DeGirum AI Client for C++ Reference Guide

Contents

Chapter 1

DeGirum AI Client for C++

1.1 Introduction

The DeGirum AI Client package referenced in this document is the part of DeGirum AI Software Suite.

Note: AI Client package consists of multiple programming language support packs. This document describes AI Client for C++. AI client packs for other languages are described in separate documents.

The DeGirum AI Software Suite provides means to perform fast and efficient AI model inferencing on locally installed AI hardware accelerator platforms.

The list of supported hardware accelerator platforms includes all DeGirum Orca AI hardware accelerators and Google Edge TPU accelerators. AI inferencing on local host CPU is also supported.

The DeGirum AI Software Suite consists of DeGirum AI Server package, AI Model Zoo, and DeGirum AI Client package.

1.1.1 AI Server Package

The AI Server package is installed on a computer system where AI hardware accelerators are connected to. The purpose of the AI Server is the following:

- It controls AI hardware accelerators
- It provides distributed and concurrent access of multiple AI clients to AI inferencing functionality of AI hardware accelerators over the TCP/IP network
- It maintains local AI model zoo with AI models to be used for AI inferencing
- It provides Web-based GUI to manage local model zoo

The following computer platforms are currently supported for the server installation:

- Intel x64 with Linux 64-bit OS

The support for following computer platforms is currently under development:

- Intel x64 with Windows 10 OS
- ARM with Linux 64-bit OS

The AI Server for Linux is typically distributed as fully configured Docker container, which simplifies AI Server deployment to the variety of Linux platforms.

1.1.2 AI Model Zoo

The AI Model Zoo is the place where AI models reside.

The local AI Model Zoo of DeGirum AI Server is implemented as the directory on the AI Server local file system. It contains files of all locally deployed AI models. Only models in the local Model Zoo can be used for AI inferencing on particular AI server installation.

Currently, AI Server supports the following AI model types:

- N2X models: models in proprietary DeGirum N2X format compiled for DeGirum Orca AI accelerator hardware and CPU
- TFLite Edge TPU models: models in Google TFLite format compiled for Google Edge TPU AI accelerator hardware
- TFLite CPU models: models in Google TFLite format compiled for execution on CPU

The management of the local Model Zoo is performed either via Web-based AI Server control GUI or using `dg_zoo_mgr.py` Python script, both included in the distribution of AI Server package.

1.1.3 AI Client Package

The AI Client package is installed on a computer system, where client-side ML software runs. It can be the same computer system, where the AI Server is deployed, or it can be any other computer system, which has TCP/IP network access to the AI Server computer system.

AI Client Package includes the following components:

- AI Client library
- AI Client examples
- Build scripts

AI Client library provides APIs to control and use the AI Server. It includes the following functionality:

- List all AI models with their attributes available for AI inferencing in particular AI Server local Model Zoo
- Find AI model in that local Model Zoo matching particular model attributes
- Open and configure streaming connection to particular AI Server for inferencing using particular AI model from local Model Zoo
- Stream data frames to the that streaming connection to perform AI inferencing on these data frames

AI Client library provides integration into the following programming languages: C++ and Python. This document describes AI Client for C++. AI client packs for other languages are described in separate documents.

All AI Client Package software is distributed in the source code and requires compilation.

The following computer platforms are currently supported for the client installation:

- Intel x64 with Linux 64-bit OS

The support for following computer platforms is currently under development:

- Intel x64 with Windows 10 OS
- ARM with Linux 64-bit OS

1.2 AI Client Package Installation and Build

1.2.1 Installation Pre-Requisites

For Linux OS the following software packages need to be installed in order to build AI Client package software:

- Git
- CMake
- GCC compiler
- Thread building blocks library
- Python 3.8

To install Git use the following command:

```
sudo apt install git-all
```

To install CMake use the following command:

```
sudo apt install cmake
```

To install GCC compiler use the following command:

```
sudo apt install g++
```

To install thread building blocks library use the following command:

```
sudo apt install libtbb-dev
```

To install Python 3.8 you can install miniconda package. First download miniconda 64-bit for Python 3.8 for example from this link:

```
https://docs.conda.io/en/latest/miniconda.html#linux-installers
```

Then install it from download directory by running the following command:

```
bash Miniconda3-latest-Linux-x86_64.sh
```

1.2.2 AI Client Package Installation

The AI Client library is distributed in source code and is available for DeGirum customers on GitHub at the following link:

```
https://github.com/DeGirum/SWReleases
```

You clone the AI Client library repo by executing command:

```
git clone https://github.com/DeGirum/SWReleases.git
```

The `client` directory contains the AI Client library collaterals. The subdirectory structure is as follows:

Directory	Contents
/examples	sample applications
/extern	third-party libraries
/images	sample images to be used for AI inference
/inc	AI Client library C++ include files
/lib	AI Client library compiled binary files
/src	AI Client library C++ source files

1.2.3 AI Client Package Build

The CMakeLists.txt file is provided for the CMake build. It builds AI Client C++ library, AI Client Python bindings library, and C++ examples.

To perform the build under Linux OS, execute the following commands:

```
cd client
mkdir build
cd build
cmake ..
```

To build the Python binding as well, call CMake the following way:

for x64 Python:

```
cmake .. -DDG_BUILD_PYTHON=ON -A x64 -T host=x64
```

for x32 Python:

```
cmake .. -DDG_BUILD_PYTHON=ON
```

For Windows OS and Visual Studio, CMake will create the Visual Studio project files and a solution. The library can be built by opening and building AICLIENTLIB.sln solution. To install Python bindings, INSTALL target from the CMakePredefinedTargets folder should be built.

For Linux OS, CMake will create make files (unless otherwise specified with the -G flag). The library can be built by invoking make in the /build subdirectory. To install Python bindings, the following command should be executed:

```
sudo make install
```

After building, the library and example executables are copied into /lib subdirectory.

1.3 AI Client C++ API Description

DeGirum AI Client library provides the following functionality:

- modelzooListGet() function to list all AI models with their attributes available for AI inferencing in particular AI Server local Model Zoo
- modelFind() function to find AI model in that local Model Zoo matching particular model attributes
- DG::AIModel class to connect to AI server and to perform AI model inference in a simple non-pipelined sequential manner (see AIModel: Synchronous Sequential AI Inferencing API)
- DG::AIModelAsync class to connect to AI server and to perform AI model inference in efficient pipelined asynchronous manner using mechanism of callbacks (see AIModelAsync: Asynchronous Pipelined AI Inferencing API)
- the family of ModelParams classes to handle model parameters (see Model Parameters Management)
- functions and methods for error handling (see Error Handling)

1.3.1 AI Server Connection and Addressing

The client software communicates with AI server through TCP/IP network using DeGirum proprietary communication protocol. In order to establish a connection to AI server, the client side must know the host name or IP address of the AI server, and TCP port, which AI server listens.

The server address is passed to AI inference classes as a string in the following format:

If AI server domain name is known, the address string is specified in the form

```
domain_name:port
```

If only IP address of AI server is known, the address string is specified in the form

```
xxx.xxx.xxx.xxx:port
```

The port number together with ':' separator can be omitted, in this case default port 8778 is used. When AI server is started without specifying the port, the same default port 8778 is used for the server side. In this case it is safe to omit the port. Otherwise you should match the port on the client side to the port used on the server side.

1.3.2 Listing and Searching Models in Model Zoo

The AI server can perform AI inferences only on models located in the local model zoo of that AI server. To initiate an inference of a model on AI server, the client must provide to the server the *model identifier*, which uniquely identifies the desired model in the local model zoo. There are two ways for the client code to obtain the model ID.

1. By obtaining the full list of models available in the model zoo using modelzooListGet() function
2. By searching the model zoo for a model satisfying certain search attributes using modelFind() function.

Either way you obtain the instance of DG::ModelInfo structure, which contains all necessary model identification information, including:

- DG::ModelInfo::id - unique model ID
- DG::ModelInfo::name - model string name
- DG::ModelInfo::W - input frame width
- DG::ModelInfo::H - input frame height
- DG::ModelInfo::C - input color depth
- DG::ModelInfo::N - input frame depth
- DG::ModelInfo::device_type - device type on which model runs
- DG::ModelInfo::runtime_agent - runtime agent type on which model runs
- DG::ModelInfo::model_quantized - 'is model quantized' flag
- DG::ModelInfo::model_pruned - 'is model pruned (not dense)' flag
- DG::ModelInfo::input_type - input data type: "Image", ...
- DG::ModelInfo::input_tensor_layout - for image inputs, image tensor layout the model expects: "NHWC", "↩ NCHW"

- DG::ModelInfo::input_color_space - for image inputs, image color-space the model expects: "BGR", "RGB"
- DG::ModelInfo::input_image_format - for image inputs, image format: "JPEG", "RAW" (this is user-tunable runtime parameter)
- DG::ModelInfo::input_raw_data_type - for "RAW" image inputs, image pixel data type: "DG_FLT", "DG_UINT8" (this is user-tunable runtime parameter)

Then you pass this structure to constructors of classes used for AI inference: DG::AIModel or DG::AIModelAsync.

The modelzooListGet() function returns an array of ModelInfo structures listing all models found in the local model zoo. It is then your responsibility to iterate over all elements of this array to find the model you want to use.

The modelFind() function provides simpler means to find the desired model. It accepts the model query: the instance of ModelQuery structure, which contains some model attributes you want to look for. The modelFind() function first obtains the full list of models from the AI server, and then iterates over this list to find a model matching all attributes specified in the model query. It returns the first model matching all attributes.

You may specify the following attributes:

- Any part of the model name. Only models which have the specified substring in their model name would be considered.
- Device type to use for inference. Only models which are designed for specified device type would be considered.
- Runtime agent to use for inference. Only models which are designed for specified runtime agent would be considered.
- Model should be quantized flag. Only quantized models would be considered.
- Model should be pruned/sparse flag. Only sparse/pruned models would be considered.

Only the model name substring attribute is mandatory, all other attributes are optional and can be omitted if not needed.

You specify the device type by providing the Device ID string. The following device types are supported:

Device ID	Device
"ORCA"	DeGirum Orca AI accelerator
"EDGETPU"	Google Edge TPU AI accelerator
"CPU"	AI server host computer

The *runtime agent* is the software module, which controls the device used for AI inference. You specify the runtime agent type by providing the Agent ID string. The following runtime agent types are supported:

Agent ID	Agent
"N2X"	DeGirum N2X runtime agent. Used for all Orca devices and N2X model inference on CPU.
"TFLITE"	Google TFLite runtime agent. Used for Edge TPU devices and direct TFlite model inference on CPU.

Note: the runtime agent attribute is rarely required because in most use cases the AI hardware uniquely defines the runtime agent to use, so it is recommended to avoid setting this attribute.

Examples:

Find a model which has 'MobileNet_v2' substring in the model name:

```
auto model_id = DG::modelFind( server, { "MobileNet_v2" } );
```

Find a model which has 'MobileNet_v2' substring in the model name, is designed to run on DeGirum Orca AI accelerator hardware, is quantized, and is sparse/pruned:

```
auto model_id = DG::modelFind( server, { "MobileNet_v2",
    "ORCA", "", DG::ModelQuery::Yes, DG::ModelQuery::Yes } );
```

1.3.3 AIModel: Synchronous Sequential AI Inferencing API

The DG::AIModel class is used to perform AI model inference on AI server in simple non-pipelined sequential manner.

Note: for more efficient (but somewhat more complex to use) asynchronous pipelined inference use DG::AIModelAsync class.

On construction this class performs connection to AI server, selection of AI model, and setting model run-time parameters.

You specify the AI server address as a string as described in the section AI Server Connection and Addressing.

You specify the model ID as described in the section Listing and Searching Models in Model Zoo.

If you want to modify some of the model parameters (see more on that in the section Model Parameters Management), you define the instance of ModelParamsWriteAccess-derived class, change model parameters you want to adjust by invoking setter methods, and pass this instance as one of DG::AIModel constructor arguments.

