Artificial Intelligence (AI) and computer vision function pack for STM32H7 microcontrollers

Introduction

FP-AI-VISION1 is a function pack (FP) demonstrating the capability of STM32H7 Series microcontrollers to execute a Convolutional Neural Network (CNN) efficiently in relation to computer vision tasks. FP-AI-VISION1 contains everything needed to build a CNN-based computer vision application on STM32H7 microcontrollers.

FP-AI-VISION1 also demonstrates several memory allocation configurations for the data involved in the application. Each configuration enables the handling of specific requirements in terms of the amount of data required by the application. Accordingly, FP-AI-VISION1 implements examples describing how to place the different types of data efficiently in both the on-chip and external memories. These examples enable the users to understand easily which memory allocation fits their requirements the best.

This user manual describes the content of the FP-AI-VISION1 function pack and details the different steps to be carried out in order to build a CNN-based computer vision application on STM32H7 microcontrollers.

1 General information

The FP-AI-VISION1 function pack runs on the STM32H7 microcontrollers based on the Arm® Cortex®-M7 processor.

Note: Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

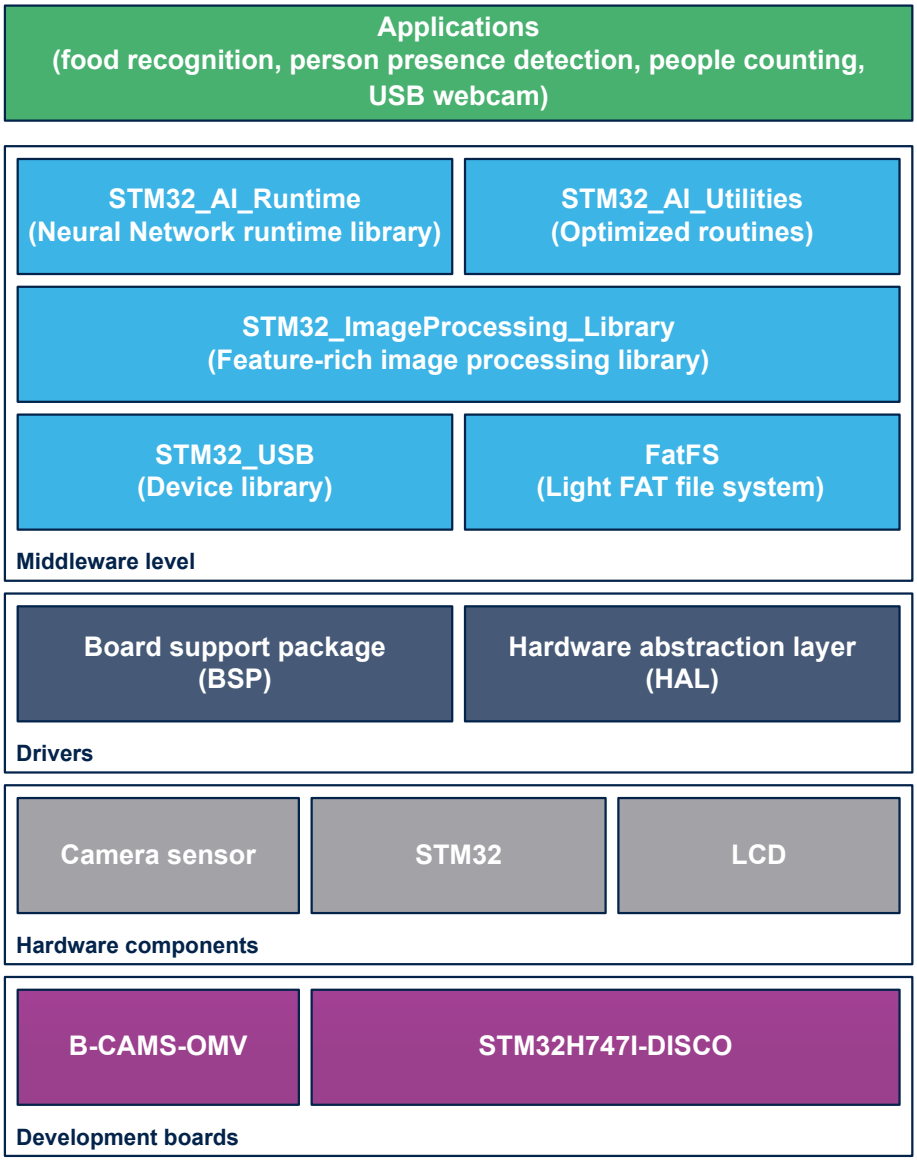
1.1 FP-AI-VISION1 function pack feature overview

- Runs on the STM32H747I-DISCO board connected with the B-CAMS-OMV camera module bundle (advised) or STM32F4DIS-CAM camera daughterboard (legacy only)
- Includes three image classification application examples based on CNN:
  - One food recognition application operating on color (RGB 24 bits) frame images
  - One person presence detection application operating on color (RGB 24 bits) frame images
  - One person presence detection application operating on grayscale (8 bits) frame images
- Includes one people counting application (counting the number of persons in a scene) based on an object-detection CNN model
- Includes complete application framework for camera capture, frame image preprocessing, inference execution, and output post-processing
- Includes feature-rich image processing library supporting for more than 100 processing operations
- Includes examples of integration of both floating-point and 8-bit quantized C models
- Supports several configurations for data memory placement in order to meet application requirements
- Includes test and validation firmware in order to test, debug, and validate the embedded application
- Includes capture firmware enabling dataset collection
- Includes support for file handling (on top of FatFS) on an external microSD™ card
- Includes a USB webcam application, which can be used to create image and video datasets as well as to perform live testing on the host

1.2 Software architecture

The top-level architecture of the FP-AI-VISION1 function pack usage is shown in Figure 1.

Figure 1. FP-AI-VISION1 architecture



1.3 Terms and definitions

Table 1 presents the definitions of the acronyms that are relevant for a better contextual understanding of this document.

Table 1. List of acronyms

Acronym	Definition
API	Application programming interface
BSP	Board support package
CNN	Convolutional Neural Network

Acronym	Definition
DMA	Direct memory access
FAT	File allocation table
FatFS	Light generic FAT file system
FP	Function pack
FPS	Frame per second
HAL	Hardware abstraction layer
LCD	Liquid-crystal display
MCU	Microcontroller unit
microSD™	Micro secure digital
MIPS	Million of instructions per second
NN	Neural Network
RAM	Random-access memory
QVGA	Quarter VGA
SRAM	Static random-access memory
USB	Universal Serial Bus
VGA	Video graphics array resolution

1.4 Overview of available documents and references

Table 2 lists the complementary references for using FP-AI-VISION1.

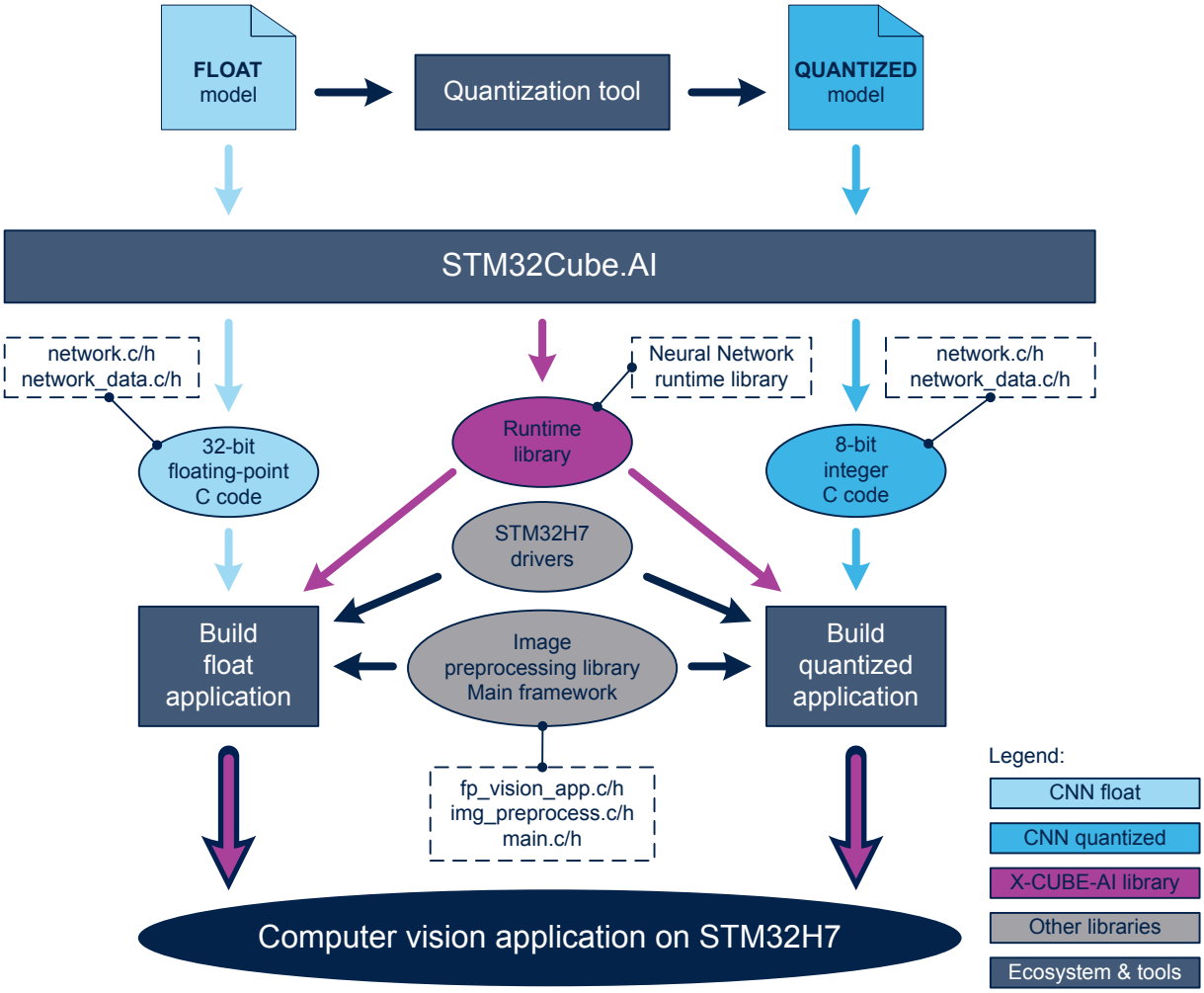
Table 2. References

ID	Description
[1]	User manual: *Getting started with X-CUBE-AI Expansion Package for Artificial Intelligence (AI)* (UM2526).
[2]	Reference manual: *STM32H745/755 and STM32H747/757 advanced Arm®-based 32-bit MCUs* (RM0399).
[3]	MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications: *https://arxiv.org/pdf/1704.04861.pdf*
[4]	The Food-101 Data Set: *https://data.vision.ee.ethz.ch/cvl/datasets_extra/food-101/*
[5]	STM32CubeProgrammer software for programming STM32 products: STM32CubeProg
[6]	Keras - The Python Deep Learning library: *https://keras.io/*
[7]	STM32Cube initialization code generator: STM32CubeMX

2 Building a CNN-based computer vision application on STM32H7

Figure 2 illustrates the different steps to obtain a CNN-based computer vision application running on the STM32H7 microcontrollers.

Figure 2. CNN-based computer vision application build flow



Starting from a floating-point CNN model (designed and trained using a framework such as Keras), the user generates an optimized C code (using the STM32Cube.AI tool, [1]) and integrates it in a computer vision framework (provided as part of FP-AI-VISION1) in order to build his computer vision application on STM32H7.

The user has the possibility to select one of two options for generating the C code:

- Either generating the floating-point C code directly from the CNN model in floating-point
- Or quantizing the floating-point CNN model to obtain an 8-bit model, and subsequently generating the corresponding quantized C code

For most CNN models, the second option enables to reduce the memory footprint (Flash and RAM) as well as inference time. The impact on the final output accuracy depends on the CNN model as well as on the quantization process (mainly the test dataset and the quantization algorithm).

As part of the FP-AI-VISION1 function pack, three image classification application examples are provided including the following material:

- One application for food recognition:
  - Floating–point Keras model (`.h5` file)
  - 8-bit quantized model (`.h5` file + `.json` file) obtained using STM32Cube.AI (X-CUBE-AI) quantizer
  - Generated C code in both floating point and 8-bit quantized format
  - Example of computer vision application integration based on C code generated by STM32Cube.AI (X-CUBE-AI)
- Two applications for person presence detection:
  - 8-bit quantized models (`.tflite` file) obtained using the TFLiteConverter tool(1)
  - Generated C code in 8-bit quantized format
  - Examples of computer vision application integration based on C code generated by STM32Cube.AI (X-CUBE-AI)

1. *TensorFlow is a trademark of Google Inc.*

2.1 Integration of the generated code

From a float or quantized model, the user must use the STM32Cube.AI tool (X-CUBE-AI) to generate the corresponding optimized C code.

When using the GUI version of STM32Cube.AI (X-CUBE-AI) with the user's own `.ioc` file, the following set of files is generated in the output directory:

- `Src\network.c` and `Inc\network.h`: contain the description of the CNN topology
- `Src\network_data.c` and `Inc\network_data.c`: contain the weights and biases of the CNN

Note: For the network, the user must keep the default name, which is "network". Otherwise, the user must rename all the functions and macros contained in files `ai_interface.c` and `ai_interface.h`. The purpose of the `ai_interface.c` and `ai_interface.h` files is to provide an abstraction interface to the NN API.

From that point, the user must copy and replace the above generated `.c` files and `.h` files respectively into the following directories:

- `\Projects\STM32H747I-DISCO\Applications\<app_name>\CM7\Src`

  `<app_name>` is any of
  - `FoodReco_MobileNetDerivative\Float_Model`
  - `FoodReco_MobileNetDerivative\Quantized_Model`
  - `PersonDetection\Google_Model`
  - `PersonDetection\MobileNetv2_Model`
  - `PeopleCounting`
- `\Projects\STM32H747I-DISCO\Applications\<app_name>\CM7\Inc`

  `<app_name>` is any of
  - `FoodReco_MobileNetDerivative\Float_Model`
  - `FoodReco_MobileNetDerivative\Quantized_Model`
  - `PersonDetection\Google_Model`
  - `PersonDetection\MobileNetv2_Model`
  - `PeopleCounting`

An alternate solution is to use the CLI (command-line interface) version of STM32Cube.AI (X-CUBE-AI), so that the generated files be directly copied into the `Src` and `Inc` directories contained in the output directory provided on the command line. This solution does not require any manual copy/paste operation.

The application parameters are configured in files `fp_vision_app.c` and `fp_vision_app.h` where they can be easily adapted to the user's needs.

In file `fp_vision_app.c`:

- The `output_labels[]` table of strings (where each string corresponds to one output class of the Neural Network model) is the only place where adaptation is absolutely required for a new application.
- The `App_Context_Init()` function is in charge of initializing the different software components of the application. Some changes may be required to:
  - adapt the camera orientation
  - adapt the path to read input images from the microSD™ card when in *Onboard Validation* mode
  - **adapt to the NN input data range used during the training phase**

    This is achieved by updating the value of both the `nn_input_norm_scale` and `nn_input_norm_zp` variables during initialization. The `nn_input_norm_scale` and `nn_input_norm_zp` variables affect the *pixel format adaptation* stage (refer to Figure 7. in section Section 3.2.5.1).
  - **adapt the pixel color format of the NN input data**

    This is achieved by updating the initialization value of the `Pfc_Dst_Img.format` variable. The `Pfc_Dst_Img.format` variable affects the *pixel color format conversion* stage (PFC; refer to Figure 7. in section Section 3.2.5.1).

3 Package content

3.1 CNN models

The FP-AI-VISION1 function pack is demonstrating two CNN-based image classification applications:

- A food-recognition application recognizing 18 types of food and drink
- A person presence detection application identifying whether a person is present in the image or not

The FP-AI-VISION1 function pack is also demonstrating a people counting application relying on object-detection Neural Network model.

3.1.1 Food recognition application

The food-recognition CNN is a derivative of the MobileNet model (refer to [3]).

MobileNet is an efficient model architecture [3] suitable for mobile and embedded vision applications. This model architecture was proposed by Google®.

The MobileNet model architecture includes two simple global hyper-parameters that efficiently trade off between latency and accuracy. Basically these hyper-parameters allow the model builder to determine the application right-sized model based on the constraints of the problem.

The food recognition model that is used in this FP has been built by adjusting these hyper-parameters for an optimal trade-off between accuracy, computational cost and memory footprint, considering the STM32H747 target constraints.

The food-recognition CNN model has been trained on a custom database of 18 types of food and drink:

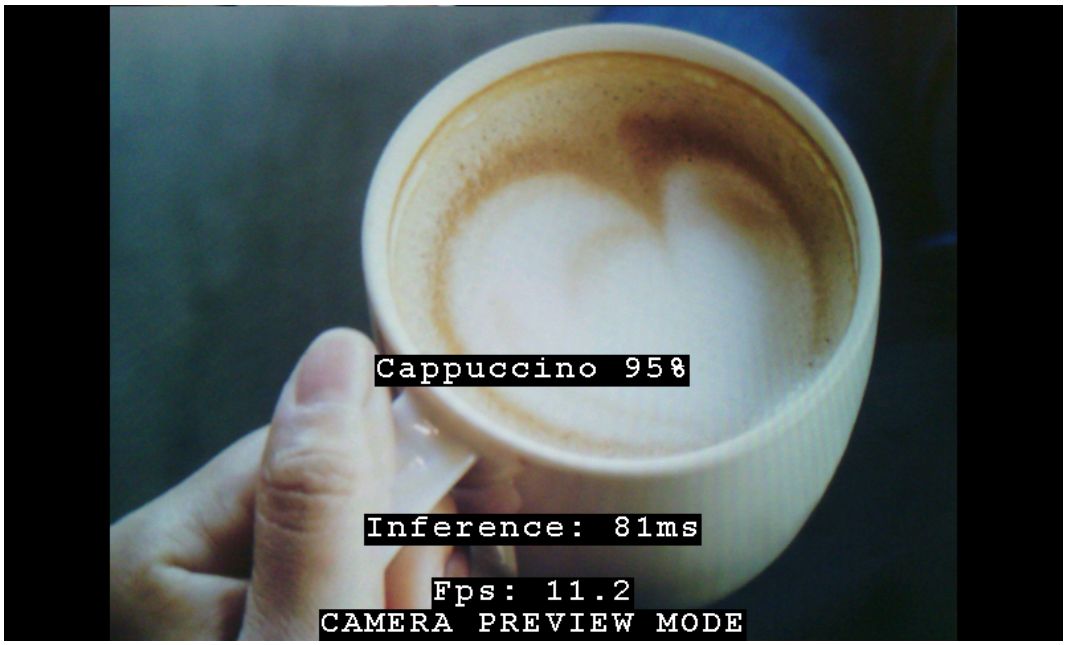
- Apple pie
- Beer
- Caesar salad
- Cappuccino
- Cheesecake
- Chicken wings
- Chocolate cake
- Coke™
- Cupcake
- Donut
- French fries
- Hamburger
- Hot dog
- Lasagna
- Pizza
- Risotto
- Spaghetti bolognese
- Steak

The food-recognition CNN is expecting color image of size 224 × 224 pixels as input, each pixel being coded on three bytes: RGB888.

The FP-AI-VISION1 function pack includes two examples based on the food recognition application: one example implementing the floating-point version of the generated code, and one example implementing the quantized version of the generated code.

Figure 3 illustrates the food recognition application.

