



EIQGLOWAOTUG

eIQ Glow Ahead of Time User Guide

Rev. 6 — 1 June 2022

User guide

Document information

Information	Content
Keywords	eIQ, Glow, Ahead of Time, AoT
Abstract	This document describes steps to download, start using Glow AOT, and create an application that integrates bundles generated using the Glow AOT compiler.



1 Overview

Glow is a machine learning compiler for neural network graphs. It is designed to optimize the neural network graphs and generate code for various hardware devices available at <https://github.com/pytorch/glow>. Glow comes in two flavors: Just in Time (JIT) and Ahead of Time (AOT) compilations. JIT compilation is performed at runtime just before the model is executed. AOT compilation is performed offline and generates an object file (bundle) which is later linked with the application code.

MCUXpresso Software Development Kit (MCUXpresso SDK) provides a set of helper functions that allows integration of Glow AOT bundles.

Ensure to see the official Glow documentation for building AOT applications located at: <https://github.com/pytorch/glow/blob/master/docs/AOT.md> before proceeding. However, the eIQ Glow Ahead of Time User Guide provides extra information regarding the NXP deliverable of the Glow compiler which includes some extra features and optimizations.

This document describes steps to download, start using Glow AOT, and create an application that integrates bundles generated using the Glow AOT compiler.

To run the Glow AOT project examples, the following packages must be installed on your system.

- MCUXpresso IDE
- MCUXpresso SDK loaded into MCUXpresso IDE
- A serial connection client like PuTTY or Tera Term
- Python 3.6 with pip package installer
- The Glow AOT compiler which is available for Windows [here](#).

After installing the Glow AOT compiler, you must add the location of the Glow binary tools to the PATH system environment variable.

2 Deployment

The eIQ Glow AOT is part of the eIQ machine learning software package, which is an optional middleware component of MCUXpresso SDK. The eIQ component is integrated into the MCUXpresso SDK Builder delivery system available on mcuxpresso.nxp.com. To include eIQ machine learning into an MCUXpresso SDK package, the eIQ middleware component has to be selected in the software component selector on the SDK Builder when building a new package. For details, see [Figure 1](#).

	Name	Category	Description	Dependencies
<input checked="" type="checkbox"/>	SDMMC Stack	Middleware	Stack supporting SD, MMC, SDIO	
<input type="checkbox"/>	CANopen	Middleware	MicroCANopen Stack from Embedded Solutions Academy	
<input type="checkbox"/>	cJSON	Middleware	Ultralightweight JSON parser in ANSI C	
<input checked="" type="checkbox"/>	CMSIS DSP Library	CMSIS DSP Lib	CMSIS DSP Software Library	
<input checked="" type="checkbox"/>	eIQ	Middleware	eIQ machine learning SDK containing: - ARM CMSIS-NN library ... (more)	CMSIS DSP Library
<input type="checkbox"/>	Embedded Wizard GUI	Middleware	Embedded Wizard GUI from TARA Systems	
<input type="checkbox"/>	emWin	Middleware	emWin graphics library	
<input type="checkbox"/>	Azure RTOS		Azure RTOS	
<input type="checkbox"/>	FreeRTOS		Real-time operating system for microcontrollers from Amazon	

Figure 1. MCUXpresso SDK Builder software component selector for RT1050/RT1060/RT1064/RT1160/RT1170

Filter by Name, Category, or Description...			Select All	Unselect All
	Name	Category	Description	Dependencies
<input checked="" type="checkbox"/>	multicore	Middleware	Multicore Software Development Kit	
<input checked="" type="checkbox"/>	SDMMC Stack	Middleware	Stack supporting SD, MMC, SDIO	
<input checked="" type="checkbox"/>	CMSIS DSP Library	CMSIS DSP Lib	CMSIS DSP Software Library	
<input type="checkbox"/>	DSP Audio Streamer	Middleware	DSP Audio Streamer Framework based on Xtensa Audio Framework... (more)	Essential Audio Processing Library
<input checked="" type="checkbox"/>	DSP Neural Networks	Middleware	DSP Neural Networks Framework based on Xtensa Neural Network... (more)	
<input checked="" type="checkbox"/>	eIQ	Middleware	eIQ machine learning SDK containing: - ARM CMSIS-NN library ... (more)	CMSIS DSP Library, DSP Neural Networks
<input type="checkbox"/>	emWin	Middleware	emWin graphics library	
<input checked="" type="checkbox"/>	FreeRTOS		Real-time operating system for microcontrollers from Amazon	
<input type="checkbox"/>	AWS IoT Core	Middleware	Amazon Web Service (AWS) IoT Core SDK	FreeRTOS, lwIP, mbedTLS, NXP Wi-Fi
<input type="checkbox"/>	NXP Wi-Fi	Middleware	NXP Wi-Fi	FreeRTOS, lwIP, SDMMC Stack
<input type="checkbox"/>	Wireless edgefast_bluetooth stack	Middleware	BT/BLE edgefast_bluetooth stack	Fatfs, FreeRTOS, lwIP, mbedTLS, NXP Wi-Fi, SDMMC Stack, USB Host, Device, OTG Stack

Figure 2. MCUXpresso SDK Builder software component selector for RT600

Once the MCUXpresso SDK package is downloaded, it can be extracted on a local machine or imported into the MCUXpresso IDE. To get familiar with the MCUXpresso SDK folder structure, see the *Getting Started with MCUXpresso SDK* document.

The package directory structure might look like [Figure 3](#). The eIQ Glow library directories are highlighted in red.

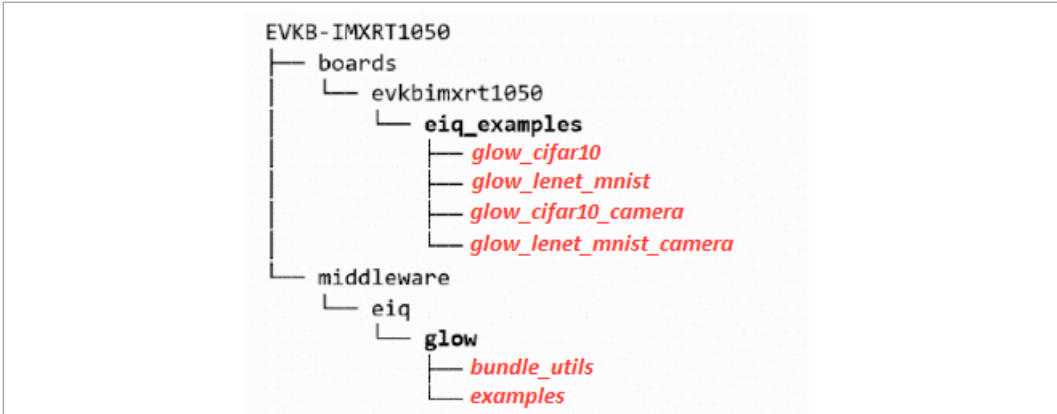


Figure 3. MCUXpresso EVKB-IMRT1050 SDK structure

The package directory structure might look like [Figure 4](#). The eIQ Glow library directories are highlighted in red.

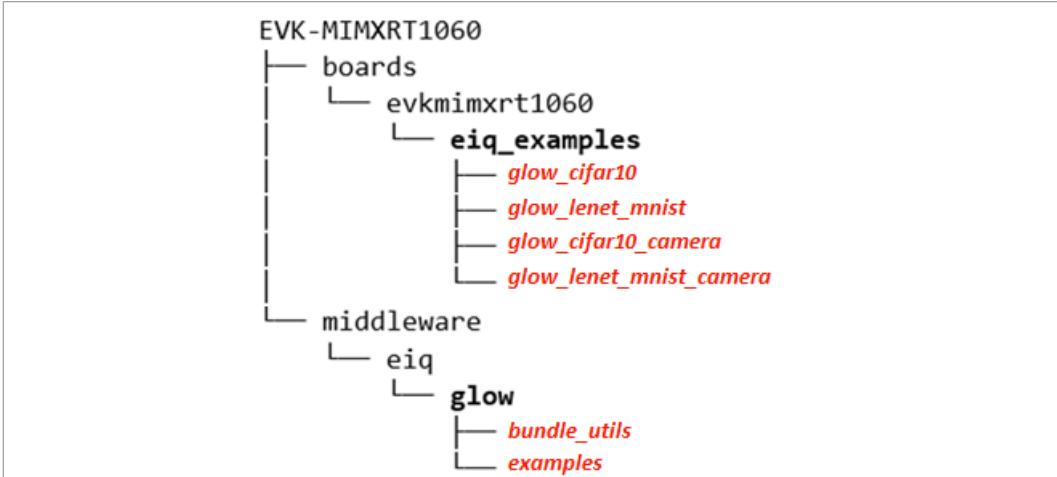


Figure 4. MCUXpresso EVKB-IMRT1060 SDK structure

The package directory structure might look like [Figure 5](#). The eIQ Glow library directories are highlighted in red.

