NeuroPilot-Micro MT3620 User Guide

Version:	0.2 (Beta Release)
Release date:	2020-07-10

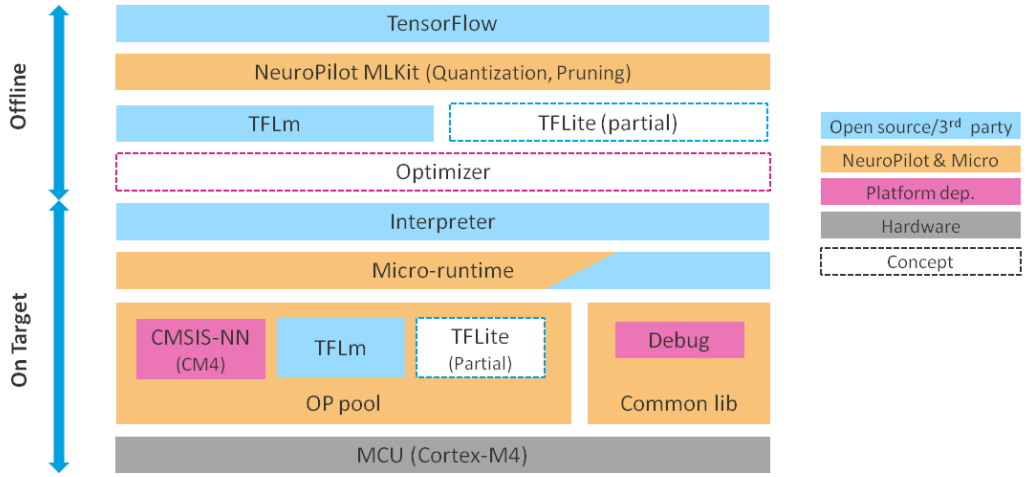
User Guide for TinyML at MT3620

- SDK User Guide
  - NeuroPilot-Micro SDK
  - Verify Your Model
  - Optimize Your Model
- Demo SDK User Guide
  - Tiny Vision SDK
  - Dual Core Processing SDK
  - Handwriting Recognition SDK

NeuroPilot-Micro SDK User Guide

NeuroPilot-Micro SDK Architecture

NeuroPilot-Micro SDK can help you deploy your model on MCU platform with a high efficiency, lower power consuming and easy developing way. NeuroPilot-Micro is fully compatible with TensorFlow Lite for microcontrollers neural network interface and supports TensorFlow Lite ecosystem. Therefore, it can leverage TensorFlow and NeuroPilot helpful and powerful neural network frameworks and tools. We also plan to support more TensorFlow Lite operations (OP) for full compatible with NeuroPilot.



Micro-runtime is the key for platform dependent optimization. Micro-runtime is in charge of the optimization for MCU and system architecture. Next step, the offline optimizer will help for static optimization to reduce micro-runtime overhead and to enhance its ability.

The memory usage of NeuroPilot-Micro interpreter and micro-runtime is about 34KB. It is very light weight for MCU system. OP (Operation) library is about 90KB for all supported OPs. It can be optimized to register used OP only for saving memory.

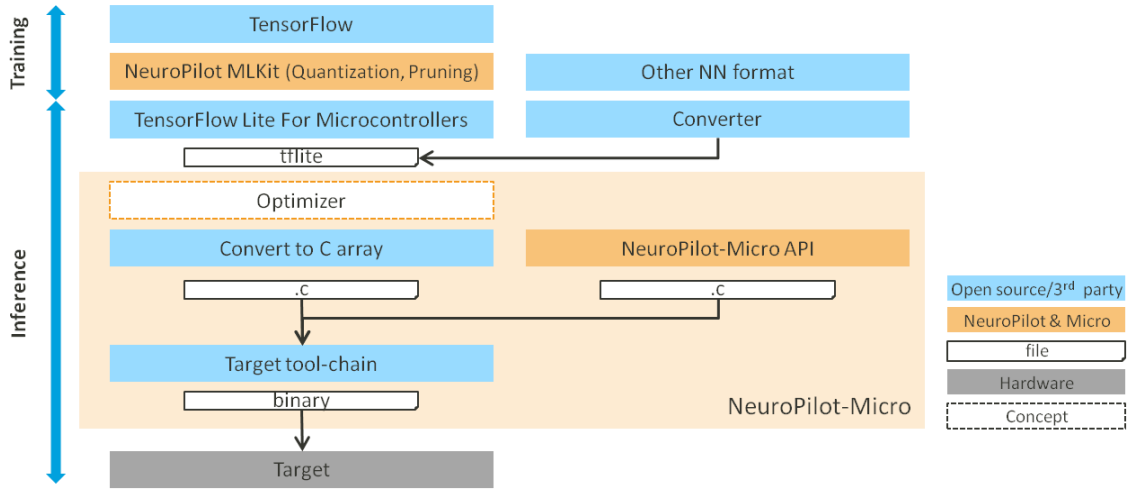
With NeuroPilot-Micro optimization, the model execution at flash is as fast as TCM. Therefore, we can put models at flash for saving TCM memory.

Below is model inference memory usage example for three demos.

Take an example, if we want to execute these three models sequentially, it needs 138KB TCM (sharing) and 498KB Flash (three models) only.

Item	Person detection	Object detection	Handwriting recognition	Three models example
Model	230 KB (Flash)	228 KB (Flash)	40 KB (Flash)	498 KB (Flash)
Tensor arena	73 KB	45 KB	22 KB	73 KB
Used OP	20 KB	23 KB	31 KB	31 KB
Framework	34 KB	34 KB	34 KB	34 KB
Total	TCM: 127 KB Flash: 230 KB	TCM: 102 KB Flash: 228 KB	TCM: 87 KB Flash: 40 KB	TCM: 138 KB Flash: 498 KB

Training and Inference



A neural network model includes two steps, training and inferencing.

