trunks).

Bounding box based tags are described in the following table:

Tag task name	Tag type	Available for views	Notes
tag_ambulance	yes/no	all	Ambulance or a medic.
tag_animal_transport	yes/no	all except frontal exact	Vehicle is carrying livestock. Racing horses transport is not recognized.
tag_caravan	yes/no	all	Vehicle is a caravan.
tag_fire_brigade	yes/no	all	Fire brigade.
tag_helmet	yes/no	all	The driving person has a helmet. Only use if tag_rider_present=yes and tag_rickshaw=no.
tag_law_enforcement	yes/no	all	Police, Douane, Sheriff, Highway Patrol, etc.
tag_mixer_truck	yes/no	all	The vehicle is a concrete mixer truck (agitator truck).
tag_pickup	yes/no	all	Vehicle is a pickup.
tag_push_bumper	yes/no	any frontal	Vehicle's frontal side is obstructed by a push bumper (typical for Australia). This significantly reduces MMR recognition accuracy, so in some cases it may be better to remove vehicles with push bumper from MMR recognition.
tag_rear_mount	yes/no	any rear	The vehicle's rear side is obstructed by a mounted device, typically a bike or a bike holder. This significantly reduces MMR recognition accuracy, so in some cases it may be better to remove vehicles with rear mount from MMR recognition.
tag_rickshaw	yes/no	all	The vehicle is an auto rickshaw(tuk-tuk).
tag_rider_present	yes/no	all	There is a rider present, only for ANIMAL, CYCLE, KICKBIKE and MTB.
tag_tank_truck	yes/no	all	The vehicle is a tank truck (cistern).
tag_taxi	yes/no	all	The vehicle is a taxi.
tag_towed	yes/no	all	The vehicle is towed by another vehicle and so is not moving on its own.
tag_truck_is_semi	yes/no	all except frontal exact and rear exact	The vehicle is a semi-truck. Otherwise it is a straight-truck. Only for LGT and HVT.
tag_wood_truck	yes/no	all	Vehicle is carrying wood (tree trunks).

7 Examples

This chapter contains the examples description which are contained in the SDK package. The examples are used to demonstrate the functionality of the SDK, the source codes are included in the package and are in detail described in this chapter.

7.1 Eyedea MMR SDK Example

The Eyedea MMR SDK contains an example which is used to demonstrate the basic functionality of the MMR SDK on several input images. The example uses known position of the road users in the images to recognize their view and category; in case of vehicles also make, model, color, and tags using the SDK library. This chapter describes in detail the example together with references to important parts of this document. The example is in the folder **[EyedeaMMR]/examples/example-mmr-API/**. The folder contains all source code and files needed for a successful build. In the case of Windows Visual Studio 2019 project is included, in case of Linux a Makefile is included.

7.1.1 API Linking

The EdfAPI structure serves to access library API independently on type of the libeyedentify linking (explicit or implicit). To fills up the API structure, the *edfLinkAPI* function can be used. For explicit linking, the library must be first loaded and *edfLinkAPI* function linked. In the example linking macros from **er_explink.h** are used for that. For implicitly linked library the function should be called with "*nullptr*" instead of library handle.

```
EdfAPI edfAPI;
#ifdef EXPLICIT_LINKING
// explicitly link library
std::string edf_lib_path = std::string(EDF_SDK_PATH) + "lib/" + std::string(EDF_SHLIB_NAME);
shlib_hnd hdll = nullptr;
ER_OPEN_SHLIB(hdll, edf_lib_path.c_str());
if (hdll==nullptr) {
    std::cout << "Library␣'" << edf_lib_path << "'␣not␣loaded!\n" << ER_SHLIB_LASTERROR << "\n";
    return -1;
}
fcn_edfLinkAPI pfLinkAPI=nullptr;   /* The function which will link all other api functions */
ER_LOAD_SHFCN(pfLinkAPI, fcn_edfLinkAPI, hdll, "edfLinkAPI");
if (pfLinkAPI==nullptr) {
    std::cout << "Loading␣function␣'esLinkAPI'␣from␣" << edf_lib_path << "␣failed!\n";
    return -1;
}
if ( pfLinkAPI(hdll, &edfAPI) != 0 ){
    std::cout << "Function␣edfLinkAPI()␣returned␣with␣error!\n";
     return -1;
}
#else
edfLinkAPI(nullptr, &edfAPI);
#endif
```