Figure 3. Food recognition application



3.1.2 Person presence detection application

Two person presence detection applications are provided in this package:

- One based on a low-complexity CNN model (so-called `Google_Model`) working on grayscale images (8 bits per pixel) with a resolution of 96 × 96 pixels. The model is downloaded from storage.googleapis.com.
- One based on a higher-complexity CNN model (so-called `MobileNetv2_Model`) working on color images (24 bits per pixel) with a resolution of 128 × 128 pixels.

The person presence detection models contain two output classes: `Person` and `Not Person`.

The FP-AI-VISION1 function pack demonstrates 8-bit quantized models.

Figure 4 illustrates a person presence detection application.

Figure 4. Person presence detection application

3.1.3 People counting application

The people counting application relies on an object-detection model, optimized for STM32H7 microcontrollers, in charge of processing the captured image and providing as output the bounding boxes corresponding to people's positions in the image.

The people counting application enables people recognition, counting and tracking in images.

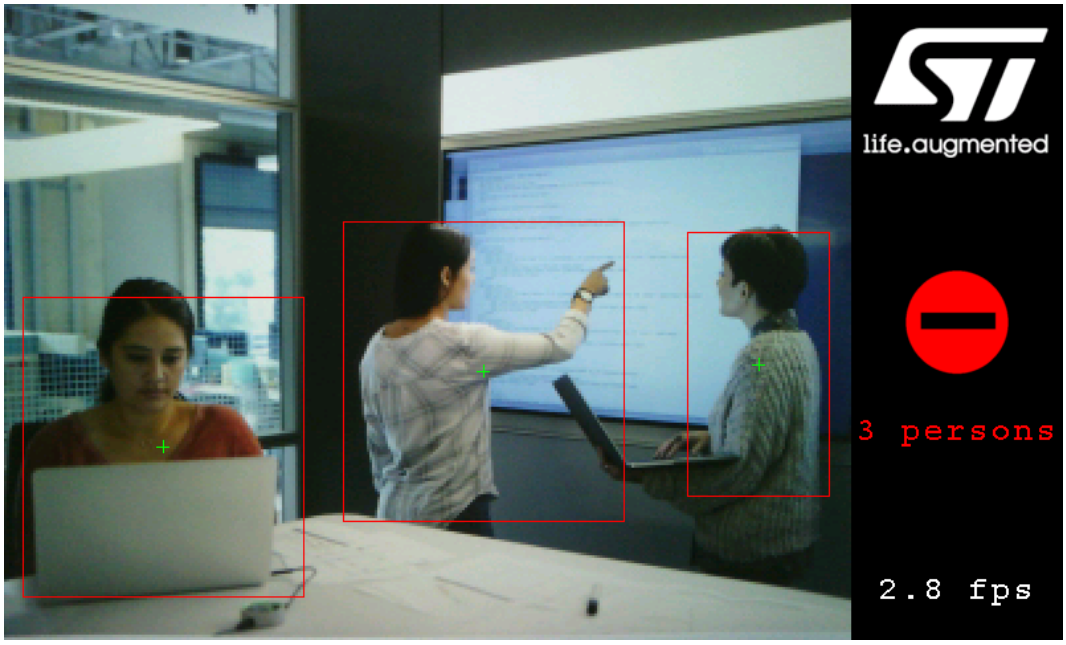
The object-detection model used is part of STMicroelectronics model zoo and is a derivative of the YOLO model, quantized on 8-bit and expecting a 240 × 240 resolution input image in RGB888 format.

The people couting application implements a full internal memory allocation scheme.

The object-detection Neural Network model used is provided as a binary library. Contact STMicroelectronics for more details.

Figure 5 illustrates the people counting application.

Figure 5. People counting application

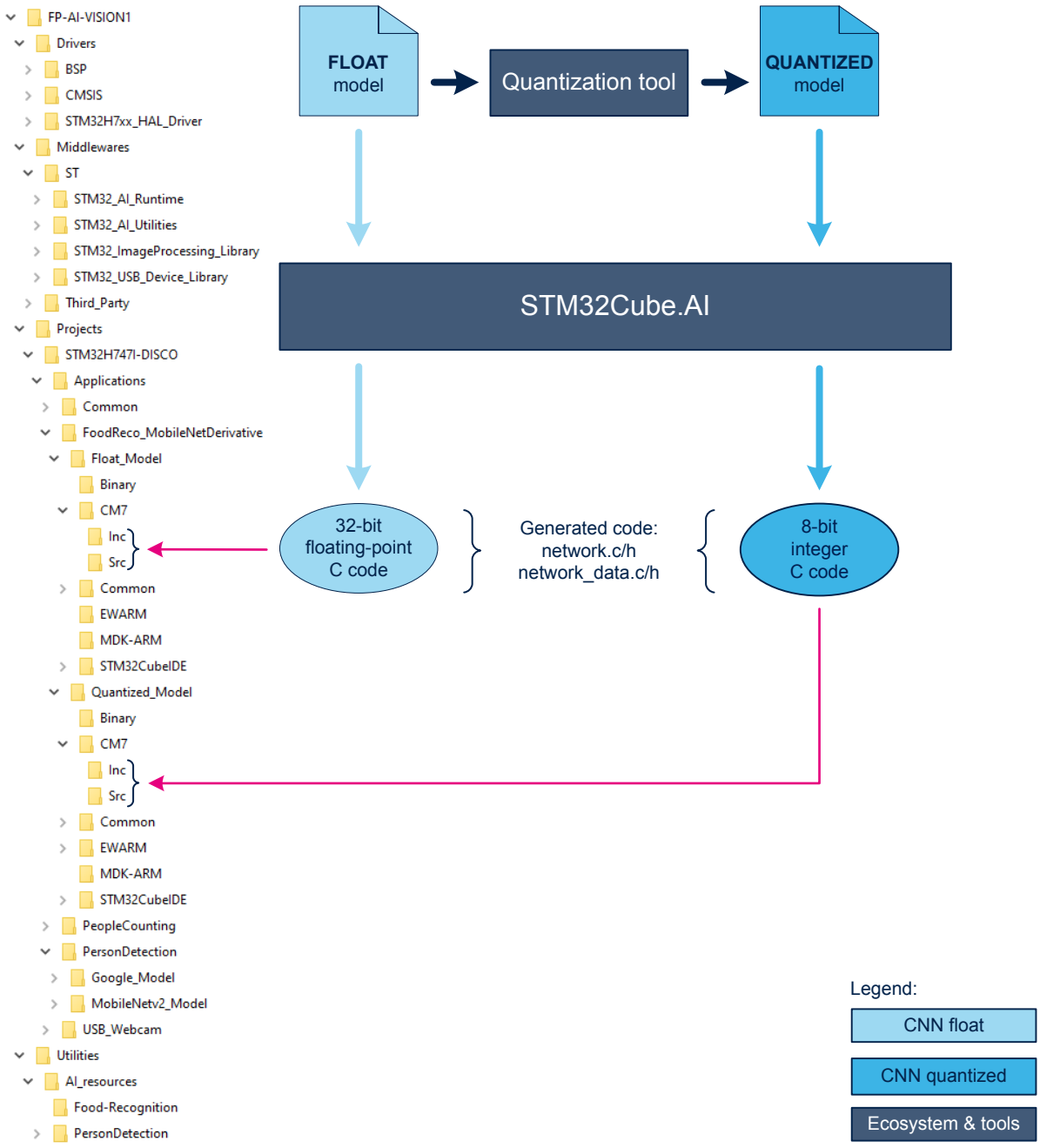


3.2 Software

3.2.1 Folder organization

Figure 6 shows the folder organization in FP-AI-VISION1 function pack.

Figure 6. FP-AI-VISION1 folder tree



`Driver`

Contains all the BSP and STM32H7 HAL source code.

Middlewares

Contains five sub-folders:

- `ST/STM32_AI_Runtime`

  The `lib` folder contains the Neural Network runtime libraries generated by STM32Cube.AI (X-CUBE-AI) for each IDE: IAR Embedded Workbench® from IAR Systems (EWARM), MDK-ARM from Keil®, and STM32CubeIDE from STMicroelectronics. These libraries do not need to be replaced when converting a new Neural Network.

  The `Inc` folder contains the include files required by the runtime libraries.

  These two folders do not need to be replaced when converting a new Neural Network, unless using a new version of the X-CUBE-AI code generator.
- `ST/STM32_AI_Utilities`

  Contains optimized routines.
- `ST/STM32_ImageProcessing_Library`

  Contains a library of functions for image processing. These functions are used to preprocess the input frame image captured by the camera. The purpose of this preprocessing is to generate the adequate data (such as size, format, and others) to be input to the Neural Network during the inference. Refer to Section 3.2.2 for more details about this ibrary.
- `ST/STM32_USB_Device_Library`

  Contains the USB device library for STM32.
- `Third_Party/FatFS`

  Third party middleware providing support for FAT file system.

Project/STM32H747I-DISCO/Applications

Contains the projects and source codes for the applications provided in the FP-AI-VISION1 FP. These applications are running on the STM32H747 (refer to [2]), which is a dual-core microcontroller based on the Cortex®-M7 and Cortex®-M4 processors. The application code is running only on the Cortex®-M7 core.

Project/STM32H747I-DISCO/Applications/Common

This folder contains the source code common to all applications:

- `ai_interface.c` and `ai_interface.h`

  Provide an abstraction of the NN API.
- `fp_vision_ai.c` and `fp_vision_ai.h`

  Provide the utilities that are required to adapt the representation of the NN input data, post-process the NN output data, initialize the NN, and run an inference of the NN. These files require to be adapted by the user for application parameters when integrating a new Neural Network model.
- `fp_vision_camera.c` and `fp_vision_camera.h`

  Provide the functions to configure and manage the camera module.
- `fp_vision_display.c` and `fp_vision_display.h`

  Provide the functions to configure and manage the LCD display.
- `fp_vision_preproc.c` and `fp_vision_preproc.h`

  Provide an abstraction layer to the image preprocessing library (located in `Middlewares/ST/STM32_Image`).
- `fp_vision_test.c` and `fp_vision_test.h`

  Provide a set of functions for testing, debugging and validating the application.
- `fp_vision_utils.c` and `fp_vision_utils.h`

  Provide a set of miscellaneous utilities.

Project/STM32H747I-DISCO/Applications/FoodReco_MobileNetDerivative

This folder contains all the source code related to the food recognition application. It contains two sub-folders, one sub-folder per application example:

- One demonstrating the integration of the float C model (32-bit float C code)
- One demonstrating the integration of the quantized C model (8-bit integer C code)

Each sub-folder is composed as follows:

- `Binary`

  Contains the binaries for the applications:

  – `STM32H747I-DISCO_u_v_w_x_y_z.bin`

  Binaries generated from the source files contained in the `Float_Model/CM7` and `Quantized_Model/CM7` folders.

  ◦ `u` corresponds to the application name. For the food recognition application, the value is:

  • `Food`

  ◦ `v` corresponds to the model type. For the food recognition application, it can be:

  • `Std` (for standard)

  • `Optimized` (for optimized)

  When `v` is `Opt`, it means that the binary is generated from sources that are not released as part of the FP-AI-VISION1 function pack since they are generated from a specific version of the food recognition CNN model. This specific version of the model is further optimized for a better trade-off between accuracy and embedded constraints such as memory footprint and MIPS. Contact STMicroelectronics for information about this specific version.

  ◦ `w` corresponds to the data representation of the model type. For the food recognition application, it can be:

  • `Float` (for float 32 bits)

  • `Quant8` (for quantized 8 bits)

  ◦ `x` corresponds to the configuration for the volatile data memory allocation. For the food recognition application, it can be:

  • `Ext` (for external SDRAM)

  • `Split` (for split between internal SRAM and external SDRAM)

  • `IntMem` (for internal SRAM with memory optimized)

  • `IntFps` (for internal SRAM with FPS optimized)

  ◦ `y` corresponds to the memory allocation configurations for the non-volatile data. For the food recognition application, it can be:

  • `IntFlash` (for internal Flash memory)

  • `QspiFlash` (for external Q-SPI Flash memory)

  • `ExtSdram` (for external SDRAM)

  ◦ `z` corresponds to the version number of the FP-AI-VISION1 release. It is expressed as `Vab c` where `a`, `b` and `c` represent the major version, minor version, and patch version numbers respectively. For the food recognition application corresponding to this user manual, the value is:

  • `V300`

- `CM7`

  Contains the source code specific to the food recognition application example that is executed on the Cortex®-M7 core. There are two types of files:

  – Files that are generated by the STM32Cube.AI tool (X-CUBE-AI):
    - `network.c` and `network.h`: contain the description of the CNN topology
    - `network_data.c` and `network_data.h`: contain the weights and biases of the CNN

  – Files that contain the application:
    - `main.c` and `main.h`
    - `fp_vision_app.c` and `fp_vision_app.h`

      Used to configure the application specific settings.
    - `stm32h7xx_it.c` and `stm32h7xx_it.h`

      Implement the interrupt handlers.

- `CM4`

  This folder is empty since all the code of the food recognition application is running on the Cortex®-M7 core.

- `Common`

  Contains the source code that is common to the Cortex®-M7 and Cortex®-M4 cores.

- `EWARM`

  Contains the IAR Systems IAR Embedded Workbench® workspace and project files for the application example. It also contains the startup files for both cores.

- `MDK-ARM`

  Contains the Keil® MDK-ARM workspace and project files for the application example. It also contains the startup files for both cores.

- `STM32CubeIDE`

  Contains the STM32CubeIDE workspace and project files for the application example. It also contains the startup files for both cores.

Note: For the `EWARM`, `MDK-ARM` and `STM32CubeIDE` sub-folders, each application project may contain several configurations. Each configuration corresponds to:

- *A specific data placement in the volatile memory (RAM)*
- *A specific placement of the weight-and-bias table in the non-volatile memory (Flash)*

`Project/STM32H747I-DISCO/Applications/PeopleCounting`

This folder contains the source code that is specific to the people counting application.

The organization of sub-folders is similar to the one of the sub-folders described above in the context of the food recognition application examples. The `CM7` folder has one more sub-folder named `lib`, which contains a binary library implementing the object-detection model.

The Binary sub-folder contains the binaries for the applications. The binaries are named as STM32H747I-DISCO_u_w_x_y_z.bin where:

- u corresponds to the application name. For the people counting applications, the value is:
  - PeopleCounting
- w corresponds to the data representation of the model type. For the people counting application, the value is:
  - Quant8 (for quantized 8 bits)
- x corresponds to the memory allocation configurations for the volatile data. For the people counting applications, the value is:
  - IntFps (for internal SRAM with FPS optimized)
- y corresponds to the memory allocation configurations for the non-volatile data. For the people counting applications, the value is:
  - IntFlash (for internal Flash memory)
- z corresponds to the version number of the FP-AI-VISION1 release. It is expressed as Vabc where a, b and c represent the major version, minor version, and patch version numbers respectively. For the people counting application corresponding to this user manual, the value is:
  - V300

Project/STM32H747I-DISCO/Applications/PersonDetection

This folder contains the source code that is specific to the person presence detection applications. It contains two sub-folders, one sub-folder per application example:

- One demonstrating the integration of a low-complexity model (so-called Google_Model)
- One demonstrating the integration of a medium-complexity model (so-called MobileNetv2_Model)

The organization of sub-folders is identical to the one of the sub-folders described above in the context of the food recognition application examples.

The Binary sub-folder contains the binaries for the applications. The binaries are named as STM32H747I-DISCO_u_v_w_x_y_z.bin where:

- u corresponds to the application name. For the person presence detection applications, the value is:
  - Person
- v corresponds to the model type. For the person presence detection applications, it can be:
  - Google
  - MobileNetV2
- w corresponds to the data representation of the model type. For the person presence detection applications, the value is:
  - Quant8 (for quantized 8 bits)
- x corresponds to the memory allocation configurations for the volatile data. For the person presence detection applications, the value is:
  - IntFps (for internal SRAM with FPS optimized)
- y corresponds to the memory allocation configurations for the non-volatile data. For the person presence detection applications, the value is:
  - IntFlash (for internal Flash memory)
- z corresponds to the version number of the FP-AI-VISION1 release. It is expressed as Vabc where a, b and c represent the major version, minor version, and patch version numbers respectively. For the person presence detection application corresponding to this user manual, the value is:
  - V300

Project/STM32H747I-DISCO/Applications/USB_Webcam

This folder contains the source code specific to the USB webcam application. The organization of the folder is identical to the one of the sub-folders described above in the context of the food recognition application examples.

The Binary sub-folder contains the binary for the application. The binary are named as STM32H747I-DISCO_u_z.bin where:

- u corresponds to the application name. For the USB webcam application, the value is:
  - Webcam
- z corresponds to the version number of the FP-AI-VISION1 release. It is expressed as V.abc where a, b and c represent the major version, minor version, and patch version numbers respectively. For theUSB webcam application corresponding to this user manual, the value is:
  - V300

Utilities/AI_resources/Food-Recognition

This sub-folder contains:

- The original trained model (file FoodReco_MobileNet_Derivative_Float.h5) for the food recognition CNN used in the application examples. This model is used to generate:
  - Either directly the floating-point C code via STM32Cube.AI (X-CUBE-AI)
  - Or the 8-bit quantized model via the quantization process, and then subsequently the integer C code via STM32Cube.AI (X-CUBE-AI)
- The files required for the quantization process (refer to Section 3.2.3 Quantization process):
  - config_file_foodreco_nn.json: file containing the configuration parameters for the quantization operation
  - test_set_generation_foodreco_nn.py: file containing the function used to prepare the test vectors used in the quantization process
- The quantized model generated by the quantization tool (files FoodReco_MobileNet_Derivative_Quantized.json and FoodReco_MobileNet_Derivative_Quantized.h5)
- The re-training script (refer to Section 3.2.4 Training scripts): FoodDetection.py along with a Jupyter™ notebook (FoodDetection.ipynb)
- A script (create_dataset.py) to convert a dataset of images in the format expected by the piece of firmware performing the validation on board (refer to Onboard Validation mode in Section 3.2.12 Embedded validation, capture and testing)

Utilities/AI_resources/PersonDetection

This sub-folder contains:

- MobileNetv2_Model/README.md: describes how to retrain a new person detection image classifier from a pretrained network using TensorFlow™.
- MobileNetv2_Model/create_dataset.py: Python™ script to create the ***Person20*** dataset from the previously downloaded COCO dataset as described in the README.md file.
- MobileNetv2_Model/train.py: Python™ script to create an image classifier model from a pretrained MobileNetV2 head.
- MobileNetv2_Model/quantize.py: Python™ script to perform post-training quantization on a Keras model using the TFLiteConverter tool from TensorFlow™. Sample images are required to run the quantization operation.

3.2.2 STM32 image processing library

The STM32_ImageProcessing_Library is a feature-rich image processing library supporting for more than 100 image processing operations. It includes not only common operation (like image rescaling and image color conversion) but also advanced operations (like blob tracking and face detection).

The library supports for read and write operations of several file formats such as .bmp and .jpeg. For more details, refer to the documentation located in the Middlewares/ST/STM32_ImageProcessing_Library folder.

3.2.3 Quantization process

The quantization process consists in quantizing the parameters (weights and biases) as well as the activations of a NN in order to obtain a quantized model having parameters and activations represented on 8-bit integers.

Quantizing a model reduces the memory footprint because weights, biases, and activations are on 8 bits instead of 32 bits in a float model. It also reduces the inference execution time through the optimized DSP unit of the Cortex®-M7 core.

