SIEMENS

TM NPU MicroPython

Programming Manual

06/2025
A5E52728839-AB

Legal information

Warning notice system

This manual contains notices you have to observe in order to ensure your personal safety, as well as to prevent damage to property. The notices referring to your personal safety are highlighted in the manual by a safety alert symbol, notices referring only to property damage have no safety alert symbol. These notices shown below are graded according to the degree of danger.

⚠ DANGER

indicates that death or severe personal injury **will** result if proper precautions are not taken.

⚠ WARNING

indicates that death or severe personal injury **may** result if proper precautions are not taken.

⚠ CAUTION

indicates that minor personal injury can result if proper precautions are not taken.

NOTICE

indicates that property damage can result if proper precautions are not taken.

If more than one degree of danger is present, the warning notice representing the highest degree of danger will be used. A notice warning of injury to persons with a safety alert symbol may also include a warning relating to property damage.

Qualified Personnel

The product/system described in this documentation may be operated only by **personnel qualified** for the specific task in accordance with the relevant documentation, in particular its warning notices and safety instructions. Qualified personnel are those who, based on their training and experience, are capable of identifying risks and avoiding potential hazards when working with these products/systems.

Proper use of Siemens products

Note the following:

⚠ WARNING

Siemens products may only be used for the applications described in the catalog and in the relevant technical documentation. If products and components from other manufacturers are used, these must be recommended or approved by Siemens. Proper transport, storage, installation, assembly, commissioning, operation and maintenance are required to ensure that the products operate safely and without any problems. The permissible ambient conditions must be complied with. The information in the relevant documentation must be observed.

Trademarks

All names identified by ® are registered trademarks of Siemens Aktiengesellschaft. The remaining trademarks in this publication may be trademarks whose use by third parties for their own purposes could violate the rights of the owner.

Disclaimer of Liability

We have reviewed the contents of this publication to ensure consistency with the hardware and software described. Since variance cannot be precluded entirely, we cannot guarantee full consistency. However, the information in this publication is reviewed regularly and any necessary corrections are included in subsequent editions.

Siemens Aktiengesellschaft
Digital Industries
Postfach 48 48
90026 NÜRNBERG
GERMANY

A5E52728839-AB
Ⓟ 07/2025 Subject to change

Table of contents

Introduction

1

Disclaimer

Changes to the API may occur with future versions.

Therefore, please consult the release notes to get a first impression of changes between different versions.

Official MicroPython documentation references

This documentation details the additional modules added to the TM NPU port of MicroPython. For further details regarding MicroPython consult the official MicroPython documentation version 1.12.

Furthermore, if not specified in TM NPU MicroPython documentation or the official MicroPython documentation, refer to the official CPython documentation version 3.9.6.

Structure of this documentation

Refer to the table of contents for an overview of the chapters. The structure of the documentation for each customized module of the TM NPU MicroPython API is as follows. Chapters that do not apply are left out.

- General description: Description of the module
- Module functions: Functions of the module itself
- Class: Available class that comes with the module
- Example: Overall example of the module

Release Notes: Changes in API

2

The `release_notes` document relevant changes to the TM NPU MicroPython API.

2.1 Releases

V2.0.2

- module camera:
  camera.camera() the supported resolutions for GigE vision and external image have been extended up to 2MP and can be set freely within the defined range
- module *vid_pipeline*:
  start_streaming() the timeout parameter has been added
  read_raw() the timeout parameter has been added
  read_processed() the timeout parameter has been added
- API-Update with V2.0.2

V2.0.1

- Initial release

See also

Class: NeuralNet (Page 24)
General description (Page 17)
Class: Camera (Page 14)
Troubleshooting list (Page 33)

TM NPU-specific MicroPython modules 3

3.1 Introduction to the modules

The inbuilt TM NPU MicroPython interpreter provides specifically created modules which allow access to the module interfaces and functionalities such as communication with the PLC, access to the camera, initialization and running of the inference.

The TM NPU MicroPython modules are listed in the order they would be used in for realizing a typical classification or object detection application. When creating the MicroPython script, this order can be rearranged and altered as required by the actual application.

- For communication between the CPU and TM NPU via the process image, for example in order to trigger functions within the MicroPython application. The PLC_COMMUNICATION module can be used.
- The TM NPU's SD card as well as the FTP server can be accessed using the *npufs* (Page 11) module, which allows access to the module's file system. This includes read and write access to the module's SD Card, as well as a potentially connected FTP Server with the module's inbuilt FTP Client. This can be used for saving images to the SD Card or the FTP Server, or downloading updated Neural networks for example.
- The *camera* (Page 14) module is available for grabbing a (raw) image from the connected camera for further processing .
- The (raw) image from the camera can be modified and prepared as input for the neural network using the *vid_pipeline* (Page 17) module, which supports with scaling, format conversion, etc.
- The received output frame from the vid_pipeline module can be used as input for an initialized neural network. With the help of the *neural_network* (Page 23) module a neural network (blob file) e.g. from the SD Card or an FTP Server can be initialized, provided with a frame and executed. The neural network makes use of the hardware accelerator automatically and returns results of the neural network. These can either be used by the MicroPython application or be sent back to the PLC again via the process image.

Beside these TM NPU specific modules, a range of standard MicroPython functions are also available and can be applied. Please refer to Chapter 5 "Supported modules on TM NPU (Page 8)" for a list of supported Micropython functions as well as possible restrictions.Hotspot-Text (Page 30).

3.2 PLC_WITHOUT_PROTOCOL: Communicating with a connected PLC

3.2.1 General description

This module gives access to the process image of the PLC.

The first two bytes of the entire process image of inputs and outputs are reserved as status, error and control bytes. These can not be accessed through the MicroPython API. The remaining bytes are user-defined message bytes and can be freely adapted by the application. The user can access them by using the functions `plc.read()` and `plc.write()`.

The figure below shows the exchange of a message between the PLC and TM NPU using the process image.