For example, you want to change the input frame image type to JPEG. You can pass the following expression to DG::AIModel constructor:

```
DG::ModelParamsWriter().InputImgFmt_set( "JPEG" )
```

Once constructed, it can be used to perform sequential AI inference by invoking DG::AIModel::predict() method. Predict method accepts the input frame data in various formats (see section Format of Data Frames for more details) and returns the inference results, which are described in the section Format of Inference Results. This method is blocking method, i.e. it returns the execution only when the inference of an input frame is complete. Working with blocking methods simplifies the programming, but loses some run-time efficiency because (unlike for DG::AIModelAsync class) some actions like sending data to AI server, frame pre-processing, result post-processing, and receiving results from AI server are not pipelined between frames.

1.3.4 Format of Inference Results

The predict() method of DG::AIModel class returns Json array with inference results. The internal structure of this Json array greatly depends on the AI model type: it is different for classification models, detection models, and so on. But in general, that Json array is an array of identical Json objects, and each Json object contains inference results for one detected entity.

The following tables describe Json array format for some model types.

For classification model types each classified object is represented by the following Json object:

Json Field Name	Description	Data Type
"label"	Classified object class label	string
"category_id"	Classified object class index	integer
"score"	Probability of classified object	floating point

For detection model types each detected object is represented by the following Json object:

Json Field Name	Description	Data Type
"label"	Detected object class label	string
"category_id"	Detected object class index	integer
"score"	Probability of detected object	floating point
"bbox"	Detected object rectangle	four-element Json array [xmin, ymin, xmax, ymax]
"landmarks"	Landmark points (optional)	Json array of two-element arrays [[x1, y1], [x2, y2], ...]

For pose detection types each detected object is represented by the following Json object:

Json Field Name	Description	Data Type
"score"	Probability of detected pose	floating point
"landmarks"	Pose description landmarks	array of landmark Json objects

The following is the structure of pose description landmark Json object:

Json Field Name	Description	Data Type
"label"	Classified object class label	string
"category_id"	Classified object class index	integer
"score"	Probability of classified landmark	floating point
"landmark"	landmark point	two-element Json array [x, y]
"connect"	list of adjacent landmarks indices	Json array of integers

When you specify post-processing type as "None" (see section Model Parameters Management) then the inference result Json array contains network output tensors. Each element of that array corresponds to one tensor. Each tensor is represented by the following Json object:

Json Field Name	Description	Data Type
"id"	Tensor numeric ID as specified in network	integer
"name"	Tensor name as specified in network	string
"shape"	Tensor shape: sizes of each dimension	integer array
"quantization"	Tensor quantization parameters	object
"type"	Tensor element type	string
"data"	Tensor linear data buffer contents	binary field

The following tensor data types are supported:

Type String	Type Description
DG_FLT	32-bit floating point
DG_UINT8	8-bit unsigned integer

The following is the structure of quantization Json object:

Json Field Name	Description	Data Type
"axis"	Quantization axis or -1 for global quantization	integer
"scale"	Quantization scale array	floating point array
"zero"	Quantization zero offset	integer array

For convenient handling of tensors the DG::BasicTensor class is provided. You can convert Json array containing a tensor into DG::BasicTensor object by using DG::JsonHelper::tensorDeserialize() static method:

```
for( const auto &json_tensor: inference_result )
{
    DG::BasicTensor basic_tensor( DG::JsonHelper::tensorDeserialize( json_tensor ) );
    auto shape = basic_tensor.shape();              // get tensor shape
    auto element_type = basic_tensor.dataTypeGet(); // get tensor element type
    if( element_type == DG_UINT8 )                  // get pointer to tensor linear buffer
        const uint8_t *data = basic_tensor.data< uint8_t >();
}
```

There is a special case of Json array returned by AI server in case when runtime error happens during inference. In this case the runtime error info object is returned. It is described in details in the Error Handling section.

1.3.5 AIModelAsync: Asynchronous Pipelined AI Inferencing API

The DG::AIModelAsync class is used to perform AI model inference on AI server in efficient pipelined asynchronous manner using the mechanism of callbacks.

Note: for simple (but less efficient) synchronous non-pipelined inference use DG::AIModel class.

On construction this class performs connection to AI server, selection of AI model, setting model run-time parameters, and installation of the client callback function. The client callback function is used to pass the inference results from the AI server to the client code.

You specify the AI server address as a string as described in the section AI Server Connection and Addressing.

You specify the model ID as described in the section Listing and Searching Models in Model Zoo.

You should define the callback function somewhere in your code in the following form:

```
void MyAiCallback( const json &result, const std::string &info ){}
```

Then you pass your callback name (in this example it is MyAiCallback) as one of DG::AIModelAsync constructor arguments. The callback function has two arguments: the inference result and the frame info string. The inference result is Json array, described in the section Format of Inference Results. The frame info string is the string passed to predict() method along with the frame data. More on that below.

If you want to modify some of the model parameters (see more on that in the section Model Parameters Management), you define the instance of DG::ModelParamsWriteAccess derived class, change model parameters you want to adjust by invoking setter methods, and pass this instance as one of DG::AIModelAsync constructor arguments.

For example, you want to change the input frame image type to JPEG. You can pass the following expression to DG::AIModelAsync constructor:

```
DG::ModelParamsWriter().InputImgFmt_set( "JPEG" )
```

Once constructed, it can be used to perform asynchronous AI inference by invoking DG::AIModelAsync::predict() method. The predict method accept the input frame data in various formats (see section Format of Data Frames for more details) and initiates the inference on the AI server. The method is a non-blocking method, i.e. it returns execution immediately after posting the frame data to the AI server. This allows calling this method in a loop without waiting for the inference results, achieving the maximum AI server utilization.

To simplify matching inference results and corresponding frames in the client callback, additional argument can be passed to predict() method. The frame_info argument is optional frame information string. It will be passed to the client callback along with the frame result. You can pass arbitrary information as the frame info, for example frame data filenames, or frame numbers, or any other frame identification information, which you need to know in the scope of client callback function in order to interpret inference results in respect to the input frames.

Once the inference of a frame is complete, and the inference result is received from the AI server, the client callback is invoked to dispatch the inference result. Such result handling via callback mechanism is performed in a thread, separate from the main execution thread. It means that the client callback function is called in asynchronous manner, thus the name of the class.

During construction you may also specify the depth of the internal frame queue. The internal frame queue works the following way. If predict() method is invoked too often and the number of non-processed ("outstanding") frames exceeds the specified queue depth, the consecutive call to predict() method will be blocked until the number of outstanding frames in the queue becomes smaller than the queue depth thus allowing to post one more frame.

The class destructor waits until all outstanding results are received and then closes the connection to AI server. You may use DG::AIModelAsync::waitCompletion() to explicitly wait for completion of all outstanding inferences. It is a blocking call: it returns only when all outstanding frames are processed by AI server and all results are dispatched via client callback.

To monitor the number of outstanding inferences in real-time you use DG::AIModelAsync::outstandingResultsCountGet() method.

To check for runtime errors which happened during inferences you use DG::AIModelAsync::lastError() method. If ever during consecutive calls to predict() method the AI server reported a run-time error, then this method will return the error message string, otherwise it returns an empty string. Alternatively, you may check for runtime errors directly in the callback function by invoking errorCheck() function on inference result argument.

Note: in case of server runtime error, all frames posted after the error was detected, will not be processed.

1.3.6 Model Parameters Management

Each AI model in the model zoo comes with the model parameters file in Json format. This Json file defines all model parameters such as model name, model input dimensions, pre-processing parameters, post-processing parameters, etc. Some of the model parameters are fixed and cannot be altered without breaking the model behavior, and some of the model parameters can be changed to adjust the model behavior. Later parameters are called model *runtime parameters* and can be adjusted when constructing instances of model inference classes DG::AIModel and DG::AIModelAsync.

To work with model parameters the following classes are provided:

- DG::ModelParamsReadAccess
- DG::ModelParamsWriteAccess
- DG::ModelParams
- DG::ModelParamsWriter

DG::ModelParamsReadAccess class provides programmatic type-safe read access to model parameters defined in Json model configuration. It keeps non-owning const reference to underlying Json array. For each model parameter it provides getter method, which name matches the parameter name as it appears in Json array.

DG::ModelParamsWriteAccess inherits DG::ModelParamsReadAccess and provides programmatic type-safe read *and write* access to model parameters defined in Json model configuration. It keeps non-owning non-const reference to underlying Json array. For each model parameter it provides both getter and setter methods. Getter method name matches the parameter name as it appears in Json array. Setter method name is constructed from the parameter name by appending `_set` suffix. For example, `InputImgFmt_set()` Setter methods return reference to self, so they can be called in a daisy-chain manner, for example:

```
params.InputImgFmt_set( "RAW" ).InputImgRawDataType_set( "DG_UINT8" );
```

DG::ModelParams class provides programmatic type-safe access to model parameters defined in Json model configuration while *owning* that Json array. Access (read or write) is defined by the template parameter, which is used as the base class. You can use any DG::ModelParamsReadAccess-derived base class: it can be DG::ModelParamsReadAccess or DG::ModelParamsWriteAccess.

And finally DG::ModelParamsWriter is DG::ModelParams template instantiation with write access. Instances of this class you typically use to pass adjusted runtime model parameters to constructors of inference classes.

The following table summarizes all runtime parameters which are accessible to the client code:

Parameter Name	Type	Description	Possible Values	Default Value
InputImgFmt	string	Input frame format: JPEG or raw binary	"JPEG", "RAW"	"JPEG"
InputImgRawDataType	string	Data type for raw binary frame format: byte of 32-bit float	"DG_UINT8", "DG_FLT"	"DG_UINT8"
InputImgRotation	int	Input image rotation angle in degrees, clockwise	0, 90, 180, 270	0
OutputConfThreshold	double	Confidence threshold	0..1	0.1
OutputNMSThreshold	double	Threshold for non-max suppression	0..1	0.6
OutputPostprocessType	string	Post-processor type	see below	"None"

If you do not specify a runtime parameter, then the value for this parameter will be taken from the model Json configuration file from the local model zoo. If the model Json configuration file also does not specify this parameter, then the default value will be used.

The following table lists currently available post-processor types.

Post-Processor Name	Applicability	Description
"None"	All models	Pass-through post-processor. Network output tensors are sent without any post-processing
"Classification"	EfficientNet, MobileNet_v1, MobileNet_v2	Classification post-processor for all classification models
"TFLiteDetection"	MobileNet_v2 SSD TFlite	Object detection post-processor for TF Lite detection models
"Detection"	MobileNet_v2 SSD N2X	Object detection post-processor for DeGirum N2X detection models

Note: not all post-processors are compatible with all models. Special care should be taken when specifying post-processor type other than "None".

1.3.7 Format of Data Frames

The AI Client package supports two formats of the input data frames: JPEG images and binary bitmaps, where binary format is used by default and typically set in Json model configuration files for all DeGirum models.

In any case the HxW dimension of the input data frame must match the input dimension of the AI model used for the inference. The input dimensions of the AI model are stored in the DG::ModelInfo data structure fields DG::ModelInfo::W and DG::ModelInfo::H, returned by either modelzooListGet() or modelFind() functions (see section Listing and Searching Models in Model Zoo).

The frame data in the binary bitmap format should be arranged as HWC three-dimensional dense array, meaning that color dimension 'C' is the fastest dimension, while the vertical dimension 'H' is the slowest. The color order is dependent on how the model was originally trained. The pixel data type can be either 8-bit unsigned integer number or 32-bit floating point number.

You specify input frame type when you construct instances of model inference classes DG::AIModel and DG::AIModelAsync, passing instance of ModelParams class as the constructor parameter (see more details in the section Model Parameters Management).

To specify JPEG frame type you need to set InputImgFmt parameter to "JPEG".

To specify binary frame type you need to set both InputImgFmt and InputImgRawDataType parameters. For example, the byte binary frame type you set this way:

```
std::string server = "192.168.0.1"; // some server address
auto model_id = DG::modelFind( server, { "MobileNet_v2" } );
DG::AIModel model( server, model_id,
    DG::ModelParamsWriter.InputImgFmt_set( "RAW" ).InputImgRawDataType_set( "DG_UINT8" ) );
```

1.3.8 Error Handling

There are two mechanisms of error handling employed in AI Client package: using standard library exceptions and using specially formatted Json inference results.