Several quantization schemes are supported by the STM32Cube.AI (X-CUBE-AI) tool:

- Fixed point Qm,n (deprecated feature)
- Integer arithmetic (signed and unsigned)

X-CUBE-AI can import different types of quantized model in 8-bit integer format:

- A Keras float model associated with its tensor format configuration file. The conversion of each 32-bit float weight-and-bias tensors to 8-bit integer format is directly achieved by the importer through the settings provided.
- A quantized TensorFlow™ Lite model (generated by a post-training or training-aware process). In this case, the calibration has been performed by the TensorFlow™ Lite framework, principally through the TFLiteConverter utility exporting the TensorFlow™ Lite file.

Refer to the STM32Cube.AI tool (X-CUBE-AI) documentation in [1] for more information on the different quantization schemes and how to run the quantization process.

Note:
- *Two datasets are required for the quantization operation. It is up to the users to provide their own datasets.*
- *The impact of the quantization on the accuracy of the final output depends on the CNN model (that is its topology), but also on the quantization process: the test dataset and the quantization algorithm have a significant impact on the final accuracy.*

3.2.4 Training scripts

Training scripts are provided for each application.

3.2.4.1 Food recognition application

File `Utilities/AI_ressources/Food-Recognition/FoodDetection.ipynb` contains an example script showing how to train the MobileNet derivative model used in the function pack. As the dataset used to train the model provided in the function pack is not publicly available, the training script relies on a subset of the Food-101 dataset (refer to [4]). This publicly available dataset contains images of 101 food categories with 1000 images per category.

In order to keep the training process short, the script uses only 50 images per food category, and limits the training of the model to 20 epochs. To achieve a training on the whole dataset, the variable `max_imgs_per_class` in section *Prepare the test and train datasets* must be updated to `np.inf`.

Note: The use of the GPU is recommended for the complete training on the whole dataset.

The Jupyter™ notebook is also available as a plain Python™ script in the `Utilities/AI_ressources/Food-Recognition/FoodDetection.py` file.

3.2.4.2 Person presence detection application

File `Utilities/AI_ressources/PresenceDetection/MobileNetv2_Model/train.py` contains an example script showing how to retrain the MobileNetV2 model by using transfer learning. The training script relies on the `***Person20***` dataset. Instructions on how to build the `***Person20***` dataset from the publicly-available `COCO-2014` dataset can be found in `Utilities/AI_resources/PersonDetection/MobileNetv2_Model/README.md` along with the `Utilities/AI_resources/PersonDetection/MobileNetv2_Model/create_dataset.py` Python™ script to filter COCO images. An example Python™ script to perform post-training quantization is available in `Utilities/AI_resources/PersonDetection/MobileNetv2_Model/quantize.py`. The post-training quantization is performed on a Keras model using the TFLiteConverter tool from TensorFlow™. Sample images are required to run the quantization operation. Sample images can be extracted from the model training set.

3.2.4.3 People counting application

Contact STMicroelectronics to obtain the retraining script.

3.2.5 Memory requirements

When integrating a C model generated by the STM32Cube.AI (X-CUBE-AI) tool, the following memory requirements must be considered:

- Volatile (RAM) memory requirement: memory space is required to allocate:
  - The inference working buffer (called the `activation` buffer in this document). This buffer is used during inference to store the temporary results of the intermediate layers within the Neural Network.
  - The inference input buffer (called the `nn_input` buffer in this document), which is used to hold the input data of the Neural Network.
- Non-volatile (Flash) memory requirement: memory space is required to store the table containing the weights and biases of the network model.

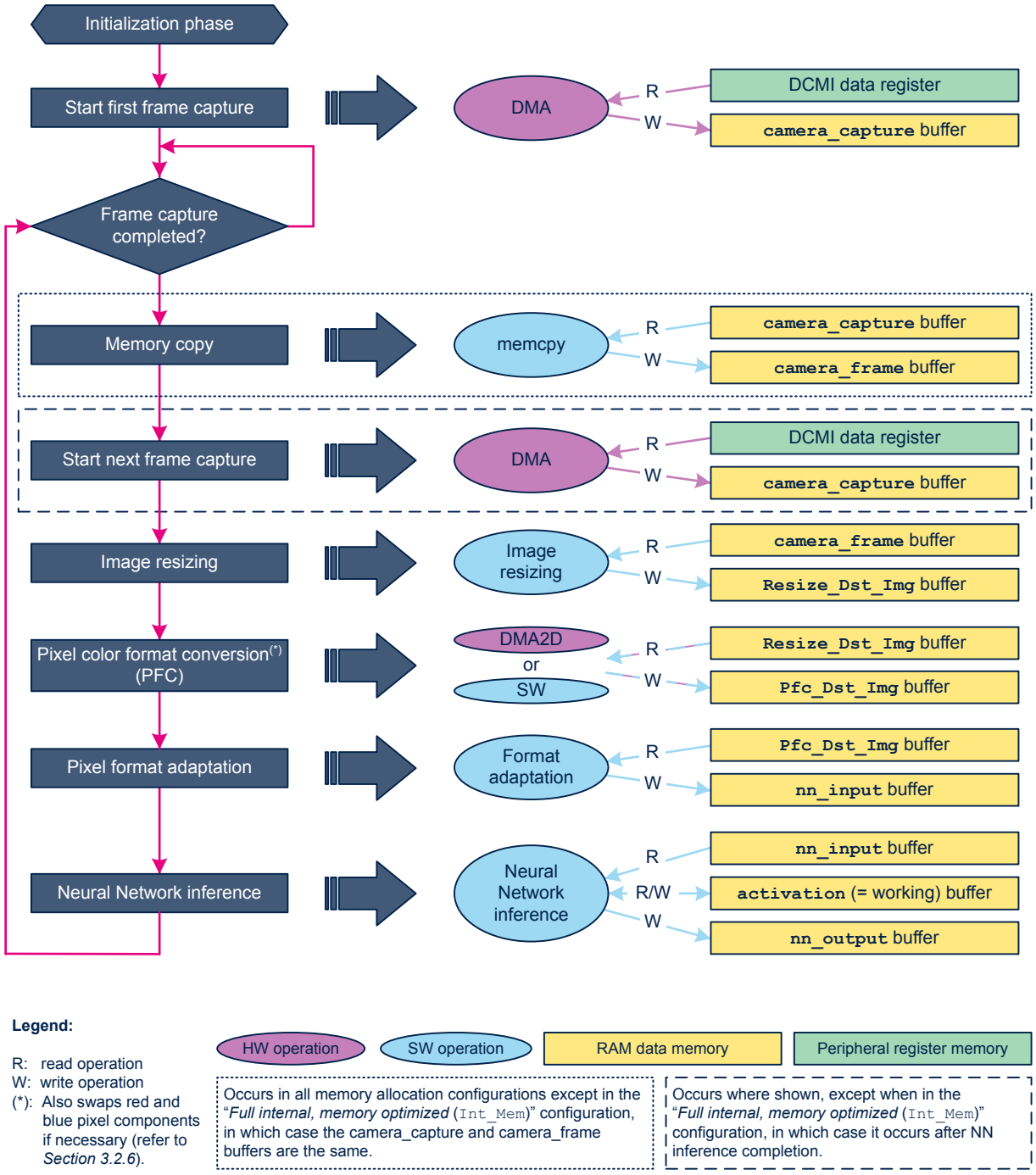
On top of the above-listed memory requirements, some more requirements come into play when integrating the C model for a computer vision application:

- Volatile (RAM) memory requirement: memory space is required in order to allocate the various buffers that are used across the execution of the image pipeline (camera capture, frame preprocessing).

3.2.5.1 *Application execution flow and volatile (RAM) data memory requirement*

In the context of a computer vision application, the integration requires several data buffers as illustrated in Figure 7. Table 3, Table 4 and Table 5 show the different data buffers required during the execution flow of the various applications.

Figure 7. Data buffers during execution flow



The application executes the following operations in sequence:

1. Acquisition of the camera frame (via the DMA engine from DCMI data register) in the `camera_capture` buffer. After acquisition completion, the content of the `camera_capture` buffer is copied into the LCD frame buffer (not shown in Table 3, Table 4 and Table 5, and always located in the external SDRAM) via the DMA2D engine with the transformation from the RGB565 capture format to the ARGB8888 display format.
2. At this point, depending on the memory allocation configuration chosen, the `camera_capture` buffer content is copied onto the `camera_frame` buffer and the capture of the subsequent frame is launched.
3. Rescale the image contained in the `camera_frame` buffer into the `Resize_Dst_Img` buffer to match the expected CNN input tensor dimensions. For instance, the food recognition NN model requires an input tensor such as `Height` × `Width` = 224 × 224 pixels.
4. Perform pixel color format conversion and red blue color channel swapping (refer to Section 3.2.7) from the `Resize_Dst_Img` buffer into the `Pfc_Dst_Img` buffer. These operations can be performed using either the DMA2D hardware engine or software routine. For instance, in the food recognition example, the RGB565 capture format is converted to the RGB888 format to provide the three input channels expected by the food recognition CNN model.
5. Adapt the format of each pixel contained in the `Pfc_Dst_Img` buffer content into the `nn_input` buffer. The adaptation consists in changing the representation of each pixel to fit with the range defined by the NN model training and the quantization format expected by a quantized NN.

   Example: if the food recognition float model is trained with input data normalized between 0 and 1, during inference, the input data (which are values comprised between 0 and 255) must be divided by 255 to fit in the range [0,1].
6. Run inference of the NN model: the `nn_input` buffer as well as the `activation` buffer are provided as input to the NN. The results of the classification are stored into the `nn_output` buffer.
7. Post-process the `nn_output` buffer content and display the results on the LCD display.

The `activation` buffer is the working buffer used by the NN during the inference. Its size depends on the NN model used.

Note: The `activation` buffer size is provided by the STM32Cube.AI tool (X-CUBE-AI) when analyzing the NN (refer to Section 3.2.5.1 for an example).

The `nn_output` buffer is where the classification output results are stored. For instance, in the food recognition NN provided in the FP-AI-VISION1 function pack, the size of this buffer is 18 × 4 = 72 bytes: 18 corresponds to the number of output classes and 4 corresponds to the fact that the probability of each output class is provided as a float value (single precision, coded on 4 bytes).

Table 3 details the amount of data RAM required by the food recognition applications when integrating the quantized C model or the float C model.

Table 3. SRAM memory buffers for food recognition applications

SRAM data buffer		SRAM data buffer size (byte)			
		Food recognition CNN			
Name	Pixel format	Quantized C model		Float C model	
		VGA capture (640 × 480)	QVGA capture (320 × 240)	VGA capture (640 × 480)	QVGA capture (320 × 240)
camera_capture	16 bits (RGB565)	600 K (640 × 480 × 2)	150 K (320 × 240 × 2)	600 K (640 × 480 × 2)	150 K (320 × 240 × 2)
camera_frame	16 bits (RGB565)	600 K (640 × 480 × 2)	150 K (320 × 240 × 2)	600 K (640 × 480 × 2)	150 K (320 × 240 × 2)
Resize_Dst_Img	16 bits (RGB565)	98 K (224 × 224 × 2)			
Pfc_Dst_Img	24 bits (RGB888)	147 K (224 × 224 × 3)			
nn_input(1)	24 bits (RGB888)	147 K (224 × 224 × 3)		588 K (224 × 224 × 3 × 4)	
activation(1)	-	98 K		395 K	
nn_output	-	72 (18 × 4)		72 (18 × 4)	

1. *When generating the C code with the STM32Cube.AI tool: if the "allocate input in activation" option is selected (either via the `--allocate-inputs` option in the CLI or via the "Use activation buffer for input buffer" checkbox in the advanced settings of the GUI), STM32Cube.AI overlays the `nn_input` buffer with the `activation` buffer; it might result in a bigger activation size. However in this case the `nn_input` buffer is not required any longer. In the end, the overall amount of memory required is reduced. In the given food recognition quantized C model, when the "allocate input in activation" is selected, the size of the generated `activation` buffer is ~148 Kbytes, which is bigger than 98 Kbytes but lower than 245 Kbytes (98 + 147). Regarding the given food recognition float C model: the "allocate input in activation" option is not selected since the `nn_input` buffer (which size is equal to 588 Kbytes) does not fit in internal SRAM.*

Table 4 details the amount of data RAM required by the presence detection applications when integrating the MobileNetV2 model or the Google model.

Table 4. SRAM memory buffers for person presence detection applications

SRAM data buffer	SRAM data buffer size (byte)	
	Person presence detection CNN	
Name	MobileNetV2 model	Google model
	QVGA capture (320 × 240)	
camera_capture	150 K (320 × 240, 16-bit RGB565)	150 K (320 × 240, 16-bit RGB565)
camera_frame	150 K (320 × 240, 16-bit RGB565)	150 K (320 × 240, 16-bit RGB565)
Resize_Dst_Img	32 K (128 × 128, 16-bit RGB565)	18 K (96 × 96, 16-bit RGB565)
Pfc_Dst_Img	48 K (128 × 128, 24-bit RGB888)	9 K (96 × 96, 8-bit grayscale)
nn_input(1)	48 K (128 × 128, 24-bit RGB888)	9 K (96 × 96, 8-bit grayscale)
activation(1)	197 K	37 K
nn_output	8 (2 × 4)	8 (2 × 4)

1. *With the `--allocate-inputs` option selected when generating the C model using the STM32Cube.AI tool. As a consequence, the `nn_input` buffer and the `activation` buffer are overlaid in a memory area of size 197 Kbytes.*

Table 5 details the amount of data RAM required by the people counting application.

Table 5. SRAM memory buffers for people counting application

SRAM data buffer	SRAM data buffer size (byte)
Name	People counting application (QVGA 320 × 240)
camera_capture	150 K (320 × 240, 16-bit RGB565)
camera_frame	150 K (320 × 240, 16-bit RGB565)
Resize_Dst_Img	112.5 K (240 × 240, 16-bit RGB565)
Pfc_Dst_Img	168.75 K (240 × 240, 24-bit RGB888)
nn_input(1)	168.75 K (240 × 240, 24-bit RGB888)
activation(1)	233 K
nn_output	26 K (15 × 15 × 30 × 4)

1. *With the `--allocate-inputs` option selected when generating the C model using the STM32Cube.AI tool. As a consequence, the `nn_input` buffer and the `activation` buffer are overlaid in a memory area of size 233 Kbytes.*

3.2.5.2 *STM32H747 internal SRAM*

Table 6 represents the internal SRAM memory map of the STM32H747XIH6 microcontroller:

Table 6. STM32H747XIH6 SRAM memory map

Memory	Address range	Size (Kbyte)
DTCM RAM	0x2000 0000 - 0x2001 FFFF	128
Reserved	0x2002 0000 - 0x23FF FFFF	-
AXI SRAM	0x2400 0000 - 0x2407 FFFF	512
Reserved	0x2400 8000 - 0x2FFF FFFF	-
SRAM1	0x3000 0000 - 0x3001 FFFF	128
SRAM2	0x3002 0000 - 0x3003 FFFF	128
SRAM3	0x3004 0000 - 0x3004 7FFF	32
Reserved	0x3004 8000 - 0x37FF FFFF	-
SRAM4	0x3800 0000 - 0x3800 FFFF	64
Reserved	0x3801 0000 - 0x387F FFFF	-
BCKUP SRAM	0x3880 0000 - 0x3880 0FFF	4
Total	-	**1 Mbyte**

The STM32H747XIH6 features about 1 Mbyte of internal SRAM. However, it is important to note that:

- This 1 Mbyte is not a contiguous memory space. The largest block of SRAM is the AXI SRAM, which has a size of 512 Kbytes.
- The `nn_input` buffer must be placed in a continuous area of memory. This is the same for the `activation` buffer. As a result, the basic following rule applies: if either the `nn_input` buffer or the `activation` buffer is larger than 512 Kbytes, the generated C model is not able to execute by relying only on the internal SRAM. In this case, additional external data RAM is required.

3.2.5.3 *Buffer placement in volatile (RAM) data memory*

The comparison of the sizes in Table 3, Table 4, Table 5 and Table 6 leads to the following conclusions regarding the integration of the C models for the different application examples provided in the FP-AI-VISION1 FP:

- Regarding the food recognition application, the float C model implementation does not fit completely into the internal SRAM, whatever the camera resolution selected: some external RAM is required in order to run this use case.

  Concerning the implementation of the quantized C model:

  – If the VGA camera resolution is selected, the implementation does not fit completely into the internal SRAM. Some external RAM is required to run this use case.

  – If the QVGA camera resolution is selected, the implementation fits completely into the internal SRAM. No external RAM is required.
- Regarding the person presence detection application, all the example implementations provided fit completely into the internal SRAM. No external RAM is required.

In the context of the FP-AI-VISION1 FP, two memory allocation configurations are supported for the implementations that do not entirely fit into the STM32 internal SRAM:

- *Full External* (Ext) memory allocation configuration: consists in placing all the buffers (listed in Table 3) in the external SDRAM.
- *Split External/Internal* (Split) memory allocation configuration: consists in placing the `activation` buffer (which includes the `nn_input` buffer if the *allocate input in activation* option in selected in the STM32Cube.AI (X-CUBE-AI) tool) as well as the `Resize_Dst_Img` and `Pfc_Dst_Img` buffers (since in the current version of the FP, both buffers are by design overlaid with the `activation` buffer) in the internal SRAM and the `camera_capture` and `camera_frame` buffers in the external SDRAM. The `nn_input` buffer is also placed in the external SDRAM (like in the food recognition float C model for instance), unless its size is such that it enables the `nn_input` buffer to be overlaid with the `activation` buffer (via the selection of the *allocate input in activation* option).

Choosing one configuration or the other comes down to the following trade-off:

- Release as much internal SRAM as possible so that it is available for other user applications possibly requiring a significant amount of internal SRAM
- Reduce the inference time by having the working buffer (`activation` buffer) allocated in faster memory such as the internal SRAM

In the context of the FP-AI-VISION1 FP, two memory allocation configurations are supported for the implementations that do entirely fit into the STM32 internal SRAM:

- *Full internal, memory optimized* (Int_Mem) memory allocation configuration: consists in placing all the buffers listed in Table 3, Table 4 or Table 5 in internal memory. The placement is such that it enables the SRAM occupation optimization.
- *Full internal, frame per second optimized* (Int_Fps) memory allocation configuration: consists in placing all the buffers listed in Table 3, Table 4 or Table 5 in internal memory. The placement is such that it enables the optimization of the number of frames processed per second.

Section 3.2.5.4 Optimizing the internal SRAM memory space describes in details the *Full internal, memory optimized* and *Full internal, frame per second optimized* memory allocation configurations.

3.2.5.4 *Optimizing the internal SRAM memory space*

For the use case fitting integrally in the internal SRAM, two memory allocation schemes are supported with a view to optimizing the internal SRAM memory space as much as possible.

These two memory allocation schemes rely on the *allocate input in activation* feature offered by the STM32Cube.AI tool (X-CUBE-AI) with a view to optimize the space required for the `activation` and `nn_input` buffers: basically, the `nn_input` buffer is allocated within the `activation` buffer, making both buffer *"overlaid"*. (Refer to [1] for more information on how to enable these optimizations when generating the Neural Network C code.)

First memory allocation scheme: *Full internal, memory optimized* (Int_Mem)

A single and unique physical memory space is allocated for all the buffers: `camera_capture`, `camera_frame`, `Resize_Dst_Img`, `Pfc_Dst_Img` , `nn_input`, `nn_output` and `activation` buffers. In other words, these seven buffers are *"overlaid"*, which means that a unique physical memory space is used for seven different purposes during the application execution flow.

The size of this memory space is equal to the size of the largest among the `activation` buffer and the `camera_frame` buffer. For instance, in the food recognition quantized model in QVGA, the size of this memory space is equal to the size of the `camera_frame` buffer amounting to 150 Kbytes.