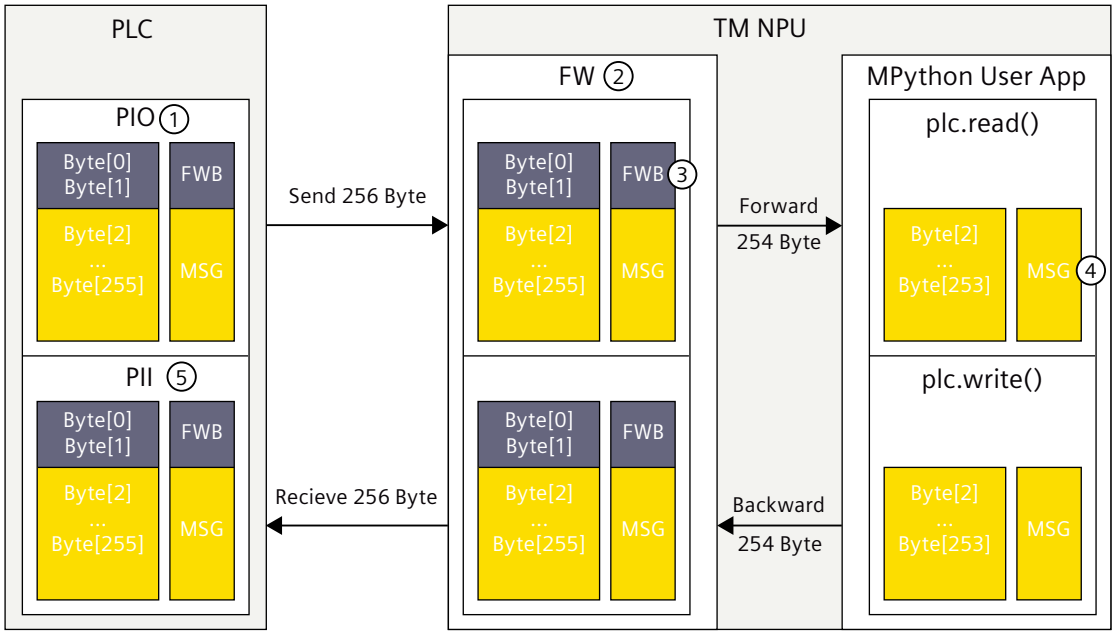

Figure 3-1 ① PIO: Process image output, ② FW: Firmware, ③ FWB: Firmware reserved bytes, ④ MSG: User-defined message bytes, ⑤ PII: Process image input

NOTE

General information on the process image and the content of the status, error and control bytes can be found in the TM NPU User Manual.

NOTE

There is no prepared protocol implemented for message handling between the TM NPU and PLC, e.g. indicating whether the module is in a ready, busy or done state for a task, or whether the information was updated. If required, this would have to be realized by the application. Examples of how this can be implemented are available in the application example.

3.2.2 Module functions

By creating an instance of the plc class the user initializes the underlying PLC communication subsystem. This should be done during the "initialization phase" of the script (please see TM NPU User Manual for more details and proposals on how to structure the script). The returned instance can then be used to call read() and write() functions.

An instance of the class is created by calling:

```
plc_without_protocol.plc()
```

This returns a plc object.

NOTE

Only one instance of the plc class can be created. In order to create a new instance of class plc, call the defined procedure to clean up resources. Please consult the information below on the use of plc.destroy() and the note on using 'del' for more details.

See also

Class: Plc (Page 9)

3.2.3 Class: Plc

`plc.read()`

Reads and returns a raw message as a bytes object from the PLC, not including the firmware reserved Control Bytes. This function performs a destructive read of the data coming from the PLC and returns 'None' if no new data is available. Read() always returns the whole process image message (254 bytes).

`plc.write(`***msg***`)`

Sends a raw byte object message (msg) to the PLC. The firmware reserved Status and Error Byte cannot be written by the application. Write() always writes to the whole process image message (254 Bytes) starting at Byte [0] (== Byte[2] in the process image).

NOTE

The write process works acyclically regarding the PROFINET cycle. The written message is collected from the backplane bus in cyclic events and is not directly triggered by the write() function. Please be aware that calling write() multiple times in close succession can result in the prior information being overwritten before it was delivered to the PLC.

`plc.destroy()`

Destroys an existing plc instance. Before creating a new plc object, call this function first. It frees underlying resources. Afterwards call the standard delete function (using the keyword 'del') to fully release the plc object. This procedure ensures only one valid plc instance at a time.

NOTE

Only calling 'del' but not 'destroy' on the object causes the reference counter of the plc object to be reduced by one, while not freeing the resources for the object. Using the garbage collection in this case causes the plc module to remain in a blocked state where no further plc instances can be created.

Procedure for safely freeing resources:

```
plc = plc_without_protocol.plc()
plc.destroy()
del plc
```

3.2.4 Examples

Receiving a message:

```
import plc_without_protocol

#continuously read without protocol
plc = plc_without_protocol.plc()
input_bytes = None
while(input_bytes is None):           #blocking until value is received
     input_bytes = plc.read()

input_buffer = bytearray(input_bytes)

#...
```

Or sending a message with the same format as above:

```
#write without protocol
output_buffer=bytearray(254)    #fill with data to be sent
plc.write(bytes(output_buffer))

#...
```

3.3 NPUFS: Accessing the NPU filesystem

3.3.1 General description

This module can be used to read from and write to the TM NPU's filesystem. This includes the module's SD-Card as well as an external FTP Server, using the module's FTP Client.

3.3.2 Module functions

`npufs.open(`***filename, mode='r'***`)`

NOTE

Current limitation: Only one file can be opened at a time.

Open a file named `filename` on the NPU file system, and return a NpufsFile object to access it. The NpufsFile object may be used as a context manager.

`filename` is a string with the name and path to the file on the file system. The file name must be compliant with standard filenames and paths with respect to the FAT32 format. This means that certain special characters, such as ':', are prohibited.

`mode` is a string describing how to open the file: read, write, text, binary, append, etc. See the mode parameter of Python's `open()` for details (https://docs.python.org/3/library/functions.html#open).

The supported 'mode' values are listed in the table below:

Mode	Function read()	Function write()	Notes
r, rb, rt	+ [1]	- [2]	
r+, rb+, rt+	+	-	
w, wb, wt	-	+	
w+, wb+, wt+	-	+	Function read() not supported on TM NPU
a, ab, at	-	-	Not supported - will result in overwriting the file
a+, ab+, at+	-	-	Not supported - will result in overwriting the file
x	-	+	

[1] supported

[2] not supported

Accessing the SD Card or FTP Server using npufs: In order to either access the SD Card or a FTP Server use the appropriate path as `filename`.

