

# **ZTACHIP**

## **Vision-AI Stack**

### **Programmer Guide**

#### **v1.0**

Author: Vuong Nguyen

<https://github.com/ztachip/ztachip>

[vuongdnguyen@hotmail.com](mailto:vuongdnguyen@hotmail.com)

# 1-INTRODUCTION

---

ztachip provides many pre-built acceleration functions for vision and AI applications.

To support these acceleration functions, a graph based framework is introduced. Different vision and AI functions are connected together into a graph of execution nodes.

Users can use this graph framework to integrate their own custom acceleration functions with ztachip vision-ai stack.

# 2- GRAPH FRAMEWORK

---

Graph is a ztchip framework used to connect different processing nodes together. It is particular popular framework used by many other AI and vision frameworks such as OpenCV, TensorFlow, OpenVX, etc...

Each ztchip acceleration functions are packaged as a graph node.

The flow of execution is then specified by how the graph nodes are connected together to form a graph.

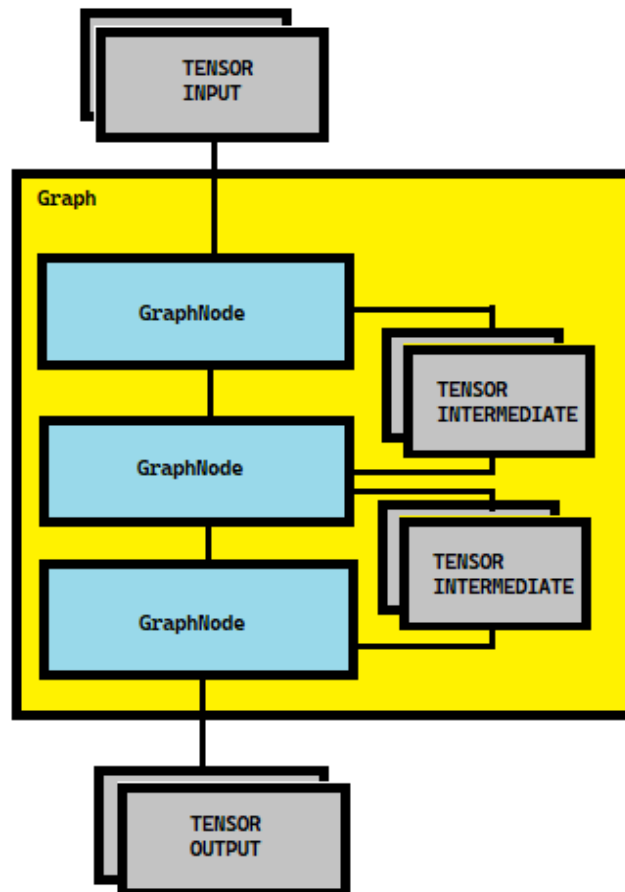
There can be multiple Graph objects representing different execution flow.

ztchip graph framework is composed of the following C++ classes:

- TENSOR: Objects that encapsulate tensor data objects. They are used as...
  - Input tensor data to a graph
  - Output result tensor data from a graph.
  - Intermediate tensor to transfer data between graph nodes.
- GraphNode
  - Unit of execution in a graph. GraphNode takes input data tensor from previous graph nodes and transfer output data tensor to next graph nodes.
- Graph
  - Objects that represent the graph.

## 1.1 *Graph structure*

Diagram below illustrates how the main objects of a Graph are interconnected.



## 1.2 TENSOR

TENSOR class encapsulates tensor data objects.

Data exchange between graph nodes are TENSOR objects.

### 1.2.1 Class Interface

#### 1.2.1.1 TENSOR()

Default constructor without initialization.

#### 1.2.1.2 TENSOR( TensorDataType \_dataType, TensorFormat \_fmt, TensorObjType objType, std::vector<int> &dim, void \*shm)

Constructor with initialization.

#### Input

_dataType	Data type of this tensor. Reference 1.2.2.1 for TensorDataType definition.
_fmt	Layout format of this tensor Reference 1.2.2.2 for TensorFormat definition
objType	Object type of this tensor Reference 1.2.2.3 for TensorObjType definition.
dim	Dimension size of this tensor.
shm	If non-zero then use this parameter as memory allocation block for this tensor. In this case, this object does not own this memory block and will not free it when done. If zero then allocate new memory block for this tensor. The memory is owned by this object will be freed by this object's destructor.