All runtime errors *except errors happened during inference*, are handled by throwing standard library exceptions - objects of std::exception type. To handle such errors the recommended practice is to place all the code which interacts with the AI Client library into one try/catch block, and then catch exceptions of std::exception type:

```
try
{
    ...
}
catch( std::exception &e )
{
    std::cout << e.what();  // handle exception
}
```

Errors which happened during the inference are handled special way. Instead of throwing an exception, the special Json runtime error info object is returned by DG::AIModel::predict() method in case you use synchronous API, and that Json object is passed as an argument into your callback in case you use asynchronous API.

The Json runtime error info object has the following structure:

Json Field Name	Description
"success"	Boolean field containing "false" in case of error
"msg"	Error message string

If you want to check that the inference result actually defines runtime error, you use errorCheck() function. If the AI server Json response contains the run-time error, then this function will return the error message string, otherwise it returns an empty string. The following code demonstrates the typical use case for synchronous API:

```
json result;
model.predict( frame, result );
auto possible_error = DG::errorCheck( result );
if( !possible_error.empty() )
    throw std::runtime_error( possible_error );
```

The following code demonstrates the typical use case for asynchronous API:

```
void MyAiCallback( const json &result, const std::string &info )
{
    auto possible_error = DG::errorCheck( result );
    if( possible_error.empty() )
    {
        ... accept inference result
    }
    else // do not throw exception, since it will be ignored by AI Client thread
}
```

Note: do not throw any exceptions in your callback, since the AI Client callback thread, which handles user callback invocation, suppresses all exceptions thrown in callbacks to not to terminate the callback thread unexpectedly.

Another approach for the inference error handling for asynchronous API would be to check for inference errors at the very end of the inference sequence. You can use DG::AIModelAsync::lastError() method: if ever during consecutive calls to predict() method the inference errors are detected, then this method will return the error message string, otherwise it returns an empty string. To guarantee that all outstanding frames are processed before the error check, you can use DG::AIModelAsync::waitCompletion() method:

```
model.waitCompletion();
auto possible_error = model.lastError();
if( !possible_error.empty() )
    throw std::runtime_error( possible_error );
```

Note: in case of inference error in asynchronous client, all frames posted after the error was detected, will not be processed.

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 5

Namespace Documentation

5.1 DG Namespace Reference

DG is DeGirum namespace which groups all DeGirum software declarations.

Classes

- class AIModel

  AIModel is DeGirum AI client API class for simple non-pipelined sequential inference.
- class AIModelAsync

  AIModelAsync is DeGirum AI client API class for efficient pipelined asynchronous inference.
- class BasicTensor
- class JsonHelper

  Json helper class: contains assorted static methods to operate with Json configurations.
- struct ModelInfo

  ModelInfo is the model identification structure. It keeps AI model key attributes.
- class ModelParams

  ModelParams is model parameters collection with user-defined access rights.
- class ModelParamsReadAccess

  ModelParamsReadAccess is read-only accessor to model parameters.
- struct ModelParamsSection

  Model parameters section descriptor.
- class ModelParamsWriteAccess

  ModelParamsWriteAccess is read/write accessor to model parameters.
- struct ModelQuery

  Model query structure used to search models on AI server which match a set of provided model attributes.
- struct ServerAddress

  ServerAddress is the server address structure. It keeps AI server TCP/IP address.

Typedefs

- using BasicTensorVector = std::vector< BasicTensor >

  Collection of tensors.
- using json = nlohmann::json

  Json library.
- typedef struct DG::ModelInfo ModelInfo

  ModelInfo is the model identification structure. It keeps AI model key attributes.
- using ModelParamsWriter = ModelParams< ModelParamsWriteAccess >

  ModelParamsWriter is ModelParams template instantiation with write access.
- typedef struct DG::ServerAddress ServerAddress

  ServerAddress is the server address structure. It keeps AI server TCP/IP address.

Enumerations

- enum DetectionStatus { DetectionStatus::OK, DetectionStatus::ProtocolMismatch, DetectionStatus::HostAlive }

  Enumerator showing server status as detected by the below function.
- enum RUNTIME_AGENT_TYPES {
  RUNTIME_AGENT_TYPES::DEFAULT, RUNTIME_AGENT_TYPES::N2X, RUNTIME_AGENT_TYPES::TFLITE, RUNTIME_AGENT_TYPES::ONNX,
  RUNTIME_AGENT_TYPES::OPENVINO, RUNTIME_AGENT_TYPES::DUMMY }

  Runtime agent types.

Functions

- std::vector< std::tuple< std::string, DG::DetectionStatus > > detectHostnameServers (const std::string &pattern, const int range_start, const int range_end, const int numeral_width=3)
- std::vector< std::tuple< std::string, DetectionStatus > > detectSubnetServers (const std::string &root_ip, const std::string &subnet_mask)
- std::string errorCheck (const json &json_response)
- template<typename T >
  T jsonGetMandatoryValue (const json &json_params, const std::string &section, int index, const std::string &key)
- template<typename T >
  T jsonGetOptionalValue (const json &json_params, const std::string &section, int index, const std::string &key, const T &default_value)
- static bool jsonKeyExist (const json &json_params, const std::string &section, int index, const std::string &key)
- template<typename T >
  void jsonSetOptionalValue (json &json_params, const std::string &section, int index, const std::string &key, const T &value)
- json labelDictionary (const std::string &server, const std::string &model_name)
- static std::string messagePrepare (const json &in)
- ModelInfo modelFind (const std::string &server, const ModelQuery &query)
- void modelzooListGet (const std::string &server, std::vector< ModelInfo > &modelzoo_list)
- bool operator< (const ServerAddress &lhs, const ServerAddress &rhs)

  Comparison operator for server address.
- std::ostream & operator<< (std::ostream &os, const BasicTensor &t)

  Print tensor to stream.
- bool operator== (const ServerAddress &lhs, const ServerAddress &rhs)

  Comparison operator for server address.
- std::string versionGet ()

Variables

- const int CURRENT_PROTOCOL_VERSION = 2

  current client-server protocol version
- const int DEFAULT_PORT = 8778

  Default TCP port of AI server.
- const int MIN_COMPATIBLE_PROTOCOL_VERSION = 2

  minimum compatible client-server protocol version
- const int MODEL_PARAMS_CURRENT_VERSION = 3
- const int MODEL_PARAMS_MIN_COMPATIBLE_VERSION = 1
- constexpr const char * PROTOCOL_VERSION_TAG = "VERSION"

  client-server protocol version tag
- constexpr ModelParamsSection SECT_DEVICE = { "DEVICE", true }

  device parameters section
- constexpr ModelParamsSection SECT_INTERNAL = { "INTERNAL", true }

  internal parameters section
- constexpr ModelParamsSection SECT_MODEL_PARAMETERS = { "MODEL_PARAMETERS", true }

  model parameters section
- constexpr ModelParamsSection SECT_POST_PROCESS = { "POST_PROCESS", true }

  post-processing parameters section
- constexpr ModelParamsSection SECT_PRE_PROCESS = { "PRE_PROCESS", false }

  pre-processing parameters section
- constexpr ModelParamsSection SECT_TOP = { "", true }

  Json file top-level sections.

5.1.1 Detailed Description

DG is DeGirum namespace which groups all DeGirum software declarations.

5.1.2 Enumeration Type Documentation

5.1.2.1 DetectionStatus

```
enum DG::DetectionStatus  [strong]
```

Enumerator showing server status as detected by the below function.

Enumerator

OK	Server fully operational.
ProtocolMismatch	Server is up, but is outdated.
HostAlive	Hardware is alive, but no server found. Unimplemented!

5.1.2.2 RUNTIME_AGENT_TYPES

```
enum DG::RUNTIME_AGENT_TYPES  [strong]
```

Runtime agent types.

Enumerator

DEFAULT	default runtime agent
N2X	DeGirum nnExpress runtime agent.
TFLITE	TF-Lite runtime agent.
ONNX	Onnx runtime agent.
OPENVINO	OpenVINO runtime agent.
DUMMY	dummy agent (for DeGirum internal use only)

5.1.3 Function Documentation

5.1.3.1 detectHostnameServers()

```
std::vector< std::tuple< std::string, DG::DetectionStatus > > DG::detectHostnameServers (
            const std::string & pattern,
            const int range_start,
            const int range_end,
            const int numeral_width = 3 )
```

Detect all ORCA servers with hostnames generated from a prefix and a range

Parameters

in	*prefix*	is a string prefix, formatted as "nnnn[:port]". This will be used to generate two sets of hostnames, one with the pattern "nnnn#", and one "nnnn###". For example, the prefix "farm", range_start 1, range_end 2 and numeral_width 3 will scan the hosts "farm1", "farm001", "farm2", "farm002". Setting numeral_width to 0 disables pattern generation, and will just scan the prefix. The port part is optional. If included, all servers will be scanned on the port. If omitted, the default port 8778 will be used.
in	*range_start*	is the lowest numeral to be applied to the pattern
in	*range_end*	is the highest numeral to be applied to the pattern, inclusive
in	*numeral_width*	is the width of the numeral in the padded set. This is optional, set to 3 by default. Set to 0 to disable pattern generation.

Returns

Vector of strings containing hostnames of ORCA servers on the network.

5.1.3.2 detectSubnetServers()

```
std::vector< std::tuple< std::string, DetectionStatus > > DG::detectSubnetServers (
            const std::string & root_ip,
            const std::string & subnet_mask )
```

Detect all ORCA servers on a given subnet

Parameters

in	*root_ip*	is an IP address in the network, formatted as "xxx.xxx.xxx.xxx[:port]". The port part is optional. If included, all servers will be scanned on the port. If omitted, the default port 8778 will be used.
in	*subnet_mask*	specifies the subnet with the associated IP, formatted as "xxx.xxx.xxx.xxx", where "xxx" is either 255 for fixed bits or 0 for variable bits.

Returns

Vector of pairs containing IPs of ORCA servers on the network and their statuses

5.1.3.3 errorCheck()

```
std::string DG::errorCheck (
            const json & json_response )
```

Check given AI server Json response for run-time errors. If AI server reported run-time error in the given Json response, then this function extracts and returns the error message string from the provided Json array, otherwise it returns an empty string.

5.1.3.4 jsonGetMandatoryValue()

```
template<typename T >
T DG::jsonGetMandatoryValue (
            const json & json_params,
            const std::string & section,
            int index,
            const std::string & key )
```

Get value from Json array. Raise error, if not there

Parameters

in	*json_params*	- Json array
in	*section*	- top section name, which contains array of records; if empty, key is taken from topmost level
in	*index*	- index in array of records
in	*key*	- key of key-value pair

Returns

parameter value

5.1.3.5 jsonGetOptionalValue()

```
template<typename T >
T DG::jsonGetOptionalValue (
            const json & json_params,
            const std::string & section,
            int index,
            const std::string & key,
            const T & default_value )
```

Get value from Json array, which may not be present there

Parameters

in	*json_params*	- Json array
in	*section*	- top section name, which contains array of records if empty, key is taken from topmost level
in	*index*	- index in array of records
in	*key*	- key of key-value pair
in	*default_value*	- value to return, if no such key is found

Returns

parameter value

5.1.3.6 jsonKeyExist()

```
static bool DG::jsonKeyExist (
            const json & json_params,
            const std::string & section,
            int index,
            const std::string & key )  [inline], [static]
```

Check if given key exists in Json array

Parameters

in	*json_params*	- Json array
in	*section*	- top section name, which contains array of records if empty, key is checked on topmost level
in	*index*	- index in array of records
in	*key*	- key of key-value pair

Returns

true if such key exists in array

5.1.3.7 jsonSetOptionalValue()

```
template<typename T >
void DG::jsonSetOptionalValue (
            json & json_params,
            const std::string & section,
            int index,
            const std::string & key,
            const T & value )
```

Set value to Json array; if it is not present, skip assignment

Parameters

in	*json_params*	- Json array
in	*section*	- top section name, which contains array of records if empty, key is set on topmost level
in	*index*	- index in array of records
in	*key*	- key of key-value pair
in	*value*	- value to set

5.1.3.8 labelDictionary()

```
json DG::labelDictionary (
            const std::string & server,
            const std::string & model_name )
```

Get model label dictionary

Parameters

in	*server*	is a string specifying server domain name/IP address and port. Format: "domain_name:port" or "xxx.xxx.xxx.xxx:port". If port is omitted, the default port is 8778.
in	*model_name*	specifies the AI model. To obtain valid model name, either modelFind() or modelzooListGet() functions should be used.

