AN13590

Training Custom Model on eIQ Portal

Rev. 0 — 15 March 2022 Application Note

Contents

1 Introduction

eIQ portal provides a friendly and customizable interface to handle image classification and bounding boxes detection problems.

The model instances constitute the more important piece of the process. They can be selected according to the problem you want to solve. eIQ provides some public models with pre-trained weights for public datasets. To solve classification and detection task, user can retrain/tune. However, in some cases, customer prefers to use their own models for performance or accuracy considerations. eIQ portal can handle this requirement: customer could use plug-in to build and train their own models. This documentation introduces:

- how to bring your own code into eIQ to define your model
- how to use the eIQ Portal to train and evaluate them
- how to reuse the eIQ model tools to quantize and convert for deploying to target platforms with different architectures

2 eIQ portal plug-in

eIQ portal brings a plug-in system that define common interfaces and let plug-ins implement different models. eIQ portal is shipped with some built-in plug-ins, called as base plug-ins. These plug-ins are what the model wizard selects. On the other hand, user plug-ins provide their own models from the installation location outside the tool.

There are two places to store the user define plug-in:

- `DEEPVIEW_BASE_PLUGINS`: `%APPDATA%\eIQ Portal\Plugins`
- `DEEPVIEW_USER_PLUGINS`: Points out to any location chosen by the user for experimental or development purposes.

The base plug-ins are shipped with the tool and this plug-in is what the wizard selects. On the other hand, user plug-ins provide their own models from the installation location outside the tool. When creating a model, the first task or problem type to solve is image classification or object detection. The following sections explain how to create shipped plug-ins.

Take CIFAR10 (project importer: `CIFAR10_uploader.py` shipped with eIQ) as example for building a classification model. We must implement the model class in a python script under classification folder in the `DEEPVIEW_BASE_PLUGINS` (defaults to <eIQ_root>\plugins). In our case, the script is placed in `DEEPVIEW_BASE_PLUGINS`, as shown in Figure 1 (`cifar10.py` is shipped in attachments).

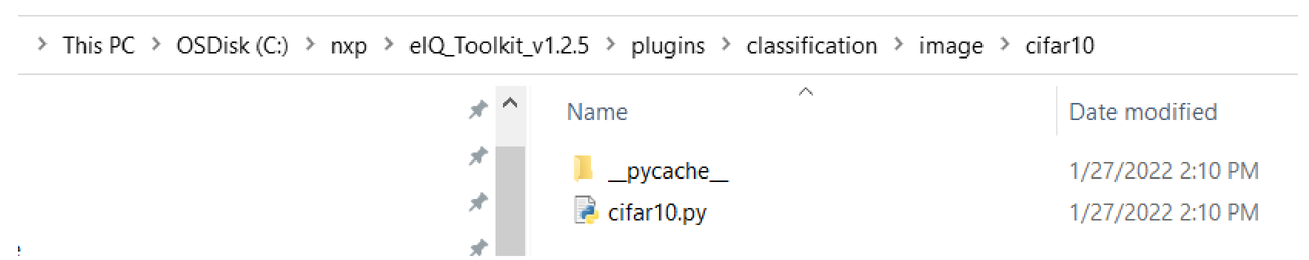

Figure 1. Python script

3 Building custom model class

Classification models inherit from the following class:

- `deepview.trainer.extensions.interfaces.ImageClassificationInterface`

eIQ portal software defines this interface and invokes methods in it to coordinate with any plug-in that implements it.

The `ImageClassificationInterface` class describes the model and provides additional information. It is used in run time to configure the GUI and options the model can handle. Also, this class is used as an interface to handle the plug-in structure and organize the models according to the problem type. The interface handles mandatory functionalities provided by each model. Mandatory function is as below:

```
def get_plugin():
    return cifar10
```

This function is very important because the trainer uses it to retrieve the main class. In this example, it returns the `cifar10` class which handles the classification model.

The model object is implemented in `cifar10` class (`interfaces.ImageClassificationInterface`) which includes some mandatory functions.

- `def get_name` (`self`):

  This function returns the model name. It is shown in eIQ portal.