**1.2.1.3      ZtaStatus Create(**  
                   TensorDataType \_dataType,  
                   TensorFormat \_fmt,  
                   TensorObjType \_objType,  
                   std::vector<int> &dim,  
                   ZTA\_SHARED\_MEM \_shm=0)

Call to initialize this object when default constructor was used.

Parameters are similar to 1.2.1.2

Output:

- ZtaStatusOk if successful
- ZtaStatusFail otherwise.

**1.2.1.4      ZtaStatus Clone(TENSOR \*other)**

To initialize this object to have the same parameters as another tensor.

New memory block is also allocated and initialized to have the same contents as the other tensor.

Input

other	Reinitialize this tensor with contents of 'other' tensor.
-------	-----------------------------------------------------------

Output:

- ZtaStatusOk if successful
- ZtaStatusFail otherwise.

**1.2.1.5      ZtaStatus Alias(TENSOR \*other)**

Initialize this object to be a reference to another TENSOR object.

Input

other	This object is just a reference to the 'other' tensor. This tensor does not own its data contents since it is just referencing other tensor's data contents.
-------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------

Output:

- ZtaStatusOk if successful
- ZtaStatusFail otherwise.

#### 1.2.1.6 ZtaStatus Alias(void \*\_shm)

Data content for this tensor is a reference to a given allocated memory block.  
This tensor does not own the memory block and will not free it upon completion.

##### Input

_shm	This tensors data content is referencing '_shm' memory block.
------	---------------------------------------------------------------

##### Output:

- ZtaStatusOk if successful
- ZtaStatusFail otherwise.

#### 1.2.1.7 ZtaStatus CreateWithBitmap( const char \*bmpFile, TensorFormat fmt=TensorFormatSplit)

Initialize this tensor with the dimensions of a bitmap.  
Load the bitmap content into this tensor.

##### Input

bmpFile	File name of the bitmap to initialize this tensor with. Bitmap file must be 24-bit BMP format.
Fmt	Layout format of this tensor. Reference 1.2.2.2 for TensorFormat definition.

##### Output:

- ZtaStatusOk if successful
- ZtaStatusFail otherwise.

#### 1.2.1.8 TensorDataType GetDataType()

Return data type of this tensor.  
Reference 1.2.2.1 for TensorDataType definition.

#### 1.2.1.9 TensorFormat GetFormat()

Return data layout format of this tensor.  
Reference 1.2.2.2 for TensorFormat definition.

#### 1.2.1.10 TensorObjType GetObjType()

Return object type of this tensor.  
Reference 1.2.2.3 for TensorObjType definition.

#### 1.2.1.11 `std::vector<int> *GetDimension()`

Return dimension list of this tensor.

The list starts with size of outer-most dimension and ends with size inner-most dimension.

#### 1.2.1.12 `int GetDimension(int _idx)`

Return size of a dimension of this tensor.

##### Input

<code>_idx</code>	Dimension index to return its size. <code>_idx</code> ranges from 0 to (num_dimension-1) with 0 means outer-most dimension and (num_dimension-1) means inner-most dimension.
-------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

##### Output:

- `ZtaStatusOk` if successful
- `ZtaStatusFail` otherwise.

#### 1.2.1.13 `void *GetBuf()`

Return data buffer address of this tensor.

#### 1.2.1.14 `int GetBufLen()`

Return total length of data buffer of this tensor.

#### 1.2.1.15 `static size_t GetTensorSize(std::vector<int>& shape)`

This is a utility function that returns the buffer size for a tensor with a particular dimension.

Size return is number of elements for the tensor.



## 1.2.2 Data Types

### 1.2.2.1 TensorDataType Enumeration

This class supports the following data types

TensorDataTypeInt8	Signed 8-bit integer.
TensorDataTypeUInt8	Unsigned 8-bit integer.
TensorDataTypeInt16	Signed 16-bit integer.
TensorDataTypeUInt16	Unsigned 16-bit integer.