Returns

JSON object containing model label dictionary

5.1.3.9 messagePrepare()

```
static std::string DG::messagePrepare (
            const json & in )  [static]
```

prepare client-server response message string from input json

Parameters

in	server response json

Returns

response string

5.1.3.10 modelFind()

```
ModelInfo DG::modelFind (
            const std::string & server,
            const ModelQuery & query )
```

Find a model on a given AI server which matches given model query.

Parameters

in	*server*	is a string specifying server domain name/IP address and port. Format: "domain_name:port" or "xxx.xxx.xxx.xxx:port". If port is omitted, the default port is 8778.
in	*query*	is the ModelQuery structure of model parameters to look for.

Returns

model descriptor to be used in AIModel and AIModelAsync constructors. If the model is not found, the model descriptor contains an empty model name.

5.1.3.11 modelzooListGet()

```
void DG::modelzooListGet (
            const std::string & server,
            std::vector< ModelInfo > & modelzoo_list )
```

Get a list of supported models from AI server. In case of server connection errors throws std::exception.

Parameters

in	*server*	is a string specifying server domain name/IP address and port. Format: "domain_name:port" or "xxx.xxx.xxx.xxx:port". If port is omitted, the default port is 8778.
out	*modelzoo_list*	is the output vector of ModelInfo structures; each structure contains essential model attributes such as model ID, model name, input width and input height. The ModelInfo structure extracted from this vector can be used as a parameter for construction of AIModel and AIModelAsync instances.

5.1.3.12 versionGet()

```
std::string DG::versionGet ( )
```

Get the version of the library.

Returns

library version in the following format: MAJOR.MINOR.REVISION.GIT_REVISION

5.1.4 Variable Documentation

5.1.4.1 MODEL_PARAMS_CURRENT_VERSION

```
const int DG::MODEL_PARAMS_CURRENT_VERSION = 3
```

The most current version of Json model configuration, supported by this version of software. Increment it each time you change any parameter definition or add/remove any parameter

5.1.4.2 MODEL_PARAMS_MIN_COMPATIBLE_VERSION

```
const int DG::MODEL_PARAMS_MIN_COMPATIBLE_VERSION = 1
```

The minimum compatible version of Json model configuration, still supported by this version of software. Increase it when the software is modified such a way, that it stops supporting older Json model configuration versions.

5.1.4.3 SECT_TOP

```
constexpr ModelParamsSection DG::SECT_TOP = { "", true }  [constexpr]
```

Json file top-level sections.

top-level section

Chapter 6

Class Documentation

6.1 DG::AIModel Class Reference

AIModel is DeGirum AI client API class for simple non-pipelined sequential inference.

`#include <dg_model_api.h>`

Public Member Functions

- AIModel (const std::string &server, const std::string &model_name, const ModelParamsReadAccess &model_params=ModelParamsReadAccess({}))
- void predict (std::vector< std::vector< char > > &data, json &json_response)
- ~AIModel ()

  Destructor. Closes connection to AI server.

6.1.1 Detailed Description

AIModel is DeGirum AI client API class for simple non-pipelined sequential inference.

This class is used to perform AI model inference on AI server in a simple non-pipelined sequential manner. For more efficient (but somewhat more complex) asynchronous pipelined inference use AIModelAsync class.

On construction this class performs connection to AI server, selection of AI model, and setting model run-time parameters.

Once constructed, it can be used to perform sequential AI inference by invoking predict() method. The predict() methods accept the input frame data in various formats and return the inference results. These are blocking methods, i.e. they return execution only when the inference of an input frame is complete.

6.1.2 Constructor & Destructor Documentation

6.1.2.1 AIModel()

```
DG::AIModel::AIModel (
            const std::string & server,
            const std::string & model_name,
            const ModelParamsReadAccess & model_params = ModelParamsReadAccess({}) )  [explicit]
```

Constructor. Performs connection to AI server, selection of AI model, and optionally setting model run-time parameters. In case of server connection errors throws std::exception.

Parameters

in	*server*	is a string specifying server domain name/IP address and port. Format: "domain_name:port" or "xxx.xxx.xxx.xxx:port". If port is omitted, the default port is 8778.
in	*model_name*	specifies the AI model to be used for inference. To obtain valid model name, either modelFind() or modelzooListGet() functions should be used.
in	*model_params*	is runtime parameter collection, which defines the model runtime behavior. This is optional parameter: if not specified, then default runtime parameters (as defined in the model zoo) are used. ModelParamsWriter class instance can be used to conveniently define runtime parameters.

6.1.3 Member Function Documentation

6.1.3.1 predict()

```
void DG::AIModel::predict (
            std::vector< std::vector< char > > & data,
            json & json_response )
```

Run the AI inference on provided byte array. The byte array contains the frame data, which depends on selected frame format. It can be either JPEG or bitmap depending on the model parameters. In case of errors throws std::exception. This is blocking method, i.e. it returns execution only when the inference of an input frame is complete.

Parameters

in	*data*	is a vector of input data for each model input where each data element is a vector of bytes.
out	*json_response*	is the result of the inference. The response format depends on the model post-processor type.

The documentation for this class was generated from the following file:

- dg_model_api.h

6.2 DG::AIModelAsync Class Reference

AIModelAsync is DeGirum AI client API class for efficient pipelined asynchronous inference.

```
#include <dg_model_api.h>
```

Public Types

- using callback_t = std::function< void(const json &inference_result, const std::string &frame_info) >

Public Member Functions

- AIModelAsync (const std::string &server, const std::string &model_name, callback_t callback, const ModelParamsReadAccess &model_params=ModelParamsReadAccess({}), size_t frame_queue_depth=8)
- std::string lastError () const
- int outstandingResultsCountGet () const

  Get the number of outstanding inference results posted so far.
- void predict (std::vector< std::vector< char > > &data, const std::string &frame_info="")
- void setCallback (callback_t callback)
- void waitCompletion ()
- ~AIModelAsync ()

6.2.1 Detailed Description

AIModelAsync is DeGirum AI client API class for efficient pipelined asynchronous inference.

This class is used to perform AI model inference on AI server in efficient pipelined asynchronous manner using the mechanism of callbacks. For simple (but less efficient) synchronous non-pipelined inference use AIModel class.

On construction this class performs connection to AI server, selection of AI model, setting model run-time parameters, and installation of the client callback function. The client callback function is used to pass the inference results from the AI server to the client code.

Once constructed, it can be used to perform asynchronous AI inference by invoking predict() method. The predict() methods accept the input frame data and initiate the inference on the AI server. Each of those methods is a non-blocking method, i.e. it returns execution immediately after posting the frame data to the AI server. This allows calling those methods in a loop without waiting for the inference results, achieving the maximum AI server utilization. Once the inference of a frame is complete, and the inference result is received from the AI server, the client callback is invoked to dispatch the inference result. Such result handling via callback mechanism is performed in a thread, separate from the main execution thread. It means that the client callback function is called in asynchronous manner, thus the name of the class.

6.2.2 Member Typedef Documentation

6.2.2.1 callback_t

```
using DG::AIModelAsync::callback_t = std::function< void( const json &inference_result, const
std::string &frame_info ) >
```

User callback type. The callback is called asynchronously from the main execution thread as soon as prediction result is ready. Consecutive prediction result in a form of Json array is passed as the inference_result argument. Corresponding frame info string (provided to predict() call) is passed as the frame_info argument.

6.2.3 Constructor & Destructor Documentation

6.2.3.1 AIModelAsync()

```
DG::AIModelAsync::AIModelAsync (
            const std::string & server,
            const std::string & model_name,
            callback_t callback,
            const ModelParamsReadAccess & model_params = ModelParamsReadAccess({}),
            size_t frame_queue_depth = 8 )  [explicit]
```

Constructor. Performs connection to AI server, selection of AI model, installing client callback, and optionally setting model run-time parameters. In case of server connection errors throws std::exception.

Parameters

in	*server*	is a string specifying server domain name/IP address and port. Format: "domain_name:port" or "xxx.xxx.xxx.xxx:port". If port is omitted, the default port is 8778.
in	*model_name*	specifies the AI model to be used for inference. To obtain valid model name, either modelFind() or modelzooListGet() functions should be used.
in	*callback*	is user callback functional, which will be called asynchronously from the main execution thread as soon as prediction result is ready.
in	*model_params*	is runtime parameter collection, which defines the model runtime behavior. This is optional parameter: if not specified, then default runtime parameters (as defined in the model zoo) are used. ModelParamsWriter class instance can be used to conveniently define runtime parameters.
in	*frame_queue_depth*	is the depth of the internal frame queue. If predict() methods are invoked too often and the number of non-processed (aka "outstanding") frames exceeds this parameter, the consecutive call to any predict() method will be blocked until the number of outstanding frames in the queue becomes smaller than the queue depth thus allowing to post one more frame. This is optional parameter: by default it is set to 8 frames.

6.2.3.2 ~AIModelAsync()

```
DG::AIModelAsync::~AIModelAsync ( )
```

Destructor. Waits until all outstanding results are received and then closes the connection to AI server. Note: in case of server runtime error, all frames posted after that error was detected, will not be processed.

6.2.4 Member Function Documentation

6.2.4.1 lastError()

```
std::string DG::AIModelAsync::lastError ( ) const
```

If ever during consecutive calls to predict() methods AI server reported a run-time error, then this method will return the error message string, otherwise it returns an empty string. Note: in case of server runtime error, all frames posted after that error was detected, will not be processed.

6.2.4.2 predict()

```
void DG::AIModelAsync::predict (
            std::vector< std::vector< char > > & data,
            const std::string & frame_info = "" )
```

Start the inference on given byte data vector. The byte vector contains the frame data, which depends on selected frame format. It can be either JPEG or bitmap depending on the model parameters. In case of errors throws std::exception. This is non-blocking call meaning that it returns execution immediately after posting the frame data to the AI server.

Parameters

in	*data*	is a vector of input data for each model input where each data element is a vector of bytes.
in	*frame_info*	is optional frame information string to be passed to the client callback along with the frame result. You can pass arbitrary information as frame info. This simplifies matching results to frames in the client callback.

6.2.4.3 setCallback()

```
void DG::AIModelAsync::setCallback (
            callback_t callback )
```

Set user callback

Parameters

in	*callback*	is user callback functional, which will be called asynchronously from the main execution thread as soon as prediction result is ready.

6.2.4.4 waitCompletion()

```
void DG::AIModelAsync::waitCompletion ( )
```

Wait for completion of all outstanding inferences. This is blocking call: it returns when all outstanding frames are processed by AI server and all results are dispatched via client callback. You can continue calling predict() after call to this method - frame processing will restart automatically.

The documentation for this class was generated from the following file:

- dg_model_api.h

6.3 DG::BasicTensor Class Reference

```
#include <dg_tensor_structs.h>
```

Classes

- struct quant_params_t

  Quantization parameters (how to convert from integer/quantized data back to floating point)

Public Types

- using shape_t = std::vector< size_t >

  Tensor shape vector type.