  Below is the code snippet of implementation of our example:

  ```
  def get_name(self):
    return cifar10
  ```

- `def is_base` (`self`):

  This function returns **True** or **False** if the model belongs to the base plug-ins or not. In the example, the function returns **True**. The code snippet is as below:

  ```
  def is_base(self):
    return True`
  ```

- `def get_model` (`self`, `input_shape`, `num_classes`, `weights`, `named_params={}`):

  This function defines the model object. We received the parameters that help us to build the classification model. We can build a model object or load a pre-trained model. This function returns the model object. The code snippet is as below:

  ```
  def get_model(self, input_shape, num_classes, weights, named_params={}):
     alpha = float(named_params.get('alpha', "0.35"))
     layer = 0
     drop = 0.25
     lstOCs = [32, 32, 64]
     model = tf.keras.Sequential()
          ….
     return model
  ```

- `def get_task` (`self`):

  This function returns the resolved model task. The code snippet is as below:

  ```
  def get_task(self):
     return "classification"
  ```

- `def get_preprocess_function` (`self`):

This function pre-processes the input images. It is passed to a `tf.DataSet` instance to internally handle the pre-fetch preprocessed images. The code snippet is as below:

```
def get_preprocess_function(self):
   return imagenet_utils.preprocess_input(x, data_format=data_format, mode='tf')
```

- `def get_metadata` (`self`, `base_path`):

  This function takes the `base_path` parameter which stores the model path (checkpoint) and returns some additional metadata. This function is used as constants while quantizing or converting the model. In this case, we only return the normalization as signed. This result tells the converter that the model needs a signed normalization during samples calibration in the quantization process. The code snippet is as below:

```
def get_metadata(self, base_path):
    response = {
         "constants": {},
      "params": {
           "normalization": "signed"
       }
   }
   return response
```

- `def get_exposed_parameters` (`self`):

  This method allows the GUI to be configured dynamically by using the returned object list. Notice how each object has the same keys and only the values are changing. In this case, we only introduce two parameters, alpha and optimizer. In your case, you can include anything as required to configure the model. Our GUI is prepared to read that properties and auto-configure it. Remember to parse this parameter inside the `get_model` method. The code snippet is as below:

```
def get_exposed_parameters(self):
   return [
               {
                   "name": "Alpha",
                   "key": "alpha",
                   "default": "0.35",
                   "values": ["0.35", "0.50", "0.75", "1.00", "1.30"],
                   "description": "Controls the width of the network"
                },
                {
                   "name": "Optimizer",
                   "key": "optimizer",
                   "default": "Adam",
                   "values": ["SGD", "Adam", "RMSprop", "Nadam", "Adadelta", "Adagrad",
"Adamax"],
                   "description": "Model optimizer"
                }
         ]
```

- `def get_losses` (`self`):

  Our dataset iterator returns classes in a categorical way so the loss function that we must use is `CategoricalCrossentropy`. Also, you can create loss functions and expose them. This documentation is introduced in further guides. The code snippet is as below:

```
def get_losses(self):
    return ["CategoricalCrossentropy"]
```

- `def get_optimizers` (`self`):

This function is a helper method that returns the default optimizer. In classification task, we use `Adam`. The code snippet is as below:

```
def get_optimizers(self):
    return ["Adam"]
```

- `def get_allowed_dimensions` (`self`):

  This function tells the GUI that our model supports any input dimension larger than or equal to 32. The code snippet is as below:

```
def get_allowed_dimensions(self):
    return ["32", '32']
```

- `def get_pretrained_dimensions` (`self`):

  This function introduces the set of dimensions with pre-trained weights and the source of the weights. In our case, we are only setting `imagenet` as pre-trained weights, but you can set anything there. Remember what you set here can be a possible option in the weights parameter of the `get_model` method. Handle properly inside it. The code snippet is as below:

```
def get_pretrained_dimensions(self):
                    return [["32", 'cifar10']]
```

- `def get_qat_support` (`self`):