7.1.2 Initialization

First thing to do is the Eyedea MMR SDK module initialization using the *EdfInitConfig* structure and the *edfInitEyedentify* function. In the example the *setEdfInitConfig* function fills up the intitialization structure.

```
EdfInitConfig setEdfInitConfig(const char *path, const char *name, ERComputationMode mode,
                               int gpu_device_id, int num_threads)
{
    EdfInitConfig config{};
    config.module_path      = path; // e.g. ../../sdk/modules/edftf2lite/
    config.model_file       = name; // e.g. MMR_VCMMCT_FAST_2024Q4.dat
                                    // or MMRBOX_VCMMCT_2024Q4.dat
    config.computation_mode = mode; // e.g. ER_COMPUTATION_MODE_CPU
    config.gpu_device_id    = gpu_device_id;
    config.num_threads      = num_threads;
    return config;
}
```

The *EdfInitConfig* structure is filled first with the path to the Eyedea MMR module, then the name of the binary model to use, then the computation mode and the ID of the GPU are specified, and finally the number of threads used for image crop and descriptor computation in CPU mode. In case of CPU computation, "*gpu_device_id*" parameter is not used (e.g. set to 0). If the initialization was successful, zero code is returned from the *edfInitEyedentify* function. For other return codes refer the function reference: *edfInitEyedentify*.

7.1.3 Input Image Loading

Before the MMR engine could be used, image data must be loaded and decoded. The example uses the Eyedea Recognition's custom image structure *ERImage* to manipulate with the images. The image is loaded to the *ERImage* structure using the *erImageRead*. In real scenario, the client would probably use *erImageAllocateAndWrap* to wrap the client's already existing image memory.

```
// Create the ERImage.
ERImage image;
// Read the input image
int image_read_code = api.erImageRead(&image, input.imageFilename.c_str());
// Check whether the image was loaded.
if (image_read_code != 0) {
    // Handle errors
}
// Work with image
// ...
```

7.1.4 Cropping the Input Image

The engine requires the input image to be cropped and transformed to have the object of interest aligned and to have the image in correct resolution and color and data format. For such image transformations, the *edfCropImage* function is used. First the cropping parameters in the *EdfCropParams* must be set. The cropping parameters are allocated with the *edfCropParamsAllocate* function. There are two possible setups.

```
EdfCropParams cropParams;
edfCropParamsAllocate(EDF_MMR_CROP_POINTS, EDF_MMR_CROP_VALUES, &cropParams);
// Set license plate center in the input image.
EDF_LP_CENTER_X(cropParams)       = 475.0;
EDF_LP_CENTER_Y(cropParams)       = 573.0;
// Set license plate resolution in pixels per meter.
// -> License plate has 134 pixels in the image and
//    Czech LP is 0.52 m wide:    134/0.52 = 257.7
EDF_LP_SCALE_PX_PER_M(cropParams) = 257.7;
// Set license plate rotation compensation in degrees.
EDF_LP_ROTATION(cropParams)       =   2.0;

// Create the pointer to the cropped EdfImage.
EdfImage *cropImage = NULL;
// Create the image crop with respect to the license plate.
int cropResultCode = edfCropImage(&image, &cropParams, module_state, &cropImage, NULL);
// Set the image data pointer to NULL:
// data belongs to the cv::Mat structure and will be deallocated there.
image.data = NULL;
// Free the image structure fields. The image is not needed anymore.
freeEdfImage(&image);
// Free vehicle crop parameters.
freeEdfCropParams(&cropParams);
// Check the crop result.
if (cropResultCode != 0) {
    // Handle the error.
}
```