Public Member Functions

- template<class CONTAINER >
  auto alloc (const CONTAINER &source, bool do_copy=true, int32_t id=0, const std::string &name="", const shape_t &shape={}, const quant_params_t &quant_params={}) -> decltype(source.data(), source.size(), void())
- template<typename T >
  void alloc (int32_t id, const std::string &name, const shape_t &shape, const quant_params_t &quant_↩ params={}, T *ext_lin_buffer=nullptr)
- void alloc (int32_t id, const std::string &name, const shape_t &shape, DGType data_type, const quant_params_t &quant_params={}, void *ext_lin_buffer=nullptr)
- BasicTensor () noexcept

  Default constructor.
- BasicTensor (BasicTensor &&move) noexcept

  Move constructor.
- BasicTensor (const BasicTensor &)=delete

  Deleted copy constructor.
- template<class CONTAINER >
  BasicTensor (const CONTAINER &source, bool do_copy=true, int32_t id=0, const std::string &name="", const shape_t &shape={}, const quant_params_t &quant_params={})
- template<typename T >
  BasicTensor (int32_t id, const std::string &name, const shape_t &shape, const quant_params_t &quant_↩ params={}, T *ext_lin_buffer=nullptr)
- BasicTensor (int32_t id, const std::string &name, const shape_t &shape, DGType data_type, const quant_params_t &quant_params={}, void *ext_lin_buffer=nullptr)
- BasicTensor clone () const

  Create and return exact copy of itself.
- BasicTensor copy () const
- template<typename T >
  T * data ()
- template<typename T >
  const T * data () const
- DGType dataTypeGet () const

  Get DG data type of tensor element.
- void dealloc ()

  Deallocate tensor data and clear tensor.
- template<typename T_IN , typename T_OUT >
  void dequantize ()
- size_t elementSizeGet () const

  Get tensor element size in bytes.
- bool empty () const

  Check if tensor is empty.

- int32_t id () const
  Get id.
- FloatCompareResult< double > isEqualData (const BasicTensor &rhs, double maxRelDiff) const
- template<typename T >
  FloatCompareResult< double > isEqualData (const std::vector< T > &rhs, double maxRelDiff) const
- bool isEqualDataShape (const BasicTensor &rhs) const
- bool isEqualStruct (const BasicTensor &rhs) const
- bool isExternal () const
  Is external buffer?
- bool isNull () const
  Check if tensor is null.
- size_t linearSizeGet () const
  Get tensor linear size in elements of current type.
- size_t linearSizeGet_bytes () const
  Get tensor linear size in bytes.
- const std::string & name () const
  Get name.
- std::string numpyTypeGet () const
  Return numpy-compatible type string.
- BasicTensor & operator= (BasicTensor &&move)
  Move assignment operator.
- BasicTensor & operator= (const BasicTensor &)=delete
  Deleted copy assignment operator.
- template<typename T_IN , typename T_OUT >
  void quantize ()
- const quant_params_t & quantParams () const
  Get quantization parameters.
- void reshapeTo (size_t dim)
- void reshapeToNHWC ()
  Reshape tensor to 4-D NHWC shape.
- const shape_t & shape () const
  Get shape vector.
- const std::type_info & typeInfo () const
  Return type info.
- template<typename T = void>
  T * untypedData ()
  Return raw data buffer.
- template<typename T = void>
  T * untypedData () const
  Return raw data buffer.
- ~BasicTensor ()
  Destructor.

Static Public Member Functions

- static std::string numpyTypeGet (const std::string &dg_type)
  Return numpy-compatible type string from given DG type string.

6.3.1 Detailed Description

Basic tensor container class. Based on "linear buffer with dimension array" approach. Dynamically-typed. Supports both internally or externally allocated buffers.

6.3.2 Constructor & Destructor Documentation

6.3.2.1 BasicTensor() [1/3]

```
template<typename T >
DG::BasicTensor::BasicTensor (
            int32_t id,
            const std::string & name,
            const shape_t & shape,
            const quant_params_t & quant_params = {},
            T * ext_lin_buffer = nullptr )  [inline]
```

Constructor: allocate or assign tensor memory according to dimension vector (static typing version)

Parameters

in	*id*	- tensor ID
in	*name*	- tensor name
in	*shape*	- tensor shape vector, which defines dimensions of new tensor
in	*quant_params*	- tensor quantization parameters
in	*ext_lin_buffer*	- optional external linear buffer pointer; if nullptr, buffer will be allocated internally

6.3.2.2 BasicTensor() [2/3]

```
DG::BasicTensor::BasicTensor (
            int32_t id,
            const std::string & name,
            const shape_t & shape,
            DGType data_type,
            const quant_params_t & quant_params = {},
            void * ext_lin_buffer = nullptr )  [inline]
```

Constructor: allocate or assign tensor memory according to dimension vector (dynamic typing version)

Parameters

in	*id*	- tensor ID
in	*name*	- tensor name
in	*shape*	- tensor shape vector, which defines dimensions of new tensor
in	*data_type*	- tensor element data type id
in	*quant_params*	- tensor quantization parameters
in	*ext_lin_buffer*	- optional external linear buffer pointer; if nullptr, buffer will be allocated internally

6.3.2.3 BasicTensor() [3/3]

```
template<class CONTAINER >
DG::BasicTensor::BasicTensor (
            const CONTAINER & source,
            bool do_copy = true,
            int32_t id = 0,
            const std::string & name = "",
            const shape_t & shape = {},
            const quant_params_t & quant_params = {} )  [inline], [explicit]
```

Constructor: allocate tensor memory and copy data from given linear container. CONTAINER type should have value_type, data(), and size() methods. If shape is empty, 1-D tensor the size of the linear container will be allocated. The size of the data copied is the minimum of the linear container size and the shape dimensions product.

Parameters

in	*source*	- linear container (such as string or vector) with tensor data
in	*do_copy*	- when true, copy data from container into tensor-owned buffer, otherwise assign pointer to container data
in	*id*	- tensor ID
in	*name*	- tensor name
in	*shape*	- tensor shape vector, which defines dimensions of new tensor
in	*quant_params*	- tensor quantization parameters

6.3.3 Member Function Documentation

6.3.3.1 alloc() [1/3]

```
template<class CONTAINER >
auto DG::BasicTensor::alloc (
            const CONTAINER & source,
            bool do_copy = true,
            int32_t id = 0,
            const std::string & name = "",
            const shape_t & shape = {},
            const quant_params_t & quant_params = {} ) -> decltype( source.data(), source.↩
size(), void() )

  [inline]
```

Allocate tensor memory and copy data from given linear container. CONTAINER type should have value_type, data(), and size() methods. If shape is empty, 1-D tensor the size of the linear container will be allocated. The size of the data copied is the minimum of the linear container size and the shape dimensions product.

Parameters

in	*source*	- linear container (such as string or vector) with tensor data
in	*do_copy*	- when true, copy data from container into tensor-owned buffer, otherwise assign pointer to container data
in	*id*	- tensor ID
in	*name*	- tensor name
in	*shape*	- tensor shape vector, which defines dimensions of new tensor
in	*quant_params*	- tensor quantization parameters

6.3.3.2 alloc() [2/3]

```
template<typename T >
void DG::BasicTensor::alloc (
            int32_t id,
            const std::string & name,
            const shape_t & shape,
            const quant_params_t & quant_params = {},
            T * ext_lin_buffer = nullptr )  [inline]
```

Allocate or assign tensor memory according to dimension vector (static typing version)

Parameters

in	*id*	- tensor ID
in	*name*	- tensor name
in	*shape*	- tensor shape vector, which defines dimensions of new tensor
in	*quant_params*	- tensor quantization parameters
in	*ext_lin_buffer*	- optional external linear buffer pointer; if nullptr, buffer will be allocated internally

6.3.3.3 alloc() [3/3]

```
void DG::BasicTensor::alloc (
            int32_t id,
            const std::string & name,
            const shape_t & shape,
            DGType data_type,
            const quant_params_t & quant_params = {},
            void * ext_lin_buffer = nullptr )  [inline]
```

Allocate or assign tensor memory according to dimension vector (dynamic typing version)

Parameters

in	*id*	- tensor ID
in	*name*	- tensor name
in	*shape*	- tensor shape vector, which defines dimensions of new tensor
in	*data_type*	- tensor element data type id
in	*quant_params*	- tensor quantization parameters
in	*ext_lin_buffer*	- optional external linear buffer pointer; if nullptr, buffer will be allocated internally

6.3.3.4 copy()

```
BasicTensor DG::BasicTensor::copy ( ) const  [inline]
```

Create and return deep copy of itself. External buffer data will be duplicated.

6.3.3.5 data() [1/2]

```
template<typename T >
T* DG::BasicTensor::data ( )  [inline]
```

Return non-const typed pointer to underlying linear buffer. Returns nullptr if template type does not match actual runtime tensor element type

6.3.3.6 data() [2/2]

```
template<typename T >
const T* DG::BasicTensor::data ( ) const  [inline]
```

Return const typed pointer to underlying linear buffer. Returns nullptr if template type does not match actual runtime tensor element type

6.3.3.7 dequantize()

```
template<typename T_IN , typename T_OUT >
void DG::BasicTensor::dequantize ( )  [inline]
```

Dequantize tensor according to current quantization settings from type T_IN to type T_OUT New linear buffer will be internally allocated to receive dequantized data.

6.3.3.8 isEqualData() [1/2]

```
FloatCompareResult< double > DG::BasicTensor::isEqualData (
            const BasicTensor & rhs,
            double maxRelDiff ) const  [inline]
```

Check, if this object and given object have equal data contents

Parameters

in	*rhs*	- object to compare with
in	*maxRelDiff*	- maximum relative difference for floating point comparison (ignored for integer data)

6.3.3.9 isEqualData() [2/2]

```
template<typename T >
FloatCompareResult< double > DG::BasicTensor::isEqualData (
            const std::vector< T > & rhs,
            double maxRelDiff ) const  [inline]
```

Check, if this object and given vector have equal data contents

Parameters

in	*rhs*	- vector to compare with
in	*maxRelDiff*	- maximum relative difference for floating point comparison (ignored for integer data)

6.3.3.10 isEqualDataShape()

```
bool DG::BasicTensor::isEqualDataShape (
            const BasicTensor & rhs ) const  [inline]
```

Check, if this object and given object have equal shape, element size and number of elements

Parameters

in	*rhs*	- object to compare with

6.3.3.11 isEqualStruct()

```
bool DG::BasicTensor::isEqualStruct (
            const BasicTensor & rhs ) const  [inline]
```

Check, if this object and given object have equal structure

Parameters

in	*rhs*	- object to compare with

6.3.3.12 quantize()

```
template<typename T_IN , typename T_OUT >
void DG::BasicTensor::quantize ( )  [inline]
```

Quantize tensor according to current quantization settings from type T_IN to type T_OUT New linear buffer will be internally allocated to receive quantized data.

6.3.3.13 reshapeTo()

```
void DG::BasicTensor::reshapeTo (
            size_t dim )  [inline]
```

Reshape tensor to given shape When new shape is bigger than current, shape is appended with unity values When new shape is smaller than current, shape is reduced, and last shape dimension is multiplied by product of reduced dimensions

The documentation for this class was generated from the following file:

- dg_tensor_structs.h

6.4 DG::JsonHelper Class Reference

Json helper class: contains assorted static methods to operate with Json configurations.

`#include <dg_json_helpers.h>`

Public Types

- using serial_container_t = json::binary_t::container_type

  Container type for serialized Json array (it is std::vector< unit8_t >)

Static Public Member Functions

- static std::string errorCheck (const json &response, const std::string &source, bool do_throw=true)
- static json errorCreate (const std::string &message)

  Create and return Json error packet with given message.
- static json jsonDeserialize (const serial_container_t &v)
- static serial_container_t jsonSerialize (const json &j)
- static json parse (const std::string &json_cfg, const char *file, const char *line, const char *func)
- static json parse_ignore_errors (const std::string json_cfg)
- static BasicTensor tensorDeserialize (const json &j)
- static json tensorSerialize (const BasicTensor &t)

6.4.1 Detailed Description

Json helper class: contains assorted static methods to operate with Json configurations.