### 1.2.2.2 TensorFormat Enumeration

This enumeration supports for the following data layout format

<i>TensorFormatInterLeaved</i>	<p>For example with a tensor 3x2 In this layout format, tensor elements layout in data buffer is as followed.</p> <p>[0][0] [1][0] [2][0] [0][1] [1][1] [2][1]</p>
<i>TensorFormatSplit</i>	<p>For example with a tensor 3x2 In this layout format, tensor elements layout in data buffer is as followed.</p> <p>[0][0] [0][1] [1][0] [1][1] [2][0] [2][1]</p>

### 1.2.2.3 TensorObjType Enumeration

This enumeration supports the following tensor object types

TensorObjTypeRGB	Object type is an image with pixel color in RGB order
TensorObjTypeBGR	Image with pixel color in BGR order
TensorObjTypeYUYV	Image in YUYV color space.
TensorObjTypeMonochrome	Monochrome image but in RGB format with R,G,B having same values
TensorObjTypeMonochromeSingleChannel	Monochrome image but only with 1 bytes per pixel representing the intensity
TensorObjTypeUnknown	Unknown data object type

## 1.3 *GraphNode Class*

This is a class template with virtual functions to be implemented by a derived class. Objects with GraphNode as base class are the execution units of a graph.

ztachip acceleration functions implemented by corresponding tensor programs and pcore programs are encapsulated within a derived class of GraphNode.

### 1.3.1 Class Interface

#### 1.3.1.1 GraphNode()

Default constructor

#### 1.3.1.2 ZtaStatus Verify()

This is a virtual function to be implemented by a derived class. The derived class verifies the integrity of this graph node and performs any necessary initialization required before the start of execution.

Output:

- ZtaStatusOk if successful
- ZtaStatusFail otherwise.

#### 1.3.1.3 ZtaStatus Execute(int queue,bool stepMode)

This is a virtual function to be implemented by a derived class. The derived class perform the execution associated with this node.

Input

queue	There may be multiple graphs running simultaneously. Each graph has a unique queue id. Graph node will use this queue id to generate a unique job-id which will then be passed to ztachip for task completion notification.
stepMode	If false then execute this node till completion. If true then partially execute this node. This function will be invoked again for the node to continue with the execution. This is useful when a graph node may take a long execution time, and step mode allows execution of a slow graph to be preempted by other more critical graph.

Output:

- ZtaStatusOk if processing is completed successfully.
- ZtaStatusPending if processing is successful but not fully completed. More processing is still

required.

- ZtaStatusFail if errors are encountered.

#### 1.3.1.4      `uint32_t GetJobId(int queue)`

Generate a unique job id for a tensor program execution.

Refer to [1] on how tensor program would use this job-id.

Example in [1] shows that tensor program is waiting for the completion of the task by waiting for the notification message from ztchip about the completion of job-id. However, when tensor program is called from within a graph framework, tensor program must not wait for the completion message since this is done by graph framework instead.

##### Input

queue	The same as queue id passed from Execute function (1.3.1.3) Each graph has a unique queue id.
-------	--------------------------------------------------------------------------------------------------

##### Output:

Unique job id for a tensor processing task.

### 1.3.2 Example of implementing a graph node.

Below is an example that shows how a new graph node is implemented.

GraphNode primary function is to provide wrapper functions for a tensor program so that tensor program can be invoked as part of a graph execution.

```
// Declare a new graph node. It is derived from GraphNode
class MyGraphNode : public GraphNode {
    MyGraphNode();
    ~MyGraphNode();
    ZtaStatus Create(TENSOR *in,TENSOR *out);
    ZtaStatus Prepare() {}
    ZtaStatus Verify() {}
    ZtaStatus Execute(int queue,bool stepMode);
private:
    TENSOR *m_in;
    TENSOR *m_out;
}

// Initialize this node.
ZtaStatus MyGraphNode ::Create(TENSOR *in,TENSOR *out) {
    // In this example, output tensor has same format as input tensor
    m_in=in;
    m_out=out;
    m_out->Clone(m_in);
    return ZtaStatusOk;
}

// Verify this node.
ZtaStatus MyGraphNode ::Verify() {
    return ZtaStatusOk;
}

// Prepare for new execution run.
ZtaStatus MyGraphNode ::Prepare() {
    return ZtaStatusOk;
}
```

```
// Execute this node
ZtaStatus MyGraphNode ::Execute(int queue,stepMode) {
    // Get a job id and run the tensor program
    my_tensor_program(GetJobId(queue),
        (uint8_t *)m_in->GetBuf(),
        (uint8_t *)m_out->GetBuf(),
        m_in->GetBufLen());
    return ZtaStatusOk;
}
```

## 1.4 Graph

Object of this class implements a flow of execution of multiple steps with each steps are performed by a graph node.