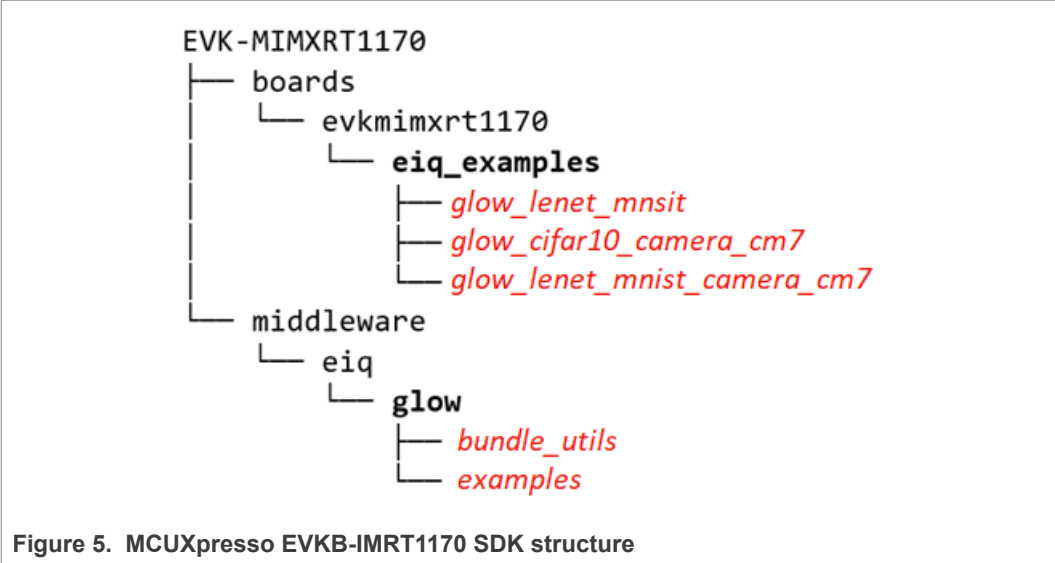


Figure 5. MCUXpresso EVKB-IMRT1170 SDK structure

The package directory structure might look like [Figure 6](#). The eIQ Glow library directories are highlighted in red.

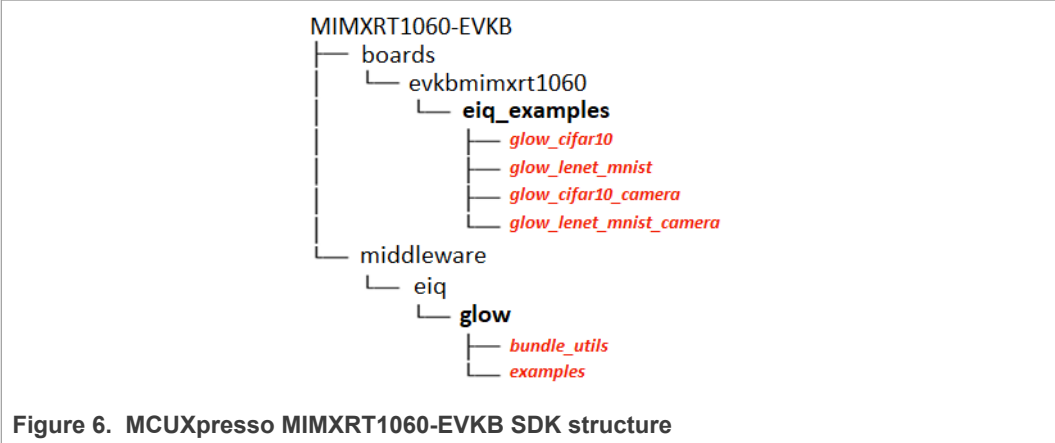


Figure 6. MCUXpresso MIMXRT1060-EVKB SDK structure

The package directory structure might look like [Figure 7](#). The eIQ Glow library directories are highlighted in red.

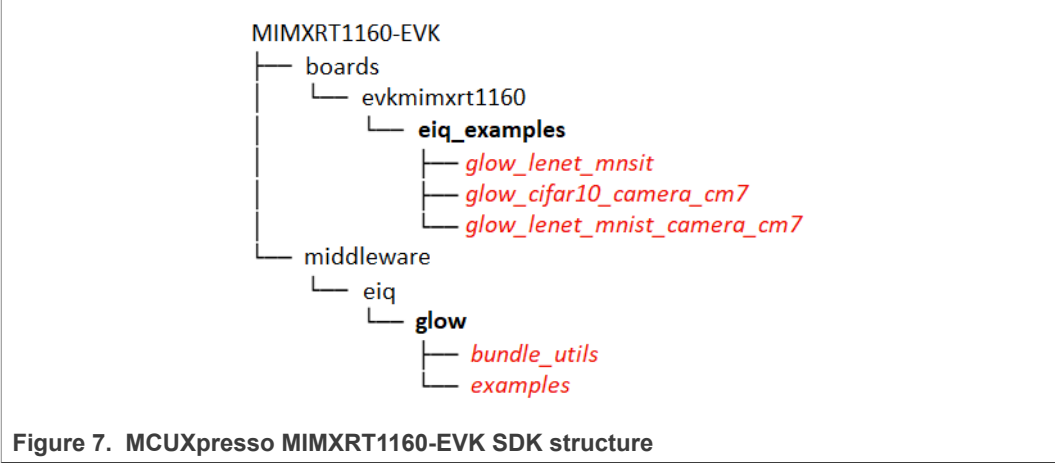


Figure 7. MCUXpresso MIMXRT1160-EVK SDK structure

The package directory structure might look like [Figure 8](#). The eIQ Glow library directories are highlighted in red.

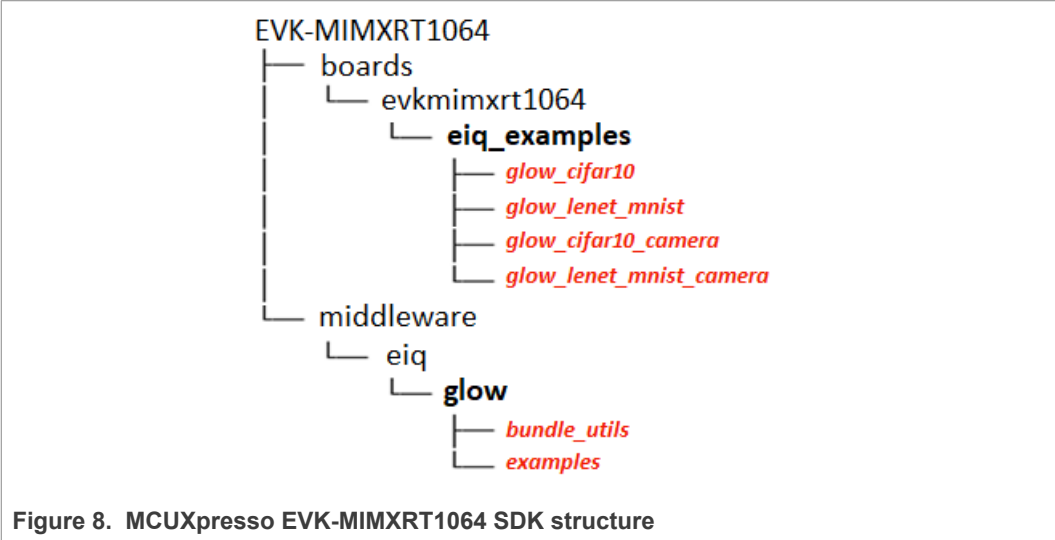


Figure 8. MCUXpresso EVK-MIMXRT1064 SDK structure

The package directory structure might look like [Figure 9](#). The eIQ Glow library directories are highlighted in red.

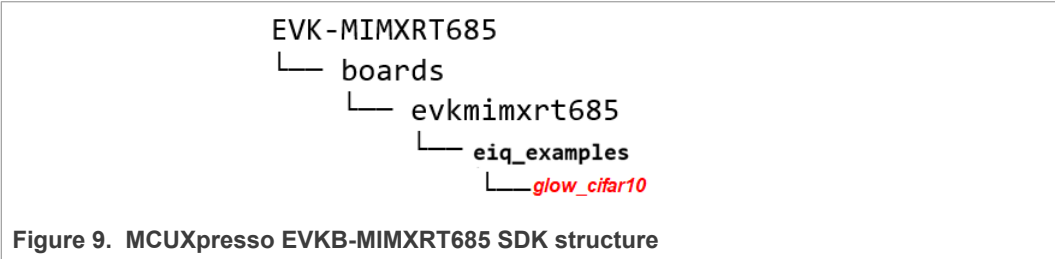


Figure 9. MCUXpresso EVKB-MIMXRT685 SDK structure

The *boards* directory contains example application projects for supported toolchains (for the list of supported toolchains, see the *MCUXpresso SDK Release Notes* document). The *middleware* directory contains the eIQ library source codes, pre-compiled library binaries and example application source codes and data.

Note: *Installing the Glow AOT tool on the target machine is out of the scope of this document. It is assumed that the tool is already installed on the workstation and its location is included in the system path. You can find the Glow AOT Windows installer [here](#).*

3 Example applications

The eIQ Glow AOT is provided with a set of example applications. For details, see [Table 1](#). The applications demonstrate the usage of the Glow AOT in several use cases.

Table 1. List of example applications

Name	Boards	Description
glow_cifar10	RT1050/RT1060/RT1064/ RT1160/RT1170	CIFAR-10 classification of 32x32 RGB pixel images into 10 categories using a small convolutional neural network (CNN).
glow_lenet_mnist	RT1050/RT1060/RT1064/ RT1160/RT1170	Performs handwritten digit classification using the LeNet neural network trained on MNIST database.
glow_cifar10	RT600	CIFAR-10 classification of 32x32 RGB pixel images into 10 categories using a small convolutional neural network (CNN). This project is using the HiFi-NN firmware. Requires FreeRTOS on Cortex-M33.

For details on how to build and run the example applications with supported toolchains, see the *Getting Started with MCUXpresso SDK* document. When using MCUXpresso IDE, the example applications can be imported through the SDK Import Wizard as shown in [Figure 10](#).