Training a Model by Tensorflow

We can train a simple model by Keras and Tensorflow2 on Python3.

Install Python3 & pip3

```
$ sudo apt update
$ sudo apt install python3-dev python3-pip python3-venv
```

Install Tensorflow2 via pip3

```
$ pip3 install tensorflow

(optional, tensorflow2 already includes keras)
$ pip3 install keras
```

More details about Install Tensorflow with pip
More about Keras example: Train a simple deep CNN on the CIFAR10 small images dataset.

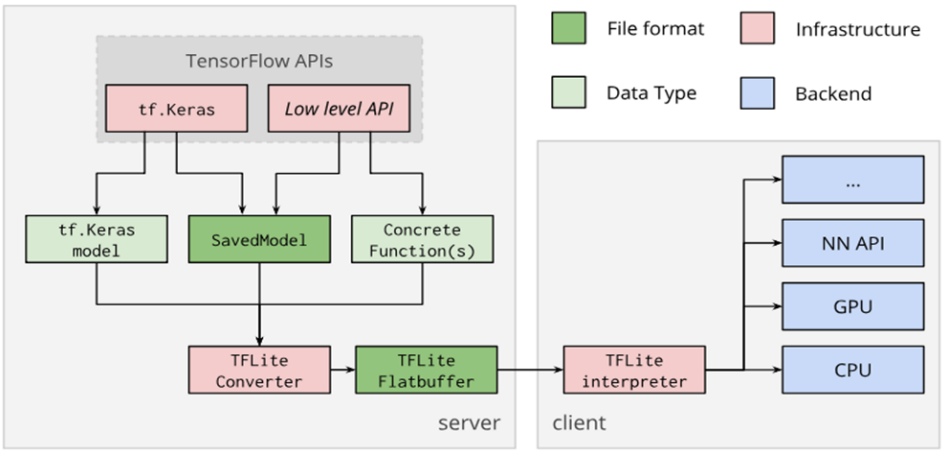
NeuroPilot-Micro ML Kit

Inference a Model by Tensorflow Lite

The trained model cannot be executed efficiently by Edge devices. Therefore, TensorFlow Lite is used for model execution, called inference. For deployment, the first step is to convert TensorFlow model to TensorFlow Lite format. The second step is to write the target code for model inference.

Convert to Tensorflow Lite Model

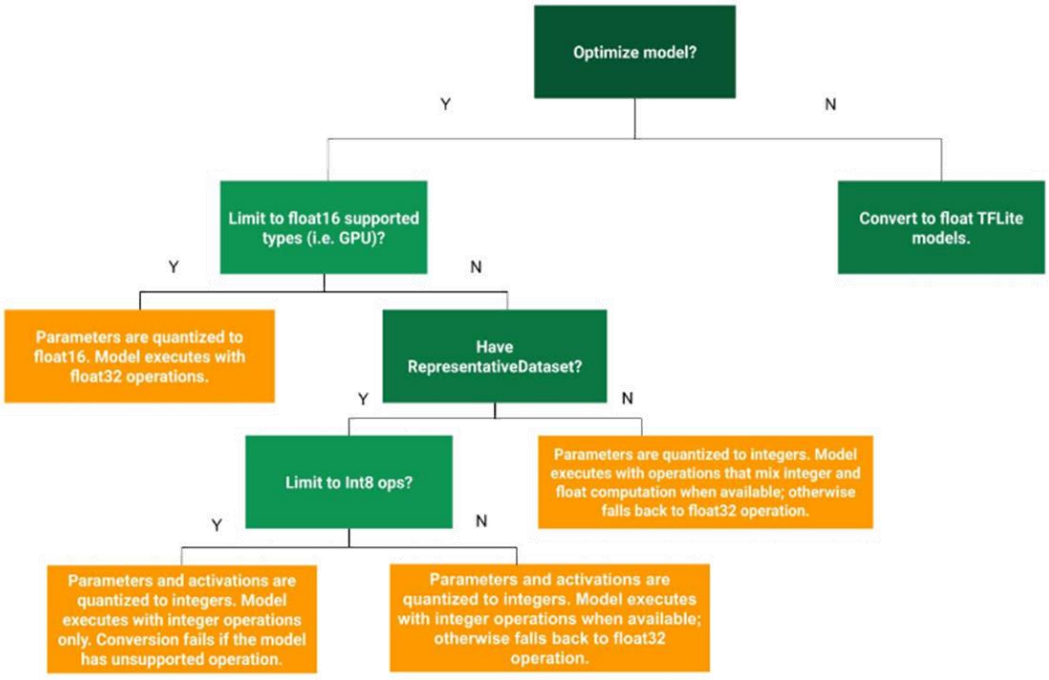
We can convert a TensorFlow model to a TensorFlow Lite model for inference by Tensorflow Lite Converter. The output will be a ".tflite" file. Devices can execute the model with TensorFlow Lite inference framework supporting.



Source from https://www.tensorflow.org/lite/convert

Quantize Tensorflow Lite Model

Because MCU system has constrained memory and limited performance, it is very important to use less bits fixed-point format. Quantization is a method to reduce bits with acceptable error. We can quantize a TensorFlow Lite model by [Tensorflow Lite Quantization Python API]

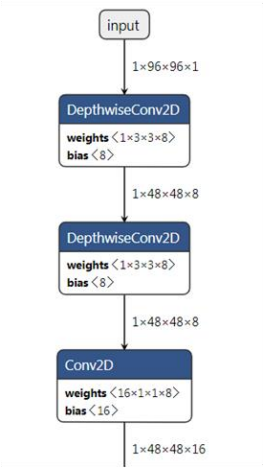


Source from https://www.tensorflow.org/lite/performance/post_training_quantization

Preview of a tflite Model

1. Visualizing Graph of a tflite Model

There is a very helpful online tool Netron for preview the model. Visualized user interface can help developers to preview the model more easily. We can check the data type, tensor shape, if the model is quantized or not and so on.