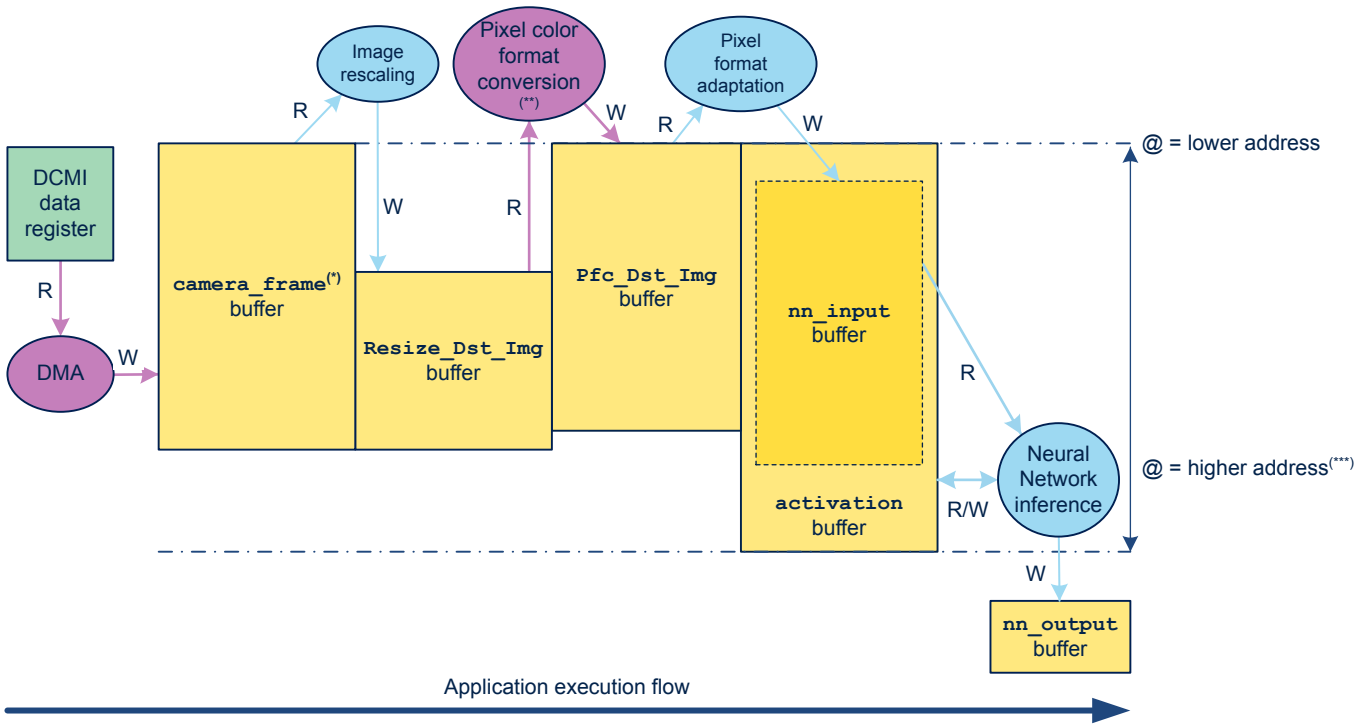
The advantage of this approach is that the required memory space is optimized at best.

This approach has also two main drawbacks:

- The data stored in the memory space are overwritten (and thus no longer available) as the application is moving forward along the execution flow.
- The memory space is not available for a new camera capture until the NN inference is completed. As a consequence, the maximum number of frames that can be processed per second is not optimized.

The whole amount of memory required is allocated in a single region as illustrated in Figure 8.

Figure 8. SRAM allocation - Memory optimized scheme



Notes:

(*): When in *Full internal, memory optimized* allocation configuration, the **camera_capture** and **camera_frame** buffers are one single and unique buffer since the subsequent capture starts only after the NN inference is completed.
(**): The pixel color format conversion can be performed either in hardware via the DMA2D or in software.
(***): higher address = lower address + Max(**activation** buffer size, **camera_frame** buffer size)

Legend:

R: read operation
W: write operation

Note: For the food recognition application (quantized model and QVGA capture), by enabling the memory optimization feature in the STM32Cube.AI tool ("allocate input in activation"), only 148 Kbytes of memory are required to hold both the `activation` and the `nn_input` buffers (versus 147 Kbytes + 98 Kbytes = 245 Kbytes if this feature is not enabled).

Second memory allocation scheme: *Full internal, frame per second optimized* (Int_Fps)

- A first physical memory space is allocated only for the `camera_capture` buffer. The advantage of this approach is that a subsequent frame can be captured as soon as the content of the `camera_capture` buffer is copied onto the `camera_frame` buffer without waiting for the inference to complete. As a result, the number of frames processed per second is optimal.

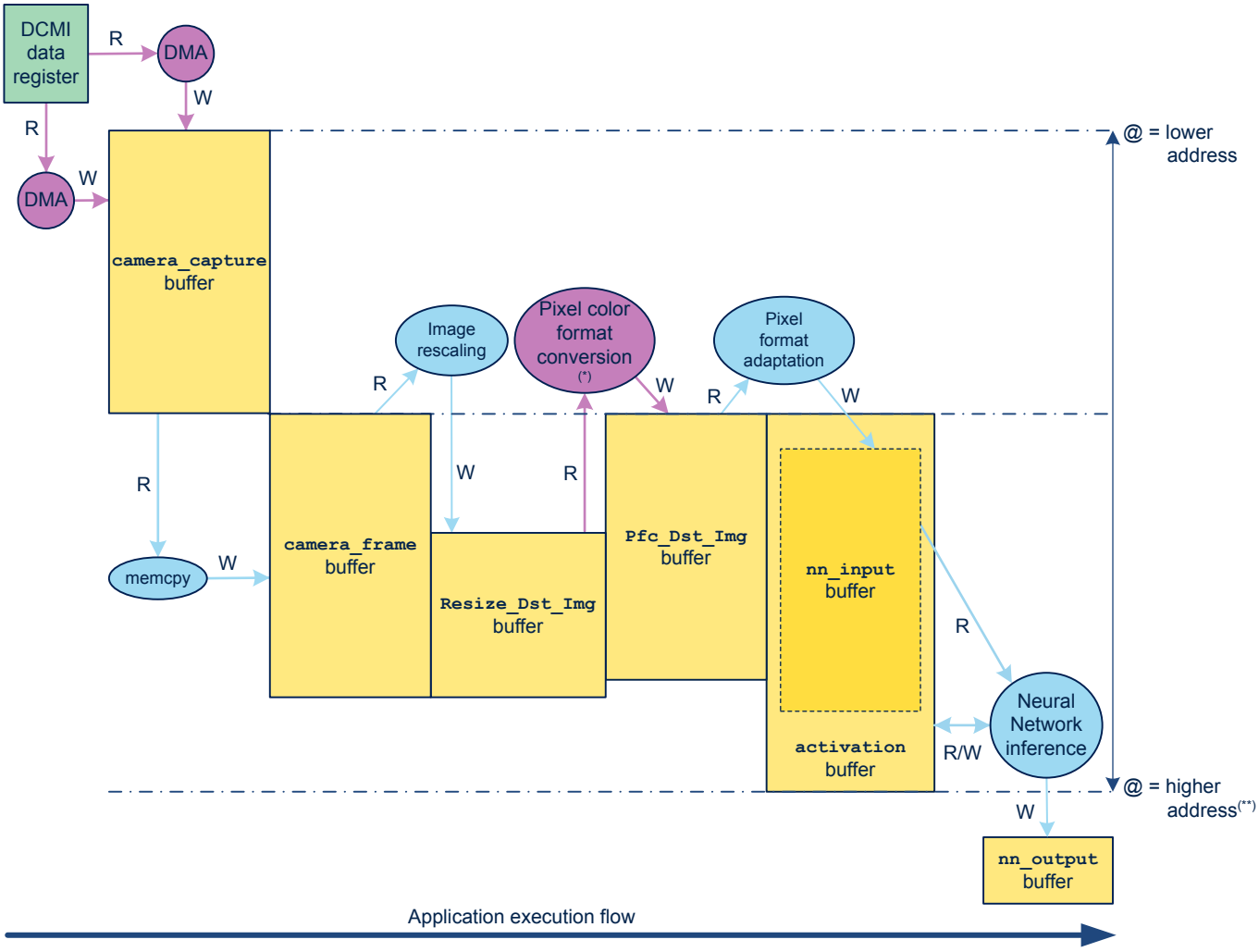
  This is at the expense of the amount of memory space required.
- A second unique physical memory space is allocated for all the other buffers: `camera_frame`, `Resize_Dst_Img`, `Pfc_Dst_Img`, `nn_input`, `nn_output` and `activation` buffers. In other words, these six buffers are *"overlaid"*, which means that a unique physical memory space is used for six different purposes during the application execution flow.

  The size of this memory space is equal to the size of the biggest of the five *"overlaid"* buffers: in our case, it is the largest among the `activation` and `camera_frame` buffers. For instance, in the food recognition quantized model in QVGA, the size of this memory space is equal to the size of the `camera_frame` buffer, amounting to 150 Kbytes.

Conclusion: For the food recognition application (quantized model and QVGA capture), the overall memory size required by the second allocation scheme is 150 Kbytes + 150 Kbytes = 300 Kbytes. The amount of memory required by the second scheme is higher than the one in the first scheme but the second scheme enables to optimize the number of frames processed per second.

The whole amount of memory required is allocated in a single region as illustrated in Figure 9.

Figure 9. SRAM allocation - FPS optimized scheme



Notes:

(*): The pixel color format conversion can be performed either in hardware via the DMA2D or in software.
(**): higher address = lower address + **camera_frame** buffer size + Max(**activation** buffer size, **camera_frame** buffer size)

Legend:

R: read operation
W: write operation

HW operation | SW operation | Peripheral register memory | SRAM data memory | **nn_input** allocated in activation

Note: For the food recognition application (quantized model and QVGA capture), by enabling the memory optimization feature in the STM32Cube.AI tool ("allocate input in activation"), only 148 Kbytes of memory are required to hold both the `activation` and the `nn_input` buffers (versus 147 Kbytes + 98 Kbytes = 245 Kbytes if this feature is not enabled).

3.2.5.5 *Weight and bias placement in non-volatile (Flash) memory*

The STM32Cube.AI (X-CUBE-AI) code generator generates a table (defined as constant) containing the weights and biases. Therefore, by default, the weights and biases are stored into the internal Flash memory.

There are use cases where the table of weights and biases does not fit into the internal Flash memory (the size of which being 2 Mbytes, shared between read-only data and code). In such situation, the user has the possibility to store the weight-and-bias table in the external Flash memory.

The STM32H747I-DISCO Discovery board supported in the context of this FP has a serial external Flash that is interfaced to the STM32H747I MCU via a Quad-SPI. An example illustrating how to place the weight-and-bias table in the external Flash memory is provided in this FP under `Projects/STM32H747I-DISCO/Applications/FoodReco_MobileNetDerivative/Float_Model` directory (configurations `STM32H747I-DISCO_FoodReco_Float_Ext_Qspi`, `STM32H747I-DISCO_FoodReco_Float_Split_Qspi`, `STM32H747I-DISCO_FoodReco_Float_Split_Sdram` and `STM32H747I-DISCO_FoodReco_Float_Ext_Sdram`).

Follow the steps below to load the weight-and-bias table in the Q-SPI external Flash memory:

1. Define flags `WEIGHT_QSPI` to 1 and `WEIGHT_QSPI_PROGED` to 0 to define a memory placement section in the memory range corresponding to the Quad-SPI memory interface (0x9000 0000) and place the table of weights and biases into it.

   In order to speed up the inference, it is possible for the user to set flag `WEIGHT_EXEC_EXTRAM` to 1 (as demonstrated for instance in configurations `STM32H747I-DISCO_FoodReco_Float_Split_Sdram` and `STM32H747I-DISCO_FoodReco_Float_Ext_Sdram`) so that the weight-and-bias table gets copied from the external Q-SPI Flash memory into the external SDRAM memory at program initialization.

2. Generate a binary of the whole application as an `.hex` file (not as a `.bin` file). Load the `.hex` file using the STM32CubeProgrammer (STM32CubeProg) tool [5]. Select first the external Flash loader for the STM32H747I-DISCO Discovery board. As shown in Figure 10, the user must select the *External loaders* tab on the left to access the *External loaders* view. After selecting the right Flash loader, the user must select the *Erasing & programming* tab on the left to access the *Erasing & programming* view shown on Figure 11.

   Figure 10 and Figure 11 are snapshots of the STM32CubeProgrammer (STM32CubeProg) tool showing the sequence of steps to program the full binary of the application into the Flash memory.

Figure 10. Flash programming (1 of 2)

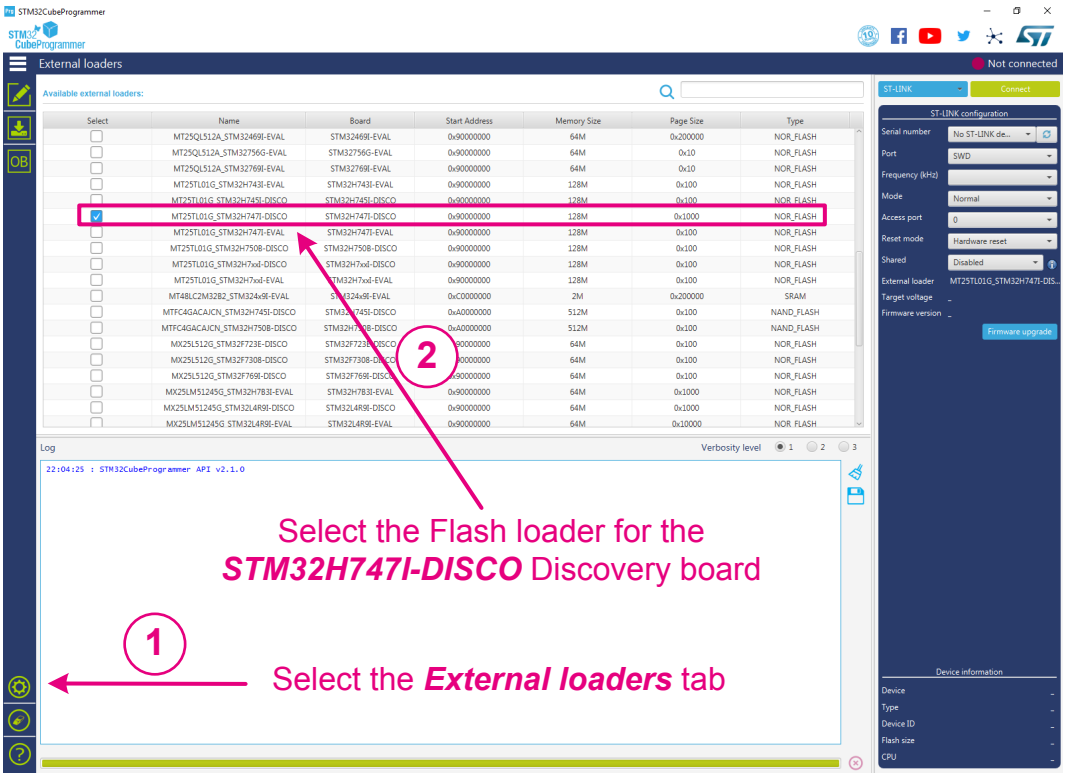
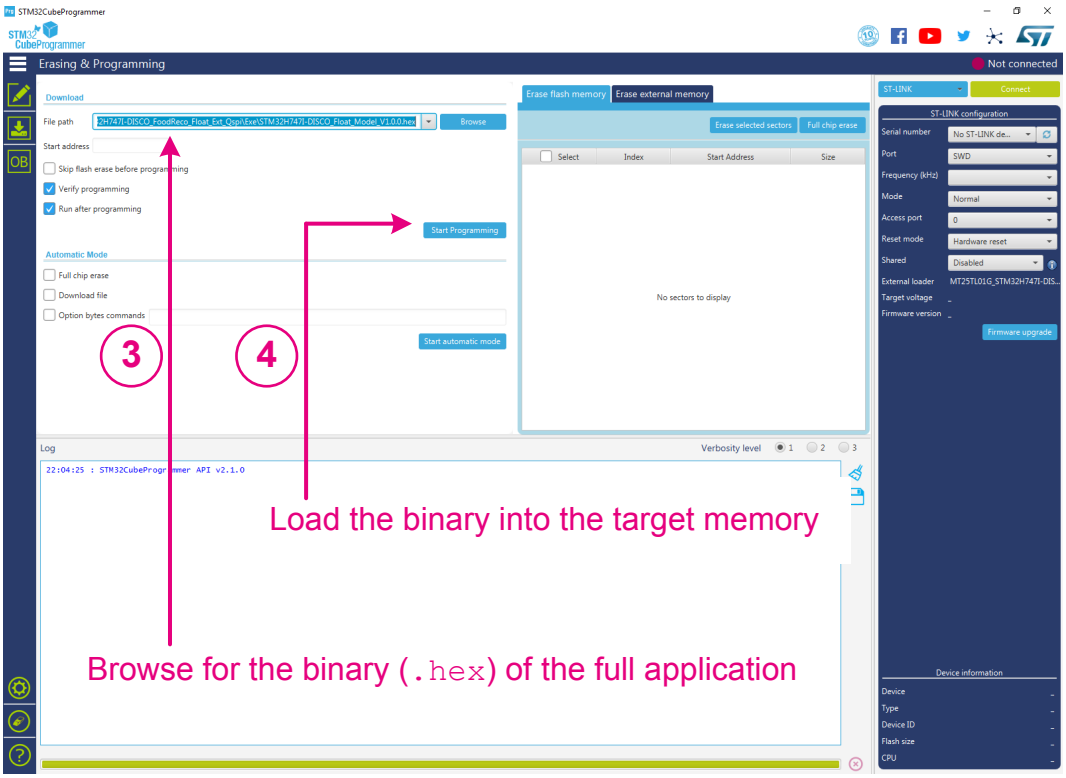


Figure 11. **Flash programming (2 of 2)**



3. Once the table of weights and biases is loaded into the external memory, it is possible to continue the debugging through the IDE by defining flag `WEIGHT_QSPI_PROGED` to 1

   Defining flag `WEIGHT_QSPI` to 0 generates a program with the weight-and-bias table located in the internal Flash memory.

Table 7 summarizes the combinations of the compile flags.

Table 7. **Compile flags**

Effect		Compile flag		
		WEIGHT_QSPI	WEIGHT_QSPI_PROGED	WEIGHT_EXEC_EXTRAM
Generates a binary with the weight-and-bias table placed in the internal Flash memory		0	0	0
		0	0	1
		0	1	0
		0	1	1
Generates a binary with the weight-and-bias table placed in the external Q-SPI Flash memory(1)	-	1	0	0
	Weight-and-bias table copied from the external Q-SPI Flash memory into the external SDRAM at startup	1	0	1
Generates a binary without the weight-and-bias table (since already loaded)	-	1	1	0
	Weight-and-bias table copied from the external Q-SPI Flash memory into the external SDRAM at startup	1	1	1

1. *To be loaded using the STM32CubeProgrammer tool (STM32CubeProg).*

Refer to Section 3.2.8 Applications performance for the performance when the weight-and-bias table is accessed from the external Q-SPI Flash memory and external SDRAM versus internal Flash memory.