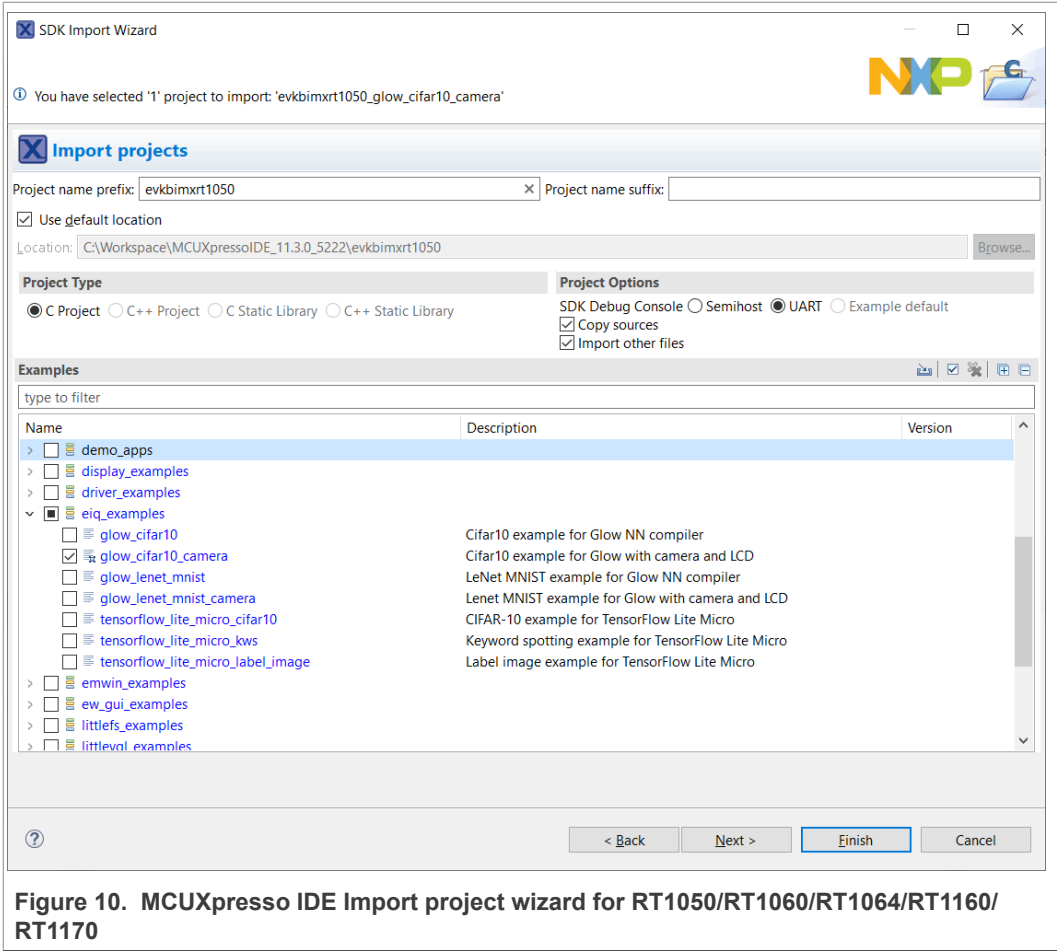


Figure 10. MCUXpresso IDE Import project wizard for RT1050/RT1060/RT1064/RT1160/RT1170

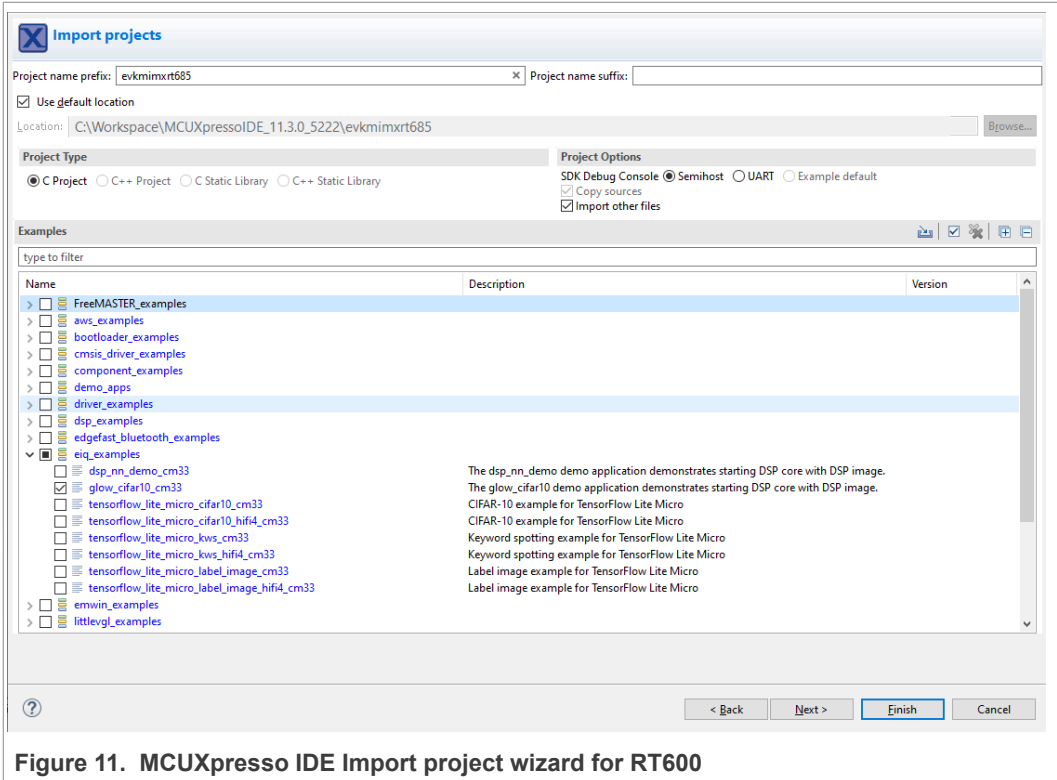


Figure 11. MCUXpresso IDE Import project wizard for RT600

Each example application contains a `readme.txt` file in the path `<project_folder>\doc` which describes the required hardware and the steps required to build and run the application. After building the example application and downloading it to the target, the execution stops in the `main` function. When the execution is resumed, an output message should be displayed on the connected terminal. For example, [Figure 12](#) shows the output of the `glow_lenet_mnist` example application for RT1050 printed to the serial client window when UART is selected in the SDK Import Wizard.



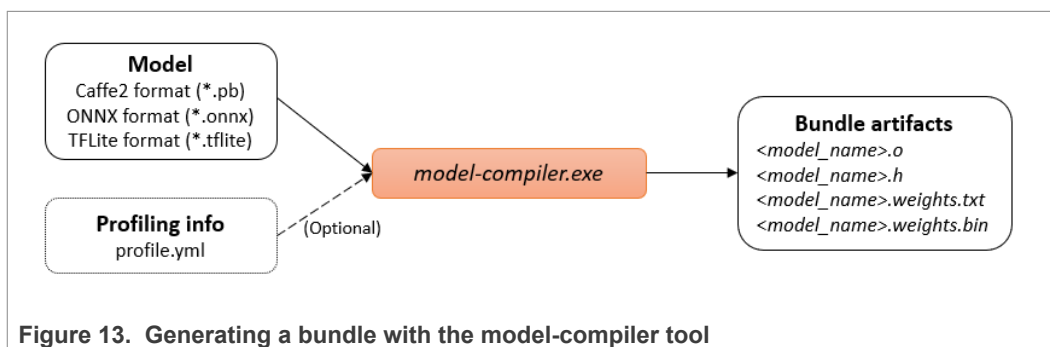
Figure 12. Serial connection window

4 Model compilation

- [Section 4.1](#)
- [Section 4.2](#)
- [Section 4.3](#)

4.1 Bundle generation

Bundle generation represents the model compilation to a binary object file (bundle). Bundle generation is performed using the `model-compiler` tool as shown in [Figure 13](#).



It is possible to generate both floating point and quantized bundles using the `model-compiler` tool. Generating quantized bundles is recommended as it can significantly reduce both the inference time and the memory footprint of the application.

The Glow compiler has an LLVM backend and is capable to cross-compile bundles for different target architectures. The following compile options apply when targeting some of the basic architectures of interest. Later this document explains how to significantly increase the performance by utilizing Arm CMSIS-NN and Cadence NN Library.

- For cross-compiling a bundle for the Arm Cortex M7 core from the i.MX RT1050/RT1060/RT1064/RT1160/RT1170 board: `-target=arm -mcpu=cortex-m7 -float-abi=hard`
- For cross-compiling a bundle for the Arm Cortex M33 core from the i.MX RT 685 board: `-target=arm -mcpu=cortex-m33 -float-abi=hard`

The `model-compiler` tool can be used to:

- Compile a **float32** model to a **float32** bundle.
- Compile a pre-quantized **int8** model to an **int8** bundle, using for example TensorFlow Lite models quantized during training with quantization aware training (QAT).
- Quantize and compile a **float32** model to an **int8** bundle using post-training quantization through profiling.