2. Json of a tflite model

tflite model is created in Flatbuffer format. By flatc tool, we can convert a tflite model to a Json file. NeuroPilot-Micro SDK provides a pre-built flatc for Linux x86. You can find it at tools folder.

```
$ flatc -t --strict-json --defaults-json schema.fbs -- model.tflite
```

schema.fbs can be found in source/tensorflow/lite/schema

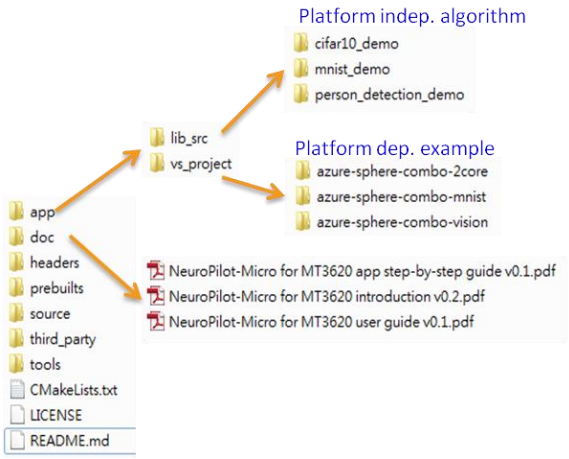
The result Json will look similar to the following.

```
{
  "version": 3,
  "operator_codes": [
    {
      "builtin_code": "AVERAGE_POOL_2D",
      "version": 1
    },
    {
      "builtin_code": "CONV_2D",
      "version": 1
    },
    {
      "builtin_code": "DEPTHWISE_CONV_2D",
      "version": 1
    }
  ],
  "subgraphs": [
    {
      "tensors": [
        {
          "shape": [
            8
          ],
        ......
```

flatc also can convert Json to a tflite model.

```
$ flatc -b -c schema.fbs model.json
```

How to Run Your Model



We provide servals APP examples in our NeuroPilot-Micro SDK. Please refer to NeuroPilot-Micro SDK app/lib_src/examples.

1. Convert tflite File to C Array

Filesystem is not widely adopted at MCU system, and it will bring lots of overhead (memory, performance). A simple way to run an inference without filesystem is to convert a model to a hexdump and make it as a char array. The following command shows how to generate the hexdump of a tflite model by unix builtin tool xxd.

```
$ xxd -i model.tflite model_tflite.cc
```

The result will look like the following.

```
unsigned char model_tflite[] __attribute__((aligned(32))) = {
  0x1c, 0x00, 0x00, 0x00, 0x54, 0x46, 0x4c, 0x33, 0x00, 0x00, 0x12, 0x00,
  0x1c, 0x00, 0x04, 0x00, 0x08, 0x00, 0x0c, 0x00, 0x10, 0x00, 0x14, 0x00,
  // lots of data
};
unsigned int model_tflite_len = 234048;
```

The file with model table can be compiled and linked to target binary.

Please add attribute __attribute__((aligned)) or __attribute__((aligned(32))) to model declaration to make sure the model is placed with correct alignment.

2. Load Model

Model must be loaded before use. Load our char array model to a tflite::model instance.

```
model = ::tflite::GetModel(model_tflite);
```

3. Setup Operator Resolver

Tensorflow Lite Micro provides three ways to instantiate an OP resolver.
- AllOpsResolver

AllOpsResolver will pull in all operator implementations. It is easy and simple to use, but it takes more memory space. The following shows how to instantiate a AllOpsResolver

```
#include "tensorflow/lite/micro/all_ops_resolver.h"

tflite::AllOpsResolver resolver;
```

Refer to NeuroPilot-Micro : app\lib_src\simple_example or tensorflow hello_world example for examples.

- MicroMutableOpResolver

MicroMutableOpResolver can only pull in the operators we need. To instantiate a MicroMutableOpResolver, we need to know which operators will be used in the model and add the certain operators to the resolver.

```
#include "tensorflow/lite/micro/micro_mutable_op_resolver.h"

tflite::MicroMutableOpResolver<3> resolver;
resolver.AddBuiltin(
  tflite::BuiltinOperator_DEPTHWISE_CONV_2D,
  tflite::ops::micro::Register_DEPTHWISE_CONV_2D());
resolver.AddBuiltin(
  tflite::BuiltinOperator_CONV_2D,
  tflite::ops::micro::Register_CONV_2D());
resolver.AddBuiltin(
  tflite::BuiltinOperator_FULLY_CONNECTED,
  tflite::ops::micro::Register_FULLY_CONNECTED());
```

MicroMutableOpResolver is a template class and it needs the number of operators. There are some ways to find out which operators are used in the model.

- Convert the tflite model to Json.
- Preview the tflite model by Netron.

Refer to NeuroPilot-Micro: app\lib_src\cifar10_demo, app\lib_src\mnist_demo or app\lib_src\person_detection_demo for examples.

- Custom Operators

Tensorflow Lite Micro provides an interface for adding custom operators. We can add self-implemented operators or special operators to the resolver.
Implement Init/free/Prepare/Invoke functions and set custom_name. You can reference to the implementation of other operators in source\tensorflow\lite\micro\kernels.