When generating new C code from a new model, the usual way to do (as described in Section 2.1 Integration of the generated code) is to replace the existing `network.c`, `network.h`, `network_data.c` and `network_data.h` files by the ones generated. If placement in external memory is required, the existing `network_data.c` file must not be replaced. Instead, it is requested to replace only the content of the weight-and-bias table (named `s_network_weights[]`) contained in the existing `network_data.c` file by the content of the table contained in the generated `network_data.c` file.

3.2.5.6 *Summary of volatile and non-volatile data placement in memory*

The FP-AI-VISION1 function pack demonstrates the implementation of four different schemes for allocating the volatile data in memory:

- Full internal memory with FPS optimized (Int_Fps):

  All the buffers (listed in Table 3, Table 4 and Table 5) are located in the internal SRAM. To enable the whole system to execute from the internal memory, some memory buffers are overlaid (as shown in Figure 9).

  In this memory layout scheme, the `camera_capture` buffer is not overlaid so that it enables a new camera capture while the inference is running, hence maximizing the number of frames per second (FPS).
- Full internal memory with memory optimized (Int_Mem):

  All the buffers (listed in Table 3, Table 4 and Table 5) are located in the internal SRAM. In order to enable the whole system to execute from the internal memory, some memory buffers are overlaid (as shown in Figure 8).

  In this memory layout scheme, the `camera_capture` and `camera_frame` buffers are one and single unique buffer, which is overlaid so that it optimizes the internal SRAM occupation as much as possible.
- Full external memory (Ext):

  All the buffers (listed in Table 3, Table 4 and Table 5) are located in the external SDRAM.
- Split internal / external memory (Split):

  The `camera_capture` and `camera_frame` buffers are located in the external SDRAM. The `activation` buffer (which includes the `nn_input` buffer if the *allocate input in activation* option is selected in the STM32Cube.AI (X-CUBE-AI) tool) as well as the `Resize_Dst_Img` and `Pfc_Dst_Img` buffers (since in the current version of the FP, both buffers are by design overlaid with the `activation` buffer), are located in the internal SRAM. In this allocation scheme, the `nn_input` buffer is also placed in the external SDRAM (like in the food recognition float C model for instance), unless its size is such that it enables the `nn_input` buffer to be overlaid with the `activation` buffer (via the selection of the *allocate input in activation* option).

The FP-AI-VISION1 function pack also demonstrates the implementation of three different ways to access (at inference time) the non-volatile data stored in memory:

- Access from the internal Flash memory
- Access from the external Q-SPI Flash memory
- Access from the external SDRAM: in this case the non-volatile data are stored either in the internal Flash memory or in the external Q-SPI Flash memory and are copied into the external SDRAM at program startup

Table 8 lists the IAR Embedded Workbench® projects available in FP-AI-VISION1 with the various configurations.

Table 8. Summary of IAR Embedded Workbench® project configurations versus memory schemes

IAR Embedded Workbench® project configuration name	Memory allocation scheme for volatile data	Memory used to store the weights and biases
STM32H747I-DISCO_FoodReco_Float_Ext	Full external	Internal Flash memory
STM32H747I-DISCO_FoodReco_Float_Ext_Qspi		External Q-SPI Flash memory
STM32H747I-DISCO_FoodReco_Float_Ext_Sdram		External SDRAM
STM32H747I-DISCO_FoodReco_Float_Split	Split internal / external	Internal Flash memory
STM32H747I-DISCO_FoodReco_Float_Split_Qspi		External Q-SPI Flash memory
STM32H747I-DISCO_FoodReco_Float_Split_Sdram		External SDRAM
STM32H747I-DISCO_FoodReco_Quantized_Ext	Full external	Internal Flash memory
STM32H747I-DISCO_FoodReco_Quantized_Split	Split internal / external	
STM32H747I-DISCO_FoodReco_Quantized_Int_Mem	Full internal with memory optimized	
STM32H747I-DISCO_FoodReco_Quantized_Int_Fps	Full internal with FPS optimized	
STM32H747I-DISCO_PersonDetect_Google		
STM32H747I-DISCO_PersonDetect_MobileNetv2		
STM32H747I-DISCO_PeopleCounting		

3.2.6 Camera pixel clock

3.2.6.1 *USB webcam application*

To comply with the FCC certification, the PCLK camera pixel clock of the B-CAMS-OMV camera module bundle (OV5640 camera) is limited to 12 MHz. With PCLK = 12 MHz, the OV5640 camera output frame rate is 7.5 fps for both the VGA and QVGA resolutions.

To reach a higher FPS, the pixel clock can be increased using `CAMERA_BOOST_PCLK = 1`. This increases the OV5640 camera pixel clock as follows:

- 24 MHz for the VGA resolution to reach 15 fps
- 48 MHz for the QVGA resolution to reach 30 fps

Note: *`CAMERA_BOOST_PCLK` has no effect when using the STM32F4DIS-CAM (OV9655) camera.*

For the USB webcam application, to reach the frame rates defined in the USB descriptor, the OV5640 pixel clock must be increased to 24 MHz, or even 48 MHz depending on resolution. The `CAMERA_BOOST_PCLK` preprocessor define can be enabled in the compiler settings or directly in file `stm32h747i_discovery_camera_ex.c`.

```
/*
 Set CAMERA_BOOST_PCLK=1 to operate the OV5640 camera in optimal mode.
 Note: when operating the camera in optimal mode, the pixel frequency is
 above 12MHz and FCC certification is no longer guaranteed.
*/

#ifndef CAMERA_BOOST_PCLK
#define CAMERA_BOOST_PCLK   1
#endif
```

Statement about terms and conditions

By setting the `CAMERA_BOOST_PCLK` compiler flag to 1, users change the default setting of the kit provided by STMicroelectronics. With such new setting, the users are reminded that they shall comply with the following terms and conditions:

Notice applicable to evaluation boards not FCC-approved.

This kit is designed to allow: (1) Product developers to evaluate electronic components, circuitry, or software associated with the kit to determine whether to incorporate such items in a finished product, and (2) Software developers to write software applications for use with the end product. This kit is not a finished product and when assembled may not be resold or otherwise marketed unless all required FCC equipment authorizations are first obtained. Operation is subject to the condition that this product not cause harmful interference to licensed radio stations and that this product accept harmful interference. Unless the assembled kit is designed to operate under part 15, part 18 or part 95 of 47 CFR, Chapter I ("FCC Rules"), the operator of the kit must operate under the authority of an FCC license holder or must secure an experimental authorization under part 5 of this chapter.

3.2.6.2 *Other applications*

For the other applications (person presence detection, food recognition and people counting), the OV5640 camera pixel clock is set to 48 MHz by default. This enables a rate of 30 fps for both the VGA and QVGA camera capture modes.

3.2.7 Pixel data order

Depending on the operating mode (refer to Section 3.2.12 Embedded validation, capture and testing), the input data image comes either from the camera sensor or from a `.bmp` file stored onto the microSD™ present on the STM32H747I-DISCO board.

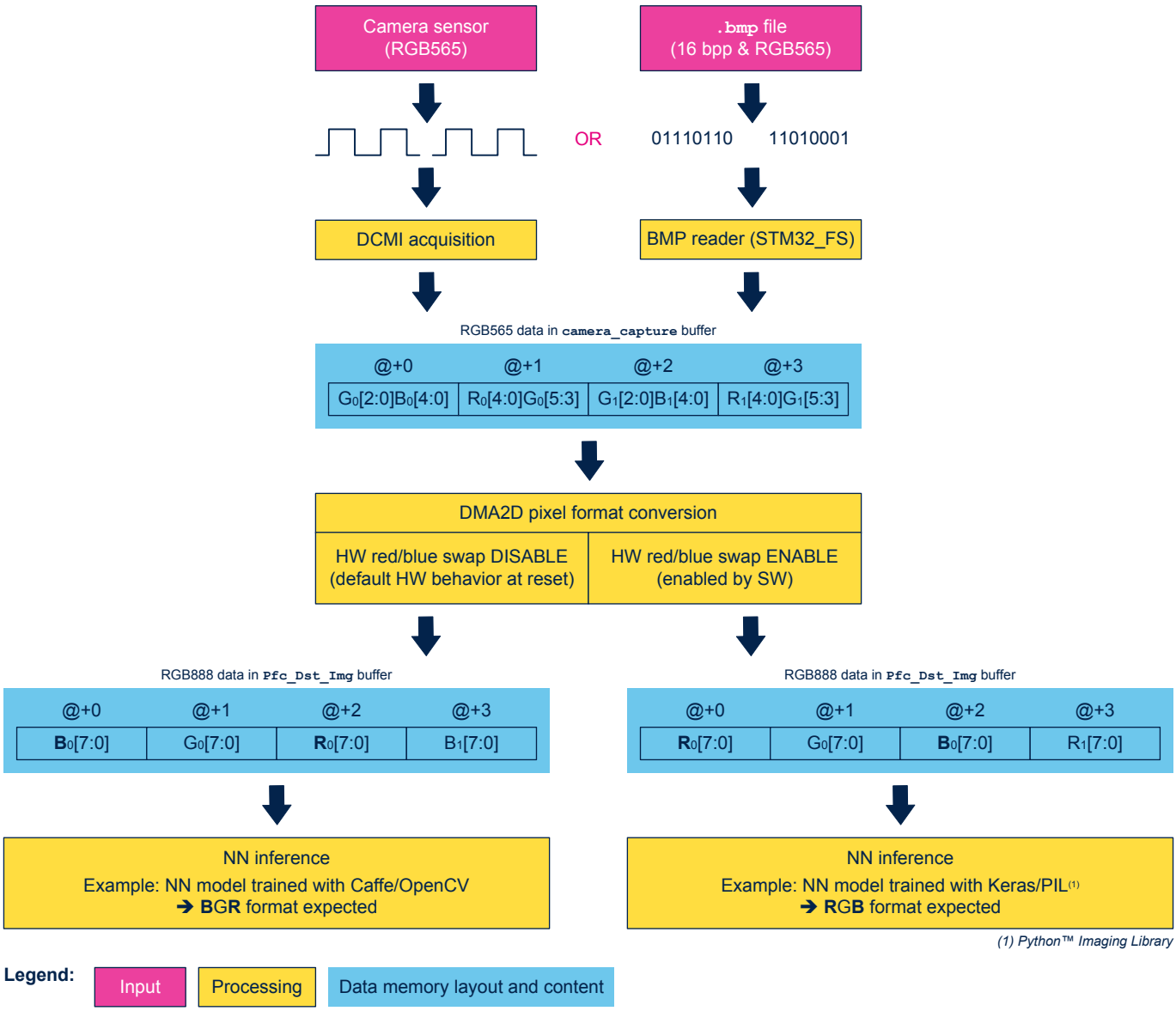
The camera sensor is configured to output data in the RGB565 format. During camera acquisition, the bytes are transmitted via the DCMI interface in the following order: the first byte G[2:0]B[4:0] is transmitted first, followed by the second byte R[4:0]G[5:3].

The `.bmp` file is expected to contain 16-bit-per-pixel data stored in the RGB565 format. In such case, when the input data image is read from the `.bmp` file, byte G[2:0]B[4:0] is stored first in memory and byte R[4:0]G[5:3] is stored in the next memory byte.

As a result, whether the input data image is coming from the camera sensor or from a `.bmp` file, the data memory layout in the `camera_capture` buffer is exactly the same.

Figure 12 depicts the pixel data memory layout after each stage.

Figure 12. Pixel order and data memory layout



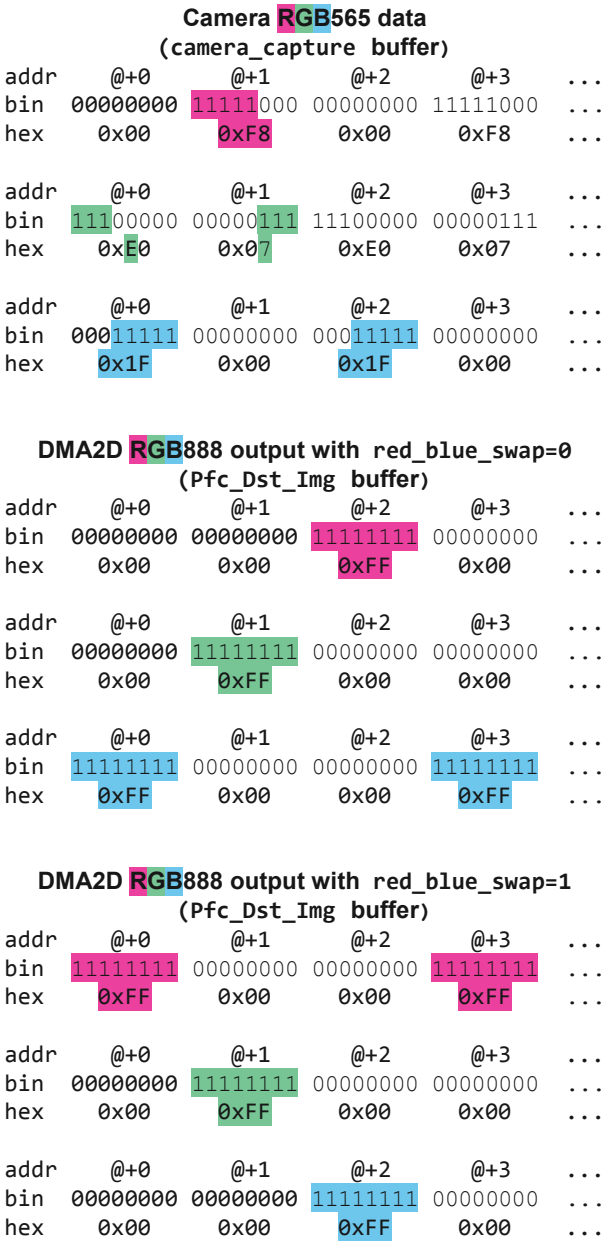
The food recognition NN models as well as the MobileNet-V2 person presence detection NN model are all trained in such a way that they expect the input data tensor to be stored in memory in the RGB format.

Consequently, a swap between the red and the blue component is required at some point. In the function pack firmware, the swap is performed by the DMA2D during the pixel format conversion (PFC) operation.

Upon hardware reset, and when configured to perform PFC, the DMA2D engine stores the blue component first in memory as shown in Figure 12. If it is required to have the red component stored first in memory, the DMA2D must be configured so to perform a red/blue component swap: for this purpose the user must set the variable `red_blue_swap` (which is part of the `PreprocContext`) to 1 before starting the DMA2D pixel format conversion operation (see function `Run_Preprocessing()` in file `ai_utilities.c`).

Figure 13 provides memory content examples for three different pixels: one red, one green and one blue.

Figure 13. Pixel order and data memory layout examples



3.2.8 Applications performance

3.2.8.1 *Preprocessing timings*

Table 9, Table 10 and Table 11 present the camera capture time and preprocessing time measured for each of the supported applications.

Table 9. Measurements of frame capture and preprocessing times for the food recognition application examples

Measurements(1) (ms)	Food recognition		
	VGA capture (640 × 480)		QVGA capture (320 × 240)
	Quantized C model	Float C model	Quantized C model
Camera frame capture time (2) (3)	~35		~35
Image rescaling time(4)	~5 to ~7		~5
Pixel color conversion(5)	< 1 to ~2		< 1
Pixel format adaptation time(6)	~1 to ~3	~9 to ~11	~1

1. *Measurement conditions: STM32H747 at 400 MHz CPU clock, code compiled with EWARM v8.50.6 with option `-O3`.*
2. *The values depend on the lighting conditions since the exposure time may vary.*
3. *Values for the B-CAMS-OMV camera module bundle connected with the STM32H747I-DISCO Discovery board.*
4. *In the FP-AI-VISION1, the "Nearest Neighbor" algorithm is used as the rescaling method. The memory locations of the `camera_capture` and `Resize_Dest_Img` buffers have an impact on the timing.*
5. *The memory locations of the `Resize_Dest_Img` and `Pfc_Dest_Img` buffers have an impact on the timing.*
6. *A pre-computed look-up table is used for the pixel format adaptation when dealing with the quantized model. The memory locations of the `Pfc_Dest_Img` and `nn_input` buffers have an impact on the timing.*

Table 10. Measurements of frame capture and preprocessing times for the person presence detection application examples

Measurements(1) (ms)	Person presence detection QVGA capture (320 × 240)	
	MobileNetV2 model (128 × 128, 24-bit RGB888)	Google model (96 × 96, 8-bit grayscale)
Camera frame capture time(2) (3)	~35	
Image rescaling time(4)	~2	~1
Pixel color conversion(5)	< 1	
Pixel format adaptation time(6)	< 1	

1. *Measurement conditions: STM32H747 at 400 MHz CPU clock, code compiled with EWARM v8.50.6 with option `-O3`.*
2. *The values depend on the lighting conditions since the exposure time may vary.*
3. *Values for the B-CAMS-OMV camera module bundle connected with the STM32H747I-DISCO Discovery board.*
4. *In the FP-AI-VISION1, the "Nearest Neighbor" algorithm is used as the rescaling method.*
5. *The memory locations of the `Resize_Dest_Img` and `Pfc_Dest_Img` buffers have an impact on the timing.*
6. *A pre-computed look-up table is used for the pixel format adaptation when dealing with the quantized model.*

Table 11. Measurements of frame capture and preprocessing times for the people counting application example

Measurements[1] (ms)	People counting: 240 × 240 × 3, 24-bit RGB888 QVGA capture (320 × 240)
Camera frame capture time[2] [3]	~35
Image rescaling time[4]	~5
Pixel color conversion	< 1
Pixel format adaptation time[5]	< 1

1. *Measurement conditions: STM32H747 at 400 MHz CPU clock, code compiled with EWARM v8.50.6 with option -O3.*
2. *The values depend on the lighting conditions since the exposure time may vary.*
3. *Values for the B-CAMS-OMV camera module bundle connected with the STM32H747I-DISCO Discovery board.*
4. *In the FP-AI-VISION1, the "Nearest Neighbor" algorithm is used as the rescaling method.*
5. *A pre-computed look-up table is used for the pixel format adaptation when dealing with the quantized model.*

3.2.8.2 *Configurations supported*

Considering the different possible models (float and quantized), the different possible camera resolutions (VGA and QVGA), the different possible locations for data in volatile memory (full external, split, full internal FPS optimized, full internal memory optimized), and the different possible locations of the weight-and-bias table during inference execution (internal Flash memory, external Q-SPI Flash memory, external SDRAM), the number of possible configuration is high.

Table 12, Table 13 and Table 14 summarize the configurations supported in the FP-AI-VISION1 function pack for each of the supported applications.