Graph object owns all the graph nodes and coordinates the execution of these nodes.

There can be multiple instances of Graph objects with each instance performing a separate task.

### 1.4.1 Class Interface

#### 1.4.1.1 Graph()

Default constructor of this class

#### 1.4.1.2 ZtaStatus Clear()

Reset the graph by clearing all the nodes.

Output:

- ZtaStatusOk if successful
- ZtaStatusFail otherwise.

#### 1.4.1.3 ZtaStatus Add(GraphNode \*node)

Add a graph node to the end of the graph.

Output:

- ZtaStatusOk if successful
- ZtaStatusFail otherwise.

#### 1.4.1.4 ZtaStatus Verify()

Verify the integrity of the graph.

Graph will then call Verify function of each nodes in the graph.

Output:

- ZtaStatusOk if successful
- ZtaStatusFail otherwise.

#### 1.4.1.5 ZtaStatus Prepare()

This function marks the beginning of a new graph execution. Previous execution results are discarded.

Output:

- ZtaStatusOk if successful
- ZtaStatusFail otherwise.

#### 1.4.1.6      ZtaStatus RunSingleStep()

Execute this graph in step mode.

In this mode, this function may have to be called multiple times to reach execution completion.

This mode is useful when we want to run multiple graphs at the same time and we don't want a slow graph to block the execution of other graphs that are more time critical.

##### Output:

- ZtaStatusOk if graph processing is completed successfully.
- ZtaStatusPending if graph processing is successful but not fully completed. More processing is still required by calling RunSingleStep() function again.
- ZtaStatusFail otherwise.

#### 1.4.1.7      ZtaStatus RunUntilCompletion()

To execute the graph until completion.

##### Output:

- ZtaStatusOk if successful
- ZtaStatusFail otherwise.

#### 1.4.1.8      bool IsRunning()

Return true if graph is currently busy, false if graph is idle and ready to accept a new execution run.



## 1.4.2 Example running a single graph

Example below shows how a single graph is created and executed.

```
// Declare graph,node and tensor objects
Graph graph;
Task1GraphNode node1;
Task2GraphNode node2;
TENSOR tensor_input,tensor_output,tensor_temp;

// Initialize tensor_input dimension and content from a bitmap image
tensor_input.CreateWithBitmap("bitmap.bmp");

// Initialize and attach graph nodes to the graph
// node1 executes first, then node1 passes its output to node2 via tensor_temp,
// then node2 is the final stage of the graph.
// input to the graph is the tensor_input.
// output of the graph is the tensor_output.
node1.Create(&tensor_input,&tensor_temp);
node2.Create(&tensor_temp,&tensor_output);
graph.Add(&node1);
graph.Add(&node2);

// Verify the graph
graph.Verify();

// Prepare for execution
graph.Prepare();

// Execute the graph till completion
graph.RunUntilCompletion();

// Done. Result is now in tensor_output
```

### 1.4.3 Example running multiple graphs

Example below shows how to create and execute multiple graphs simultaneously  
This is similar to previous example except that there are 2 instances of the graph.

```
// Declare graph,node and tensor objects
Graph graph[2];
Task1GraphNode node1;
Task2GraphNode node2;
Task3GraphNode node3;
Task4GraphNode node4;
TENSOR tensor_input[2],tensor_output[2],tensor_temp[2];

// Initialize tensor_input dimension and content from a bitmap image
tensor_input[0].CreateWithBitmap("bitmap1.bmp");
tensor_input[1].CreateWithBitmap("bitmap2.bmp");

// Create first graph
node1.Create(&tensor_input[0],&tensor_temp[0]);
node2.Create(&tensor_temp[0],&tensor_output[0]);
graph[0].Add(&node1);
graph[0].Add(&node2);
graph[0].Verify();
graph[0].Prepare();

// Create second graph
node3.Create(&tensor_input[1],&tensor_temp[1]);
node4.Create(&tensor_temp[1],&tensor_output[1]);
graph[1].Add(&node3);
graph[1].Add(&node4);
graph[1].Verify();
graph[1].Prepare();
```

```
// We can run each graph to completion consecutively.
// But in this example, we interleave the execution of both graphs by running
// them in step mode.
// Since graphs are executed in steps, we have control on how to schedule the
// execution of these graphs or even interleaving graph execution with other
// tasks that are not related to graph.
while(graph[0].IsRunning() || graph[1].IsRunning()) {
    if(graph[0].IsRunning())
        graph[0].RunSingleStep();
    if(graph[1].IsRunning())
        graph[1].RunSingleStep();
}
// Done. Result are now in tensor_output[0] and tensor_output[1]
```

# 2-VISION STACK

---

ztachip has a library of graph nodes that perform many common vision processing tasks.