  This function tells the GUI whether our model supports Quantization Aware Training or not. If the model supports QAT, we provide the input/output type and the framework where the per-channel or per-tensor quantization is provided. `TensorFlow` only provides per-channel quantization. Our converter can add this feature and create very accurate approximations for per-tensor quantization. The code snippet is as below:

```
def get_qat_support(self):
    return [{
                   # Per-Channel Quantization
                   "supported": "true",
                   "types": ['uint8', 'int8', 'float32'],
                   "frameworks": ['Tensorflow', 'Converter']
              }, {
                   # Per-Tensor Quantization
                   "supported": "true",
                   "types": ['uint8', 'int8', 'float32'],
                   "frameworks": ["Converter"]
         }]
```

- `def get_ptq_support` (`self`):

  This function tells the GUI whether our model supports Post Training Quantization or not. If the model supports PTQ, we can provide the input/output type and the framework where the per-channel or per-tensor quantization is provided. `Tensorflow` only provides per-channel quantization. Our converter can add this feature and create very accurate approximations for per-tensor quantization. The code snippet is as below:

```
def get_ptq_support(self):
   return [{
                   # Per-Channel Quantization
                   "supported": "true",
                   "types": ['uint8', 'int8', 'float32'],
                   "frameworks": ['Tensorflow', 'Converter']
           }, {
                   # Per-Tensor Quantization
                   "supported": "true",
                   "types": ['uint8', 'int8', 'float32'],
```

```
                "frameworks": ["Converter"]
        }]
```

4 Training and validating model

Open eIQ portal and select `cifar10` project. In the model selection window, select **base models** (`cifar10` plug-in placed in `DEEPVIEW_BASE_PLUGINS`). `cifar10` model is shown in the list, as shown in Figure 2.

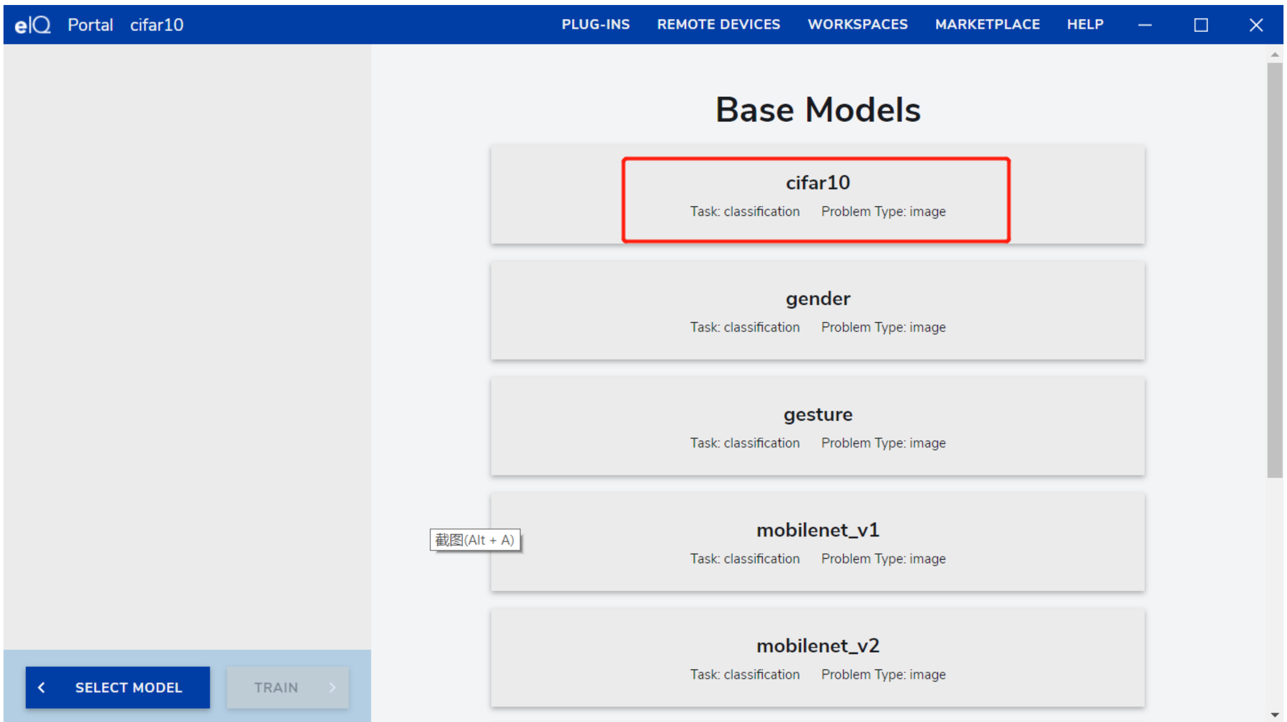

Figure 2. Base models window

Select `cifar10` model and enter the **Trainer** window.