NOTE

The TM NPU has a startup period for enabling all its services at the very beginning of the execution of the application. Therefore, ensure that the FTP Server can be accessed by delaying the 'npufs.open()' call. This can be realized by calling sleep prior to the 'npufs.open()' call. Another option is to retry the 'npufs.open()' procedure until the startup of the TM NPU is complete.

NOTE

Only absolute paths are allowed to open files on SD Card or FTP Server.

NOTE

The standard Python open() method is not available on the TM NPU. To access files on the TM NPU the user must instead use the TM NPU-specific module 'npufs.open()'

Accessing the SD Card

To access files on the SD Card, use the npufs module.

```
PATH = SDCARDMOUNT = "/media/mmcsd-0-0/"
with npufs.open(PATH + 'testFile.txt', 'r') as fp:
```

Accessing the FTP Server

The FTP client functionality can be used by applying the *npufs.open()* function and providing the path information of the FTP Server as shown in the example below.

NOTE

The configuration and setup of the FTP client in the TM NPU is done via the 'ftpclient.conf' config file which has to be present on the SD Card of the TM NPU. Details are described in the TM NPU User Manual.

The following parameters affect the 'path' parameter of npufs.open():

ftpclient.conf:

```
{
...,
"server":"192.168.1.1",
"localPath":"/SERVERROOT",
...,
}
```

Use the mapped path to access FTP server:

```
PATH = FTPMOUNT = "/SERVERROOT/testFolder/"
with npufs.open(PATH + 'testFile.txt', 'r') as fp:
```

3.3.3 Class: NpufsFile

An instance of this class is returned by `npufs.open()` and provides access to the open file.

`NpufsFile.read(`***num_bytes=- 1***`)`

Read `num_bytes` bytes from the file. If no argument is provided, the file is read until the end. A byte object or string (depending on the mode the file was opened with) is returned. For available modes see *npufs.open()* (Page 11).

`NpufsFile.write(`***data***`)`

The given data is written to the file. The data can be a byte object, string, or general buffer object, such as an *ExternalBuffer* (Page 22).

NOTE

If the SD Card is write protected, the procedure to write to SD Card results in a runtime error.

`NpufsFile.close()`

Closes the file. The NpufsFile should not be used after this call. It is called automatically when leaving a managed context.

3.3.4 Example

```
import npufs

PATH = "/media/mmcsd-0-0/"

#write 'TestDataTestData' as binary data in the testdata.bin file
with npufs.open(PATH + 'testdata.bin', 'wb') as f:
     f.write(b'TestDataTestData')

#read data as binary data from the testdata.bin file
#read entire file to the end
with npufs.open(PATH + 'testdata.bin', 'rb') as f:
     print(f.read())
```

3.4 CAMERA: Accessing connected cameras

3.4.1 General description

The camera module provides access to a connected camera. It creates an object which allows for definition of the camera's properties in preparation for the initialization of the video pipeline (vid_pipeline, see Chapter 3.5).
A camera object can be created as one of three versions: "GigE Vision", "REALSENSE_D435", or"External Image", defined via the "camera_id". The camera object must be created to fit to the connected camera.
If an image is to be processed by ta video pipeline not coming from a physically connected camera connected to the TM NPU, the camera object can be initialized as "EXTERNAL IMAGE". This allows for images loaded from the file system (SD-Card or FTP Server) to be processed via the video pipeline (and the inference).

NOTE

A camera object must be created before initializing the video pipeline using vid_pipeline. Any changes to the initialized camera object after initialization of the video pipeline have no effect on it.

See also

General description (Page 17)

3.4.2 Class: Camera

An instance of the ‘camera’ class is created by calling:

`camera.camera(`*camera_id='GIGE_VISION', resolution=(1280, 720), pixel_format='YUV', pixel_format_description='YUYV', layout='PLANAR'*`)`

This returns a camera object, to be used with the *vid_pipeline* (Page 17) module.

Multiple instances of the ‘camera’ class can be created. A camera object can be erased using the standard MicroPython ‘del’ function.

- `camera_id`

  The unique identifier of the camera to be used. Can be ‘GIGE_VISION’, ‘REALSENSE_D435’ or ‘EXTERNAL_IMAGE’.

  NOTE

  The value ‘EXTERNAL_IMAGE’ has to be used in combination with *vid_pipeline.set_Image()* (Page 18) for manual input of images into the vid_pipeline. This can be used for testing the inference on TM NPU, for example by loading an image from the SD Card and checking the expected results of the inference.

NOTE

Apart from the configuration in the MicroPython script (main.py), the corresponding ports of the TM NPU must be activated to enable access to the connected camera. This is done in the engineering by enabling or disabling the USB and Ethernet interface. For detailed information on the module configuration, please refer to the TM NPU User Manual.

To use a USB camera, select:

– Engineering: Enable USB interface and camera
– main.py: camera_id = REALSENSE_D435' and pixel_format_description = 'YUYV'

To use a GIGE camera, select:

– Engineering: Enable Ethernet interface and the underlying 'GigE Vision' service
– main.py: `camera_id` = 'GIGE_VISION' and `pixel_format_description` = 'UYVY'

If the engineering and MicroPython user application do not match in the camera selection, frames received from the *vid_pipeline.read_processed()* (Page 18) will misshappen and with color coding (green/violet).

- `resolution`

  The desired output resolution for the camera, and must be a tuple of (width, height). The supported resolution for using the Intel RealSENSE camera is fixed. For using a GigE Vision Camera or an external image the resolution can be set freely within the range of 640 x 480 pixels to 2MP. As the max. supported resolution for 2MP cameras can be either 1920 x 1080 pixels or 1600 x 1200 pixel, the limiting factor of the used resolution is the multiplication of width and height (X x Y) which must be equal or lower than 2.073.600 pixels (=1920 x 1080 pixels). The output resolution can therefor be set as follows:

  – GIGE_VISION: (X, Y), with 640 <= x <= 1920, 480 <= Y <= 1200 and X*Y <= 1920*1080
  – REALSENSE_D435: (1280, 720)
  – EXTERNAL_IMAGE: (X, Y), with 640 <= x <= 1920, 480 <= Y <= 1200 and X*Y <= 1920*1080

  NOTE

  If the resolution for the camera object of this module differs from the configuration of the physical camera the initialization of vid_pipeline will fail. This error can not be caught by an exception. The module is forced to make an app error that can only be solved by applying the power cycle.