The following is an example.

```
TfLiteRegistration* Register_CUSTOM_OP() {
  static TfLiteRegistration r = {/*init=*/custom_op::Init,
                                 /*free=*/nullptr,
                                 /*prepare=*/custom_op::Prepare,
                                 /*invoke=*/custom_op::Invoke,
                                 /*profiling_string=*/nullptr,
                                 /*builtin_code=*/0,
                                 /*custom_name=*/"CUSTOM_OP",
                                 /*version=*/0};
  return &r;
}
```

Declare TfLiteRegistration function(e.g., Register_CUSTOM_OP()) in header file MT3620_M4_NeuroPilot-Micro/source/tensorflow/tensorflow/lite/micro/kernels/micro_ops.h.

```
  TfLiteRegistration* Register_TANH();
  TfLiteRegistration* Register_CUSTOM_OP(); /* */
}  // namespace micro
```

Add Custom Operator to your resolver by AddCustom.

```
tflite::MicroMutableOpResolver<1> resolver;
resolver.AddCustom("CUSTOM_OP",
  tflite::ops::micro::Register_CUSTOM_OP());
```

Notice: Resolver finds the implementation of custom operator by matching string custom_name. Therefore, the custom_code in operator_codes field of tflite model, custom_name of TfLiteRegistration function and the name string used in resolver.AddCustom() are required to be the same.

More detail about Tensorflow Lite Micro Custom Operators.

4. Set the Size of Tensor Arena

Tensor arena is a memory space used to store everything needed by runtime, such as temporary buffer for operations, runtime information, etc. The size varies with model. We can get a optimize value from experiment.

```
constexpr int kTensorArenaSize = 71 * 1024;
static uint8_t tensor_arena[kTensorArenaSize];
```

We suggest trying arena size on Linux or Windows. To find the tensor_arena size of a new model, we set a value to tensor_arena at first and execute it on Linux or Windows. If the binary executes properly, we can reduce the size. If it pops up some error messages of Failed to allocate memory, we should increase the size. Repeat this experiment to find a smallest value of tensor_arena size.

5. Setup Interpreter

Instantiate an interpreter and allocate resource from tensor_arena.

```
static tflite::MicroInterpreter static_interpreter(model, resolver,
      tensor_arena, tensor_arena_size, error_reporter);
interpreter = &static_interpreter;
TfLiteStatus allocate_status = interpreter->AllocateTensors();
```

6. Setup Input Data

Get the pointer of input buffer from interpreter API interpreter->input(0).

```
model_input = interpreter->input(0);
```

Then, you can set the input data.

```
for (int i = 0; i < kMaxImageSize; i++) {
      model_input->data.uint8[i] = input[i];
}
```

7. Run the Model

After setup interpreter and input data, we are ready to run the model. Execute the model by following code.

```
interpreter->Invoke();
```

8. Get Result

After interpreter->Invoke() is finished, we can get the result from interpreter->output(0). The following code show how to get output result.

```
uint8_t score1 = output->data.uint8[1];
uint8_t score2 = output->data.uint8[2];
```

Notice: Both interpreter->input(0) and interpreter->output(0) are pointer of structure TfLiteTensor. TfLiteTensor allows to store data by several data types. The data type you used should be the same as the data type of the model. That is, if the model input type is uint8, we should store the input data to array data.uint8.

More about: Tensorflow Lite Micro

9. Test Your APP on Linux NeuroPilot-Micro SDK

Add your folder name to list TEST_APPS in MT3620_M4_NeuroPilot-Micro/CMakeLists.txt.
For example, add a folder name my_new_model in list TEST_APPS.

```
### Add app folder name here ###
list(APPEND TEST_APPS simple_example my_new_model)
```

Create a build folder, generate cmake cache and build the files.

```
# go to root dictionary of NeuroPilot-Micro SDK
$ cd MT3620_M4_NeuroPilot-Micro
$ mkdir build
$ cd build
$ cmake .. -DBUILD_TYPE=linux
$ make
```

The executable ELF my_new_model will be generated in current folder.

10. Put the tflite Model to MT3620 FLASH

All read-only data will be placed in serial flash. We configure the linker.ld in rtcore to place all read-only data to serial flash.

```
REGION_ALIAS("CODE_REGION", TCM);
REGION_ALIAS("RODATA_REGION", FLASH);
REGION_ALIAS("DATA_REGION", TCM);
REGION_ALIAS("BSS_REGION", TCM);
```

We can place a model to serial flash by declaring with const.

```
const unsigned char model_tflite[] __attribute__((aligned(32))) = {
  0x1c, 0x00, 0x00, 0x00, 0x54, 0x46, 0x4c, 0x33, 0x00, 0x00, 0x12, 0x00,
  0x1c, 0x00, 0x04, 0x00, 0x08, 0x00, 0x0c, 0x00, 0x10, 0x00, 0x14, 0x00,
  // lots of data
};
const unsigned int model_tflite_len = 234048;
```

Next Steps

After creating an APP for the new model, you can verify the behavior of the model.
If the model works properly on Linux x86, you can try to deploy the model on MT3620 by following the way of our example and demo.
Then, you can evaluate the performance and optimize your model.
If something goes wrong, you may need some tips for debugging.

- **Deploy on MT3620**
- **Verify Your Model**
- **Evaluate model performance**
- **Optimize Your Model**
- **Debug**

Verify Your Model

How to verify at Your PC

***We strongly suggest testing the model on Linux NeuroPilot-Micro SDK before you deploy on MCU.**
Debugging on Linux environment is more efficient than on MCU platform and running NeuroPilot-Micro on Linux is quick and simple. Moreover, there are also many

debugging tool on Linux. These tool can help us address problems at first time.

1. Write Code

Create a new folder for your model in app/lib_src. You can write you own APP for your model by following the chapter How to run your model and simple_example in NeuroPilot-SDK. Then, use error_reporter to print message.

```
TF_LITE_REPORT_ERROR(error_reporter, "get result %d\r\n", res);
```

2. Build Code

Follow the commands show at Test Your APP on Linux NeuroPilot-Micro SDK and you can generate an executable ELF. Run the ELF and check if the behavior of model and APP is correct or not.