4.1.1 Compile a float32 model to a float32 bundle

In [Section 4.1.1.1](#) there is a sample command line that generates a **floating-point** bundle for the LeNet Caffe2 model. The model (files `init_net.pb` and `predict_net.pb`) is stored in the folder `models\lenet_mnist` and the bundle is saved in the folder `bundle` specified with the `emit-bundle` option. You can download the model files for LeNet from the following links:

- http://fb-glow-assets.s3.amazonaws.com/models/lenet_mnist/predict_net.pb
- http://fb-glow-assets.s3.amazonaws.com/models/lenet_mnist/init_net.pb

Parameter `-model-input` is used to specify the name of the input tensor (*data*), the type (*float*) and the shape (*[1,1,28,28]*). [Section 4.1.1.1](#) assumes that the target architecture is the Arm Cortex M7 core.

The parameter `-model-input` is only required for Caffe2 models which do not incorporate information about the inputs of the model. For ONNX and TensorFlowLite models the parameter is not required.

4.1.1.1 Example: Compile a float32 model to a float32 bundle

```
model-compiler.exe ^
    -model=models\lenet_mnist -model-input=data,float,
[1,1,28,28] -emit-bundle=bundle ^
    -backend=CPU -target=arm -mcpu=cortex-m7 -float-
abi=hard
```

4.1.2 Compile an int8 model to an int8 bundle

You can use the tool `model-compiler` to compile pre-quantized int8 or uint8 TensorFlowLite models with either:

- Asymmetric quantization for activations, asymmetric per-tensor quantization for weights
- Asymmetric quantization for activations, symmetric per-channel quantization for weights

Note: Models with uint8 precision will be converted internally by Glow to int8. Therefore the bundle generated for uint8 will be always int8 so you should be careful how to provide the input data to the bundle or how to consume the output data from the bundle.

For example you can download a MobileNet v1 model in TensorFlowLite format from [here](#) and compile it using the following command:

4.1.2.1 Example: Compile an int8 model to an int8 bundle

```
model-compiler.exe ^
    -model=mobilenet_v2_1.0_224_quant.tflite -emit-
bundle=bundle ^
    -backend=CPU -target=arm -mcpu=cortex-m7 -float-abi=hard
```

Note: When compiling a pre-quantized int8 model, the `model-compiler` tool should not be provided with any quantization parameters (`quantization-schema`, `quantization-precision`, `quantization-precision-bias` or `enable-channelwise`) because these parameters are determined by the model and cannot be modified. This is because re-quantization of a model with other quantization parameters is a process which loses significant model accuracy.

4.1.3 Compile a float32 model to an int8 bundle

To generate a **quantized** bundle, the profiling information is required for the model. To generate the model profile (`profile.yml`) see the instructions in [Model Profiling](#).

[Section 4.1.3.1](#) shows an example of generating a quantized bundle by loading the profile `profile.yml` assumed available in the current directory (note the use of `-load-`

profile=profile.yml). By default, quantization is performed according to an asymmetric 8-bit schema.

4.1.3.1 Example: Compile a float32 model to an int8 bundle

```
model-compiler.exe ^
    -model=models\lenet_mnist -model-input=data,float,
[1,1,28,28] -emit-bundle=bundle ^
    -backend=CPU -target=arm -mcpu=cortex-m7 -float-
abi=hard ^
    -load-profile=profile.yml
```

When generating a quantized bundle, you can choose from the following options.

- **quantization-schema:** specifies the quantization schema:
 - asymmetric (Default)
 - symmetric
 - symmetric_with_uint8
 - symmetric_with_power2_scale
- **quantization-precision:** specifies the precision used for quantized the nodes:
 - Int8 (Default)
 - Int16
- **quantization-precision-bias:** specifies the precision used to quantize the bias operand of some of the nodes (Convolution, Fully Connected):
 - Int32(Default)
 - Int8
- **enable-channelwise:** specifies whether the quantization is done using per-channel quantization (by default is per-tensor quantization when this option is not used)

4.1.4 Compile with CMSIS-NN and HiFi-NN optimizations

Parameter *-use-cmsis* instructs the compiler to generate a bundle that uses the kernel implementations from the CMSIS-NN library.

4.1.4.1 Example: Generating quantized bundle with CMSIS-NN

```
model-compiler.exe ^
    -model=models\lenet_mnist -model-input=data,float,
[1,1,28,28] -emit-bundle=bundle ^
    -backend=CPU -target=arm -mcpu=cortex-m7 -float-abi=hard ^
    -load-profile=profile.yml ^
    -use-cmsis
```

Note: CMSIS-NN is a library developed for Arm Cortex M4, M7 and M33 series implementing the following NN operations: Convolution, Fully Connected, Pooling and Activation layers. The library implementations are available for both float32 and int8 quantized models with asymmetric schema with both per-tensor or per-channel quantization.

For the i.MX RT 685 board which has an additional HiFi DSP core, we can instruct the compiler to use the HiFi-NN firmware for DSP acceleration with the flag *-use-hifi*. The generated bundle will run natively on the Arm Cortex M33 core but will dispatch NN operations to the HiFi DSP. In [Section 4.1.4.2](#) we have the command for building

the floating-point bundle from Example 1 but using the HiFi-NN firmware support for acceleration.

4.1.4.2 Example: Generating floating-point bundle with HiFi-NN

```
model-compiler.exe ^
  -model=models\lenet_mnist -model-input=data,float,
  [1,1,28,28] -emit-bundle=bundle ^
  -backend=CPU -target=arm -mcpu=cortex-m33 -float-abi=hard ^
  -use-hifi
```

In [Section 4.1.4.3](#) we have the command for building the quantized bundle from [Section 4.1.4.4](#) but using the HiFi-NN firmware support for acceleration.

4.1.4.3 Example: Generating quantized bundle with HiFi-NN

```
model-compiler.exe ^
  -model=models\lenet_mnist -model-input=data,float,
  [1,1,28,28] -emit-bundle=bundle ^
  -backend=CPU -target=arm -mcpu=cortex-m33 -float-
  abi=hard ^
  -load-profile=profile.yml ^
  -quantization-schema=symmetric_with_power2_scale ^
  -quantization-precision-bias=Int8 -use-hifi
```

Note: The HiFi-NN firmware is a software interface used by the Arm core to dispatch NN operations to the HiFi4 DSP. The DSP has a library with kernel implementations for NN operations. The library features both floating-point and quantized implementations using the symmetric_with_power2_scale quantization schema and int8 precision for the bias or asymmetric quantization schema with -enable-channelwise flag and int32 precision for the bias of Convolution and Fully Connected operators. Therefore, use the compilation flag -use-hifi only when building floating-point bundles or quantized bundles with the mentioned schema for the i.MX RT 685.

To run inference on DSP standalone (with HiFi-NN optimizations), generate a Glow bundle using xt-clang compiler version with -use-hifi flag and insert the .o and .h files in your DSP project in "glow_bundle" folder.

For specifying the xt-clang as external compiler, use these flags for model-compiler:

```
-llvm-compiler=<path_to_xtensa>\XtDevTools\install\tools
\RI-2020.5-win32\XtensaTools\bin\xt-clang.exe
-llvm-compiler-opt="-mlsp=<path_to_RT600_SDK>
\devices\MIMXRT685S\xtensa\sim -c --xtensa-
core=nxp_rt600_RI2020_5_newlib"
```

In [Section 4.1.4.4](#) we have the command for building the quantized bundle using both the HiFi-NN firmware and the CMSIS-NN support for acceleration.

4.1.4.4 Example: Generating quantized bundle with HiFi-NN and CMSIS-NN

```
model-compiler.exe ^
  -model=models\lenet_mnist -model-input=data,float,
  [1,1,28,28] -emit-bundle=bundle ^
  -backend=CPU -target=arm -mcpu=cortex-m33 -float-abi=hard ^
  -load-profile=profile.yml ^
```

```
-quantization-schema=symmetric_with_power2_scale ^
-quantization-precision-bias=Int8 -use-hifi -use-cmsis
```

While generating the bundle using model-compiler you can also dump the visual representation of the graph using the option `-dump-graph-DAG=graph.dot` which exports the graph representation in DOT format. You can find details about how to convert the DOT format in other formats in section [Section 6.2](#).

After running any of the above commands the model-compiler produces the following set of files:

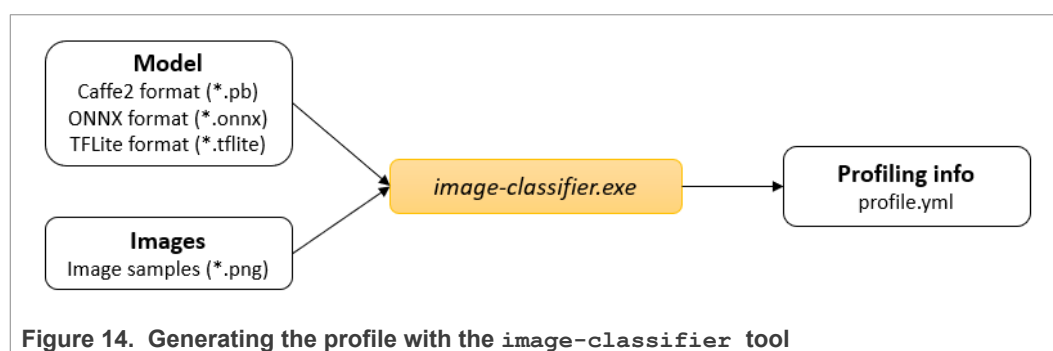
- `<model_name>.o` – the bundle object file.
- `<model_name>.h` – the bundle header file (API).
- `<model_name>.weights.txt` – the weights of the model as C array data.
- `<model_name>.weights.bin` – the weights of the model stored in binary format.