These vision processing functions are very efficient and fast since they are implemented based on tensor programming as described in [1].

This vision library is used under the graph framework as described in chapter 1.

Vision stack provides the following vision processing acceleration:

- Edge detection using Canny Algorithm
- Color space conversion
- Tensor reshaping
- Image Gaussian blurring
- Feature detection using Harris-Corner Algorithm
- Motion detection with Optical-flow algorithm
- Image resize

## 2.1 *GraphNodeCanny*

GraphNodeCanny is graph node implementing edge detection algorithm based on canny edge detector algorithm.

### 2.1.1 GraphNodeCanny(TENSOR \*input,TENSOR \*output)

Constructor for this graph node.

#### Input

input	Input image to perform edge detection
output	Output tensor will be initialized to have the same width and height as input tensor but with the following tensor attributes <ul style="list-style-type: none"><li>– DataType = TensorDataTypeUInt8</li><li>– DataFormat = TensorFormatSplit</li><li>– ObjType = TensorObjTypeMonochromeSingleChannel</li></ul>

### 2.1.2 Create(TENSOR \*input,TENSOR \*output)

Call to initialize graph node when default constructor was used.

Parameters are similar to 2.1.1

### 2.1.3 void SetThreshold(int \_loThreshold,int \_hiThreshold)

Setting edge detection threshold

#### Input

_loThreshold	_loThreshold must be <= 255. If pixel gradient is below _loThreshold then pixel is rejected as edge. Default low threshold is 81.
_hiThreshold	_hiThreshold must be <= 255 If pixel gradient is above _hiThreshold then pixel is accepted as edge. Default high threshold is 163.

#### Output:

None

2.1.4 **void** GetThreshold(**int** \*\_loThreshold,**int** \*\_hiThreshold)

Return current edge detection threshold

## 2.2 *GraphNodeColorAndReshape*

This graph node performs color space conversion and tensor reshaping.

### 2.2.1 GraphNodeColorAndReshape(

```
TENSOR *input,  
TENSOR *output,  
TensorObjType _dstColorSpace,  
TensorFormat _dstFormat,  
int clip_x=0,  
int clip_y=0,  
int clip_w=0,  
int clip_h=0,  
int dst_x=0,  
int dst_y=0,  
int dst_w=0,  
int dst_h=0)
```

Constructor for this graph node.

Transform and copy input tensor to output tensor.

Transform from a source tensor with a DataType/DataFormat to a destination tensor with a different DataType/DataFormat.

#### Input

input	Input tensor
output	Output tensor
_dstColorSpace	Object type of destination tensor
_dstFormat	Data format layout of destination tensor
clip_x clip_y clip_w clip_h	Identifies the region within input tensor to be used as source tensor. clip_x and clip_y are the origin of the region. clip_w and clip_h are the dimension of the region.
dst_x dst_y dst_w dst_h	Identifies the region within output tensor to write result to. dst_x and dst_y are the origin of the region. dst_w and dst_h are the dimension of the region.

### 2.2.2 Create()

```
TENSOR *input,  
TENSOR *output,  
TensorObjType _dstColorSpace,  
TensorFormat _dstFormat,  
int clip_x=0,  
int clip_y=0,  
int clip_w=0,  
int clip_h=0,  
int dst_x=0,  
int dst_y=0,  
int dst_w=0,  
int dst_h=0)
```

Call to initialize graph node when default constructor was used.

Parameters are similar to 2.2.1



## 2.3 *GraphNodeGaussian*

This graph node performs image blurring using a Gaussian filter.

### 2.3.1 `GraphNodeGaussian(TENSOR *input, TENSOR *output)`

Constructor for this graph node.

Input:

input	Input tensor to apply the gaussian filter.
output	Output tensor.