More about debugging: Debug

Modify Code of Tensorflow Lite Micro

NeuroPilot-Micro SDK contains source code of Tensorflow Lite Micro in folder source/tensorflow. We do not build the Tensorflow Lite Micro source code together with RT-core. Instead, we build them as a standalone static library and link the library to RT-core. You can find the pre-built library libtensorflow-microlite.a in prebuilts/lib.

- **Why We Build Static Lib & Link?**
  Some of MCU OS, such as FreeRTOS, are written in C and build by C compiler.
  NeuroPilot-Micro is written in C++. To build NeuroPilot-Micro with MCU OS, we need to change the C environment to C++ environment.
  We may encounter some annoying problems when changing whole building environment.
  Therefore, we build NeuroPilot-Micro as standalone static libraries.
  Just build with the same architecture as target system. We can simply link these libraries into target system.

If you need, you can modify the source code. The following shows how to build tensorflow lite micro as a static library and link to vs_project.

1. Modify the code in source/tensorflow
2. Open CMakeLists.txt in root folder of NeuroPilot-Micro SDK.
   Set CMAKE_CXX_COMPILER and CMAKE_C_COMPILER to your MT3620 toolchain path.

3. Type following commands

```
# In NeuroPilot-Micro root folder
$ mkdir build
$ cd build
$ cmake .. -DBUILD_TARGET=mt3620
$ make
```

This CMakeLists.txt accepts a user input variable BUILD_TARGET and it can be set to mt3620 or linux.

```
$ cmake .. -DBUILD_TARGET=mt3620 # to build static library for MT3620
$ cmake .. -DBUILD_TARGET=linux # to build executable for Linux x86
```

4. Result library libtensorflow-microlite.a will be shown at the build folder

```
[ 98%] Linking CXX static library libtflite-micro.a
[ 98%] Built target tflite-micro
Scanning dependencies of target tensorflow-microlite
[100%] Generating libtensorflow-microlite.a
ar: creating MT3620_M4_NeuroPilot-Micro/build/libtensorflow-microlite.a
[100%] Built target tensorflow-microlite
MT3620_M4_NeuroPilot-Micro/build$ ls
CMakeCache.txt  CMakeFiles  cmake_install.cmake  libsimple_example_static.a  libtensorflow-microlite.a  libtflite-micro.a  Makefile
MT3620_M4_NeuroPilot-Micro/build$
```

5. Modify the CMakeLists.txt of RT-core to use new libtensorflow-microlite.a. Change the following path to new library.

```
54  # Libraries
55  add_library(libtensorflow-microlite STATIC IMPORTED)
56  SET_TARGET_PROPERTIES( libtensorflow-microlite PROPERTIES IMPORTED_LOCATION ${NPU_ROOT_DIR}/prebuilts/lib/libtensorflow-microlite.a)
57
```

Debug

We suggest to test your model on Linux or Windows before you deploy to MCU. Due to the lack of exception handlers and tools, debugging on MCU is quite difficult. It is quick and simple to run NeuroPilot-Micro on Linux and Windows. Moreover, there are many debugging tools and these tools can help us address problems at first time.

Debug on Linux

This section shows how to debug by gdb.

If there are some error while executing your APP, you can use gdb to check what happened. The following shows a segmentation fault.

```
$ ./person_detection
arena size 69632.

Arena size is too small for activation buffers. Needed 55296 but only 52912
was available.
AllocateTensors() failed.

Setup & Init

Segmentation fault (core dumped)
```

With gdb, we can obtain more detail messages.

```
$ gdb ./person_detection
# skip some gdb message....
Reading symbols from ./person_detection...done.
(gdb) r
Starting program: person_detection
arena size 69632.

Arena size is too small for activation buffers. Needed 55296 but only 52912
was available.
AllocateTensors() failed.

Setup & Init

Program received signal SIGSEGV, Segmentation fault.
0x00005555555569ab in loop (input_buf=<optimized out>)
    at ...
      model_input->data.uint8[i] = input_buf[i];
(gdb)
```

Here list some basic commands of gdb.

- Run the program

```
(gdb) run
or
(gdb) r
```

- Show backtrace

```
(gdb) backtrace
#0  0x00005555555569ab in loop (input_buf=<optimized out>)
    at ...
#1  test ()
    at ...
#2  0x0000555555556079 in main (argc=<optimized out>, argv=<optimized out>)
    at ...
```

- Exit gdb

```
(gdb) quit
or
(gdb) q
```

More about GDB commands and tutorial.

Debug on Windows

This section shows how to execute and debug by Visual Studio.

1. Download NeuroPilot-Micro SDK

2. Install Visual Studio

3. Install MinGW - GNU tool and GGC Compiler on Windows System

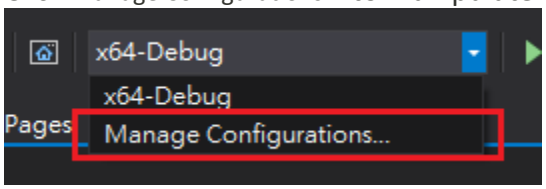
If you already have installed MinGW, you can skip this step.
Download MinGW installer and install the necessary packages (mingw32-base-bin, mingw32-gcc-g++.bin and mingw32-gcc-objc.bin). After installation, it also has to add MinGW folder to Path of Environment Variable.
You can reference to this tutorial: How To install MinGW on Windows 10.