- `pixel_format`

  The pixel format of the images delivered from the camera. Can be 'YUV'.

  NOTE

  'RGB' and 'MONO' are not supported.

- `pixel_format_description`
  Further specifies the pixel format of the images delivered from the camera. Can be 'YUYV' or 'UYVY'.
- `layout`
  Defines how the image data should be arranged in the memory. Can be 'PLANAR'.

NOTE

'ROWMAJOR' and 'INTERLEAVED' are currently not implemented.

3.4.3 Example

```
pfd_gige = "UYVY"
cam_info_gige = camera.camera(camera_id="GIGE_VISION",
resolution=(1280, 720),pixel_format_description=pfd_gige)
```

3.5 VID_PIPELINE: Configuring the video pipeline

3.5.1 General description

This module enables the use of an image preprocessing pipeline which utilizes the TM NPU integrated hardware accelerators. Images from this module can later be used e.g. as input for the neural network.

The figure below shows the entire flowchart of using the `vid_pipeline` and its functions, as well as interfaces of potentially interacting modules.

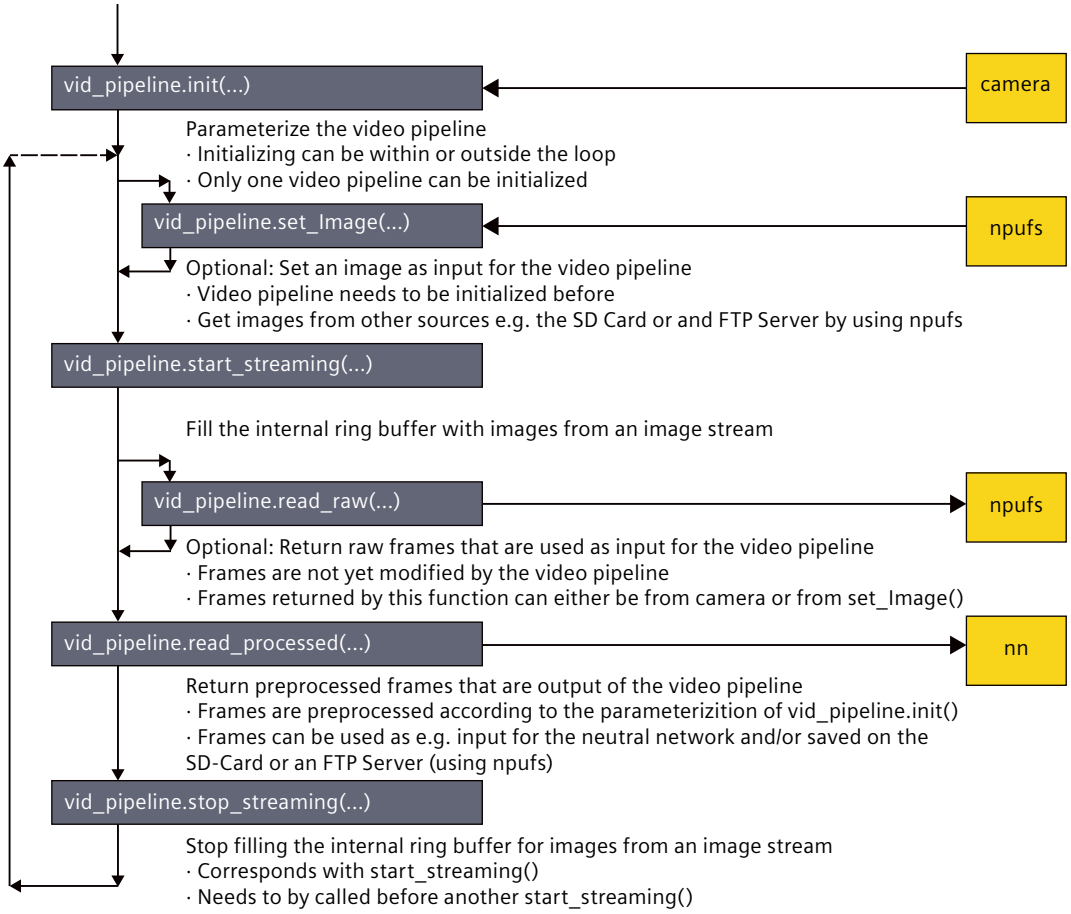

Figure 3-2 Example functionality for using the video pipeline for (pre)processing images

3.5.2 Module functions

`vid_pipeline.init(`***camera, target_resolution, target_format='RGB', target_normalization=(0.0, 1.0), target_output='FLOAT', target_layout='PLANAR', shaves_scaling=4, shaves_conversion=4***`)`

Initializes a video pipeline for preprocessing a raw image before integrating it into the neural network.

NOTE

The video pipeline can only be initialized once. Reinitializing the video pipeline or initializing more than one instance of vid_pipeline is not supported.

- `camera`

  Has to be an object produced via *camera.camera()* (Page 14).

- `target_resolution`

  A tuple (width, height) describing the size required for the pipeline's output frames. Currently, the target_resolution result of the multiplication of width and height is limited to a total of 307,200 pixels.

- `target_format`

  Can be 'RGB'.

  NOTE

  'MONO' is not supported.

- `target_normalization`

  A tuple (*scale_mean, scale_norm*) with a default value of (0.0, 1.0). For normalization *scale_mean* is the value to be subtracted from each pixel, and *scale_norm* is the value to scale each pixel after mean subtraction. These are only used when the target_output is 'FLOAT'.

- `target_output`

  Can be 'FLOAT'. It determines the format of the individual pixel data delivered.

  NOTE

  'INTEGER' is not supported.

- `target_layout`

  How the output frame data should be arranged in the memory. Possible values: 'PLANAR'.

  NOTE

  'ROWMAJOR' and 'INTERLEAVED' are not supported.

- `shaves_scaling`

  The number of SHAVEs dedicated for scaling a frame.