6.4.2 Member Function Documentation

6.4.2.1 errorCheck()

```
static std::string DG::JsonHelper::errorCheck (
            const json & response,
            const std::string & source,
            bool do_throw = true )  [inline], [static]
```

Check server Json response for errors and throw exception, if any

Parameters

in	*response*	- Json response from server/core
in	*source*	- description of the server command initiator
in	*do_throw*	- if true, throws exception in case of error

Returns

original error string, if error is detected, empty string otherwise

6.4.2.2 jsonDeserialize()

```
static json DG::JsonHelper::jsonDeserialize (
            const serial_container_t & v )  [inline], [static]
```

Deserialize given byte vector with msgpack data into Json array

Parameters

in	*v*	- byte vector with msgpack data to deserialize

Returns

Json array

6.4.2.3 jsonSerialize()

```
static serial_container_t DG::JsonHelper::jsonSerialize (
            const json & j )  [inline], [static]
```

Serialize given Json array into byte vector using conversion to msgpack

Parameters

in	*j*	- Json array to serialize

Returns

byte vector with Json array contents converted to msgpack representation

6.4.2.4 parse()

```
static json DG::JsonHelper::parse (
            const std::string & json_cfg,
            const char * file,
            const char * line,
            const char * func )  [inline], [static]
```

Parse given Json string with proper exception handling

Parameters

in	*json_cfg*	- Json string to parse
in	*file*	- file where parsing happens
in	*line*	- line in file where parsing happens
in	*func*	- function where parsing happens

Returns

Json array

6.4.2.5 parse_ignore_errors()

```
static json DG::JsonHelper::parse_ignore_errors (
            const std::string json_cfg )  [inline], [static]
```

Parse given Json string with swallowing exceptions

Parameters

in	*json_cfg*	- Json string to parse

Returns

Json array or empty array, if parsing failed

6.4.2.6 tensorDeserialize()

```
static BasicTensor DG::JsonHelper::tensorDeserialize (
            const json & j )  [inline], [static]
```

Convert Json array to basic tensor

Parameters

in	*j*	- Json array

Returns

basic tensor

6.4.2.7 tensorSerialize()

```
static json DG::JsonHelper::tensorSerialize (
            const BasicTensor & t )  [inline], [static]
```

Convert basic tensor to Json array

Parameters

in	*t*	- basic tensor

Returns

Json array

The documentation for this class was generated from the following file:

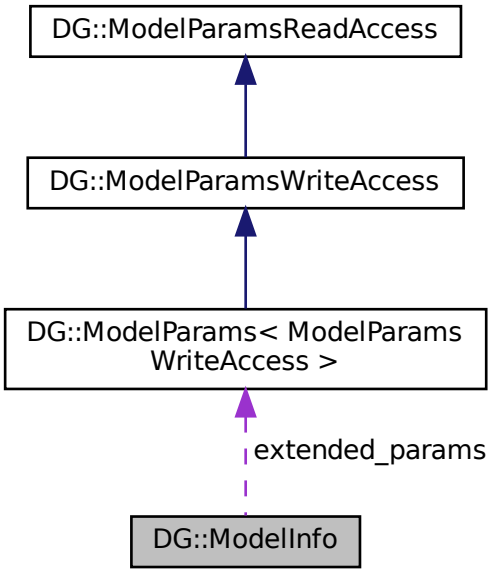
- dg_json_helpers.h

6.5 DG::ModelInfo Struct Reference

ModelInfo is the model identification structure. It keeps AI model key attributes.

```
#include <dg_client_structs.h>
```

Collaboration diagram for DG::ModelInfo:



Public Attributes

- int C
  input color depth
- std::string device_type
  device type on which model runs
- DG::ModelParamsWriter extended_params
  extended model parameters
- int H
  input height
- size_t id
  unique model ID
- std::string input_color_space
  for image inputs, image color-space the model expects: "BGR", "RGB"
- std::string input_image_format
  for image inputs, image format: "JPEG", "RAW" (this is user-tunable runtime parameter)
- std::string input_raw_data_type
  for "RAW" image inputs, image pixel data type: "DG_FLT", "DG_UINT8" (this is user-tunable runtime parameter)
- std::string input_tensor_layout
  for image inputs, image tensor layout the model expects: "NHWC", "NCHW"
- std::string input_type
  input data type: "Image", ...
- bool model_pruned
  'is model pruned (not dense)' flag
- bool model_quantized

'is model quantized' flag

- int N
  input frame depth
- std::string name
  model string name
- std::string runtime_agent
  runtime agent type on which model runs
- int W
  input width

6.5.1 Detailed Description

ModelInfo is the model identification structure. It keeps AI model key attributes.

The documentation for this struct was generated from the following file:

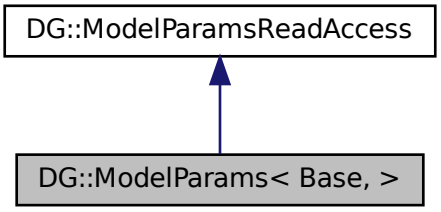
- dg_client_structs.h

6.6 DG::ModelParams< Base, > Class Template Reference

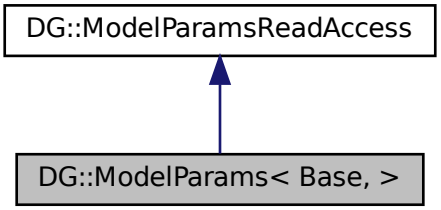
ModelParams is model parameters collection with user-defined access rights.

`#include <dg_model_parameters.h>`

Inheritance diagram for DG::ModelParams< Base, >:



Collaboration diagram for DG::ModelParams< Base, >:



Public Member Functions

- ModelParams (const char *json_text="{}")
- ModelParams (const json &json_cfg)
- ModelParams (const std::string &json_text)
- ModelParams (ModelParams &&rhs)

  Move constructor.
- ModelParams (ModelParams const &rhs)

  Copy constructor.
- ModelParams & operator= (ModelParams &&rhs)

  Move assignment.
- ModelParams & operator= (ModelParams const &rhs)

  Copy assignment.

Additional Inherited Members

6.6.1 Detailed Description

template<class Base = ModelParamsReadAccess, std::enable_if_t< std::is_base_of_v< ModelParamsReadAccess, Base >, bool > = false>
class DG::ModelParams< Base, >

ModelParams is model parameters collection with user-defined access rights.

This class provides programmatic type-safe access to model parameters defined in Json model configuration while owning that Json array. Access (read or write) is defined by the Base template parameter, which is used as the base class.

Template Parameters

Base	- ModelParamsReadAccess-derived base class: it can be ModelParamsReadAccess or ModelParamsWriteAccess

6.6.2 Constructor & Destructor Documentation

6.6.2.1 ModelParams() [1/3]

```
template<class Base = ModelParamsReadAccess, std::enable_if_t< std::is_base_of_v< Model↩
ParamsReadAccess, Base >, bool > = false>
DG::ModelParams< Base, >::ModelParams (
            const char * json_text = "{}" )  [inline]
```

Constructor. Creates model parameter collection by parsing Json text in character array

Parameters

in	*json_text*	- Json text with model parameters. It will be parsed and stored in internal Json array.

6.6.2.2 ModelParams() [2/3]

```
template<class Base = ModelParamsReadAccess, std::enable_if_t< std::is_base_of_v< Model↩
ParamsReadAccess, Base >, bool > = false>
DG::ModelParams< Base, >::ModelParams (
            const std::string & json_text )  [inline], [explicit]
```

Constructor. Creates model parameter collection by parsing Json text in std::string

Parameters

in	*json_text*	- Json text with model parameters. It will be parsed and stored in internal Json array.

6.6.2.3 ModelParams() [3/3]

```
template<class Base = ModelParamsReadAccess, std::enable_if_t< std::is_base_of_v< Model↩
ParamsReadAccess, Base >, bool > = false>
DG::ModelParams< Base, >::ModelParams (
            const json & json_cfg )  [inline], [explicit]
```

Constructor. Creates model parameter collection from given Json array

Parameters

in	*json_cfg*	- Json array with model parameters. It will be copied into internal Json array.

The documentation for this class was generated from the following file:

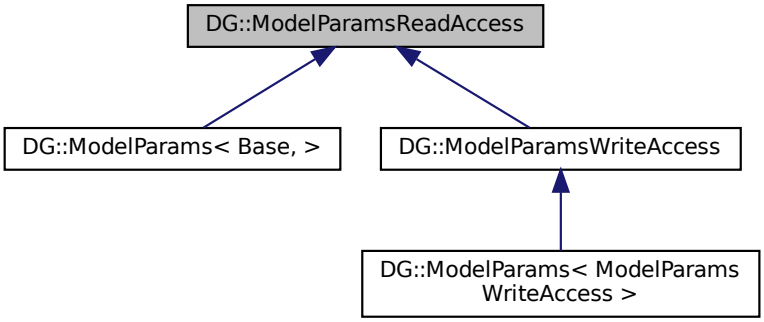
- dg_model_parameters.h

6.7 DG::ModelParamsReadAccess Class Reference

ModelParamsReadAccess is read-only accessor to model parameters.

```
#include <dg_model_parameters.h>
```

Inheritance diagram for DG::ModelParamsReadAccess:



Public Member Functions

- const json & jsonGet () const
- ModelParamsReadAccess (const json &config)
- template<typename T >
  T None (size_t idx=0) const

  Stub for empty getter fallback.
- bool None_exist (size_t idx=0) const

  Stub for empty existence checker fallback.
- template<typename T >
  T None_get (size_t idx=0) const

  Stub for empty getter fallback.
- operator const json & ()
- operator std::string ()
- const json & sectionGet (const std::string &section, size_t idx=0) const
- size_t sectionHashGet (const std::string &section, size_t idx=0) const
- size_t sectionSizeGet (const std::string &section) const

Protected Member Functions

- bool paramExist (const char *section, const char *key, size_t idx, bool(ModelParamsReadAccess::*fallback)(size_t) const) const
- template<typename T >
  T paramGet (const char *section, const char *key, bool is_mandatory, const T &default_value, size_t idx, T(ModelParamsReadAccess::*fallback)(size_t) const) const

Protected Attributes

- const json & m_cfg_ro

  non-owning reference to Json array with model configuration

6.7.1 Detailed Description

ModelParamsReadAccess is read-only accessor to model parameters.

This class provides programmatic type-safe read access to model parameters defined in Json model configuration. It keeps non-owning const reference to underlying Json array. For each model parameter it provides getter method, which name matches the parameter name as it appears in Json array.

6.7.2 Constructor & Destructor Documentation

6.7.2.1 ModelParamsReadAccess()

```
DG::ModelParamsReadAccess::ModelParamsReadAccess (
            const json & config )  [inline]
```

Constructor. Attaches model parameter read-only accessor to Json array

Parameters

in	*config*	- Json array with model configuration

6.7.3 Member Function Documentation

6.7.3.1 jsonGet()

```
const json& DG::ModelParamsReadAccess::jsonGet ( ) const  [inline]
```

Access to underlying Json array

Returns

constant non-owning reference to Json array with model configuration

6.7.3.2 operator const json &()

```
DG::ModelParamsReadAccess::operator const json & ( )  [inline]
```

Access to underlying Json array

Returns

constant non-owning reference to Json array with model configuration

6.7.3.3 operator std::string()

```
DG::ModelParamsReadAccess::operator std::string ( )  [inline]
```

Access to underlying Json array as string

Returns

Json text with model configuration

6.7.3.4 paramExist()

```
bool DG::ModelParamsReadAccess::paramExist (
            const char * section,
            const char * key,
            size_t idx,
            bool(ModelParamsReadAccess::*)(size_t) const fallback ) const  [inline], [protected]
```

Check parameter for existence

Template Parameters

T	- parameter type

Parameters

in	*section*	- top section name; if empty, key is taken from topmost level
in	*key*	- key of key-value pair
in	*idx*	- array index inside section object
in	*fallback*	- existence checker for fallback parameter

Returns

parameter value

6.7.3.5 paramGet()

```
template<typename T >
T DG::ModelParamsReadAccess::paramGet (
            const char * section,
            const char * key,
            bool is_mandatory,
            const T & default_value,
            size_t idx,
            T(ModelParamsReadAccess::*)(size_t) const fallback ) const  [inline], [protected]
```

Get parameter from Json array

Template Parameters

T	- parameter type

Parameters

in	*section*	- top section name; if empty, key is taken from topmost level
in	*key*	- key of key-value pair
in	*is_mandatory*	- if parameter mandatory
in	*default_value*	- value to return, if no such key is found
in	*idx*	- array index inside section object
in	*fallback*	- fallback parameter getter

Returns

parameter value

6.7.3.6 sectionGet()

```
const json& DG::ModelParamsReadAccess::sectionGet (
            const std::string & section,
            size_t idx = 0 ) const  [inline]
```

Access to underlying Json sub-array for particular section

Parameters

in	*section*	- section name to access (one of SECT_xxx constants)
in	*idx*	- array index inside section object

Returns

constant non-owning reference to Json array with particular model section configuration

6.7.3.7 sectionHashGet()

```
size_t DG::ModelParamsReadAccess::sectionHashGet (
            const std::string & section,
            size_t idx = 0 ) const  [inline]
```

Compute the hash of the parameter values in a given section

Parameters

in	*section*	- section name to compute hash for
in	*idx*	- array index inside section object

Returns

std::hash for a given section

6.7.3.8 sectionSizeGet()

```
size_t DG::ModelParamsReadAccess::sectionSizeGet (
            const std::string & section ) const  [inline]
```

Get size of underlying Json sub-array for particular section

Parameters

in	*section*	- section name to access (one of SECT_xxx constants)

Returns

section sub-array size in elements

The documentation for this class was generated from the following file:

- dg_model_parameters.h

6.8 DG::ModelParamsSection Struct Reference

Model parameters section descriptor.

```
#include <dg_model_parameters.h>
```

Public Attributes

- bool is_scalar

  scalar vs vector flag: vector sections contain more than one element
- const char * label

  section name string

6.8.1 Detailed Description

Model parameters section descriptor.