### 2.3.2 `ZtaStatus Create(TENSOR *input, TENSOR *output)`

Call to initialize graph node when default constructor was used.

Parameters are similar to 2.3.1

### 2.3.3 `void SetSigma(float _sigma)`

Set sigma value of the gaussian filter.

### 2.3.4 `float GetSigma()`

Return current sigma value of the the gaussian filter.

## 2.4 *GraphNodeHarris*

This graph node performs Harris-Corner feature detection on an image.

### 2.4.1 `GraphNodeHarris(TENSOR *input, TENSOR *output)`

Constructor for this graph node.

Input:

input	Input tensor.
output	Output tensor with width=input's width, height=input's height, dataType=int16. Data elements are feature detection scores. zero for no detection.

### 2.4.2 `ZtaStatus Create(TENSOR *input, TENSOR *output)`

Call to initialize graph node when default constructor was used.

Parameters are similar to 2.4.1

## 2.5 *GraphNodeOpticalFlow*

This graph node performs optical-flow for motion detection on two images captured consecutively in time.

```
2.5.1 GraphNodeOpticalFlow(TENSOR *input1,  
                             TENSOR *x_gradient,  
                             TENSOR *y_gradient,  
                             TENSOR *t_gradient,  
                             TENSOR *x_vect,  
                             TENSOR *y_vect,  
                             TENSOR *display)
```

Constructor for this graph node.

### Input

input1	input1 is expected to be an alias tensor to an image buffer. At every new execution, there must a new buffer submitted with the previous buffer still valid and unchanged. This graph node compares current image buffer with the last image buffer for motion detection.
x_gradient	buffer with dimension hwx of type int16 Holds the gradient change in x direction.
y_gradient	buffer with dimension hwx of type int16 Holds the gradient change in y direction.
t_gradient	buffer with dimension hwx of type int16 Holds the gradient change in time direction.
x_vect	x component of motion vector
y_vect	y component of motion vector
display	Buffer with dimension 3xhwx intended for display purposes. If set to 0 then display will not be generated, Pixel colour represents motion vector direction. <ul style="list-style-type: none"><li>- red means movement to the right</li><li>- green means movement to the left</li><li>- blue means vertical movement.</li></ul> Pixel intensity represents motion vector magnitude.

**2.5.2** ZtaStatus Create(TENSOR \*input1,  
TENSOR \*x\_gradient,  
TENSOR \*y\_gradient,  
TENSOR \*t\_gradient,  
TENSOR \*x\_vect,  
TENSOR \*y\_vect,  
TENSOR \*display)

Call to initialize graph node when default constructor was used.

Parameters are similar to 2.5.1

## 2.6 *GraphNodeResize*

This graph node performs image resize

2.6.1 `GraphNodeResize(TENSOR *input, TENSOR *output, int w, int h)`

Resize image

input	Input tensor to be resized
output	Output tensor
w	Width of image after resize
h	Height of image after resize

2.6.2 `ZtaStatus Create(TENSOR *input, TENSOR *output, int w, int h)`

Call to initialize graph node when default constructor was used.

Parameters are similar to 2.6.1

# 3-AI STACK

---

ztachip provides acceleration functions for the execution of Google's TensorFlow Lite model. AI stack is implemented as graph node.

The following Neural Network Layers are supported

- Convolution
- ConvolutionDepthWise
- FCN
- Add
- Concatenation
- Logistics
- ObjectDetection
- PoolAverage
- Reshape
- Relu

## 2.7 *TfliteNn*

This is a graph node that would execute a TensorFlow Lite model.

It executes an AI model using the original TensorFlow Lite trained model binary that we can be downloaded from Google website. No model retraining is required.

**2.7.1 ZtaStatus Create(const char \*fname, TENSOR \*\_input, int numOutput, ...)**

Load a TensorFlow Lite model and prepare for inferencing.

Input

fname	TensorFlow Lite model file name. It has suffix *.tflite
_input	Input tensor to the model.
numOutput	Number of output tensors expected. After this parameter, we expect numOutput numbers of tensors to follow

Output:

- ZtaStatusOk if successful

- ZtaStatusFail otherwise.

**2.7.2 ZtaStatus Load(const char \*fname, TENSOR \*\_input,  
int numOutput, ...)**

Same as 2.7.1

**2.7.3 ZtaStatus Unload()**

Unload and close the current TensorFlow Lite model.