License plate-based MMR task requires the center of the license plate in the image, scale in pixels per meter and rotation compensation to be set. The 2D center of the vehicle's license plate in the image is set using the macro **EDF_LP_CENTER_X** for the x coordinate and the macro **EDF_LP_CENTER_Y** for the y coordinate. The scale of the vehicle in pixels per meter is obtained by dividing the width of the license plate in pixels by the width of the license plate in meters (for details see the example code above) and set using the macro **EDF_LP_SCALE_PX_PER_M**. The last parameter - rotation compensation is set using the macro **EDF_LP_ROTATION**. The rotation compensation parameter defines the in-plane rotation of the input image with the center of the rotation in the center of the license plate.

```
EdfCropParams cropParams;
edfCropParamsAllocate(EDF_MMRBOX_CROP_POINTS, EDF_MMRBOX_CROP_VALUES, &cropParams);
// Set license plate center in the input image.
EDF_MMRBOX_TOP_LEFT_X(cropParams) = 100.0;
EDF_MMRBOX_TOP_LEFT_Y(cropParams) = 50.0;
EDF_MMRBOX_BOTTOM_RIGHT_X(cropParams) = 200.0;
EDF_MMRBOX_BOTTOM_RIGHT_Y(cropParams) = 250.0;

// Create the pointer to the cropped EdfImage.
EdfImage *cropImage = NULL;
// Create the image crop with respect to the license plate.
int cropResultCode = edfCropImage(&image, &cropParams, module_state, &cropImage, NULL);
// Set the image data pointer to NULL:
// data belongs to the cv::Mat structure and will be deallocated there.
image.data = NULL;
// Free the image structure fields. The image is not needed anymore.
freeEdfImage(&image);
// Free vehicle crop parameters.
freeEdfCropParams(&cropParams);
// Check the crop result.
if (cropResultCode != 0) {
    // Handle the error.
}
```

Bounding box based MMR task requires a 2D rectangle covering the whole vehicle. In case of vehicle combination (tractor+trailer, car+trailer), each vehicle needs its separate bounding box and will get a separate classification. The top left corner of the rectangle is specified using macros **EDF_MMRBOX_TOP_LEFT_X** and **EDF_MMRBOX_TOP_LEFT_Y** for column and row coordinates. The top left corner of the rectangle is specified using macros **EDF_MMRBOX_BOTTOM_RIGHT_X** and **EDF_MMRBOX_BOTTOM_RIGHT_Y** for column and row coordinates.

With the cropping parameters prepared, the input image can be processed with the function *edfCropImage* which crops and transforms the input image with respect to the cropping parameters defined above.

While only the cropped ERImage is required for further processing the original *ERImage* with the input image data can be deleted using the function *erImageFree*. The input image is loaded with the OpenCV framework to the *cv::Mat* structure and the image data are deleted with that structure, therefore the data array of the *ERImage* must be set to **NULL** to avoid the OpenCV's image data deletion with *erImageFree* function.

7.1.5 Descriptor Computation

The core of the recognition process and most time-consuming operation is done during the descriptor computation. The descriptor is a condensed representation of the recognized object, and it is optimized for efficient classification and comparison.

The descriptor computation is done using the *edfComputeDesc* function. The function requires the cropped input image. The computation takes from tens to hundreds of milliseconds depending on the binary model and hardware used (for more information about the computation times see the document Eyedea MMR SDK - Technical sheet).