The documentation for this struct was generated from the following file:

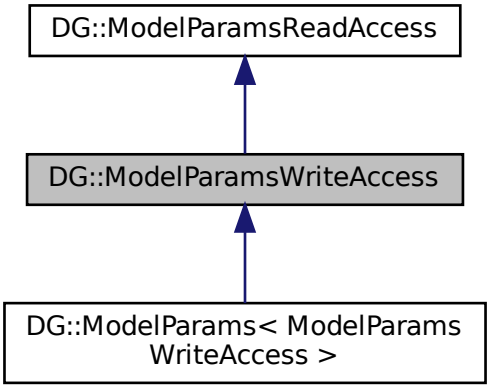
- dg_model_parameters.h

6.9 DG::ModelParamsWriteAccess Class Reference

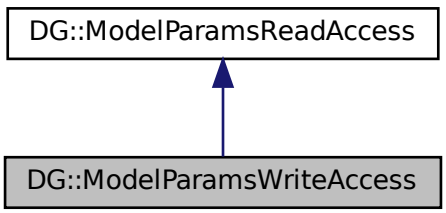
ModelParamsWriteAccess is read/write accessor to model parameters.

`#include <dg_model_parameters.h>`

Inheritance diagram for DG::ModelParamsWriteAccess:



Collaboration diagram for DG::ModelParamsWriteAccess:



Public Member Functions

- bool is_dirty () const

  Check if dirty: at least one of the parameters was changed.
- ModelParamsWriteAccess & merge (const json &config)
- ModelParamsWriteAccess (json &config)
- void set_dirty (bool state)

Protected Member Functions

- template<typename T >
  ModelParamsWriteAccess & paramSet (const char *section, const char *key, const T &value, size_t idx)

Protected Attributes

- json & m_cfg_rw

  non-owning reference to Json array with model configuration
- bool m_dirty

  'some parameter was changed' flag

6.9.1 Detailed Description

ModelParamsWriteAccess is read/write accessor to model parameters.

This class provides programmatic type-safe read and write access to model parameters defined in Json model configuration. It keeps non-owning non-const reference to underlying Json array. For each model parameter it provides both getter and setter methods. Getter method name matches the parameter name as it appears in Json array. Setter method name is constructed from the parameter name by appending _set suffix. For example, Input↩ ImgFmt_set()

6.9.2 Constructor & Destructor Documentation

6.9.2.1 ModelParamsWriteAccess()

```
DG::ModelParamsWriteAccess::ModelParamsWriteAccess (
            json & config ) [inline]
```

Constructor. Attaches model parameter read-write accessor to Json array

Parameters

in	*config*	- Json array with model configuration

6.9.3 Member Function Documentation

6.9.3.1 merge()

```
ModelParamsWriteAccess& DG::ModelParamsWriteAccess::merge (
            const json & config )  [inline]
```

Merge given model configuration Json array with self Note: only runtime parameters will be merged

Parameters

in	*config*	- model configuration Json array to be merged

6.9.3.2 paramSet()

```
template<typename T >
ModelParamsWriteAccess& DG::ModelParamsWriteAccess::paramSet (
            const char * section,
            const char * key,
            const T & value,
            size_t idx )  [inline], [protected]
```

Set parameter to Json array

Template Parameters

T	- parameter type

Parameters

in	*section*	- top section name; if empty, key is placed in topmost level
in	*key*	- key of key-value pair
in	*value*	- value to set
in	*idx*	- array index inside section object

6.9.3.3 set_dirty()

```
void DG::ModelParamsWriteAccess::set_dirty (
            bool state )  [inline]
```

Set dirty flag

Parameters

in	*state*	- dirty flag value to set

The documentation for this class was generated from the following file:

- dg_model_parameters.h

6.10 DG::ModelQuery Struct Reference

Model query structure used to search models on AI server which match a set of provided model attributes.

`#include <dg_model_api.h>`

Public Types

- enum TriState { Yes, No, Dont_care }
  Tri-state boolean enum.

Public Attributes

- std::string device_type
  device type to use for inference (optional, can be empty if don't care)
- std::string model_name
  any part of the model name (mandatory parameter)
- TriState model_pruned = Dont_care
  model should be pruned/sparse (not dense)
- TriState model_quantized = Dont_care
  model should be quantized
- std::string runtime_agent
  runtime agent to use for inference (optional, can be empty if don't care)

6.10.1 Detailed Description

Model query structure used to search models on AI server which match a set of provided model attributes.

6.10.2 Member Enumeration Documentation

6.10.2.1 TriState

`enum DG::ModelQuery::TriState`

Tri-state boolean enum.

Enumerator

Yes	yes / true
No	no / false
Dont_care	not set / don't care

The documentation for this struct was generated from the following file:

- dg_model_api.h

6.11 DG::BasicTensor::quant_params_t Struct Reference

Quantization parameters (how to convert from integer/quantized data back to floating point)

`#include <dg_tensor_structs.h>`

Classes

- struct scale_t

Public Member Functions

- bool isEqualData (const quant_params_t &rhs, double maxRelDiff) const
- bool isEqualStruct (const quant_params_t &rhs) const
- int quant_axis () const

  Get quantization axis (-1 means global quantization)
- const std::vector< scale_t > & quant_params () const

  Get quantization parameters.
- quant_params_t ()

  Default constructor.
- quant_params_t (const scale_t &global_qparam)
- quant_params_t (int axis, const std::vector< scale_t > &qparams)
- template<typename SCALE_T , typename ZERO_T >
  quant_params_t (int axis, const std::vector< SCALE_T > &scales, const std::vector< ZERO_T > &zeros)
- template<typename T >
  std::vector< T > quant_scales () const

  Get array of scales.
- template<typename T >
  std::vector< T > quant_zeros () const

  Get array of zero offsets.

6.11.1 Detailed Description

Quantization parameters (how to convert from integer/quantized data back to floating point)

6.11.2 Constructor & Destructor Documentation

6.11.2.1 quant_params_t() [1/3]

```
DG::BasicTensor::quant_params_t::quant_params_t (
            const scale_t & global_qparam )  [inline]
```

Constructor for global quantization

Parameters

in	*global_qparam*	- global quantization parameters

6.11.2.2 quant_params_t() [2/3]

```
DG::BasicTensor::quant_params_t::quant_params_t (
            int axis,
            const std::vector< scale_t > & qparams )  [inline]
```

Constructor for per-axis quantization

Parameters

in	*axis*	- quantization axis
in	*qparams*	- vector of per-axis quantization parameters

6.11.2.3 quant_params_t() [3/3]

```
template<typename SCALE_T , typename ZERO_T >
DG::BasicTensor::quant_params_t::quant_params_t (
            int axis,
            const std::vector< SCALE_T > & scales,
            const std::vector< ZERO_T > & zeros )  [inline]
```

Alternative constructor for per-axis quantization

Parameters

in	*axis*	- quantization axis
in	*scales*	- vector of per-axis scales
in	*zeros*	- vector of per-axis zero offsets

6.11.3 Member Function Documentation

6.11.3.1 isEqualData()

```
bool DG::BasicTensor::quant_params_t::isEqualData (
            const quant_params_t & rhs,
            double maxRelDiff ) const  [inline]
```

Check, if this object and given object have equal data contents

Parameters

in	*rhs*	- object to compare with
in	*maxRelDiff*	- maximum relative difference for floating point comparison

6.11.3.2 isEqualStruct()

```
bool DG::BasicTensor::quant_params_t::isEqualStruct (
            const quant_params_t & rhs ) const  [inline]
```

Check, if this object and given object have equal structure

Parameters

in	*rhs*	- object to compare with

The documentation for this struct was generated from the following file:

- dg_tensor_structs.h

6.12 DG::BasicTensor::quant_params_t::scale_t Struct Reference

```
#include <dg_tensor_structs.h>
```

Public Attributes

- double m_scale
  scale factor
- int64_t m_zero
  zero offset

6.12.1 Detailed Description

Single quantization parameter: scale and zero offset Scaling formula is real_value = m_scale $*$ (int_value - m_zero)

The documentation for this struct was generated from the following file:

- dg_tensor_structs.h

6.13 DG::ServerAddress Struct Reference

ServerAddress is the server address structure. It keeps AI server TCP/IP address.

```
#include <dg_client_structs.h>
```

Public Member Functions

- const bool is_valid () const
- ServerAddress ()

  Default constructor.
- ServerAddress (std::string ip, int port=DEFAULT_PORT)

Public Attributes

- std::string ip

  server domain name or IP address string
- int port

  server TCP port number

6.13.1 Detailed Description

ServerAddress is the server address structure. It keeps AI server TCP/IP address.

6.13.2 Constructor & Destructor Documentation

6.13.2.1 ServerAddress()

```
DG::ServerAddress::ServerAddress (
            std::string ip,
            int port = DEFAULT_PORT )  [inline]
```

Constructor

Parameters

in	*ip*	- server domain name or IP address
in	*port*	- server TCP port

6.13.3 Member Function Documentation

6.13.3.1 is_valid()

```
const bool DG::ServerAddress::is_valid ( ) const  [inline]
```

Check server address validity

Returns

true if server address is valid

The documentation for this struct was generated from the following file:

- dg_client_structs.h

Chapter 7

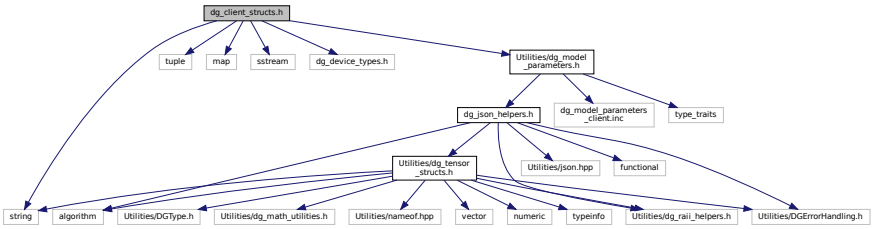
File Documentation

7.1 dg_client_structs.h File Reference

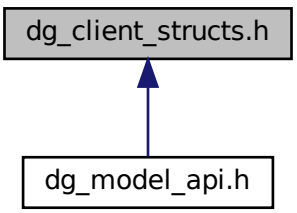
DG client API data types.

```
#include <string>
#include <tuple>
#include <map>
#include <sstream>
#include "dg_device_types.h"
#include "Utilities/dg_model_parameters.h"
```

Include dependency graph for dg_client_structs.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct DG::ModelInfo

  ModelInfo is the model identification structure. It keeps AI model key attributes.
- struct DG::ServerAddress

  ServerAddress is the server address structure. It keeps AI server TCP/IP address.

Namespaces

- DG

  DG is DeGirum namespace which groups all DeGirum software declarations.

Typedefs

- typedef struct DG::ModelInfo DG::ModelInfo

  ModelInfo is the model identification structure. It keeps AI model key attributes.
- typedef struct DG::ServerAddress DG::ServerAddress

  ServerAddress is the server address structure. It keeps AI server TCP/IP address.