4. Open Visual Studio and Open CMake File MT3620_M4_NeuroPilot-Micro/CMakeLists.txt

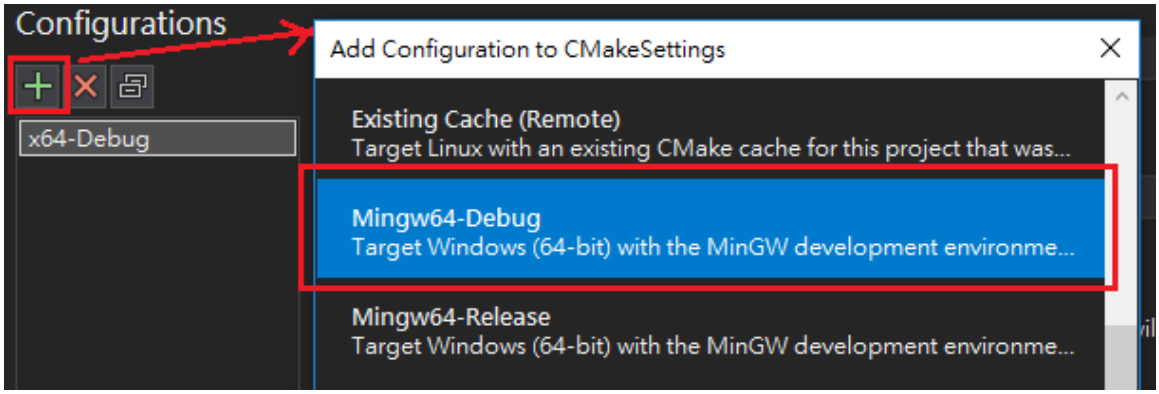
5. Setting CMakeSetting.json

The default configurations of Visual Studio is x64-Debug. We have to change to Mingw64-Debug.

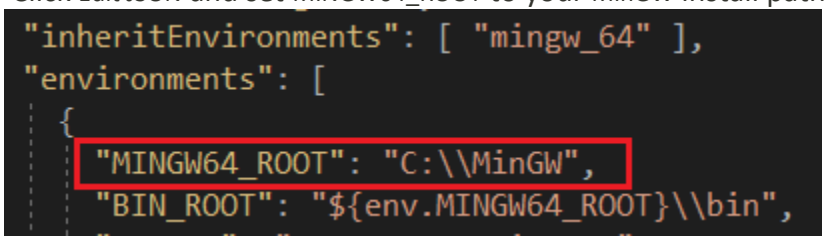
Click Manage Configurations... to manipulate CMakeSettings.



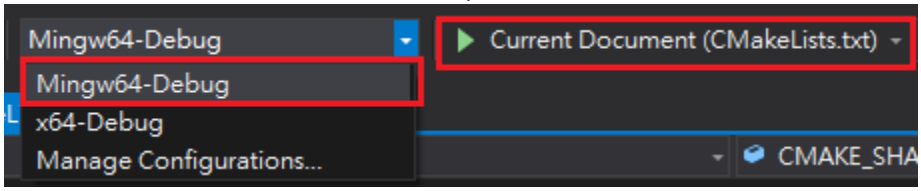
Click + and add a new Mingw64-Debug configuration.



Click Edit JSON and set MINGW64_ROOT to your MinGW install path.

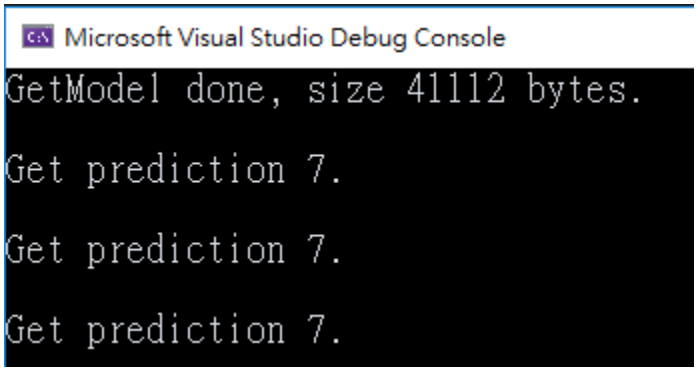


Let's go back to the CMakeLists.txt. Switch configuration to Mingw64-Debug and select Current Document in Select Startup Item.



6. Press F5 to Build and Start Debugging

The following shows the result of running simple_example.



Learn more about Visual Studio Debugger

Evaluate Model Performance

Interpreter->invoke() runs the inference one time. We can measure the execution time of this function call to evaluate model performance.

Optimize Your Model User Guide

Micro-Runtime

MCU platform usually has a limited size of TCM(or other high speed memory). For most MCU platforms, the size of TCM is usually less than 1 Mb. However, the model size of an image classification model is often several hundred Kb. Not to mention, if we add more layers to gain accuracy, the model size will become bigger. The model may not be able to put in TCM, so we only can put the model in DRAM or FLASH. But, run an inference which stored in DRAM or FLASH will lead to huge power consumption or bad performance.

To deal with this problem, we introduce Micro-runtime to dynamically manage resource to save the TCM and run with a good performance.

Micro-runtime prepares a TCM buffer and dynamically loads part of model into the TCM buffer, which performs like a software cache. By this mechanism, Micro-runtime allows to store all model in DRAM or FLASH and obtain a performance close to run inference in TCM.

The optimization is default turn-on, and developers can do fine-tuning by increasing software cache size for better performance.

Tuning of Micro-Runtime

Intuitively, the larger of Micro-runtime buffer size, the better performance we can get. In most situation, we do not need to create a maximum buffer and get a best performance, because it may cost lots of memory and may gain a little performance. We suggest to set the optimize buffer size of the model.

The default buffer size of Micro-runtime is 5 Kb.

First, include the header and declare a pointer of DynamicAgent

```
#include <dynamic_agent.h>

tflite::DynamicAgent *agent;
```

Obtain DynamicAgent from interpreter

```
agent = interpreter->GetDynamicAgent();
// operations of agent

// before AllocateTensors
TfLiteStatus allocate_status = interpreter->AllocateTensors();
```

Note: All the setting operations of dynamic_agent have to be done before interpreter->AllocateTensors().

Quick reference of Micro-runtime APIs.