  NOTE

  Currently the only valid value for shaves_scaling is '4'. For details regarding SHAVEs, consult Chapter 4 Use of *shaves and hardware accelerators* (Page 27).

- `shaves_conversion`

  The number of SHAVEs dedicated for format conversion.

  NOTE

  Currently the only valid value for shaves_conversion is '4'. For details regarding SHAVEs, consult Chapter 4 Use ofS *shaves and hardware accelerators* (Page 27).

`vid_pipeline.start_streaming(numFrames=1, timeout=10)`

Requests that the video pipeline starts and collects a frame from the connected camera (formerly defined via the camera object) and transfers it into the modules image buffer, from where it can be grabbed for preprocessing. The parameter timeout=10 checks whether an image is delivered from the camera within the set time frame. An exception is thrown if no image is available within the expected time. The default value is 10 seconds. In case the "EXTERNAL_IMAGE" image was used and the image to be processed was not sourced from a connected camera but from the filesystem (e.g. the SD Card or an FTP Server), set_Image0 has to be called beforehand in order to provide the image for the video pipeline. In this case, start_streaming will then forward this image to the image buffer.

NOTE

The image buffer can only hold one image at a time. Therefore, the only supported value for numFrames is 1.

NOTE

Timeout can be set in the following range : {1,2, . . . , 600} seconds.

`vid_pipeline.set_Image(`***frame***`)`

Manually sets an image as input for the video pipeline, regardless of the stream resulting from the camera. `frame` must be a bytes object, not a string object, such as resulting from *npufs.read()* (Page 13) when used with binary mode ("rb"). Furthermore, it can not be a frame object, but can be an *ExternalBuffer* (Page 22) resulting from *frame.data()* (Page 21).

NOTE

set_Image(frame) can only be used in combination with the 'EXTERNAL_IMAGE' parameter used in *camera.camera()* (Page 14) and matching other parameters.

NOTE

Please be aware of the sequential call structure of this function and its dependencies towards prior function calls and initializations. The following functions have to be called in the following order: vid_pipeline.init(), vid_pipeline.set_Image(), vid_pipeline.start_streaming().

`vid_pipeline.stop_streaming()`

Requests that a previously started video pipeline stops.

NOTE

This function must be called after the vid_pipeline has processed the one frame that was announced in start_streaming(). This is because stop_streaming() triggers the frame buffer to be cleared in order to be able to receive a new image.

`vid_pipeline.read_raw(timeout=10)`

Returns a frame object, before being processed by the video pipeline, e.g. received by *camera.camera()* (Page 14) or set by *set_Image(frame)*. The frame is still processed by the vid_pipeline and can later be read by *vid_pipeline.read_processed()*. The function can only be called 'numFrame' times, as specified in start_streaming(). The parameter timeout=10 checks whether an image is delivered from the camera within the set time frame. An exception is thrown if no image is available within the expected time. The default value is 10 seconds.

NOTE

Timeout can be set in the following range : {1,2, . . . , 600} seconds.

`vid_pipeline.read_processed(timeout=10)`

Returns a frame object, produced as the output from a currently streaming video pipeline after being processed. The frame object and its members are defined by the parameters used in vid_pipeline.init(). The function can only be called 'numFrame' times, as specified in start_streaming().

The parameter timeout=10 checks whether an image is delivered from the camera within the set time frame. An exception is thrown if no image is available within the expected time. The default value is 10 seconds.

The returned frame is of the class type *frame*. When applying *frame.data()* on this frame, the returned data is of the type *ExternalBuffer*. It can be used as the input for a neural network. (see: *neural_network.run()*)

NOTE

Timeout can be set in the following range : {1,2, . . . , 600} seconds.

See also

General description (Page 17)

Class: NeuralNet (Page 24)

3.5.3 Class: Frame

After preprocessing an image, the video pipeline returns an image of the class "frame".

NOTE

Current restriction: Frame must be used as a context manager.

For example:

```
with vid_pipeline.read_processed() as frame:
     print("New frame! Dimensions:", frame.width(), "x", frame.height())
```

`Frame.width()` returns the width of the frame.

`Frame.height()` returns the height of the frame.

`Frame.data()` returns an *ExternalBuffer* (Page 22) object, allowing access to the raw pixel data of the frame.

`Frame.release()` releases the external resources held by the frame. This is automatically called when leaving the context manager.

NOTE

Currently this function must not be called manually by the user.

3.5.4 Class: ExternalBuffer

A class referencing the native frame data. The data is held externally to the MicroPython heap.

It has no methods, but implements the buffer protocol and can be passed to functions expecting a buffer (it implements the buffer protocol), such as *neural_network.run()* (Page 24) or *npufs.write()* (Page 13).

It can be copied into MicroPython's heap for further processing e.g. by using it as the constructor parameter for a bytes() or bytearray() object.

NOTE

The ExternalBuffer can only be used while its originating frame object is valid. When the frame is released, it will act as an empty buffer.

3.5.5 Example

```
#...

#Initialize camera
pfd_gige = "UYVY"
cam_info_external_gige = camera.camera(camera_id="EXTERNAL_IMAGE", resolution=(1280, 720),
pixel_format_description=pfd_gige)

#Define filepath
pathSdcardRoot = '/media/mmcsd-0-0/'
filePath = pathSdcardRoot+'USERDATA/'

#Load image from sdcard
image = None
with sdcard.open(filePath + 'rawImage.yuv', 'rb') as fp:
     image = fp.read()

#Preprocess image in video pipeline
vid_pipeline.init(camera=cam_info_external_gige,
     target_resolution=(224, 224),
     target_format="RGB",
     target_normalization=(0.0, 1.0),
     target_output="float",
     shaves_scaling=4,
     shaves_conversion=4)

#set external image
vid_pipeline.set_Image(image)

vid_pipeline.start_streaming(1)

with vid_pipeline.read_processed() as frame:
          # Save rgb image to sdcard after pipeline
     imgNameProcessed = "processedImage"
```

```
    with sdcard.open(filePath + imgNameProcessed, 'w') as fp:
        fp.write(frame.data())

        # #running neural network
    results = net.run(frame.data())

vid_pipeline.stop_streaming()

#...
```

3.6 NEURAL_NETWORK: Initializing and executing neural networks

3.6.1 General description

This module provides functions and classes that allow working with a neural network.