Table 12. Configurations supported by the FP-AI-VISION1 function pack for the food recognition applications

Volatile memory layout scheme	Weight-and-bias table location	Food recognition			
		Quantized C model		Float C model	
		VGA capture (640 × 480)	QVGA capture (320 × 240)	VGA capture (640 × 480)	QVGA capture (320 × 240)
Full internal Memory optimized	Internal Flash memory	Not possible	YES	Not possible	Not possible
	External Q-SPI Flash memory		Not supported		
	External SDRAM		Not supported		
Full internal FPS optimized	Internal Flash memory	Not possible	YES	Not possible	Not possible
	External Q-SPI Flash memory		Not supported		
	External SDRAM		Not supported		
Full external	Internal Flash memory	YES	Not supported	YES	Not supported
	External Q-SPI Flash memory	Not supported		YES	
	External SDRAM	Not supported		YES	
Split external/internal	Internal Flash memory	YES	Not supported	YES	Not supported
	External Q-SPI Flash memory	Not supported		YES	
	External SDRAM	Not supported		YES	

Table 13. Configurations supported by the FP-AI-VISION1 function pack for the person presence detection applications

Volatile memory layout scheme	Weight-and-bias table location	Person presence detection	
		MobileNetV2 model	Google model
Full internal Memory optimized	Internal Flash memory	NO	NO
	External Q-SPI Flash memory		
	External SDRAM		
Full internal FPS optimized	Internal Flash memory	YES	YES
	External Q-SPI Flash memory	NO	NO
	External SDRAM		
Full external	Internal Flash memory	NO	NO
	External Q-SPI Flash memory		
	External SDRAM		
Split external/internal	Internal Flash memory	NO	NO
	External Q-SPI Flash memory		
	External SDRAM		

Table 14. Configurations supported by the FP-AI-VISION1 function pack for the people counting application

Volatile memory layout scheme	Weight-and-bias table location	People counting
Full internal Memory optimized	Internal Flash memory	NO
	External Q-SPI Flash memory	
	External SDRAM	
Full internal FPS optimized	Internal Flash memory	YES
	External Q-SPI Flash memory	NO
	External SDRAM	
Full external	Internal Flash memory	NO
	External Q-SPI Flash memory	
	External SDRAM	
Split external/internal	Internal Flash memory	NO
	External Q-SPI Flash memory	
	External SDRAM	

3.2.8.3 *Execution performance of the supported applications*

The values provided in Table 15, Table 16, Table 17 and Table 18 below result from measurements performed using application binaries built with the EWARM IDE.

Table 15 presents the execution performance of the food recognition applications with the following conditions:

- IAR Systems EWARM v8.50.6
- `-O3` option
- Cortex®-M7 core clock frequency set to 400 MHz
- B-CAMS-OMV camera module bundle

Table 15. Execution performance of the food recognition application

C model	Camera resolution	Volatile memory layout scheme	Weight-and-bias table location	NN inference time (ms)	Frames processed per second (FPS)(1)	Output accuracy (%)	Binary file(2)
Float	VGA	Full external	Internal Flash memory	242	3.7	~73	A
			Q-SPI external Flash memory	283	3.3		B
			External SDRAM	267	3.5		C
		Split	Internal Flash memory	220	4.1		D
			Q-SPI external Flash memory	261	3.6		E
			External SDRAM	243	3.9		F
Quantized(3)	VGA	Full external	Internal Flash memory	100	9.2	~72.5	G
		Split		79	11.8		H
	QVGA	Full internal Memory optimized		79	8.5		I
		Full internal FPS optimized		79	11.8		J

1. *FPS is calculated taking into account the capture, preprocessing, and inference times.*
2. *IAR Embedded Workbench® configuration > Binary file.*
3. *Food recognition CNN model quantized using the quantization tool (provided by the STM32Cube.AI tool) with the "UaUa per layer" quantization scheme.*

The FPS values provided in Table 15 are max values. The FPS values can vary depending on the light conditions. Indeed, the FPS value includes the inference time, but also the preprocessing and camera capture times (refer to Table 9). The capture time depends on the light conditions via the exposure time.

The binary files, contained in the `Binary` folder, associated with the scenarios in Table 15, are listed below:

- A: `STM32H747I-DISCO_FoodReco_Float_Ext > STM32H747I-DISCO_Food_Std_Float_Ext_IntFlash_V300.bin`
- B: `STM32H747I-DISCO_FoodReco_Float_Ext_Qspi > STM32H747I-DISCO_Food_Std_Float_Ext_QspiFlash_V300.hex`
- C: `STM32H747I-DISCO_FoodReco_Float_Ext_Sdram > STM32H747I-DISCO_Food_Std_Float_Ext_ExtSdram_V300.hex`
- D: `STM32H747I-DISCO_FoodReco_Float_Split > STM32H747I-DISCO_Food_Std_Float_Split_IntFlash_V300.bin`
- E: `STM32H747I-DISCO_FoodReco_Float_Split_Qspi > STM32H747I-DISCO_Food_Std_Float_Split_QspiFlash_V300.hex`
- F: `STM32H747I-DISCO_FoodReco_Float_Split_Sdram > STM32H747I-DISCO_Food_Std_Float_Split_ExtSdram_V300.bin`

- G: `STM32H747I-DISCO_FoodReco_Quantized_Ext > STM32H747I-DISCO_Food_Std_Quant8_Ext_IntFlash_V300.bin`
- H: `STM32H747I-DISCO_FoodReco_Quantized_Split > STM32H747I-DISCO_Food_Std_Quant8_Split_IntFlash_V300.bin`
- I: `STM32H747I-DISCO_FoodReco_Quantized_Int_Mem > STM32H747I-DISCO_Food_Std_Quant8_IntMem_IntFlash_V300.bin`
- J: `STM32H747I-DISCO_FoodReco_Quantized_Int_Fps > STM32H747I-DISCO_Food_Std_Quant8_IntFps_IntFlash_V300.bin`

Table 16 presents the execution performance of the recognition application generated from the optimized model (not provided as part of the FP-AI-VISION1 function pack. Contact STMicroelectronics for information about this specific version.) The measurement conditions are:

- IAR Systems EWARM v8.50.6
- `-O3` option
- Cortex®-M7 core clock frequency set to 400 MHz
- B-CAMS-OMV camera module bundle

Table 16. Execution performance of the optimized food recognition application

C model	Camera resolution	Volatile memory layout scheme	Weight-and-bias table location	NN inference time (ms)	Frames processed per second (FPS)(1)	Output accuracy (%)	Binary file
Float	VGA	Full external	Internal Flash memory	492	1.9	~78	K
		Split		Not possible			
Quantized(2)	VGA	Full external		220	4.3	~77.5	L
		Split		145	6.6		M
	QVGA	Full internal Memory optimized		145	5.4		N
		Full internal FPS optimized		145	6.6		O

1. *FPS is calculated taking into account the capture, preprocessing, and inference times.*
2. *Food recognition CNN model quantized using the quantization tool (provided by the STM32Cube.AI tool) with the "UaUa per layer" quantization scheme.*

The binary files, contained in the `Binary` folder, associated with the scenarios in Table 16, are listed below:

- K: `STM32H747I-DISCO_Food_Opt_Float_Ext_IntFlash_V300.bin`
- L: `STM32H747I-DISCO_Food_Opt_Quant8_Ext_IntFlash_V300.bin`
- M: `STM32H747I-DISCO_Food_Opt_Quant8_Split_IntFlash_V300.bin`
- N: `STM32H747I-DISCO_Food_Opt_Quant8_IntMem_IntFlash_V300.bin`
- O: `STM32H747I-DISCO_Food_Opt_Quant8_IntFps_IntFlash_V300.bin`

The straight model (Table 15) shows better execution performance in some cases than the optimized model (Table 16). However, this is at the expense of the accuracy, which is better when using the optimized model since in both cases the CNN working buffer (`activation` buffer) is located in the internal SRAM.

Table 17 presents the execution performance of the person presence detection applications with the following conditions:

- IAR Systems EWARM v8.50.6
- `-O3` option
- Cortex®-M7 core clock frequency set to 400 MHz
- B-CAMS-OMV camera module bundle

Table 17. Execution performance of the person presence detection applications

C model	Camera resolution	Volatile memory layout scheme	Weight-and-bias table location	NN inference time (ms)	Frames processed per second (FPS)(1)	Output accuracy (%)(2)	Binary file(3)
MobileNetV2	QVGA	Full internal	Internal Flash memory	145	6.8	92	P
Google				41	23.3	87	Q

1. *FPS is calculated taking into account the capture, preprocessing, and inference times.*
2. *Top-1 accuracy against ***Person20*** dataset.*
3. *IAR Embedded Workbench® configuration > Binary file.*

The binary files, contained in the `Binary` folder, associated with the scenarios in Table 17, are listed below:

- P: `STM32H747I-DISCO_PersonDetect_MobileNetv2 > STM32H747I-DISCO_Person_MobileNetv2_Quant8_IntFps_IntFlash_V300.bin`
- Q: `STM32H747I-DISCO_PersonDetect_Google > STM32H747I-DISCO_Person_Google_Quant8_IntFps_IntFlash_V300.bin`

CNN impact on the final accuracy

As mentioned previously in Quantization process, the type of CNN has an impact on the final accuracy. This is illustrated by the columns *Ouput accuracy* in Table 15 and Table 16:

- Table 15 contains numbers pertaining to a network which topology has been optimized in such a way to reducing the inference time. However, this is at the expense of the output accuracy (accuracy drop of about 10 %).
- Table 16 provides numbers pertaining to a network which topology has been optimized in such a way to optimize the output accuracy. As a result, Table 16 shows accuracies that are better than the ones shown in Table 15. However, this is at the expense of the inference time and the memory space required by the network (larger `activation` buffer and weight-and-bias table).

Table 17 presents the execution performance of the people counting application with the following conditions:

- IAR Systems EWARM v8.50.6
- `-O3` option
- Cortex®-M7 core clock frequency set to 400 MHz
- B-CAMS-OMV camera module bundle

Table 18. Execution performance of the people counting application

C model	Camera resolution	Volatile memory layout scheme	Weight-and-bias table location	NN inference time (ms)	Frames processed per second (FPS)(1)	Output accuracy (%)(2)	Binary file(3)
YOLO derivative 240 × 240	QVGA	Full internal	Internal Flash memory	371	2.7	55.88	R

1. *FPS is calculated taking into account the capture, preprocessing, and inference times.*
2. *Average precision obtained using a 50% IoU and against the PASCAL VOC test database.*
3. *IAR Embedded Workbench® configuration > Binary file.*

The binary files, contained in the `Binary` folder, associated with the scenarios in Table 18, are listed below:

- R: `STM32H747I-DISCO_PeopleCounting > STM32H747I-DISCO_PeopleCounting_Quant8_IntFps_IntFlash_V300.bin`

3.2.9 Memory footprint

Table 19 presents the memory requirements for all the applications. The numbers in the table account neither for the code and data related to the LCD display management nor for the code and data related to the test mode implementations.

Table 19. Memory footprints per application

Application C model	Camera resolution	Volatile data memory configuration(1)	Flash (byte)		RW data (byte)	
			Code	RO data(2)	Internal SRAM(3)	External SDRAM
Food Reco Float standard	VGA	Full external	~60 K	~520 K	~20 K	~1.6 M
		Split			~420 K	~1.2 M
Food Reco Float optimized		Full external		~590 K	~20 K	~2.0 M
Food Reco Quantized standard(4)	VGA	Full external	~80 K	~140 K	~20 K	~1.2 M
		Split			~170 K	
	QVGA	Full internal Memory optimized				0
		Full internal FPS optimized			~320 K	
Food Reco Quantized optimized(4)	VGA	Full external		~160 K	~20 K	~1.2 M
		Split			~240 K	
	QVGA	Full internal Memory optimized				0
		Full internal FPS optimized			~390 K	
Person Detect MobileNetV2	QVGA	Full internal FPS optimized		~500 K	~390 K	0
Person Detect Google				~230 K	~320 K	
People counting YOLO derivative (240 × 240)				~280 K	~430 K	

1. *The "allocate input in activation" option is selected in the STM32Cube.AI tool for C code generation (except for the Food Reco float models).*
2. *Includes the weight-and-bias table.*
3. *Includes the stack and heap size requirements.*
4. *Food recognition CNN model quantized using the quantization tool (provided by the STM32Cube.AI tool) with the "UaUa per layer" quantization scheme.*

The external SDRAM also contains the two following items:

- Read and write buffers for managing the display on the LCD: total size of 16 Mbytes since each buffer (of size 840 × 480 × 4 = 1.575 Mbytes) is placed in a separate SDRAM bank to optimize the SDRAM access
- The buffers used for the test mode implementation: total size of 5.4 Mbytes

The code size is different between the float and quantized models. This is due to the fact that both models do not involve the same set of functions from the runtime library.

3.2.10 USB webcam application

The USB webcam application enables the STM32H747I-DISCO board (connected to the B-CAMS-OMV camera module bundle or STM32F4DIS-CAM camera daughterboard) to act as a USB video camera device. USB video devices can be used to:

- create image datasets
- create video datasets
- perform Machine Learning evaluation on a host machine (x86) using the same image acquisition device as when a model is deployed onto the target STM32 device

Firmware implements the support for two formats, each offering two frame resolutions:

- MJPEG or YUY2 format
- VGA 640×480 or QVGA 320×240 resolution

In the MJPEG format, the camera is configured to capture frames in RGB565. The RGB565 frame is then fed into the STM32H7 hardware JPEG encoder for compression.

In YUY2 format, the camera is configured to capture frames in YUV 4:2:2. No compression is done by the STM32H7.

No additional driver is required to use the camera. The application implements the USB Device Class Definition for Video Devices, revision 1.1 June 1, 2005. Native USB video driver support is provided with Windows®, Ubuntu® and macOS®.

Note: *Ubuntu® is a registered trademark of Canonical Ltd.*

macOS® is a trademark of Apple Inc., registered in the U.S. and other countries and regions.

All other trademarks are the property of their respective owners.

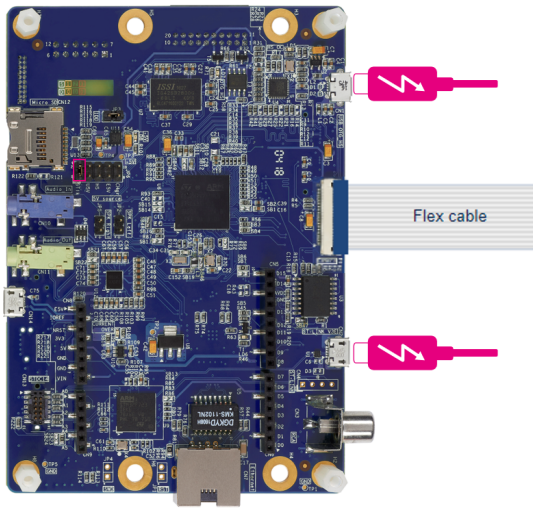
The video driver implements the following aspects of the specification:

- Device descriptor management
- Configuration descriptor management
- Interface Association Descriptor
  - Standard VC Interface Descriptor = interface 0
  - Standard VS Interface Descriptor = interface 1
- 1 Video Streaming Interface
- 1 Video Streaming Endpoint
- 1 Video Terminal Input (camera)
- Video Class-Specific AC Interfaces
- Video Class-Specific AS Interfaces
- VideoControl Requests
- Video Synchronization type: asynchronous

3.2.10.1 *USB webcam usage*

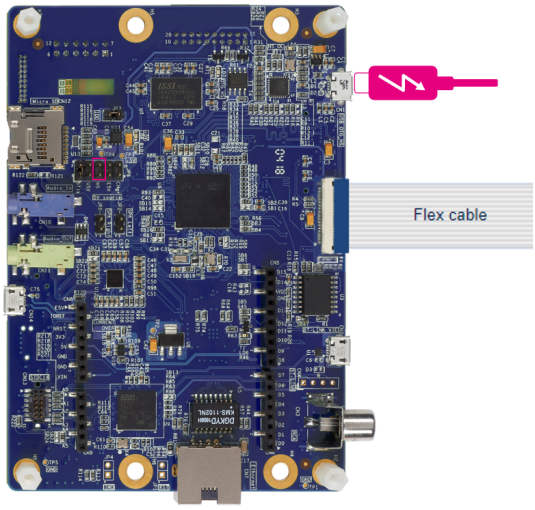
1. Load the binary onto the STM32H747I-DISCO device.
2. Connect a USB cable to the USB_OTG_HS connector. The board can be powered either through the standard ST-LINK USB or by changing jumper JP6 jumper to position HS.

Figure 14. STM32H747I-DISCO power supply (CN2)



JP6 on STLk – The STM32H747I-DISCO is supplied through the CN2 Micro-B USB receptacle.

Figure 15. STM32H747I-DISCO power supply (CN1)



JP6 on HS – The STM32H747I-DISCO is supplied through the CN1 Micro-AB USB receptacle.

Once connected, the device enumerates. Nothing is displayed on the board LCD screen.

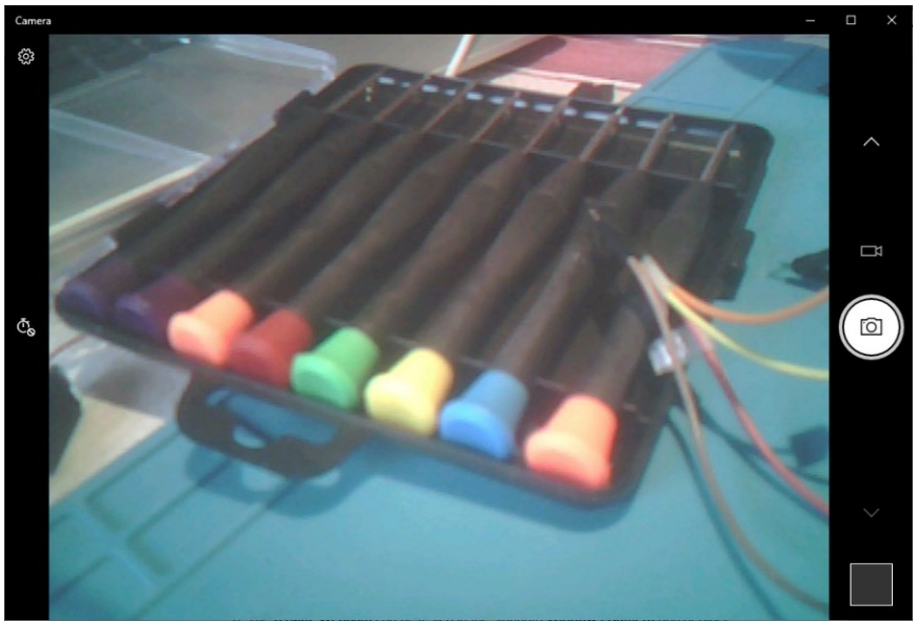
3. The STM32 webcam can be tested in the standard *Windows Camera* app or other applications with more control options like FFmpeg (see the FFmpeg usage section below).

Note: Depending on the host software, the driver may select a different format and resolution than what is configured by default.

Windows Camera app usage

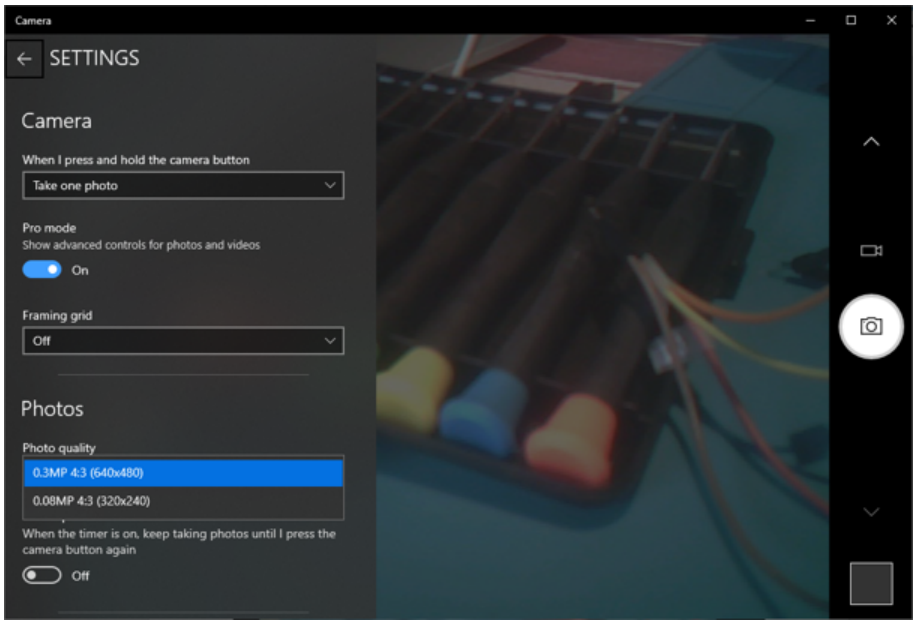
The Microsoft®-provided *Windows Camera* app can be used to capture images and video from the STM32 webcam. Images are captured using the MJPEG stream. Click on the camera icon or press the space bar to take a picture.