The image crop is not needed after the descriptor is computed; it can be deleted. The function *edf-FreeCropImage* must be used for image crop deletion because the crop was generated inside the module, so it must be also freed there. Together with the image crop data the pointer to the *ERImage* structure is freed.

```
// Create the EdfDescriptor structure.
EdfDescriptor descriptor;
// Compute the descriptor. Crop image is used as an input,
// the output is copied to the EdfDescriptor structure.
int computeDescResultCode = edfComputeDesc(cropImage, module_state, &descriptor, NULL);
// Free the crop image data. The crop is not needed anymore.
edfFreeCropImage(module_state, &cropImage);
// Check the descriptor computation result.
if (computeDescResultCode != 0) {
    // Handle error.
}
```

7.1.6 Classification

The classification is a process of getting the corresponding classes from the computed descriptor. Also, it is the process of getting the human readable results of the road user's recognition. The only input of the classification is the descriptor combined with the module which was used for the descriptor computation. The process of classification is done using the function *edfClassify* which stores the result to the *EdfClassifyResult* structure.

The *EdfClassifyResult* structure contains the array of *EdfClassifyResultValue* values. The values contain the best result for each class. In the case of MMR the classes are view, category, make, model (possibly also generation and variation), color and tags. The result contains the name of the task (i.e., "*make*"), the result class name (i.e., "*VW*"), the ID of the result class (i.e., 43) and the score of the result (i.e., 0.95041). The availability of the classes is dependent on the used model (for more information about the current binary models see the document Eyedea MMR SDK - Technical sheet).

```
// Initialize the EdfClassifyResult structure pointer.
EdfClassifyResult *classify_result = NULL;
// Run the classification. The result will be copied
// to the newly created EdfClassifyResult structure.
int resultCode = edfClassify(&descriptor, module_state, &classify_result, NULL);
// Check the classification result.
if (resultCode == 0 && classify_result) {
    // Iterate over all result entries.
    for (unsigned int i = 0; i < classify_result->num_values; i++) {
        // Get the classification result value name.
        std::string name(classify_result->values[i].task_name,
        classify_result->values[i].task_name_length);
        // Get the classification result value.
        std::string value(classify_result->values[i].class_name,
        classify_result->values[i].class_name_length);
        // Get the classification result score.
        float score = classify_result->values[i].score;
    }
} else {
    // Handle error.
}
// Free the classification result.
edfFreeClassifyResult(&classify_result, module_state);
// Free the descriptor fields.
edfFreeDesc(&descriptor);
```

The descriptor can be deleted after the classification is done. The deletion of the descriptor is done using the SDK API function *edfFreeDesc*. When the classification result *EdfClassifyResult* is not needed anymore (for example after the results were saved to the DB or printed out) it must be deleted using the *edfFreeClassifyResult* function. The function goes through the contained result values and deletes them all together with the pointer to the structure.

7.1.7 Cleaning Up

At the end, when the work with the Eyedea MMR SDK instance is finished (for example at the end of the program), the SDK instance must be deleted together with all underlying structures. To delete it use the API function *edfFreeEyedentify*, which is designed for such purpose.

```
// Free the module. All module internal structures
// will be deleted and program can be finished.
edfFreeEyedentify(&module_state);
```

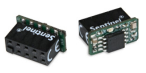
During the function run the significant amount of memory is freed because the whole binary model, which is loaded in the memory after the instance initialization, is freed.

8 MMR SDK Licensing

MMR SDK uses the third-party framework developed by Thales for software protection and licensing. The SDK is protected against reverse engineering and unlicensed execution using hardware USB keys. The SDK can not be used without a USB license key with a valid license except in tral version, which uses software key instead.

8.1 License Key Types

The SDK allows loading a license using various hardware key types which are listed in the following table. The keys differ by the number of licenses they can contain (Pro and Max versions), by physical dimensions, ability to contain time-limited licenses (Time versions) and ability to distribute licenses over the network (Net versions).

SKU	Product	SKU	Product
SH-PRO	Sentinel HL Pro	SH-BRD	Sentinel HL Max (Board form factor) 
SH-MAX	Sentinel HL Max	SH-TIM	Sentinel HL Time
SH-MIC	Sentinel HL Max (Micro form factor)	SH-NET	Sentinel HL Net
SH-CHP	Sentinel HL Max (Chip form factor)	SH-NTT	Sentinel HL NetTime

8.2 Licenses Overview

Several licenses are available for the MMR SDK. The licenses differ in the type of the binary models which can be loaded, the time period for which the license is valid, and the number of allowed function executions.