The model parameters (weights) are provided in two formats:

- `.txt` (text format) suitable to initialize the weights statically (at compile time) in the application using the `"#include"` preprocessor directive.
- `.bin` (binary format) suitable to initialize the weights dynamically (at runtime) by the application using standard library functions like `"fread"`.

4.2 Model profiling

Glow uses profile guided quantization, running inference to extract statistics regarding possible numeric values of each tensor within the neural network. Later, during model compilation, Glow uses these statistics to quantize the model. The profiling procedure is independent of the target architecture and therefore no architecture specifications are needed. The profiling information required to quantize the model can be obtained using the `image-classifier` tool as shown in [Figure 14](#).



To obtain the model profile, use the `image-classifier` tool as shown in [Section 4.2.1](#). The profiling procedure requires a set of image examples to run the inference and derive the dynamic ranges of all the values involved in inference computations in order to choose the optimal quantization parameters. Download the images provided in the following link (MNIST dataset examples from the official Glow repository) into a new folder named `images`:

- <https://github.com/pytorch/glow/tree/master/tests/images/mnist>

The following command performs profiling for the LeNet model stored in folder `models\lenet_mnist` (files [init_net.pb](#) and [predict_net.pb](#)) and stores the profile in file

`profile.yml`. Parameter `-model-input-name` is used to specify the name of the input layer name (`data`).

4.2.1 Example: Profiling a model using image-classifier tool

```
image-classifier.exe ^
  -input-image-dir=images ^
  -image-mode=0to1 ^
  -image-layout=NCHW ^
  -image-channel-order=BGR ^
  -model=models\lenet_mnist ^
  -model-input-name=data ^
  -dump-profile=profile.yml
```

When generating the model profile, it is important to preprocess the input images the same way they were processed when training the model. See below some parameters of the image-classifier tool that control the image preprocessing:

- `-image-mode`: specifies the values range for the input tensor:
 - `neg1to1` for values in $[-1, 1]$
 - `0to1` for values in $[0, 1]$
 - `neg128to127` for values in $[-128, 127]$
 - `0to255` for values in $[0, 255]$
- `-image-layout`: specifies the layout to use:
 - NHWC (channel is inner-most dimension, channels are stored in memory with stride 1)
 - NCHW (width is inner-most dimension, widths are stored in memory with stride 1)
- `-image-channel-order`: specifies the order of the channels:
 - RGB
 - BGR

The profiling dataset should be chosen carefully based on the following considerations:

- The dataset should contain at least one image from each class. Rule of thumb would be to use in the order of 10's of images from each class.
- Better still the dataset should contain at least one representative image from each class, that is one image which yields a high confidence for that class.

Note: It is important to note that the profiling phase is independent on the target or the quantization parameters so there is no need to specify the quantization schema, precision or other parameters during this phase. The profiling file `profile.yml` is obtained in the same way regardless of the target or the quantization parameters used during model compilation and can be reused for all quantization methods. Also note that the profiling phase is mandatory when quantizing models.

4.3 Model tuning

When generating the quantization profile for the first time using the `image-classifier` tool, the quantization parameters might not be the optimal ones in terms of model accuracy. When computing the quantization parameters, the `image-classifier` tool chooses the maximum dynamic ranges for the quantized tensors such that no saturation occurs. This means that the quantization step is the largest possible. The reality is that there are tensors for which most of the values are concentrated within a narrow range while also having a couple of outlier values. For these kinds of tensors, it would be

better to narrow down the dynamic range to have a finer representation (with smaller quantization step) for most values while saturating the outlier values.

For this purpose, we have the `model-tuner` tool which takes a model, an input quantization profile (obtained initially with the `image-classifier` tool), and a labeled data set (a set of pairs of images and classification labels) and optimizes (tunes) the quantization profile for maximum accuracy. The optimized quantization profile (named `profile_tuned.yml`) can be further used by the `model-compiler` tool to compile the model according to the best quantization strategy. The `model-tuner` flow is shown in [Figure 15](#).

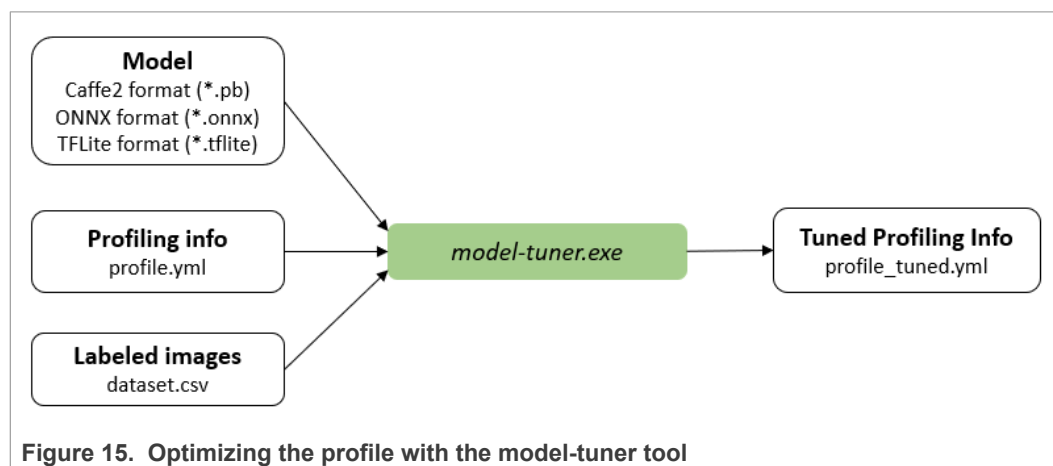


Figure 15. Optimizing the profile with the model-tuner tool

For example, in order to optimize the quantization for the model used in previous sections, we can use the following command:

4.3.1 Tuning the quantization of a model without CMSIS-NN or HIFI-NN

```

model-tuner.exe ^
  -dataset-file=dataset.csv ^
  -dataset-path=<path> ^
  -image-mode=0to1 ^
  -image-layout=NCHW ^
  -image-channel-order=BGR ^
  -model=models\lenet_mnist ^
  -model-input=data,float,[1,1,28,28] ^
  -load-profile=profile.yml ^
  -dump-tuned-profile=profile_tuned.yml ^
  -backend=CPU
  
```

4.3.2 Example: Tuning the quantization of a model for CMSIS-NN or HiFi-NN

```

model-tuner.exe ^
  -dataset-file=dataset.csv ^
  -dataset-path=<path> ^
  -image-mode=0to1 ^
  -image-layout=NCHW ^
  -image-channel-order=BGR ^
  -model=models\lenet_mnist ^
  -model-input=data,float,[1,1,28,28] ^
  -load-profile=profile.yml ^
  
```



```
-dump-tuned-profile=profile_tuned.yml ^
-backend=CPU ^
-quantization-precision-bias=Int8 ^
-quantization-schema=symmetric_with_power2_scale
```

The labeled dataset description file **dataset.csv** is a text file created by the user which has on each line an entry with an image name and an image classification label separated by space (" ") or comma (","). The classification labels are assumed to start with 0 (0, 1, ...) just as the C index variables. Below we have an example of such a dataset file (the `dataset.csv` file in the example above) where the fields are comma separated (CSV file).

4.3.3 Example: Format of the labeled dataset file

```
image0.png,0,
image1.png,5,
image2.png,9,
.....
```

As seen, the dataset file only contains the image names. In order for the tool to localize the images, the second parameter `dataset-path` is given which is the base directory where all the images can be found.

For consistency, you can create a common folder named `dataset_tuning` where you can store the dataset file `dataset.csv` and all the images. Thus all the tuning related files are found in the same place.

Note: The tuning procedure takes a long time to complete. Depending on the model size (number of tensors), the procedure might complete in hours or even days. In order to reduce the procedure time, one might choose to stop the tuning when a given accuracy has been reached, otherwise the tuning runs until completion. For example, you can choose to stop the tuning when the accuracy has reached the value 0.95 (95 %) by setting the `target-accuracy` flag:

4.3.4 Example: Tuning the quantization of a model for CMSIS-NN or HiFi-NN

```
model-tuner.exe ^
-target-accuracy=0.95 ^
-dataset-file=dataset.csv ^
-dataset-path=<path> ^
-image-mode=0to1 ^
-image-layout=NCHW ^
-image-channel-order=BGR ^
-model=models\lenet_mnist ^
-model-input=data,float,[1,1,28,28] ^
-load-profile=profile.yml ^
-dump-tuned-profile=profile_tuned.yml ^
-backend=CPU ^
-quantization-precision-bias=Int8 ^
-quantization-schema=symmetric_with_power2_scale
```

When running the above command the console output might look like this:

4.3.5 Example: Command line output for tuning

```
Computing initial accuracy ...
Initial accuracy: 100.0000 % (FLOAT)
Initial accuracy: 100.0000 % (QUANTIZED)
Target accuracy: 95.0000 % (QUANTIZED)
Number of tensors: 25
Target accuracy achieved! Tuning is stopped ...
Final accuracy: 100.0000 % (QUANTIZED)
Total time: 0 hours 0 minutes
```

Note: It is important to note that during the tuning procedure you must provide the same quantization parameters as the ones which will be later used during the compilation phase when using the model-compiler. This is because the tuning procedure optimizes the profiling information according to the specified quantization parameters. Also note that the tuning phase is optional when quantizing models.

5 Creating the application project

The easiest way to create an application with the proper configuration is to copy and modify an existing example application. For details on where to find the example applications and how to build them, see [Section 3](#).