Set the following parameters as shown in Figure 3:

- Set **Learning Rate** to **0.001**.
- Enable **Learning Rate Decay**.
- Set **Decay Rate** to **0.94** and batch size to 50.
- Set **Train Accuracy** to 20 Epoch.
- Raise **Evaluation Accuracy** to 76 %.

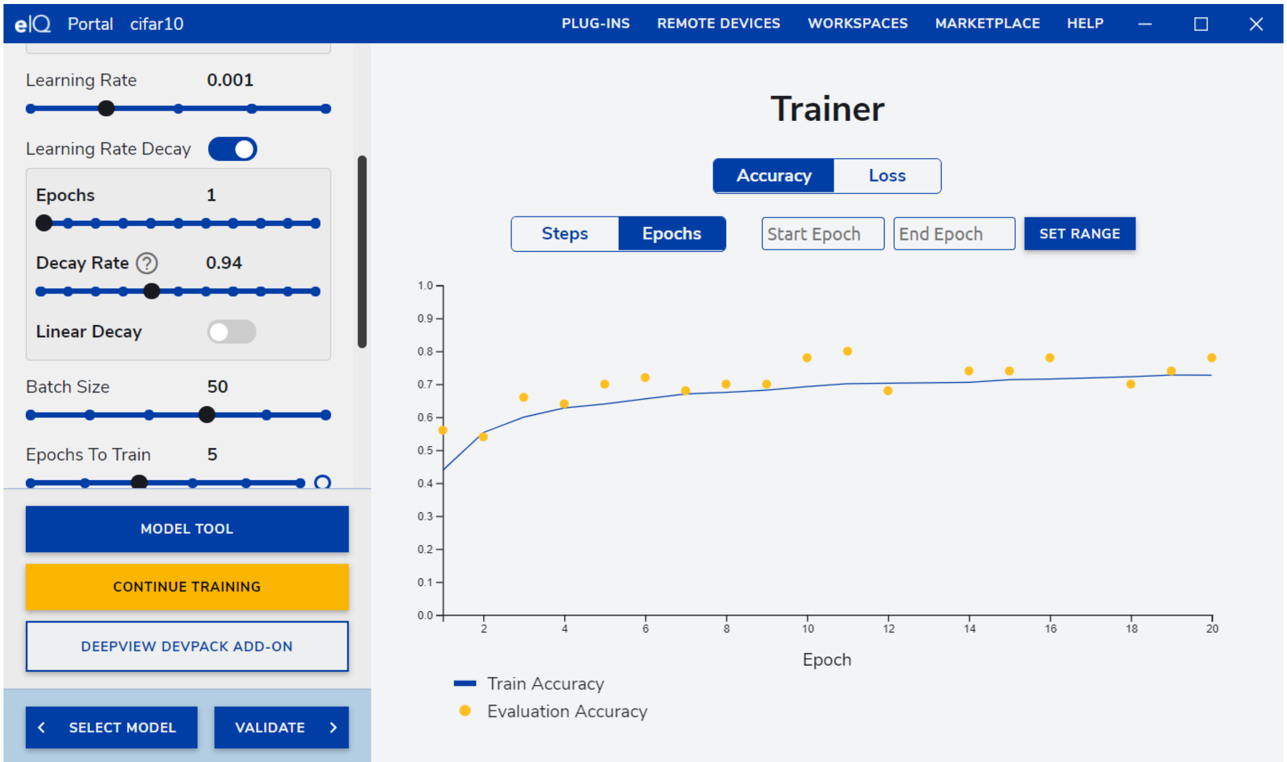

Figure 3. Trainer window

5 Deploying model on MIMXRT1060 EVK

To deploy the model, perform the following steps:

1. After validating stage, in **Deploy** window, export the model as `rtm` type with Int8 data type Quantized.
2. Open the model with the model tool, and the network is as shown in Figure 4.