Synopsis	description	input	output
TfLiteStatus EnableDynamicLoad(void)	Enable Micro-runtime	-	kTfLiteOk if succeeded, kTfLiteError if failed
TfLiteStatus DisableDynamicLoad(void)	Disable Micro-runtime	-	kTfLiteOk if succeeded, kTfLiteError if failed
TfLiteStatus SetCacheEnable(bool flag)	Enable Micro-runtime to use an additional buffer	true/false	kTfLiteOk if succeeded, kTfLiteError if failed
TfLiteStatus SetDynamicArenaSize(size_t size)	Set the maximum size can be used by Micro-runtime	size of Micro-runtime buffer size	kTfLiteOk if succeeded, kTfLiteError if failed
void PrintConfig(void)	Show the configuration of Micro-runtime	-	-

Two Ways to Add an Additional Buffer for Micro-Runtime.

- From tensor arena Micro-runtime can get unused memory space from tensor arena. In this case, only need to add more size to tensor arena.

```
//const int tensor_arena_size = 45 * 1024;
// Increase 10 Kb. The redundant memory space will be used by Micro-runtime.
const int tensor_arena_size = 55 * 1024;
uint8_t tensor_arena[tensor_arena_size];

...

agent->SetCacheEnable(true);
```

- From pre-allocated array

```
// declare a global memory buffer
const int new_buffer_size = 10 * 1024;
uint8_t new_buffer[new_buffer_size];
...

agent->SetCacheEnable(true);
agent->RegisterCache(new_buffer, new_buffer_size);
```

To Shrink the Buffer Size

Micro-runtime create a builtin memory buffer with fixed size 5 Kb. Micro-runtime allows to shrink the using size, but it still holds 5 Kb memory buffer. This API can just make Micro-runtime use less space.

```
// Micro-runtime will use 3 kb of 5kb memory buffer.
agent->SetDynamicArenaSize(3 * 1024);
```

To Disable or Enable Micro-Runtime. (Default is enabled)

Disable dynamically loading mechanism.

```
agent->DisableDynamicLoad();
```

Enable dynamically loading mechanism.

```
agent->EnableDynamicLoad();
```

Show Current Configuration of Micro-Runtime

```
// global variable
tflite::DynamicAgent *agent;
const int new_buffer_size = 10 * 1024;
uint8_t new_buffer[new_buffer_size];

...
agent = interpreter->GetDynamicAgent();
agent->SetCacheEnable(true);
agent->RegisterCache(new_buffer, new_buffer_size);

TfLiteStatus allocate_status = interpreter->AllocateTensors();

// different from other APIs, PrintConfig() only can work properly after
interpreter->AllocateTensors()
agent->PrintConfig();
```

Dump the message like the following.

```
=> Dynamic Cache size 10240
=> Dynamic Cache 0xb9ddd200
=> Dynamic Available 1
=> Dynamic buffer size 5120
=> Fine Grained size 5120
=> Enable Fine Grained 1
=> Enable Dynamic Loading 1
=> dynamic_buffer_ 0xb9ddfc60
```

CMSIS-NN API

Directly Use CMSIS-NN API

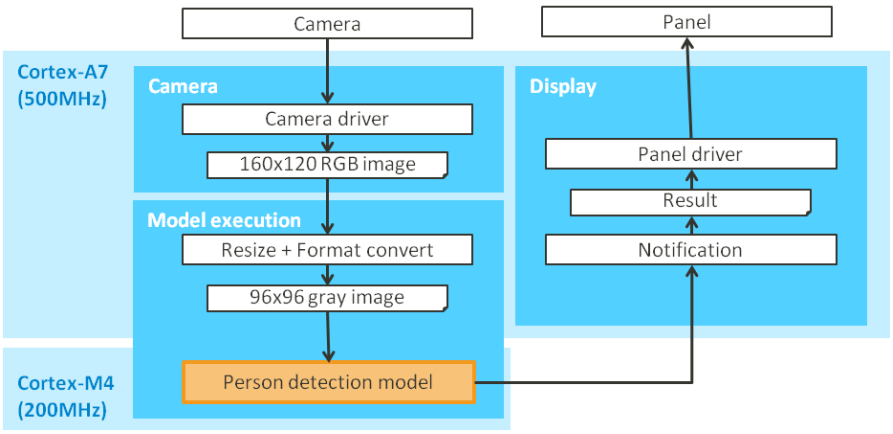
NeuroPilot-Micro provides a pre-built CMSIS-NN library and headers.
You can follow CMSIS-NN example to write a new cmsis-nn application and link the pre-built CMSIS-NN library to the application. For CMSIS-NN examples, please refer to ARM CMSIS-NN example

Tiny Vision SDK User Guide

The input of tiny vision platform is the camera. For easy developing, touch driver and related processing are at Shpere OS (Cortex-A7). The processed input will be send to Cortex-M4 for neural network model inference. And result will be back to Shpere OS for display. Person Detection and Object Detection can be used for neural network model.