3.6.2 Module functions

`neural_network.init(`***model_buffer, shaves_nn=4, use_cnns=(True, True)***`)`

Initializes and returns a *NeuralNet* (Page 24) object.

NOTE

More details on troubleshooting and exceptions can be found in Chapter 6: *General notes and troubleshooting* (Page 33).

- `model_buffer`

  The supplied raw network blob data.

  NOTE

  If a corrupted neural network is provided, an exception is thrown. However, if a formally accepted neural network that is not suitable for the purpose of the app is used (e.g. if the required buffer memory size is too large or the network includes not supported layers), no exception is thrown and the app stops working. In this case, an error is indicated by the debug server and the servicedata.

- `shaves_nn`

  The number of SHAVEs assigned to the execution of the neural network.

  NOTE

  Currently only the value of '4' is accepted, otherwise an exception is thrown. The value does not need to be assigned since it is set per default. For details regarding SHAVEs, consult Chapter 4 Use of *shaves and hardware accelerators* (Page 27).

- `use_cnns`

  Should be a tuple of boolean values, to indicate whether or not to use the available CNN accelerators (use_cnn_1, use_cnn_2).

  NOTE

  Currently only the value of (True, True) is accepted, otherwise an exception is thrown. The value does not need to be assigned since it is set per default. Both CNN accelerators are to be activated for the network. For details regarding SHAVEs, consult Chapter 4 Use of shaves and hardware accelerators (Page 27).

3.6.3 Class: NeuralNet

The NeuralNet class provides an interface to run a neural network with supplied input data.

`NeuralNet.run(`***input_data, results_dict=False***`)`

`NeuralNet.run(`*input_data, results_dict=False*`)` runs the neural network with the given input data, returning the results.

- `input_data` can be one of the following:
  - An *ExternalBuffer* (Page 22) object, where the data is passed in directly to the input tensor without performing a type conversion
  - A dictionary of input tensors
  - A list of values
  - An array of values

  If multiple input data arguments are passed, these will be assigned to the available input tensors in order.

  If a dictionary of input tensors is passed in, it must be in the form:

  - Keys are the names of the input tensors
  - Values as an array, list, or *ExternalBuffer* (Page 22) as above

  NOTE

  `input_data` must not exceed a frame size defined by the “target_resolution” parameter when initializing the video pipeline (*vid_pipeline.init()* (Page 18)).

```
results = net.run({'input': frame.data()}, results_dict=False)
```

- `results_dict` defines whether the inference results are returned as a dictionary or not, and can be either True or False:
  - `results_dict` = False:

    Results are returned as an array.

    NOTE

    Only single output tensors are supported.

  - `results_dict` = True:

    If `results_dict` is True, a dictionary is returned instead, with keys being the names of the output tensors, and values being the arrays of the corresponding tensor data.

    NOTE

    Be aware that MicroPython evaluates every value to 'True', that is not 'False', '0' or 'None'.

`NeuralNet.release()`

`NeuralNet.release()` frees underlying hardware resources, such as SHAVEs, of a previously initalized *NeuralNet* object. Only the targeted object is dealt with. The object cannot be used again afterwards. Attempting to use the object after release will result in an "Invalid NN handler" exception being thrown. The object is automatically removed from the memory by the garbage collector. Use this method to release a previously initialized neural network in order to free resources before initializing a new or updated neural network.

NOTE

Attempting to release the allocated hardware resources (SHAVEs and hardware accelerators) of an initialized NeuralNet object by using only the standard MicroPython "del" function will not result in the resources being freed. For fully freeing the resources always use "NeuralNet.release()" and "del" in succession (see example below).

3.6.4 Example

```
import neural_network
import npufs
with npufs.open('folderWithNet/example.blob', 'rb') as model_file:
     net = neural_network.init(model_file.read(), shaves_nn=4, use_cnns=(True, True))

#...

#input data from frame
print(type(frame.data()))    #output: <class 'ExternalBuffer'>
results = net.run(frame.data())
```

```
#return as array
results = net.run(input_data)
print(type(results))               #output: <class 'array'>

#return as dictionary
results = net.run(input_data, results_dict=True)
print(type(results))               #output: <class 'dict'>

#access output as dictionary
keys = [for k in results.keys()]
class_index, _ = max(list(enumerate(results[keys[0]])), key=lambda x: x[1])
print(class_index)                    #number of outputs from network

#...

#release network
net.release()
del net
```

Use of shaves and hardware accelerators

4

This chapter describes relevant aspects of the use of SHAVEs and hardware accelerators of the TM NPU with regard to MicroPython API usage.

NOTE

A general description regarding the architecture and details of the Myriad X subsystems (such as multi media subsystem, CPU subsystem, and shave subsystem) can be found in the TM NPU User Manual (https://support.industry.siemens.com/cs/us/en/view/109765877).

4.1 General information

The TM NPU offers the usage of sixteen SHAVE (Streaming Hybrid Architecture Vector Engine) vector processors. They can be used by image or computer vision applications, as well as any other general computation-intensive algorithms, to achieve highly optimized scheduling of imaging and computer vision processing pipelines. Furthermore, dedicated hardware accelerators, such as CNN hardware accelerators, ensure rapid execution of neural networks even on multiple threads.

A typical example of using SHAVE processors is to assign them to a SIPP engine (Streaming Image Processing Pipeline). SIPP is a proprietary software/hardware mechanism used by the Intel Movidius Myriad X processor to achieve these kinds of pipelines. The SIPP environment is the development framework for Media sub-system which is a collection of SIPP accelerators. It consists of a complementary collection of hardware image processing filters. These are designed primarily for use within the SIPP software framework. They allow generic or computationally intensive functionality to be offloaded from the SHAVEs. Various hardware image processing filters are available such as Sharpen Filter, Chroma Denoise or Polyphase Scaler.

A particular unit on the SHAVE or hardware accelerator is chosen depending on the specific operation and options that were provided to when compiling the neural network from Intermediate Representation (IR) into machine-readable format on the TM NPU (Intel® Movidius™ binary format). This is done automatically without the need for user intervention.