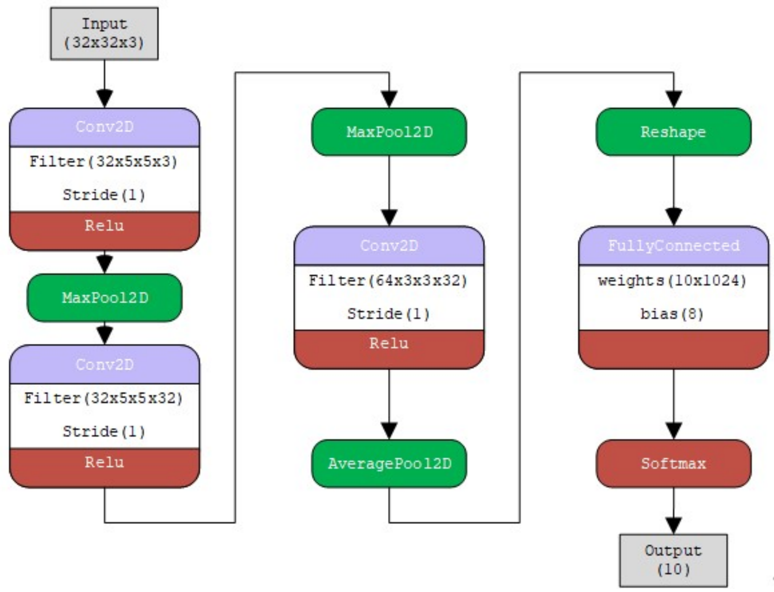

Figure 4. Network

3. Deploy the model on the board.
   - In MCUXpresso IDE import the SDK example project `deepviewrt_lableimage` from `eiq` examples into the workplace.
   - Copy the `rtm` model file, `cifar10-2022-02-16T09-11-01.769Z_in-int8_out-int8_channel_ptq.rtm`, into the source/models folder within the project.
   - Copy a ship image into `source/data/` at the same time, as shown in Figure 5.

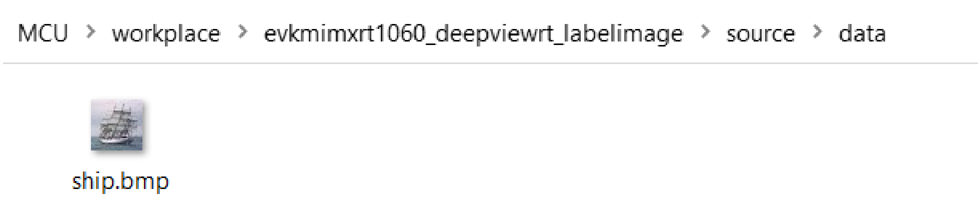

Figure 5. Copying a ship image

4. Change the `model.S` file, as shown in Figure 6.

Figure 6. Changing the model.s file

5. After building and downloading the project into board, boot up the EVK board. The debug information shows the custom model inference result, as shown in Figure 7.