8.2.1 Perpetual License

A perpetual license is the least restrictive license available. It allows the user to use the license in specified number of instances for unlimited time and unlimited number of executions. This license type is used for products which will be deployed to the end-user.

8.2.2 Time-Limited License

A time-limited license allows to set a restriction on the time for which the license is valid. The license validity end date or the number of the days for which the license is valid after the first use can be set. This license can be set on Time keys only (see *License Key Types*). This type of license is used mainly in the Developer package.

8.2.3 Execution Counting

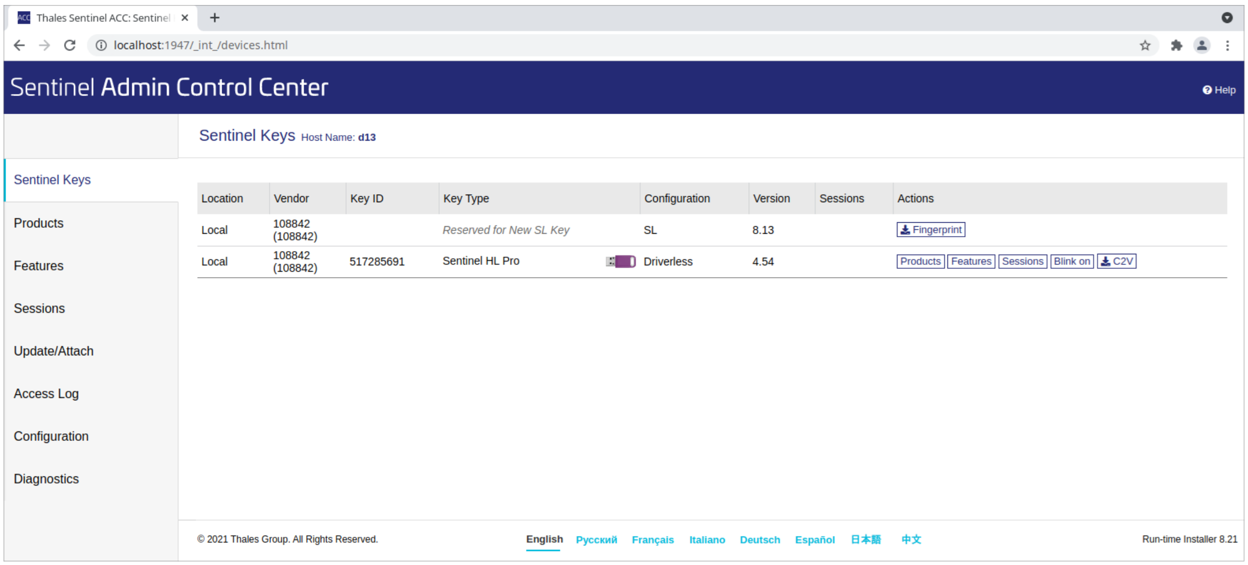
An execution counting license allows counting the number of times the license was logged in. The SDK is designed in such a way that it logs in the license every time a specified SDK function is called. It allows limiting the number of executions with the license. This type of license is used mainly in the Developer package.

8.3 License Management

The license protection software provides a web interface for license management. The web interface can be found on the address http://localhost:1947 opened in the common web browser. It allows the user to list the connected license keys, see the details of the arbitrary license key, update the license, and several other functions.

8.3.1 Connected License Keys

The list of license keys currently plugged in the computer is available at http://localhost:1947/_int_/devices.html. The list contains basic information about each key, including the location of the key (Local or IP/name of the remote machine), Vendor ID, Key ID, Key Type, Configuration, Version and the number of connected Sessions. For each key, it is possible to list the contained license products, features and sessions using the buttons Products, Features and Sessions. For easy identification, the USB key LED can be blinked using the **Blink On** button in the Actions column. The unique key identification file can be downloaded using the **C2V** button.