4.2 Relevance of the "vid_pipeline" and "neural_network" modules

The MicroPython API allows using the SHAVEs for various purposes when preprocessing an image and executing a neural network. When initializing the vid_pipeline, parameterization of the SHAVEs is (automatically) done using 'shaves_scaling' and 'shaves_conversion'. As the parameter names imply, a number of SHAVEs can be used to scale an image and apply format conversion.

With initializing the neural network, parameterization of the SHAVEs is (automatically) done using 'shaves_nn'. As the parameter name implies a number of SHAVEs can be used to run a neural network.

Furthermore, by setting the aforementioned 'use_cnn' parameter available CNN (Convolutional Neural Network) hardware accelerators are activated if the network is first in the list (find details in the chapter below). Suitable operations will then be executed with the help of the CNN hardware accelerators. This constitutes a performance boost since costly DDR accesses are avoided.

4.3 Default settings and order of SHAVEs

Since the number of SHAVEs is limited to sixteen in total, the order of allocating SHAVEs and the number of allocated SHAVEs for various purposes matters most. When applying the default settings of the afore mentioned parameters (shaves_scaling=4, shaves_conversion=4, shaves_nn=4), the SHAVEs are occupied as follows:

SHAVE number	Purpose
SHAVE 00-03	Scaling
SHAVE 04-07	Format conversion
SHAVE 08-11	Reserved
SHAVE 12-15	Neural network execution; with CNN hardware acceleration

NOTE

Refer to *vid_pipeline.init()* (Page 18) and *neural_network.init()* (Page 23) for details on parameterization.

4.4 Restrictions for the "vid_pipeline" and "neural_network" modules

To conform with the aforementioned allocations, the following restrictions apply:

- 'shaves_scaling': 4
- 'shaves_conversion': 4
- 'shaves_nn': 4
- 'use_cnn' (enforced automatically): (True, True)

Vid_pipeline: When initializing the video pipeline, eight shaves are allocated for scaling and conversion (four each).

Neural_Network: When initializing a neural network, four shaves are allocated for running it. The CNN hardware accelerators are automatically assigned to the first initialized neural network.

4.5 Performance relevance

The number of SHAVEs used for partial execution of tasks is relevant in terms of performance. However, there is no general rule of thumb for executing an application with or without SHAVEs due to the range of possible application scenarios. To gain a better understanding of performance criteria, the user can use standard MPython tools such as the 'utime' module.

Supported MPython modules on the TM NPU 5

5.1 General information

The integrated MicroPython Interpreter in the TM NPU has version 1.12. A subset of the standard MicroPython modules and libraries available can be imported in the user script ('main.py'). When creating the user application, refer to the following tables to check whether required modules, classes, functions, and constants are supported by the TM NPU. It is not possible to import or use non-supported modules. All other modules and functions can be imported and used. The modules listed in Section 5.2 "Explicitly tested MicroPython modules" are whitelisted and have been used extensively for the TM NPU system test. For a detailed description of the MicroPython standard modules and functions, please refer to the MicroPython documentation (https://docs.micropython.org/en/v1.12/).

5.2 Non-supported MicroPython modules

The following modules from the MicroPython standard are not supported on TM NPU and cannot be used:

Name of module	Classes and functions	Notes
usocket	Entire module	Provides access to the BSD socket interface.
ussl	Entire module	Provides access to Transport Layer Security encryption and peer authentication facilities for network sockets, both client-side and server-side.
utime	utime.mktime()	Inversed function of localtime.
_thread	Entire module	Inverse function of localtime.
btree	Entire module	Implements a simple keyvalue database using external storage (disk files, or in general cases, a random-access stream). Access to external storage not granted.
framebuf	Entire module	Provides a general frame buffer which can be used to create bitmap images for sending to a display.
machine	Entire module	Provides functions related to the hardware. Use TM NPU specific instead to interact with hardware.
micropython	micropython.kdb_intr(), micropython.schedule()	Allows KeyboardInterrupt exceptions and schedule the 'func' function to be executed "very soon".
network	Entire module	This module provides network drivers and routing configuration.

Name of module	Classes and functions	Notes
sys	sys.settrace()	Set the system's trace function, which allows you to implement a Python source code debugger in Python.
ubluetooth	Entire module	Provides an interface to a Bluetooth controller on a board.
ucryptolib	Entire module	Provides encryption/ decryption functionality.
uctypes	Entire module	Allows access to binary data in a structured way.
uhashlib	Entire module	This module implements binary data hashing algorithms.
uio	class uio.FileIO(), class uio.TextIOWrapper()	This module is for file access. Use the dedicated TM NPU specific module "npufs" to access files from sdcard/ ftp.
uos	Entire module	The uos module contains functions for filesystem access and mounting, terminal redirection and dplication, and the uname and urandom functions. Services for Port TM NPU are accessable via dedicated modules.
uselect	Entire module	This module provides functions to efficiently wait for evvents on multiple streams (selet streams which are ready for operations).

5.3 Explicitly tested MicroPython modules

The following modules, classes and functions have been tested in a TM NPU context, meaning that these modules, classes and functions, have been used either in full scope or partially within MicroPython scripts in an automated test environment:

Name of module	Classes and functions	Notes
builtins	bytearray, bytes, dir, len, list, print, range, round, str, sum, type, Exception, RuntimeError	Built-In classes and functions comes with MicroPython
sys	print_exception, exc_info	System functions
ure	Partially tested	Regular expressions
math	Partially tested	Mathematical functions
utime	Partially tested	Time related functions
uarray	Partially tested	Arrays of numeric data
ustruct	Partially tested	Pack and unpack primitive data types
uzlib	Partially tested	Compress and decompress files

5.4 Modules which differ from the official standard documentation

The following notes describe modules, classes, functions, constants, or exceptions that differ from the referenced MicroPython and CPython documentations. Note that the selected version of the documentation is relevant for a detailed description of enabled and disabled functionality.

Name of module	Classes and functions	Notes
utime	utime.clock()	This method has been deprecated since CPython version 3.3 and will be removed in CPython version 3.8. The behavior of this method is platform dependent. For details consult e.g. CPython documentation version 2.7 (https://docs.python.org/2.7/)

General notes and troubleshooting

6

This chapter contains general hints for the Micropython API. It lists exceptions and gives possible workarounds as well as trouble shooting for various topics.