5.1 Bundle API

The Bundle API defines the interface between the user application and the model compiled with Glow AOT. Each bundle has its own API which is generated when the model is compiled.

The most important component of the API is the entry point, the function that performs the model inference. The entry point has the same name as the model. The user may override the model name using the `-network-name` parameter of the `model-compiler` tool.

Besides the entry-point, the bundle API contains a series of macros that help in allocating and preparing the memory buffers for inference. In [Section 5.1.1](#) you can see the actual bundle API header generated for the LeNet model compiled in the previous steps.

- `<MODEL_NAME>_CONSTANT_MEM_SIZE` – defines the size of the constant memory (contains the weights of the model)
- `<MODEL_NAME>_MUTABLE_MEM_SIZE` – defines the size of the mutable memory (contains the inputs and outputs of the model)
- `<MODEL_NAME>_ACTIVATIONS_MEM_SIZE` – defines the size of the memory buffers that is internally used for storing intermediate values (activations) during inference; buffer is used as a scratch buffer and hence can be reutilized by the user code
- `<MODEL_NAME>_MEM_ALIGN` – defines the memory alignment requirement for all the allocated buffers
- `<MODEL_NAME>_<placeholder_name>` – defines the offset of the model placeholder `<placeholder_name>` within the mutable memory area. In simple terms the placeholders of a graph/model are the overall inputs and outputs.

5.1.1 Example: Bundle API example

```
// Bundle API auto-generated header file. Do not edit!
// Glow Tools version: 2020-09-28
#ifndef _GLOW_BUNDLE_LENET_MNIST_H
#define _GLOW_BUNDLE_LENET_MNIST_H
#include <stdint.h>
//
//-----
//                                     Common definitions
//
//-----
#ifndef _GLOW_BUNDLE_COMMON_DEFS
#define _GLOW_BUNDLE_COMMON_DEFS
// Glow bundle error code for correct execution.
#define GLOW_SUCCESS 0
// Memory alignment definition with given alignment size
// for static allocation of memory.
#define GLOW_MEM_ALIGN(size) __attribute__((aligned(size)))
// Macro function to get the absolute address of a
// placeholder using the base address of the mutable
// weight buffer and placeholder offset definition.
#define GLOW_GET_ADDR(mutableBaseAddr, placeholderOff) \
    (((uint8_t*)(mutableBaseAddr)) + placeholderOff)
#endif
//
//-----
//                                     Bundle API
//
//-----
// Model name: "lenet_mnist"
// Total data size: 1785408 (bytes)
// Placeholders:
//
//   Name: "softmax"
//   Type: float<1 x 10>
//   Size: 10 (elements)
//   Size: 40 (bytes)
//   Offset: 3136 (bytes)
//
//   Name: "data"
//   Type: float<1 x 1 x 28 x 28>
//   Size: 784 (elements)
//   Size: 3136 (bytes)
//   Offset: 0 (bytes)
//
// NOTE: Placeholders are allocated within the "mutableWeight"
// buffer and are identified using an offset relative to base.
//
//-----
#ifdef __cplusplus
extern "C" {
#endif
// Placeholder address offsets within mutable buffer (bytes).
#define LENET_MNIST_softmax 3136
#define LENET_MNIST_data 0
// Memory sizes (bytes).
#define LENET_MNIST_CONSTANT_MEM_SIZE 1724608
#define LENET_MNIST_MUTABLE_MEM_SIZE 3200
#define LENET_MNIST_ACTIVATIONS_MEM_SIZE 57600
```

```
// Memory alignment (bytes).
#define LENET_MNIST_MEM_ALIGN 64
// Bundle entry point (inference function). Returns 0
// for correct execution or some error code otherwise.
int lenet_mnist(uint8_t *constantWeight, uint8_t
    *mutableWeight, uint8_t *activations);
#ifdef __cplusplus
}
#endif
#endif
```

5.2 Integrating the bundle

The bundle consists in the following set of files:

- `<model_name>.o` – the bundle object file
- `<model_name>.h` – the bundle header file (API)
- `<model_name>.weights.txt` – the weights of the model as C array data
- `<model_name>.weights.bin` – the weights of the model stored in binary format

To integrate the bundle, include the first three generated files in the project. After copying these files in the *source* folder of the project, perform the following steps.

1. Change the project properties and modify the linker options such that it includes the bundle object file when linking the application. To modify the linker options:
 - **For MCUXpresso IDE**
 - a. Right-click the project and select "Properties".
 - b. Select "C/C++ Build" > "Settings".
 - c. In the "Tool Setting" tab, select "MCU C++ Linker" > "Miscellaneous".
 - d. Add the bundle to "Other objects".
 - e. Click "Add..." and specify the relative path to the bundle in the project. That is, `../source/<model_name>.o`.
 - **For IAR Embedded Workbench**
 - a. Right-click the project and select "Add" > "Add Files ...".
 - b. Choose the file category "Library/Object Files (*.r;*.a;*.lib;*.o)".
 - c. Select the bundle `<model_name>.o`.
 - d. Click "Open".
 - **For Keil uVision MDK**
 - a. Right-click the target and select "Options for Target ...".
 - b. Select the "Linker" tab.
 - c. In the "Misc controls" section, add to the existing string with a separating space character the relative path to the bundle (that is, `../source/<model_name>.o`).
2. Include the `glow_bundle_utils.h` header file and the bundle API header file in the application main source file. For example, `main.cpp`. Assuming that the model is compiled as in [Example: Including bundle](#), the header file is named `lenet_mnist.h`.

Example: Including bundle API

```
#include "lenet_mnist.h"
#include "glow_bundle_utils.h"
```

3. Declare the buffers for *constant weights*, *mutable weights*, and *activations* in the application main source file. For example, `main.cpp`. Initialize the constant weights with values included from the `.txt` file.

Example: Declaring and initializing the memory buffers file

```
// Statically allocate memory for constant weights (model
weights) and initialize.
GLOW_MEM_ALIGN(LENET_MNIST_MEM_ALIGN)
uint8_t constantWeight[LENET_MNIST_CONSTANT_MEM_SIZE] = {
#include "lenet_mnist.weights.txt"
};
// Statically allocate memory for mutable weights (model
input/output data).
GLOW_MEM_ALIGN(LENET_MNIST_MEM_ALIGN)
uint8_t mutableWeight[LENET_MNIST_MUTABLE_MEM_SIZE];
// Statically allocate memory for activations (model
intermediate results).
GLOW_MEM_ALIGN(LENET_MNIST_MEM_ALIGN)
uint8_t activations[LENET_MNIST_ACTIVATIONS_MEM_SIZE];
```

Note: In this sample, we have opted for static allocation of the buffers. If dynamic allocation is necessary, you must pay attention to memory alignment. For this purpose, we provide the `alignedAlloc` function declared in header `glow_bundle_utils.h`. The function takes as parameters the required alignment (in bytes) and the requested allocation size. The function returns a pointer to the allocated memory buffer on success and `NULL` on error. Use function `alignedFree` to deallocate memory previously allocated with `alignedAlloc`.

When initializing the **constant weights** buffer using the `*.weights.txt` file with the `#include` pre-processor directive, the compilation of the project might fail for large models. This occurs when the file `*.weights.txt` has a large size (hundreds of megabytes) resulting in the compiler remaining out of host memory and crashing while preprocessing the large text file. For example, in MCUXpresso IDE, such a compile-time error might look like this:

```
ccl.exe: out of memory allocating 268439551 bytes
make: *** [source/subdir.mk:43: source/main_v1.o] Error 1
"make -r -j8 all" terminated with exit code 2. Build might be
incomplete.
```

One way to solve this error is to use the binary weights file `*.weights.bin` instead of the text file by encapsulating it in an assembly source file (named for example `include_weights.s`) and adding the file to the project source code. The assembly file might look like this:

```
#if defined(__GNUC__)
.section .weights, "ax" @progbits @preinit_array
.global constantWeight
.type constantWeight, %object
.align 6
constantWeight:
.incbn "lenet_mnist.weights.bin"
.end
#endif
```

The above assembly source code defines a memory section named `.weights` in which a global variable named `constantWeight` (same as before) is allocated with an alignment of 64 bytes (2 to the power of 6) and is initialized with the content of the binary file `lenet_mnist.weights.bin` using the `.incbn` assembly directive. After this source code is added to project, the global variable can be referenced from a C source file by adding the following declaration:

```
extern uint8_t constantWeight[];
```

4. Use `GLOW_GET_ADDR` macro to obtain the absolute address for each of the mutable weights of the model (model inputs & outputs).

Example: Obtaining the addresses for the model input and output

```
// Bundle input data absolute address.
uint8_t *inputAddr = GLOW_GET_ADDR(mutableWeight,
    LENET_MNIST_data);
// Bundle output data absolute address.
uint8_t *outputAddr = GLOW_GET_ADDR(mutableWeight,
    LENET_MNIST_softmax);
```

5. Initialize the model input. In a real scenario, the user code obtains the input data for the inference through an acquisition procedure from sensors. For example, cameras or transmitted from other devices through wired or wireless communication. For simplicity, in this example we provide a buffer `imageData` which is statically initialized with a sample image. The buffer is initialized using the text file `input_image.inc` produced by a Python script by reading, preprocessing, and serializing a sample `png` image. In this example, copy the data from the initialized buffer `imageData` to the `inputAddr` buffer where the inference function expects to find the input. In a real application, the user can avoid this copy by filling directly the buffer pointed to by `inputAddr`.