Thales Sentinel ACC: Sentinel
localhost:1947/_int_/devices.html
Sentinel Admin Control Center
Help
Sentinel Keys
Products
Features
Sessions
Update/Attach
Access Log
Configuration
Diagnostics

Sentinel Keys Host Name: **d13**

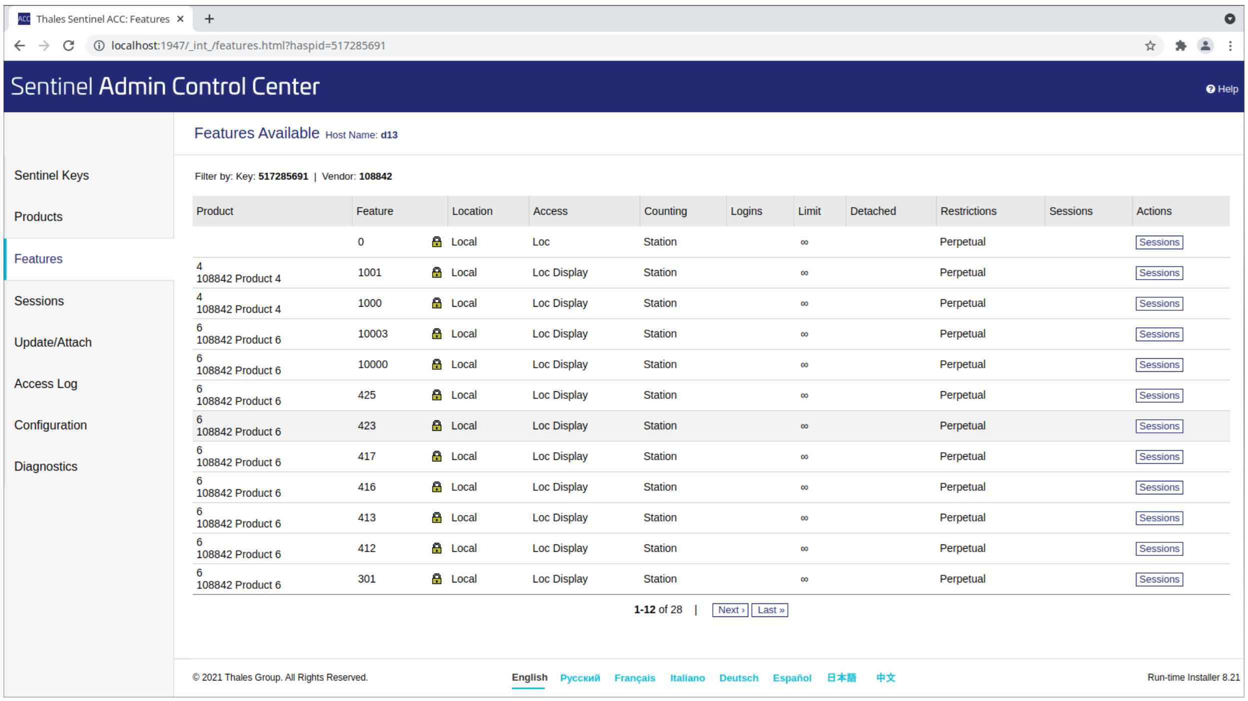
Location	Vendor	Key ID	Key Type	Configuration	Version	Sessions	Actions
Local	108842 (108842)		*Reserved for New SL Key*	SL	8.13		Fingerprint
Local	108842 (108842)	517285691	Sentinel HL Pro	Driverless	4.54		Products Features Sessions Blink on C2V

© 2021 Thales Group. All Rights Reserved. English Русский Français Italiano Deutsch Español 日本語 中文 Run-time Installer 8.21

Web interface with list of plugged keys on http://localhost:1947/_int_/devices.html

8.3.2 License Key Details

Detailed information about a key can be acquired by clicking on the **Features** button in the *Connected License Keys* list or at http://localhost:1947/_int_/features.html?haspid=KEYID, where the KEYID is the ID of the key. The web page contains information about the licenses contained on the key. The set of all the features represents the whole license. Each Feature controls a different part of the SDK workflow (initialization, binary model selection, descriptor computation, ...).