6.1 Troubleshooting list

Occurrence	Function	Description	Troubleshooting remedy

neural_network (Page 23)	neural_network.init()	Runtime error: “Error: Initialized neural network exceeds available memory”	Receiving this error during runtime indicates that the required buffer size for initializing the neural network is not available. The buffer size differs from the file size of the neural network. Exact limits of the available buffer cannot be stated as this heavily depends on the application itself. To overcome this issue, ensure that the neural network requires a smaller buffer. The required buffer size is affected by the number and size layers of the neural network, as well as its input and output layers. When creating the model_buffer and converting it with the corresponding OpenVINO version, use a config file and add the following parameters to get the required buffer size from the dumped files. Content of file:
			`MYRIAD_DUMP_ALL_PASSES YES`
			`MYRIAD_DUMP_INTERNAL_GRAPH_DIRECTORY dump`
			The required buffer size is marked as “BSS” in the dump files. This applies to OpenVINO versions >= OpenVINO 2021.2
			BSS stands for “Block Starting Symbol”.

neural_network (Page 23), *vid_pipeline* (Page 17)	neural_network.init(), vid_pipeline.init()	Calling both functions in a loop causes the app to freeze	Repeatedly initializing both the video pipeline and the neural network in a loop leads to the app freezing. Generally, the video pipeline initialization should only occur once during an "initialization phase". A release and re-initialization of the video pipeline during module runtime is not supported.
neural_network (Page 24)	neural_network.run()	Unexpected prediction for a given input	Every input provided to the given neural network is checked prior to processing it. This entails verification of whether the provided input data can be shaped into the expected dimensions of the input tensor of the neural network. Inspections regarding the order of the input dimensions are not included and therefore have to be double checked by the user. To obtain valid predictions, ensure that especially the width, height, channels, and bytes per pixel of the input data match the neural network.
neural_network (Page 24)	neural_network.run()	Unexpected number of detected objects for a given input	The size of the output tensor during object detection is encoded in the network provided (“config_number”) as a maximum number of detections per output tensor. If the number of detected objects (“output_layer_number”) is not as desired, please double check the model conversion process. A configuration file that is passed to mo_tf.py for model optimization via OpenVINO can, for instance, be used to specify the upper limit for detected objects. If the config_number is lower than the output_layer_number, less memory than required is allocated for the output buffer. Therefore, the least relevant results will be dropped. If the config_number is higher than the output_layer_number, more memory than required is allocated for the output buffer. Therefore, no results will be dropped. If

Occurrence	Function	Description	Troubleshooting remedy
			the memory for the output buffer cannot be allocated, an exception will be thrown.
model conversion via Open-VINO	mo_tf.py	FPN topologies are not supported at model conversion	OpenVINO does not support the use of FPN topologies (Feature Pyramid Networks) as they contain hard-coded shapes for some operations. Do not use mobile networks with these layers, use other SSD topologies instead. Consult the official OpenVINO website for details.
model conversion via Open-VINO	mo_tf.py	Version mismatch during model conversion	The use of corresponding versions within the conversion process is key to receiving an executable model. This applies with respect to the tensorflow version and the version used to generate the intermediate representation (IR). For generating the IR, the version is passed via the '–transformations_ config' parameter to mo_tf.py. If a tensorflow model was trained with TF version 2.0 to 2.3, apply version 2.0 as a parameter (. . . /ssd_support_api_v2.0.json). If a tensorflow model was trained with TF version 2.4 or higher, apply version 2.4 as a parameter (. . . /ssd_support_api_v2.4.json). For details regarding the config file and the conversion process itself, consult the official OpenVINO website. For example: python3 mo_tf.py –saved_model_dir /ssd_mobilenet_v2/saved_model –transformations_config /model_optimizer/extensions/front/tf/ ssd_support_api_v2.4.json –tensorflow_object_detection_api_pipeline_config /ssd_mobilenet_v2/pipeline.config –reverse_input_channels –input_shape [1,640,640,3]
neural_network (Page 24)	neural_network.run()	Predictions differ slightly when using CPU vs. NPU	There are multiple reasons for this behavior. A possible issue might be that certain neural network operations change slightly when being optimized by OpenVINO. To bypass this issue, corresponding operations can be put on the optimization blacklist using the VPU_HW_BLACK_LIST parameter in the configuration file passed to myriad_compile. To ensure this is the case, the user can temporarily turn off the hardware acceleration on the NPU using the VPU_HW_STAGES_OPTIMIZATION NO parameter.
neural_network (Page 24)	neural_network.run()	Multiple output tensors not supported	The user needs to ensure that the model is converted with a 1-D output tensor only, via OpenVINO. Multidimensional output tensors are not supported by the NPU, even if the model conversion via OpenVINO is successful.

Garbage collection	with	Object usage after dereferenced and garbage collected	If an object is to be used multiple times, ensure that it is always within the scope of the corresponding structure. Since Micropython uses a similar reference model to Python, implicitly or explicitly deleting references of objects can cause errors during execution. Therefore, in order not to lose a reference, it is recommended to stay within one specified scope.
			
			Problematic behavior:
			refImage = **None**
			#first with scope
			with vid_pipeline.read_raw() as frame:
			  #assign ref to refImage
			  refImage = frame.data()
			  #garbage collect frame.data
			  #when leaving 'first with scope'
			
			# . . .
			while True:

Occurrence	Function	Description	Troubleshooting remedy

			# . . .
			with vid_pipeline.read_raw() **as** frame:
			#refImage value no longer available
			#as ref has been garbage collected
			#before
			if fooBar(refImage,frame.data()):
			# . . .
			
			Desired behavior:
			refImage = **None**
			#first with scope
			with vid_pipeline.read_raw() as frame1:
			#assign ref to refImage
			refImage = frame1.data()
			#garbage collection not taking place
			#for refImage as still within
			#'first with scope'
			
			# . . .
			while True:
			# . . .
			with vid_pipeline.read_raw() **as** frame:
			#refImage value still available
			#still within 'first with scope'
			if fooBar(refImage,frame.data()):
			# . . .
			#garbage collect refImage here

CHANGELOG: Log-relevant API changes

7

The `changelog` documents major changes to the TM NPU MicroPython API.

8 Indices and tables

- genindex
- modindex
- search

Index