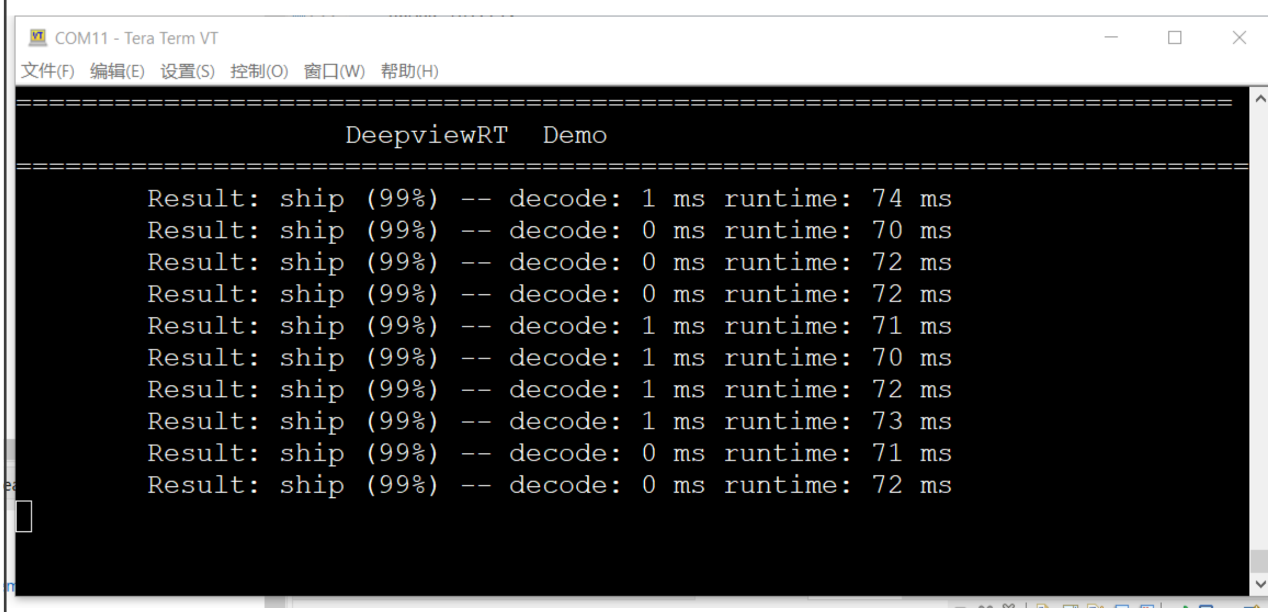

Figure 7. Custom model inference result

6 Reference

The code of the CIFAR10 plug-in is shipped in the attachments.

7 Revision history

Rev.	Date	Description
0	15 March 2022	Initial release

Legal information

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.

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 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.

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.

AMBA, Arm, Arm7, Arm7TDMI, Arm9, Arm11, Artisan, big.LITTLE, Cordio, CoreLink, CoreSight, Cortex, DesignStart, DynamIQ, Jazelle, Keil, Mali, Mbed, Mbed Enabled, NEON, POP, RealView, SecurCore, Socrates, Thumb, TrustZone, ULINK, ULINK2, ULINK-ME, ULINK-PLUS, ULINKpro, μVision, Versatile — are trademarks or registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. The related technology may be protected by any or all of patents, copyrights, designs and trade secrets. All rights reserved.

Airfast — is a trademark of NXP B.V.

Bluetooth — the Bluetooth wordmark and logos are registered trademarks owned by Bluetooth SIG, Inc. and any use of such marks by NXP Semiconductors is under license.

Cadence — the Cadence logo, and the other Cadence marks found at www.cadence.com/go/trademarks are trademarks or registered trademarks of Cadence Design Systems, Inc. All rights reserved worldwide.

CodeWarrior — is a trademark of NXP B.V.

ColdFire — is a trademark of NXP B.V.

ColdFire+ — is a trademark of NXP B.V.

EdgeLock — is a trademark of NXP B.V.

EdgeScale — is a trademark of NXP B.V.

EdgeVerse — is a trademark of NXP B.V.

eIQ — is a trademark of NXP B.V.

FeliCa — is a trademark of Sony Corporation.

Freescale — is a trademark of NXP B.V.

HITAG — is a trademark of NXP B.V.

ICODE and I-CODE — are trademarks of NXP B.V.

Immersiv3D — is a trademark of NXP B.V.

I2C-bus — logo is a trademark of NXP B.V.

Kinetis — is a trademark of NXP B.V.

Layerscape — is a trademark of NXP B.V.

Mantis — is a trademark of NXP B.V.

MIFARE — is a trademark of NXP B.V.

MOBILEGT — is a trademark of NXP B.V.

NTAG — is a trademark of NXP B.V.

Processor Expert — is a trademark of NXP B.V.

QorIQ — is a trademark of NXP B.V.

SafeAssure — is a trademark of NXP B.V.

SafeAssure — logo is a trademark of NXP B.V.

StarCore — is a trademark of NXP B.V.

Synopsys — Portions Copyright © 2021 Synopsys, Inc. Used with permission. All rights reserved.

Tower — is a trademark of NXP B.V.

UCODE — is a trademark of NXP B.V.

VortiQa — is a trademark of NXP B.V.

Date of release: 15 March 2022

Document identifier: AN13590