Example: Application buffer with static allocation and initialization with serialized data

```
uint8_t imageData[] = {
#include "input_image.inc"
};
```

Example: Serializing an image as text file using Python script

```
python scripts\glow_process_image.py ^
    -image-path=model\dataset\9_1088.png ^
    -output-path=source\input_image.inc ^
    -image-mode=0to1 -image-layout=NCHW -image-channel-
order=RGB
```

Example: Initializing the model input

```
memcpy(inputAddr, imageData, sizeof(imageData));
```

Note: When generating the input data, it is important to preprocess it before running inference. In the sample application projects delivered with the MCUXpresso SDK. There is a Python script (`glow_process_image.py`) in the project scripts subfolder that generates the C array representation of the preprocessed input image.

6. Now everything is prepared to run inference by passing the three memory buffers initialized above to the bundle entry-point function.

Example: Calling the inference function

```
lenet_mnist(constantWeight, mutableWeight, activations);
```

7. After running the inference, the results are available in the `outputAddr` buffer. Since LeNet is a hand-written digit classification model, the result is an array of 10 floating-point numbers. Each number representing the confidence score for each of the 10 digits (0 to 9). For convenience, provide the code which prints the class and the confidence score for the most certain class (with maximum confidence, also known as *top1*).

Example: Processing the inference result

```
// Get classification top1 result and confidence
float *out_data = (float*)(outputAddr);
```

```
float max_val = 0.0;
uint32_t max_idx = 0;
for(int i = 0; i < LENET_MNIST_OUTPUT_CLASS; i++) {
    if (out_data[i] > max_val) {
        max_val = out_data[i];
        max_idx = i;
    }
}
// Print classification results
PRINTF("Top1 class = %lu\r\n", max_idx);
PRINTF("Confidence = 0.%.03u\r\n", (int) (max_val*1000));
```

Note: The instructions described above apply for integrating the bundle generated for the LeNet model for RT1050/RT1060/RT1064/RT1160/RT1170 boards. For instructions related to the RT600 board where some minor differences apply you can access the RT685 Lab guide available [here](#). Also you can find additional information about the memory usage for Glow applications [here](#).

5.3 Other optimizations

Since the i.MX RT devices also have available small but fast memory units called TCM (Tightly Coupled Memory), Glow has the capability to use such fast memories to optimize the performance. For example, the i.MX RT 1052 has the following TCM units: DTC (128 kB), ITC (128 kB), and OCRAM (256 kB). To be noted that the smaller memories are commonly faster than the bigger ones.

In order to use TCM, when compiling the bundle using the `model-compiler` tool, the user must provide the information about the number and the size of all the available TCM memories in the decreasing order of their priority (a memory which is listed first is used before the other if the size allows it). This means that the memories should be listed in the decreasing order of their speed for best performance. For example, for the i.MX RT 1052, the `model-compiler` tool is provided with the `-tcm-size` option which specifies the TCM sizes as a comma-separated list of integer values (the sizes are expressed in bytes):

```
-tcm-size=131072,126976,262144
```

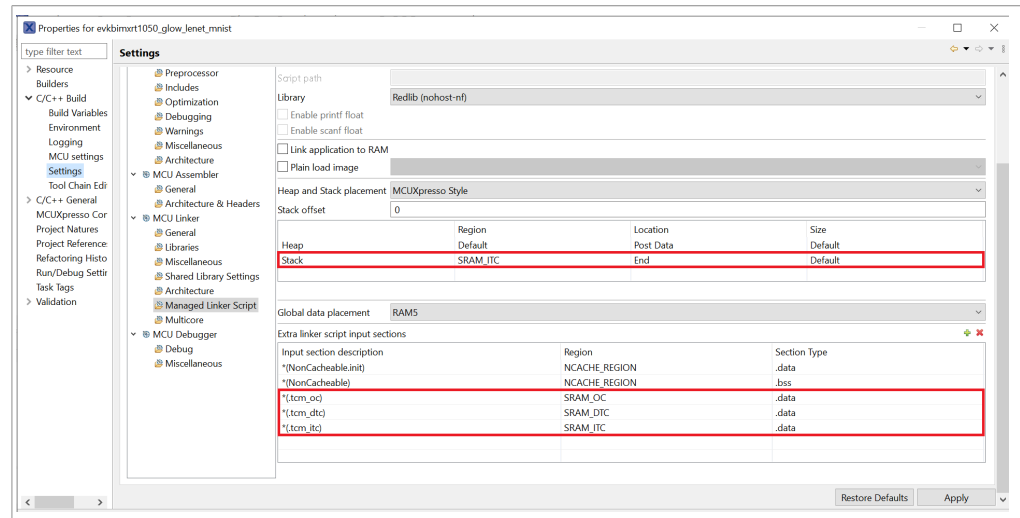
To be noted that the above sizes correspond (in order) to the DTC size (128 kB), the ITC size minus the stack size (128 kB minus 4 kB) and the OCRAM size (256 kB). We do not provide the entire ITC memory to Glow because part of it (4 kB) we use for stack which is recommended in most cases. Glow tries to use DTC first, ITC second, and OCRAM last (depending on whether a given memory can accommodate a given computation).

The generated bundle contains the following extra content in the auto-generated header file `<model_name>.h`:

```
extern uint8_t *glow_tcm_addr[];
```

What this means is that the Glow bundle now expects some information from the user application about the locations (base addresses) for each of the TCM memories through a symbol `glow_tcm_addr` which is an array of raw pointers. In order to provide the base addresses of the TCM memories, the user must perform the following steps (this example is provided only for MCUXpresso IDE although for other IDEs the steps should be similar):

1. Define a section for each TCM memory. In the picture below we can see that we define the sections `.tcm_dtc`, `.tcm_itc`, and `.tcm_oc` for DTC, ITC and OCRAM. We also see that the stack is allocated in ITC (recommended).



2. Allocate one buffer in each section:

```
__attribute__((section(".tcm_dtc")))
uint8_t dtc_memory[128 * 1024];
__attribute__((section(".tcm_itc")))
uint8_t itc_memory[128 * 1024 - 4 * 1024];
__attribute__((section(".tcm_oc")))
uint8_t oc_memory[256 * 1024];
```

3. Provide to the bundle the base address of each TCM memory by defining the `glow_tcm_addr` array of pointers.

```
uint8_t *glow_tcm_addr[] = {dtc_memory, itc_memory,
                             oc_memory};
```

The TCM usage configuration presented above is an example of how to use the maximum amount of TCM available and also how to place the stack on the ITC for the i.MX RT 1050 board. For practical reasons, we do not need to use all the TCM to have best performance and also not always placing the stack on the ITC provides best results. [Table 2](#) provides the recommended usage configuration for each board.

Table 2. Recommended usage configuration

Boards	Available TCM	Recommended TCM usage	Stack in ITC
i.MX RT 1050	128 kB (DTC) 128 kB (ITC) 256 kB (OCRAM)	128 kB (DTC)	Yes
i.MX RT 1060	128 kB (DTC) 128 kB (ITC) 768 kB (OCRAM)	128 kB (DTC)	No
i.MX RT 1064	128 kB (DTC) 128 kB (ITC) 768 kB (OCRAM)	128 kB (DTC)	No
i.MX RT 1160	256 kB (DTC) 256 kB (ITC) 64 kB (OCRAM1)	256 kB (DTC)	No
i.MX RT 1170	256 kB (DTC) 256 kB (ITC) 512 kB (OCRAM1)	256 kB (DTC)	No

Note:

The TCM optimizations are currently available only for those layers mapped to the CMSIS-NN optimized implementations when using the `-use-cmsis` option. Therefore, use the option `-tcm-size` only when using the option `-use-cmsis`.

6 Utilities

This section describes utilities which can be used to convert, visualize and debug models.

6.1 Model conversion

The Glow compiler currently has support only for Caffe2, ONNX, and TensorFlowLite model formats. Since a lot of well-known models are available in other formats, for example TensorFlow, it might be of interest to have some tools to convert models between different formats. The most used tools for format conversion are MMDNN and tf2onnx:

- MMDNN: <https://github.com/Microsoft/MMdnn>
- tf2onnx: <https://github.com/onnx/tensorflow-onnx>

We will exemplify how to convert a TensorFlow model to ONNX using the MMDNN tool. We will convert a MobileNet V1 image classification model which operates on 128 x 128 RGB images and 1001 classes. Download the MobileNet V1 model archive from here:

- http://download.tensorflow.org/models/mobilenet_v1_2018_08_02/mobilenet_v1_0.25_128.tgz

After you install MMDNN run the following command to convert the TensorFlow frozen file `mobilenet_v1_0.25_128_frozen.pb` to the ONNX model file `mobilenet_v1_0.25_128_frozen_2018.onnx`.

6.1.1 Example: Convert model from TensorFlow format to ONNX using MMDNN

```
mmconvert ^
-sf tensorflow ^
-iw mobilenet_v1_0.25_128_frozen.pb ^
--inNodeName input ^
--inputShape 128,128,3 ^
--dstNodeName MobilenetV1/Predictions/Softmax ^
-df onnx ^
-om mobilenet_v1_0.25_128_frozen_2018.onnx
```

You can find additional models in the links below, either directly in ONNX format or other formats which can be converted to ONNX using the conversion tools previously mentioned.

ONNX Model Zoo: <https://github.com/onnx/models>

MobileNetV1: https://github.com/tensorflow/models/blob/master/research/slim/nets/mobilenet_v1.md

MobileNetV2: <https://github.com/tensorflow/models/tree/master/research/slim/nets/mobilenet>

6.2 Model visualizer

A very popular tool for visualizing the **original model** before compiling with Glow is **Netron** which has an online browser version here: <https://lutzroeder.github.io/netron/>. In order to use Netron drag and drop the model file into the browser window.