Enumerations

- enum DG::RUNTIME_AGENT_TYPES {
  DG::RUNTIME_AGENT_TYPES::DEFAULT, DG::RUNTIME_AGENT_TYPES::N2X, DG::RUNTIME_AGENT_TYPES::TFLITE
  DG::RUNTIME_AGENT_TYPES::ONNX,
  DG::RUNTIME_AGENT_TYPES::OPENVINO, DG::RUNTIME_AGENT_TYPES::DUMMY }

  Runtime agent types.

Functions

- static std::string DG::messagePrepare (const json &in)
- bool DG::operator< (const ServerAddress &lhs, const ServerAddress &rhs)

  Comparison operator for server address.
- bool DG::operator== (const ServerAddress &lhs, const ServerAddress &rhs)

  Comparison operator for server address.

Variables

- const int DG::CURRENT_PROTOCOL_VERSION = 2

  current client-server protocol version
- const int DG::DEFAULT_PORT = 8778

  Default TCP port of AI server.
- const int DG::MIN_COMPATIBLE_PROTOCOL_VERSION = 2

  minimum compatible client-server protocol version
- constexpr const char * DG::PROTOCOL_VERSION_TAG = "VERSION"

  client-server protocol version tag

7.1.1 Detailed Description

DG client API data types.

This file contains declarations of various data types of DG client API:

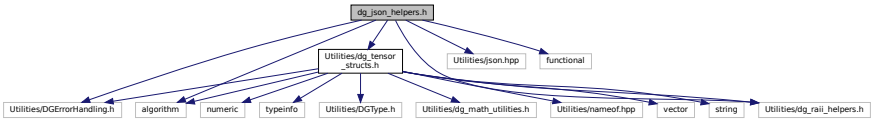
- device types
- runtime agent types
- model parameters, etc.

7.2 dg_json_helpers.h File Reference

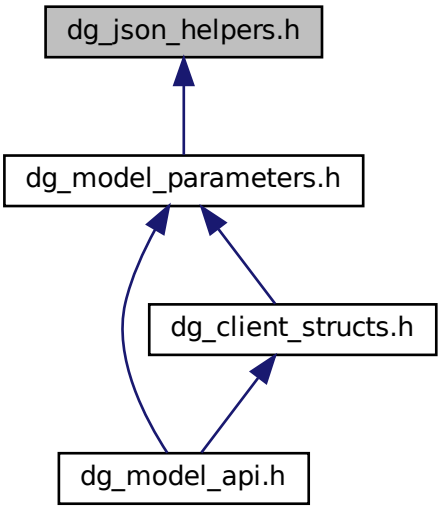
DG Core JSON helper classes and functions.

```
#include "Utilities/DGErrorHandling.h"
#include "Utilities/dg_tensor_structs.h"
#include "Utilities/dg_raii_helpers.h"
#include "Utilities/json.hpp"
#include <functional>
#include <algorithm>
```

Include dependency graph for dg_json_helpers.h:



This graph shows which files directly or indirectly include this file:



Classes

- class DG::JsonHelper

  Json helper class: contains assorted static methods to operate with Json configurations.

Namespaces

- DG

  DG is DeGirum namespace which groups all DeGirum software declarations.

Macros

- #define DG_JSON_PARSE(json_cfg) DG::JsonHelper::parse(json_cfg, __FILE__, TOSTRING(__LINE__), FUNCTION_NAME)

  Parse given Json string with proper exception handling.

Typedefs

- using DG::json = nlohmann::json

  Json library.

Functions

- template<typename T >
  T DG::jsonGetMandatoryValue (const json &json_params, const std::string &section, int index, const std::string &key)
- template<typename T >
  T DG::jsonGetOptionalValue (const json &json_params, const std::string &section, int index, const std::string &key, const T &default_value)
- static bool DG::jsonKeyExist (const json &json_params, const std::string &section, int index, const std::string &key)
- template<typename T >
  void DG::jsonSetOptionalValue (json &json_params, const std::string &section, int index, const std::string &key, const T &value)

7.2.1 Detailed Description

DG Core JSON helper classes and functions.

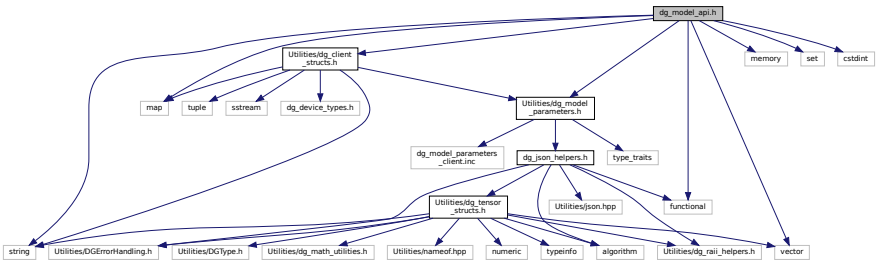
This file contains declaration of various helper classes and helper functions enhancing Json handling

7.3 dg_model_api.h File Reference

DG Client API for model inference.

```
#include <string>
#include <memory>
#include <vector>
#include <set>
#include <map>
#include <functional>
#include <cstdint>
#include "Utilities/dg_client_structs.h"
#include "Utilities/dg_model_parameters.h"
```

Include dependency graph for dg_model_api.h:



Classes

- class DG::AIModel

  AIModel is DeGirum AI client API class for simple non-pipelined sequential inference.
- class DG::AIModelAsync

  AIModelAsync is DeGirum AI client API class for efficient pipelined asynchronous inference.
- struct DG::ModelQuery

  Model query structure used to search models on AI server which match a set of provided model attributes.

Namespaces

- DG

  DG is DeGirum namespace which groups all DeGirum software declarations.

Enumerations

- enum DG::DetectionStatus { DG::DetectionStatus::OK, DG::DetectionStatus::ProtocolMismatch, DG::DetectionStatus::HostAliv
  }

  Enumerator showing server status as detected by the below function.

Functions

- std::vector< std::tuple< std::string, DG::DetectionStatus > > DG::detectHostnameServers (const std::string &pattern, const int range_start, const int range_end, const int numeral_width=3)
- std::vector< std::tuple< std::string, DetectionStatus > > DG::detectSubnetServers (const std::string &root↩ _ip, const std::string &subnet_mask)
- std::string DG::errorCheck (const json &json_response)
- json DG::labelDictionary (const std::string &server, const std::string &model_name)
- ModelInfo DG::modelFind (const std::string &server, const ModelQuery &query)
- void DG::modelzooListGet (const std::string &server, std::vector< ModelInfo > &modelzoo_list)
- std::string DG::versionGet ()

7.3.1 Detailed Description

DG Client API for model inference.

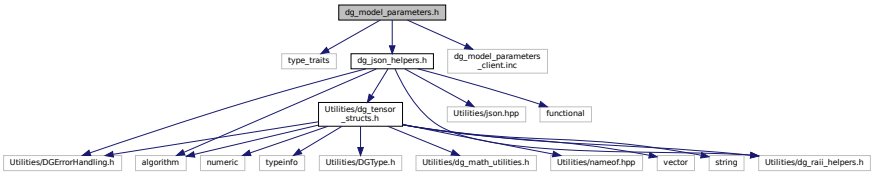
This file contains declaration of DG Client API functions and classes for model inference

7.4 dg_model_parameters.h File Reference

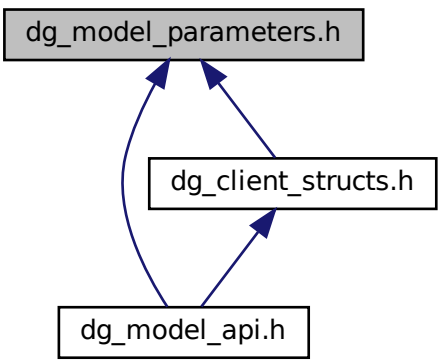
DG centralized handling of Json model parameters.

```
#include <type_traits>
#include "dg_json_helpers.h"
#include "dg_model_parameters_client.inc"
```

Include dependency graph for dg_model_parameters.h:



This graph shows which files directly or indirectly include this file:



Classes

- class DG::ModelParams< Base, >

  ModelParams is model parameters collection with user-defined access rights.
- class DG::ModelParamsReadAccess

  ModelParamsReadAccess is read-only accessor to model parameters.
- struct DG::ModelParamsSection

  Model parameters section descriptor.
- class DG::ModelParamsWriteAccess

  ModelParamsWriteAccess is read/write accessor to model parameters.

Namespaces

- DG

  DG is DeGirum namespace which groups all DeGirum software declarations.

Typedefs

- using DG::ModelParamsWriter = ModelParams< ModelParamsWriteAccess >

  ModelParamsWriter is ModelParams template instantiation with write access.

Variables

- const int DG::MODEL_PARAMS_CURRENT_VERSION = 3
- const int DG::MODEL_PARAMS_MIN_COMPATIBLE_VERSION = 1
- constexpr ModelParamsSection DG::SECT_DEVICE = { "DEVICE", true }

  device parameters section
- constexpr ModelParamsSection DG::SECT_INTERNAL = { "INTERNAL", true }

  internal parameters section
- constexpr ModelParamsSection DG::SECT_MODEL_PARAMETERS = { "MODEL_PARAMETERS", true }

  model parameters section
- constexpr ModelParamsSection DG::SECT_POST_PROCESS = { "POST_PROCESS", true }

  post-processing parameters section
- constexpr ModelParamsSection DG::SECT_PRE_PROCESS = { "PRE_PROCESS", false }

  pre-processing parameters section
- constexpr ModelParamsSection DG::SECT_TOP = { "", true }

  Json file top-level sections.

7.4.1 Detailed Description

DG centralized handling of Json model parameters.

This file implements classes for centralized handling of Json model parameters. Each AI model in DG framework is accompanied with Json configuration file, which defines all model configuration parameters. Class list: Model↩ ParamsReadAccess - read-only model parameters accessor ModelParamsWriteAccess - read/write model parameters accessor ModelParams - model parameters collection with access type defined by the template parameter

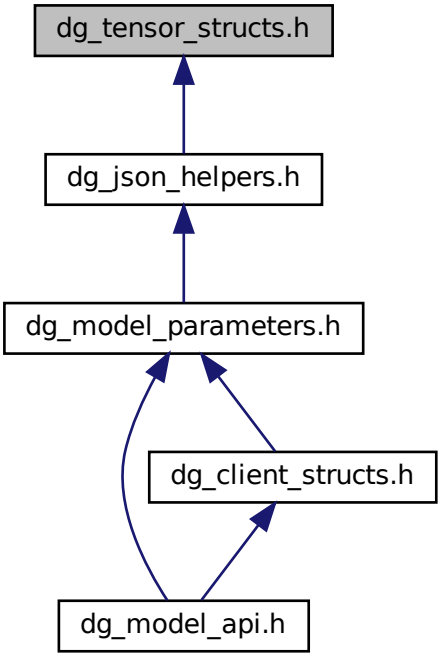
7.5 dg_tensor_structs.h File Reference

DG tensor container classes.

```
#include <vector>
#include <string>
#include <algorithm>
#include <numeric>
#include <typeinfo>
#include "Utilities/DGErrorHandling.h"
#include "Utilities/DGType.h"
#include "Utilities/dg_math_utilities.h"
#include "Utilities/dg_raii_helpers.h"
#include "Utilities/nameof.hpp"
```

Include dependency graph for dg_tensor_structs.h:

This graph shows which files directly or indirectly include this file:



Classes

- class DG::BasicTensor

- struct DG::BasicTensor::quant_params_t

  Quantization parameters (how to convert from integer/quantized data back to floating point)
- struct DG::BasicTensor::quant_params_t::scale_t

Namespaces

- DG

  DG is DeGirum namespace which groups all DeGirum software declarations.

Typedefs

- using DG::BasicTensorVector = std::vector< BasicTensor >

  Collection of tensors.

Functions

- std::ostream & DG::operator<< (std::ostream &os, const BasicTensor &t)

  Print tensor to stream.

7.5.1 Detailed Description

DG tensor container classes.

This file contains declaration of tensor container classes

Index