Thales Sentinel ACC: Features

localhost:1947/_int_/features.html?haspid=517285691

Sentinel Admin Control Center

Help

Sentinel Keys
Products
Features
Sessions
Update/Attach
Access Log
Configuration
Diagnostics

Features Available Host Name: **d13**

Filter by: Key: **517285691** | Vendor: **108842**

Product	Feature	Location	Access	Counting	Logins	Limit	Detached	Restrictions	Sessions	Actions
	0	🔒 Local	Loc	Station		∞		Perpetual		Sessions
4 108842 Product 4	1001	🔒 Local	Loc Display	Station		∞		Perpetual		Sessions
4 108842 Product 4	1000	🔒 Local	Loc Display	Station		∞		Perpetual		Sessions
6 108842 Product 6	10003	🔒 Local	Loc Display	Station		∞		Perpetual		Sessions
6 108842 Product 6	10000	🔒 Local	Loc Display	Station		∞		Perpetual		Sessions
6 108842 Product 6	425	🔒 Local	Loc Display	Station		∞		Perpetual		Sessions
6 108842 Product 6	423	🔒 Local	Loc Display	Station		∞		Perpetual		Sessions
6 108842 Product 6	417	🔒 Local	Loc Display	Station		∞		Perpetual		Sessions
6 108842 Product 6	416	🔒 Local	Loc Display	Station		∞		Perpetual		Sessions
6 108842 Product 6	413	🔒 Local	Loc Display	Station		∞		Perpetual		Sessions
6 108842 Product 6	412	🔒 Local	Loc Display	Station		∞		Perpetual		Sessions
6 108842 Product 6	301	🔒 Local	Loc Display	Station		∞		Perpetual		Sessions

1-12 of 28 | Next › Last »

© 2021 Thales Group. All Rights Reserved. English Русский Français Italiano Deutsch Español 日本語 中文 Run-time Installer 8.21

Web interface with key 517285691
details on http://localhost:1947/_int_/features.html?haspid=517285691