The Glow compiler integrates the *graphviz* utility for exporting the graph visual representation of the **compiled model** in *dot* format. The graph will depict all the optimizations and conversions performed on the original model by Glow including the node specializations when using CMSIS-NN or HIFI-NN. Note that the compile command from [Section 4.1.1.1](#) but with the addition of the *-dump-graph-DAG=graph.dot* option which exports the graph visual representation in the file *graph.dot* as presented below.

6.2.1 Example: Dump model graph visual representation to DOT file

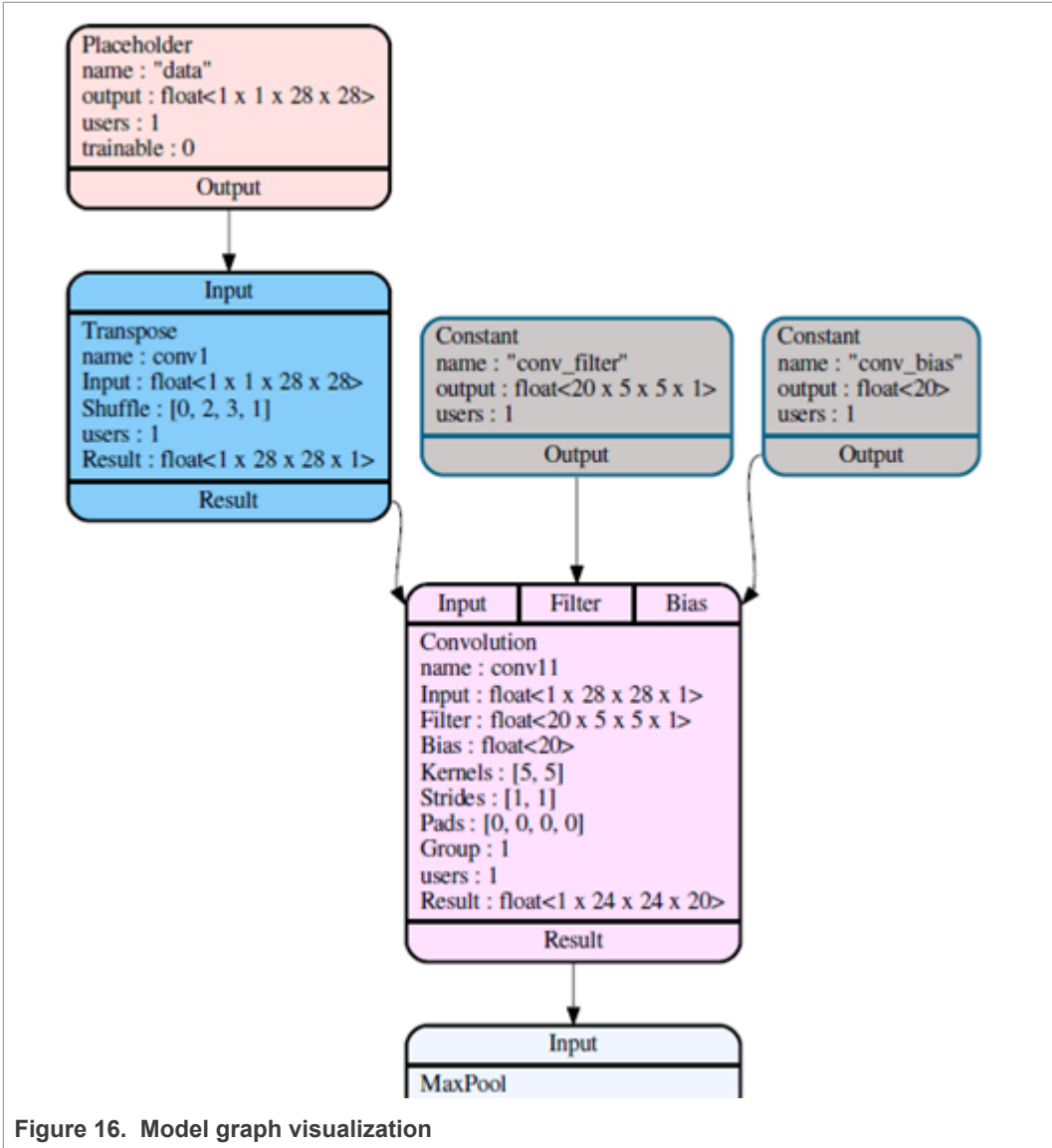
```
model-compiler.exe ^  
  -model=models\lenet_mnist -model-input=data,float,  
  [1,1,28,28] -emit-bundle=bundle ^  
  -backend=CPU -target=arm -mcpu=cortex-m7 -float-abi=hard ^  
  -dump-graph-DAG=graph.dot
```

The DOT format is a text description file which can be used to generate visual representations of the graph. We can use the “dot.exe” utility (which is installed together with the Glow tools for Windows) to convert the DOT file to PDF or PNG file formats as depicted below.

6.2.2 Example: Convert graph DOT format to PDF/PNG format

```
dot -Tpdf graph.dot -o graph.pdf -Nfontname="Times New Roman,"  
dot -Tpng graph.dot -o graph.png -Nfontname="Times New Roman,"
```

The model graph representation for LeNet generated as PDF file might be as shown in [Figure 16](#).



7 Note about the source code in the document

Example code shown in this document has the following copyright and BSD-3-Clause license:

Copyright 2019 NXP Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

8 Revision history

[Table 3](#) summarizes the changes done to the document since the initial release.

Table 3. Revision history

Revision number	Date	Substantive changes
0	01 September 2019	Initial release with CMSIS-NN support
1	15 April 2020	Updated for HiFi-NN support
2	01 October 2020	Updated for i.MX RT1170 support
3	25 November 2021	Updated for MCUXSDK 2.9.0
4	10 July 2021	Updated for MCUXSDK 2.10.0
5	19 December 2021	Updated for MCUXSDK 2.11.0
6	01 June 2022	Updated for MCUXSDK 2.12.0

9 Legal information

9.1 Definitions

Draft — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

9.2 Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification. Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Terms and conditions of commercial sale — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at <http://www.nxp.com/profile/terms>, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Suitability for use in non-automotive qualified products — Unless this data sheet expressly states that this specific NXP Semiconductors product is automotive qualified, the product is not suitable for automotive use. It is neither qualified nor tested in accordance with automotive testing or application requirements. NXP Semiconductors accepts no liability for inclusion and/or use of non-automotive qualified products in automotive equipment or applications.

In the event that customer uses the product for design-in and use in automotive applications to automotive specifications and standards, customer (a) shall use the product without NXP Semiconductors' warranty of the product for such automotive applications, use and specifications, and (b) whenever customer uses the product for automotive applications beyond NXP Semiconductors' specifications such use shall be solely at customer's own risk, and (c) customer fully indemnifies NXP Semiconductors for any liability, damages or failed product claims resulting from customer design and use of the product for automotive applications beyond NXP Semiconductors' standard warranty and NXP Semiconductors' product specifications.

Translations — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

Security — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately. Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

9.3 Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

NXP — wordmark and logo are trademarks of NXP B.V.

Contents

1	Overview	2	9	Legal information	29
2	Deployment	2			
3	Example applications	6			
4	Model compilation	10			
4.1	Bundle generation	10			
4.1.1	Compile a float32 model to a float32 bundle	10			
4.1.1.1	Example: Compile a float32 model to a float32 bundle	11			
4.1.2	Compile an int8 model to an int8 bundle	11			
4.1.2.1	Example: Compile an int8 model to an int8 bundle	11			
4.1.3	Compile a float32 model to an int8 bundle	11			
4.1.3.1	Example: Compile a float32 model to an int8 bundle	12			
4.1.4	Compile with CMSIS-NN and HiFi-NN optimizations	12			
4.1.4.1	Example: Generating quantized bundle with CMSIS-NN	12			
4.1.4.2	Example: Generating floating-point bundle with HiFi-NN	13			
4.1.4.3	Example: Generating quantized bundle with HiFi-NN	13			
4.1.4.4	Example: Generating quantized bundle with HiFi-NN and CMSIS-NN	13			
4.2	Model profiling	14			
4.2.1	Example: Profiling a model using image-classifier tool	15			
4.3	Model tuning	15			
4.3.1	Tuning the quantization of a model without CMSIS-NN or HiFi-NN	16			
4.3.2	Example: Tuning the quantization of a model for CMSIS-NN or HiFi-NN	16			
4.3.3	Example: Format of the labeled dataset file	17			
4.3.4	Example: Tuning the quantization of a model for CMSIS-NN or HiFi-NN	17			
4.3.5	Example: Command line output for tuning	18			
5	Creating the application project	18			
5.1	Bundle API	18			
5.1.1	Example: Bundle API example	19			
5.2	Integrating the bundle	20			
5.3	Other optimizations	23			
6	Utilities	25			
6.1	Model conversion	25			
6.1.1	Example: Convert model from TensorFlow format to ONNX using MMDNN	25			
6.2	Model visualizer	26			
6.2.1	Example: Dump model graph visual representation to DOT file	26			
6.2.2	Example: Convert graph DOT format to PDF/PNG format	26			
7	Note about the source code in the document	27			
8	Revision history	28			

Please be aware that important notices concerning this document and the product(s) described herein, have been included in section 'Legal information'.