Figure 16. *Windows Camera* app



Camera settings can be used to change the camera resolution from 640×480 (VGA) to 320×240 (QVGA).

Figure 17. *Windows Camera* settings - Camera resolution



In some cases, the change of resolution may result in a cropped image. To overcome this issue, close and restart the *Windows Camera* app, restoring the last settings used.

FFmpeg usage

FFmpeg is a command-line utility that can be used to capture video streams. It offers more options than the *Windows Camera* app. Specifically, it can be used to view raw uncompressed YUY2 streams.

FFmpeg can be downloaded and installed from ffmpeg.org.

FFmpeg is a command-line tool. For a graphical user interface alternative, the ContaCam (www.contaware.com) software can be used.

First, list the available devices and supported options for the *"STM32 Webcam"* device:

```
ffmpeg -f dshow -list_devices true -i dummy                ## List devices
ffmpeg -f dshow -list_options true -i video="STM32 Webcam" ## list options
```

Then, use one of the following commands to view a live stream of the *"STM32 Webcam"*:

```
ffplay -f dshow -i video="STM32 Webcam"
ffplay -f dshow -video_size 640x480 -vcodec mjpeg -i video="STM32 Webcam"
ffplay -f dshow -video_size 320x240 -vcodec mjpeg -i video="STM32 Webcam"
ffplay -f dshow -video_size 640x480 -pixel_format yuyv422 -i video="STM32 Webcam"
ffplay -f dshow -video_size 320x240 -pixel_format yuyv422 -i video="STM32 Webcam"
```

Additional information is available from the FFmpeg wiki at trac.ffmpeg.org/wiki/Capture/Webcam.

Other usage example

Refer to the STM32 MCU wiki at wiki.st.com/stm32mcu for more usage example.

3.2.10.2 *Execution performance of the USB webcam application*

The DCMI capture operates in the continuous mode. For each captured frame, the image is copied into a USB transmit buffer for asynchronous offloading on USB interrupts. The USB offloads the buffer in the isochronous mode with a packet size of 1024 bytes (USB High Speed) or 1023 bytes (USB Full Speed).

When the MJPEG format is selected, the JPEG encoding, and USB transmission are pipelined with the DCMI capture. As the JPEG encoding relies on software and hardware encoding, the processing time varies according to compiler and optimization settings.

Table 20. USB webcam application timings (with CAMERA_BOOST_PCLK = 0 / 1)

Performance	MJPEG VGA	MJPEG QVGA	Uncompressed VGA	Uncompressed QVGA
DCMI capture(1)	133.3 / 66.7 ms	133.3 / 33.3 ms	133.3 / 83.3 ms	133.3 / 33.3 ms
JPEG encoding	~46-47 ms	~11-12 ms	N/A	N/A
USB transmission	~6-9 ms	~2-3 ms	75.3 ms	18.9 ms
Frame rate(1)	7.5 / 15 fps	7.5 / 30 fps	7.5 / 12 fps	7.5 / 30 fps
Bit rate(1)	~3.3 / 9.2 Mbps	~831 / 3.9 kbps	37 / 58.9 Mbps	9.2 / 37 Mbps

1. *First value with `CAMERA_BOOST_PCLK = 0` and second value with `CAMERA_BOOST_PCLK = 1`. The `CAMERA_BOOST_PCLK` flag can be configured in the project compiler flag (refer to Section 3.2.6 for details).*

Note:
- *For MJPEG, the USB transmission time may vary with the frame size.*
- *The uncompressed VGA mode is limited by the USB transmission time.*
- *The MJPEG modes are limited by the JPEG encoding times.*

Configuration:
- All buffers in external SDRAM.
- B-CAMS-OMV camera module bundle with the MB1379 STMicroelectronics camera module (OV5640).
- JPEG quality = 90%. Can be adjusted in firmware using the JPEG_QUALITY macro.
- USB High Speed with a 1024-byte packet size. One packet is sent every microframe (125 µs). Although it is possible to specify up to 3 isochronous transfers per microframe, the HAL only supports 1 transaction per microframe.

3.2.10.3 *USB webcam application known issues and limitations*

- Incorrect frame rate when using the OV5640 camera unless `CAMERA_BOOST_PCLK = 1`. By default, the OV5640 camera is configured at 7.5 fps in all configurations. Incorrect frame rate reporting may cause video recording and video playback speed issues. Note that the reported frame rate is kept unchanged to maintain a compatibility with teachablemachine.withgoogle.com.
- When operating in the high-speed mode, the uncompressed VGA frame rate provided by default by the camera (15 fps) cannot be matched by the USB transfer speed. The camera frame rate is slowed down in this case. To achieve an uncompressed 15 fps frame rate, additional transaction per microframe would be required but this feature is not currently supported by the HAL.
- When operating in the full-speed mode, performance is degraded:
  - Slower frame rate
  - Some tearing effect can be observed

3.2.11 Visualization modes

3.2.11.1 *Food recognition applications*

The main application provides two visualization modes:

- A logo view, where each food category is represented by a logo
- An image view, which displays the camera feed

By default, the application starts with the logo view activated. Press the [**WakeUp**] button during the execution to toggle between the modes.

*Note: When pressing the [**WakeUp**] button, the user must wait for the message `Entering/Exiting CAMERA PREVIEW mode, Please release button` to show up on the screen before releasing the button. Both visualization modes also report the inference time (in ms), the number of frames per second (FPS) that are processed, and the Top-1 output class probability (in %).*

3.2.11.2 *Person presence detection applications*

For the person presence detection applications, only the camera image view mode is available. The captured camera content is displayed along with the inference time (in ms), the number of processed frames per second (FPS) and the Top-1 output class probability (in %). The person presence detection applications provide four camera orientation modes:

- Normal
- Flipped image (default)
- Mirrored image
- Mirrored and flipped image

Pressing on the blue [**WakeUp**] button causes the camera sensor to cycle through the different camera orientation modes listed above. This functionality can be used to place the camera in different orientations with respect to the board LCD display.

3.2.11.3 *People counting detection application*

For the people counting application, only the camera image view mode is available. The captured camera content is displayed along with the number of processed frames per second (FPS). A *"VACANT"* logo is displayed when no person is detected in the scene. When one or more persons are detected within the scene, the number of people detected is displayed.

The people counting application provide four camera orientation modes:

- Normal
- Flipped image (default)
- Mirrored image
- Mirrored and flipped image

Pressing on the blue [**WakeUp**] button causes the camera sensor to cycle through the different camera orientation modes listed above. This functionality can be used to place the camera in different orientations with respect to the board LCD display.

3.2.12 Embedded validation, capture and testing

The FP-AI-VISION1 function pack embeds the possibility to perform validation, capture and testing via specific operating modes:

- *Frame Capture* mode: captures frames from the camera to a microSD™ card
- *Memory Dump* mode: dumps every step of the processing pipeline to a microSD™ card for debug
- *Onboard Validation* mode: evaluates the Neural Network with images stored in a microSD™ card

This section presents each of the three modes as well as the way to start each of them.

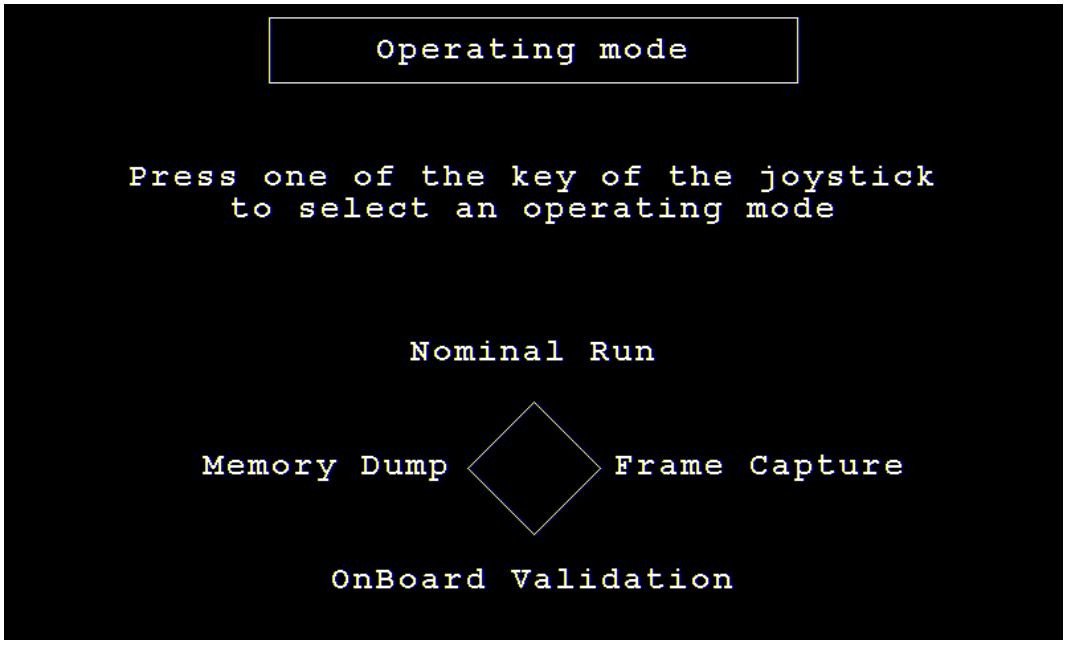
Prerequisite

The three modes require a microSD™ card to be plugged into the STM32H747I-DISCO Discovery board. It is recommended to use at least a class 10 microSD™ card to avoid slow read/write operations. Moreover, the card must be FatFS formatted and contain one single partition.

Enabling the validation, capture and testing modes

Maintain the [**WakeUp**] button pressed during the welcome screen (after a reset) to enable the test menu shown in Figure 18 instead of the main application.

Figure 18. Validation, capture and testing menu

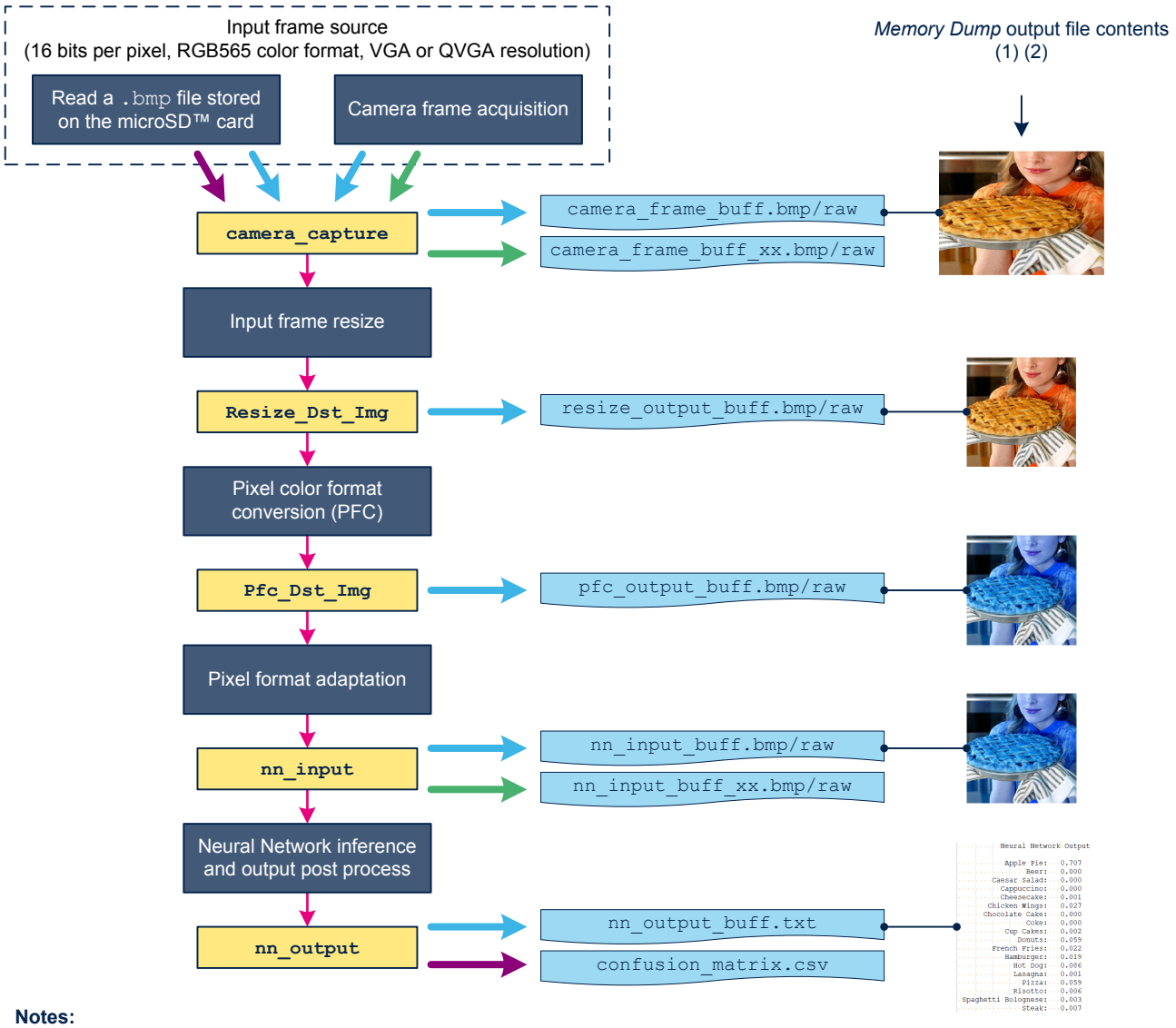


Start the desired application by pressing the corresponding arrow key of the joystick:

- [**UP**]: the default application is started (Neural Network inference with images from the camera)
- [**RIGHT**]: the *Frame Capture* mode is started
- [**LEFT**]: the *Memory Dump* mode is started
- [**DOWN**]: the *Onboard Validation* mode is started

Overview of *Onboard Validation*, *Frame Capture* and testing modes

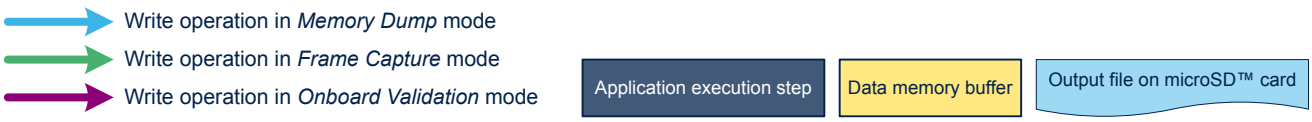
Figure 19. Overview of validation, capture and testing modes



Notes:

(1) The example is based on the food recognition (quantized C model) application. In this specific example, the pixel color format conversion (PFC) is configured (by the application) to swap the blue and red components for each pixel. This swap is required in this specific case since the food recognition NN model is trained with images containing pixels in RGB format (refer to *Section 3.2.6*) while the frames produced by the DMA2D during the PFC operation contain pixels organized in BGR format.
(2) The picture is extracted from *The Food-101 Data Set*.

Legend:



Frame Capture mode

The main purpose of the *Frame Capture* application is to enable data collection. By default in the application, the frame is captured at two stages:

- right after the camera acquisition (RGB565 format)

and

- after all the preprocessing stages, just before being fed to the input of the Neural Network

However, by modifying the application code, the frame can be captured at any stage of the execution chain. To do so, set to 1 the `PerformCapture` field of the `TestRunContext` structure that is initialized before each stage of the execution chain.

The format of the file into which the frame is captured can be configured with one of the formats below:

- `.bmp` file with data encoded in 16-bit pixel format (RGB565)
- `.raw` binary file format

This is configurable via the `capture_file_format` field of the `CaptureContext` structure during the initialization of the *Frame Capture* mode. The default format is `.bmp`.

At the end, the captured frame is stored into the microSD™ card.

For the frame capture mode, the user can also choose to swap the red and blue components of the pixels before writing the buffer content into the output file. This is done by setting to 1 the `rb_swap` field of the `TestRunContext` structure.

At application startup, the `Camera_Capture` folder is created if not already existing and a session name is generated and displayed in the top-left corner of the screen. A sub-folder, named `CAM_CAPTURE_SESS_<session name>`, is also created. The screen shows the image taken from the camera and in the top-right corner, the `READY` message indicates that the program is ready to write to the microSD™ card.

To trigger a frame capture, press on the [**WakeUp**] button. The `READY` message switches to `BUSY`, indicating that the frame is being written to the microSD™ card, preventing all other capture meanwhile. Once the write operation is finished, the `READY` message is displayed and capture enabled again.

Each capture generates two files in the sub-folder corresponding to the session:

- `camera_frame_buff_xx.bmp` or `camera_frame_buff_xx.raw`: contains the frame captured right after the camera acquisition, before the preprocessing stage.
- `nn_input_buff_xx.bmp` or `nn_input_buff_xx.raw`: contains the frame captured after the preprocessing stage, before the input of the Neural Network.

For a given session, the `xx` filename postfix corresponds to the incremental capture number.

Memory Dump mode

The *Memory Dump* application is mainly intended for debug purposes. When entering this operating mode, the `Memory_Dump` folder is created if not already existing.

The screen shows the image contained in the `camera_frame` buffer and in the top-right corner, the `READY` message indicates that the program is ready to start a memory dump session.

To trigger a memory dump session, press on the [**WakeUp**] button. The `READY` message switches to `BUSY`, indicating that the memory dump operations are being performed and that writing to the microSD™ card is taking place, preventing all other capture meanwhile. Similarly to the *Frame Capture* mode, a session name is generated and displayed in the top-left corner of the screen. A sub-folder named `DUMP_SESS_<session name>` is also created. Once the memory dump operations are completed, the `READY` message is displayed and memory dump enabled again.

During a memory dump session, each execution stage of the application (as described in Section 3.2.5.1 Application execution flow and volatile (RAM) data memory requirement) is followed by a dump of the memory portion containing the output data. The memory content is dumped into files stored in the microSD™ card. These files can be in one the following formats: `.bmp`, `.raw` or `.txt`.

Each memory dump session generates five files in the sub-folder corresponding to the session:

- `camera_frame_buff.bmp` or `camera_frame_buff.raw`: `camera_frame` buffer content following acquisition.
- `resize_output_buff.bmp` or `resize_output_buff.raw`: `Resize_Dst_Img` buffer content following resizing operation.
- `pfc_output_buff.bmp` or `pfc_output_buff.raw`: `Pfc_Dst_Img` buffer content following pixel color format conversion operation.
- `nn_input_buff.bmp` or `nn_input_buff.raw`: `nn_input` buffer content following pixel format adaptation operation.
- `nn_output_buff.txt`: `nn_ouput` buffer content following Neural Network inference.

The memory dump mode has three sub-modes depending on the source of the input image:

- *SD card*: the input image is coming from a `.bmp` file stored in the microSD™ card in a folder named `dump_src_image_xx`, `xx` being the resolution of the image (possible values are: `vga` and `qvga`).
- *Camera live*: the input image is coming from the camera acquisition (like in the nominal operating mode).
- *Test Color Bar*: the input image is coming from the camera acquisition, the camera being configured in test color bar mode.