8.4 License Update

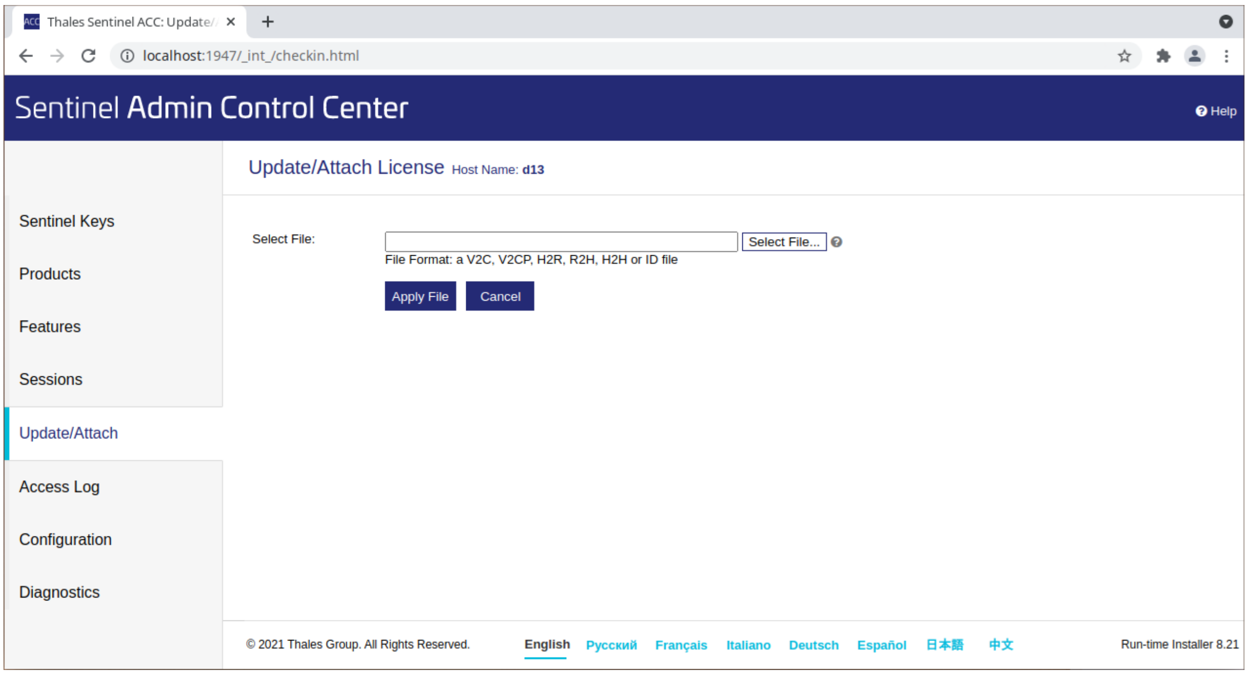
The license can be updated using a special **.v2c* file, which is emitted by the licensor of the software. The license update file is generated for a specific license key ID and only that key can be updated using the file. There are two ways of updating the license: *Web Interface* and *Command Line*.

The license update must be done on the computer where the protection software supplied with the SDK package is installed. For more information about the protection software installation see the chapter *Installation Guide*.

IMPORTANT: The hardware protection key dongle with the license to be updated needs to be connected to the machine where the license update will be applied.

8.4.1 Web Interface

The first option allows the user to update the license using the web interface of the license management software Sentinel Admin Control Center. The web interface which can be opened in all modern browsers is located at http://localhost:1947/_int_/checkin.html.



Update/Attach License Host Name: d13

Select File:

File Format: a V2C, V2CP, H2R, R2H, H2H or ID file

© 2021 Thales Group. All Rights Reserved.

Run-time Installer 8.21

Web interface for license update on http://localhost:1947/_int_/checkin.html

How to update the license:

1. Open the link http://localhost:1947/_int_/checkin.html in the web browser.
2. Click on the Select File button and select the *.v2c file which you want to use for the update.
3. Click on the Apply File button.
4. A webpage with the result of the license update is shown.

8.4.2 Command Line

The second method of updating the license is by using the Windows command line or a Linux console. This approach can be very useful when applying the update remotely or on many devices. It is also suitable for automating the license update procedure. This option requires basic knowledge of the Windows command line or some Linux console. The license update file **.v2c* is applied using the **hasp_update** utility from the folder **hasp/** located in the corresponding SDK package root.

Windows command line

Run the **hasp_update** utility with following parameter and the **.v2c* file path on the selected machine:

```
hasp_update u /path/to/v2c/license.v2c
```

If the command runs without any errors, the license has been updated successfully.

Linux console

Run the **hasp_update** utility with following parameter and the **.v2c* file path on the selected machine:

```
./hasp_update u /path/to/v2c/license.v2c
```

If the command runs without any errors, the license has been updated successfully.

9 Third Party Software

The MMR SDK uses third party software libraries in accordance with their licenses. The licenses can be found under **[MMR_SDK]/documentation/3rdparty-licenses**.

9.1 Used Libraries

Here is a complete list of all libraries used, in alphabetical order:

- Boost
- OpenBLAS
- OpenCL
- OpenCV
- OpenGL
- OpenSSL
- Protobuf
- Tensorflow Lite
- TensorRT
- ZLib

The following statements are published to fulfill the license terms of the respective libraries:

“This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/).”