The format of the file into which the memory content is dumped can be configured with one of the five formats below:

- `.bmp` file format with data encoded in 8 bits per pixel
- `.bmp` file format with data encoded in 16 bits per pixel (RGB565)
- `.bmp` file format with data encoded in 24 bits per pixel (RGB888)
- `.raw` binary file format
- `.txt` file format

Figure 19 above shows an example of *Memory Dump* output files obtained when using the food recognition quantized model in QVGA resolution. The output file obtained after the pixel color format conversion stage has red and blue pixel components swapped. The reason for the swap is because the food recognition NN model expects the memory layout of the input pixel with the red component at address offset 0. However the DMA2D (during the pixel format conversion operation) is providing frames with pixels organized with blue components at address offset 0. A swap between the red and blue pixel components is required so to feed the NN model with correct input data (refer to Section 3.2.7).

***Onboard Validation* mode**

The *Onboard Validation* application uses images stored in the microSD™ card as inputs for evaluating the Neural Network.

Important: *It is important to differentiate the "Onboard Validation" process described here and the validation process integrated into the X-CUBE-AI tool. The input data expected by the "Onboard Validation" process are raw data (`.bmp` file) meant to undergo specific preprocessing operations before being fed to the embedded Neural Network model. In situations where the user is willing to compare the results obtained with the "Onboard Validation" process against the ones obtained with the validation process integrated into the X-CUBE-AI tool, the input data used by the "Onboard Validation" process cannot be used as such as input to run the validation of the model in the X-CUBE-AI environment. For more details, refer to the X-CUBE-AI embedded documentation in section "Evaluation report and metrics/Specific attention on the provided data".*

In order for the program to find the images in the microSD™ card, a directory named `onboard_valid_dataset_xx` must exist at the root of the file system, `xx` being the resolution of the image (possible values are: `vga` and `qvga`). Inside this directory, there must be one directory per class containing the images with the same name as defined by `NN_OUTPUT_CLASS_LIST` in the code.

In the function pack, this variable is aliased to `output_labels` defined in file `fp_vision_app.c`. The following example is given for the food recognition application:

```
#define NN_OUTPUT_CLASS_LIST output_labels
const char *output_labels[AI_NET_OUTPUT_SIZE] = {
"Apple Pie", "Beer", "Caesar Salad", "Cappuccino", "Cheesecake", "Chicken Wings", "Chocolate
Cake", "Coke", "Cup Cakes", "Donuts", "French Fries", "Hamburger", "HotDog", "Lasagna",
"Pizza", "Risotto", "Spaghetti Bolognese", "Steak"};
```

All images must be stored in the BMP format (16 bits per pixel and RGB565 format) with a VGA or QVGA resolution. The class is derived directly from the directory containing the images. The filenames are not considered.

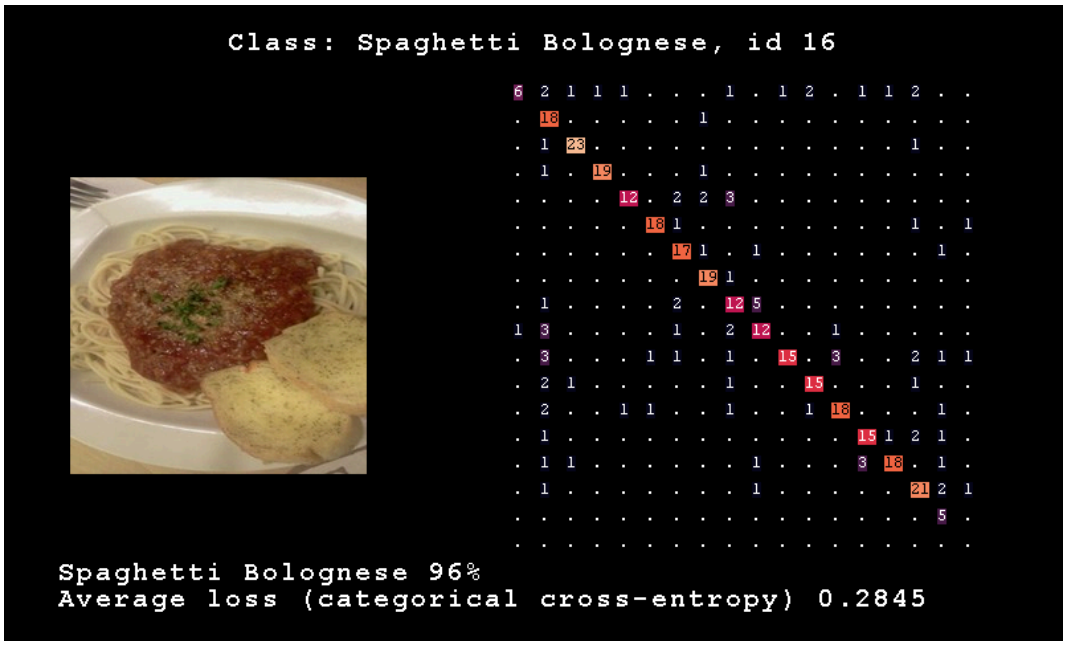
A helper script (named `create_dataset.py`) to convert any dataset of images to the BMP format is provided in the `Utilities\AI_resources\Food-Recognition` directory.

For the food recognition example, the structure of the microSD™ file system must be as follows:

```
.
|-- onboard_valid_dataset_xx
    |-- Apple Pie
    |   |-- apple_pie_example1.bmp
    |   |-- another_apple_pie.bmp
    |   `-- ...
    |-- Beer
    |   |-- beer.bmp
    |   |-- another_beer.bmp
    |   `-- ...
    |-- Caesar Salad
    |   |-- a_caesar_salad.bmp
    |   `-- ...
    |-- Cappuccino
    |   |-- cappuccino.bmp
    |   `-- ...
    |-- Cheesecake
    |   |-- cheesecake.bmp
    |   `-- ...
    |-- Chicken Wings
    |   |-- chicken_wing.bmp
    |   `-- ...
    |-- Chocolate Cake
    |   |-- chocolate_cake.bmp
    |   `-- ...
    |-- Coke
    |   |-- coke.bmp
    |   `-- ...
    |-- Cup Cakes
    |   |-- cup_cakes.bmp
    |   `-- ...
    |-- Donuts
    |   |-- donuts.bmp
    |   `-- ...
    |-- French Fries
    |   |-- french_fries.bmp
    |   `-- ...
    |-- Hamburger
    |   |-- hamburger.bmp
    |   `-- ...
    |-- Hot Dog
    |   |-- hot_dog.bmp
    |   `-- ...
    |-- Lasagna
    |   |-- lasagna.bmp
    |   `-- ...
    |-- Pizza
    |   |-- pizza.bmp
    |   `-- ...
    |-- Risotto
    |   |-- risotto.bmp
    |   `-- ...
    |-- Spaghetti Bolognese
    |   |-- spaghetti_bolognese.bmp
    |   `-- ...
    `-- Steak
        |-- steak.bmp
        `-- ...
```

At startup, the *Onboard Validation* application displays a summary of the information presented in this section. Press the [**WakeUp**] button to start the validation. The screen shown in Figure 20 is then displayed.

Figure 20. *Onboard Validation* summary of information for a food recognition example



At the top of the display, the class name is presented with its index in `NN_OUTPUT_CLASS_LIST`.

The left side shows the current image being processed by the Neural Network, which is resized to match the network input size. Below the image, the display shows the NN Top-1 output, the confidence level, and the average categorical cross-entropy loss across all processed images.

The right side of the display shows the confusion matrix, that is constantly updated while images are being processed by the neural network. The rows indicate the ground-truth label, and the columns the predicted class.

When all images are processed, a message is displayed and a press on the [**WakeUp**] button updates the display with a classification report, as shown in Figure 21.

Figure 21. *Onboard Validation* classification report for a food recognition example

	precision	recall	f1-score	support
Apple Pie	0.857	0.316	0.462	19
Beer	0.500	0.947	0.655	19
Caesar Salad	0.885	0.920	0.902	25
Cappuccino	0.950	0.905	0.927	21
Cheesecake	0.857	0.632	0.727	19
Chicken Wings	0.900	0.857	0.878	21
Chocolate Cake	0.607	0.850	0.708	20
Coke	0.760	0.950	0.844	20
Cup Cakes	0.545	0.600	0.571	20
Donuts	0.571	0.600	0.585	20
French Fries	0.938	0.536	0.682	28
Hamburger	0.789	0.750	0.769	20
Hot Dog	0.818	0.720	0.766	25
Lasagna	0.750	0.750	0.750	20
Pizza	0.900	0.720	0.800	25
Risotto	0.656	0.808	0.724	26
Spaghetti Bolognese	0.731	0.905	0.809	21
Steak	0.857	0.692	0.766	26
accuracy			0.747	395
macro avg	0.771	0.748	0.740	395
weighted avg	0.778	0.747	0.744	395

The classification report shows the precision, recall, and f1-score for each class, as well as the macro average (average of the unweighted mean per class) and weighted average (average of the support-weighted mean per class).

The confusion matrix, the list of misclassified files and the classification report are saved at the root of the microSD™ card file system as:

- `confusion_matrix.csv`
- `missclassified.txt`
- `classification_report.txt`

3.2.13 Output display

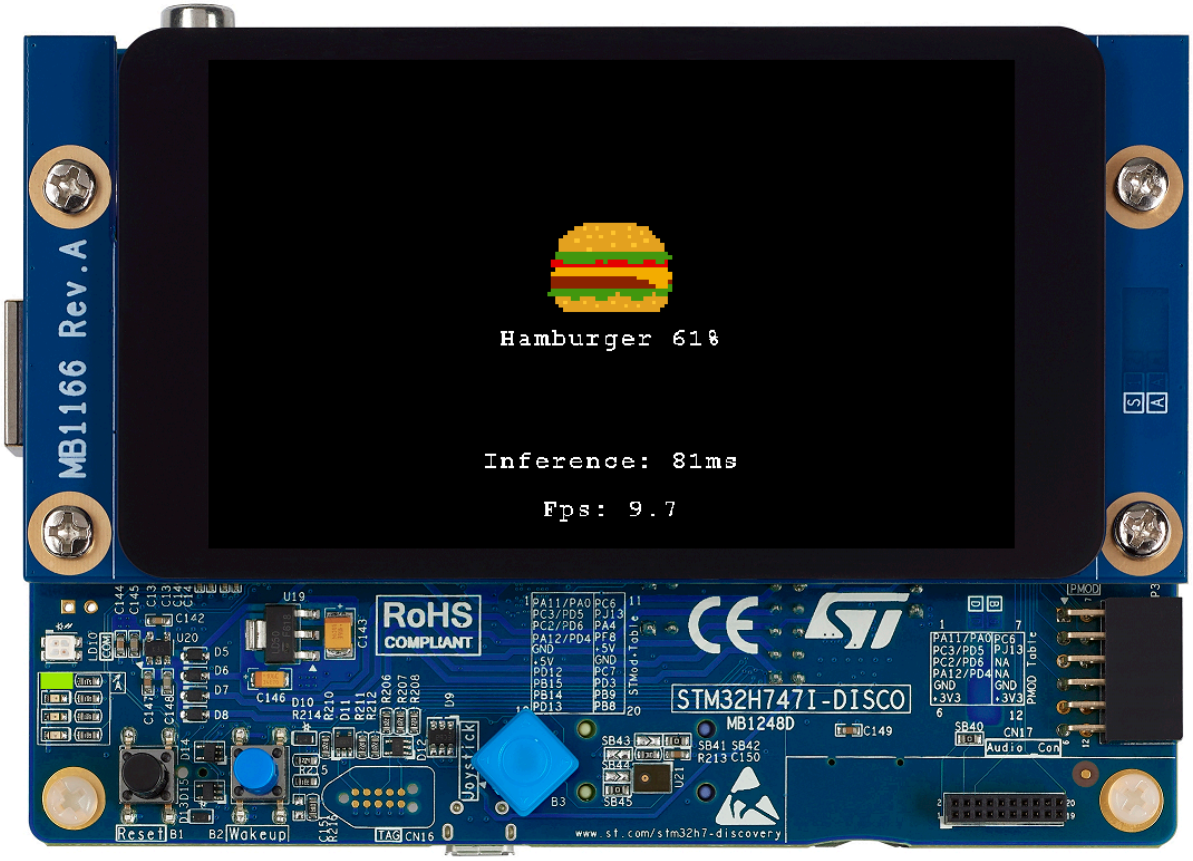
By default, only the information (class name + confidence level in %) concerning the output class with the highest probability (Top1) is displayed on the LCD after the inference.

The CNN inference time (expressed in milliseconds) as well as the number of frames processed per second (FPS) are also displayed on the LCD after the inference.

One of the three LEDs below is set after each inference:

- Red LED: if output confidence is below 55 %
- Orange LED: if output confidence ranges from 55 % to 70 %
- Green LED: if output confidence is above 70 %

Figure 22. **Food recognition example with green LED on**



3.3 Hardware setup

The FP-AI-VISION1 function pack supports the following hardware configuration: STM32H747I-DISCO Discovery board connected to the B-CAMS-OMV camera module bundle (advised) or STM32F4DIS-CAM camera daughterboard (legacy only).

The STM32H747I-DISCO Discovery board features:

- One STM32H747XIH6 microcontroller with:
  - SRAM:
    - 864 Kbytes of system SRAM (512 Kbytes + 288 Kbytes + 64 Kbytes)
    - 128 Kbytes of DTCM RAM
  - Flash memory:
    - 2 Mbytes (2 banks of 1 Mbyte each)
- One 8-bit camera connector
- One 256-Mbit external SDRAM

The B-CAMS-OMV camera module bundle features:

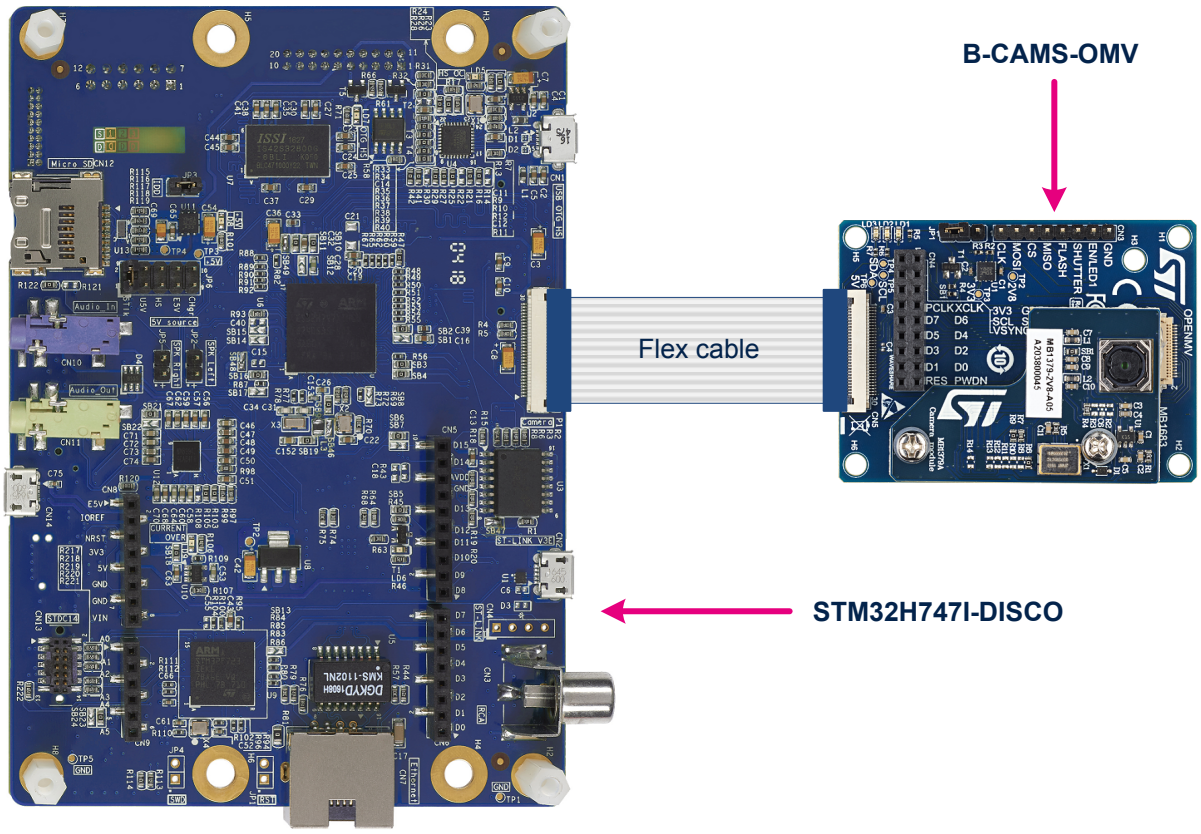
- One OmniVision® OV5640 camera sensor

The STM32F4DIS-CAM camera daughterboard features:

- One OmniVision® OV9655 camera module

The B-CAMS-OMV camera module bundle (or STM32F4DIS-CAM camera daughterboard) is connected to the STM32H747I-DISCO Discovery board through a flex cable provided with the camera module bundle (or camera daughterboard) as illustrated in Figure 23.

Figure 23. Hardware setup with STM32H747I-DISCO and B-CAMS-OMV



4 Testing the CNN

4.1 Testing conditions

The typical testing environment consists of a tablet device placed in front of the camera. In that case, the user must keep in mind that the following parameters impact the accuracy of the recognition:

- Ambiance light and illumination, for instance neon light flickering
- Quality of the input image displayed by the tablet. High-resolution is required
- Distance between the camera and the tablet
- Reflection, for instance when a tablet is used as input image source

Adjusting the camera contrast by pressing the joystick [**LEFT**] ans [**RIGHT**] buttons is a way to improve the accuracy of the recognition.

4.2 Camera and LCD setting adjustments

The LCD brightness is set using the joystick ([**UP**] and [**DOWN**]).

The camera contrast levels are adjusted using the joystick ([**LEFT**] and [**RIGHT**]).

The LCD brightness and camera contrast levels are set to their default values using the joystick ([**SEL**]).

Revision history

Table 21. **Document revision history**

Date	Revision	Changes
17-Jul-2019	1	Initial release.
24-Dec-2019	2	Replaced former section *Test mode* with *Section 3.2.8 Embedded validation and testing*. Replaced former section *Two acquisition modes* with *Section 3.2.7 Two visualization modes*. Added *Section 3.2.3 Training script*. Updated the folder tree across the entire document. Updated the entire document with respect to the possible buffer optimization offered by the *allocate input in activation* option in STM32Cube.AI. Updated measurement conditions and results in *Section 3.2.5 Execution performance and overall memory footprint*.
10-Sep-2020	3	Document entirely updated: • Added person presence detection applications • Updated data buffer and memory usage • Updated performance measurements and memory footprints • Updated folder organization and content • Updated scenarios • Updated validation, capture and testing modes
13-Apr-2021	4	Added the USB webcam application and the use of the B-CAMS-OMV camera module bundle: • Updated *FP-AI-VISION1 function pack feature overview*, *Software architecture, Folder organization*, *Execution performance for food recognition and presence detection applications* and *Hardware setup* • Added *Camera pixel clock*, *Pixel data order* and *USB webcam application*
30-Aug-2021	5	Added the people counting application: • Updated *FP-AI-VISION1 function pack feature overview*, *Software architecture*, *Integration of the generated code*, *Folder organization*, *Camera pixel clock*, and *Applications performance* • Added *People counting application*, *Table 5. SRAM memory buffers for people counting application*, *Other applications*, and *People counting detection application* Removed the dual-core MCU note from *Building a CNN-based computer vision application on STM32H7*.
17-Dec-2021	6	Replaced the STM32_Image library with the STM32_ImageProcessing_Library and removed the references to STM32_Fs: • Updated FP-AI-VISION1 function pack feature overview, Software architecture and Folder organization • Added STM32 image processing library Added People counting application.

Contents

List of tables

List of figures