The following sections includes:

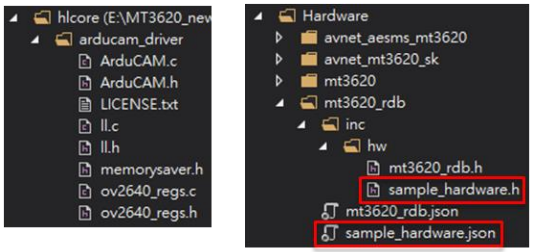
- How to use camera
- How to display with LCD
- How to execute model



How to Use Camera

Camera Setup

- hlcore/arducam_driver folder contains the lower level APIs and settings for controlling ArduCam



- Define ArduCam connection in sample_hardware.h and sample_hardware.json

```
// Connect CS to RDB Header1 Pin10(GPIO3)
#define ARDUCAM_CS MT3620_RDB_HEADER1_PIN10_GPIO
// Connect I2C to RDB Header2 Pin1(SDA) and Pin7(SCL)
#define ARDUCAM_I2C MT3620_RDB_HEADER2_ISU0_I2C
// Connect SPI to RDB Header4 Pin5(MISO), Pin7(SCLK), Pin11(MOSI)
#define ARDUCAM_SPI MT3620_RDB_HEADER4_ISU1_SPI
```

- Add capability in app_manifest.json

```
"Capabilities": {
  "Gpio": [ "$ARDUCAM_CS" ],
  "I2cMaster": [ "$ARDUCAM_I2C" ],
  "SpiMaster": [ "$ARDUCAM_SPI" ]
}
```

- Using BMP format by CFG_MODE_BITMAP configuration

Camera Initialization

Following is the steps to initialize camera. It first calls low level init and reset functions. Then after format has set, the init camera function is called. After camera init, we call functions to clear camera fifo and fifo flag.

```
// init hardware and probe camera
      arducam_ll_init();
      arducam_reset();

      // config Camera
#if defined(CFG_MODE_JPEG)
      arducam_set_format(JPEG);
#elif defined (CFG_MODE_BITMAP)
      arducam_set_format(BMP);
#endif
      arducam_InitCAM();
#if defined(CFG_MODE_JPEG)
      arducam_OV2640_set_JPEG_size(OV2640_160x120);
#endif

      delay_ms(100);
      arducam_clear_fifo_flag();
      arducam_flush_fifo();
```

Camera Input

For each frame, the following steps will be called once for getting camera input. After arducam_start_capture(), it will store the input image at camera fifo. So check the

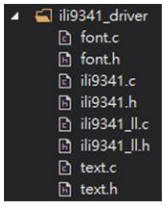
camera fifo done and then read fifo to get the input image.

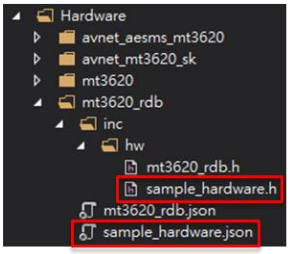
```
arducam_start_capture();
while (!arducam_check_fifo_done());

uint32_t img_len = arducam_read_fifo_length();
if (img_len > MAX_FIFO_SIZE) {
  Log_Debug("ERROR: FIFO overflow\r\n");
  return -1;
}

arducam_CS_LOW();
arducam_set_fifo_burst();
arducam_read_fifo_burst(&s_CameraBuffer[0], img_len);
arducam_CS_HIGH();
arducam_clear_fifo_flag();
```

How to Display with LCD

TFT Display API

- hlcore/ili9341_driver folder contains the lower level APIs and settings for controlling TFT display





- Define hardware in sample_hardware.h and sample_hardware.json

```
// Connect RST to RDB Header1 Pin4(GPIO0)
#define ILI9341_RST MT3620_RDB_HEADER1_PIN4_GPIO
// Connect DC to RDB Header1 Pin6(GPIO1)
#define ILI9341_DC MT3620_RDB_HEADER1_PIN6_GPIO
// Connect BL to RDB Header1 Pin8(GPIO2)
#define ILI9341_BL MT3620_RDB_HEADER1_PIN8_GPIO
// Connect SPI to RDB Header3 Pin5(SCLK), Pin7(MOSI), Pin9(MISO),
Pin11(CSA)
#define ILI9341_SPI MT3620_RDB_HEADER3_ISU3_SPI
```

- Add capability in app_manifest.json

```
"Capabilities": {
  "Gpio": [ "$ILI9341_RST", "$ILI9341_DC", "$ILI9341_BL" ],
  "SpiMaster": [ "$ILI9341_SPI" ]
}
```

TFT Display Setup

TFT display setup is relatively simple, only call ili9341_init() function. The init works will be done in driver part.

```
// write example C code
void ili9341_init(void);
```

TFT Display Output

TFT display provide several function for display output. For display output string, the lcd_set_text_cursor() and lcd_set_text_size() functions set the text position and text size. Then we call lcd_display_string() to indicate the display string content. For display output image, it provide ili9341_draw_rect() and ili9341_draw_bitmap() two functions. The ili9341_draw_rect() draws a rectangle with input position, size, and color. And the ili9341_draw_bitmap() function shows the provided bitmap image.

```
void lcd_set_text_size(uint8_t size);
void lcd_set_text_cursor(uint16_t x, uint16_t y);
void lcd_display_string(char *p_str);
void ili9341_draw_rect(uint16_t x0, uint16_t y0, uint16_t width, uint16_t
height, uint16_t color);
void ili9341_draw_bitmap(uint16_t x0, uint16_t y0, uint16_t width, uint16_t
height, uint8_t* p_bitmap);
```

Camera to Display

Both camera and display format are RGB565, but RGB order are different. So change the memory order when copy from camera buffer to display buffer.

```
#if defined(CFG_MODE_BITMAP)
  memcpy(&s_DisplayBuffer[0], &s_CameraBuffer[0], DISPLAY_WIDTH *
DISPLAY_HEIGHT * 2);
#endif

  uint8_t tmpt;
  for (uint32_t i = 0; i < DISPLAY_WIDTH * DISPLAY_HEIGHT * DISPLAY_DEPTH;
i += 2) {
    tmpt = s_DisplayBuffer[i];
    s_DisplayBuffer[i] = s_DisplayBuffer[i + 1];
    s_DisplayBuffer[i + 1] = tmpt;
  }
```

How to Execute Model

Input for CM4 Inference

The input for person detection is 96*96 and gray level image. So we need some preprocess to transfer from camera input to model input. The preprocessing function is resize_leftup_image(), it down sample to 96*96 by selecting left up point and convert to gray level.

```
resize_leftup_image(&s_CameraBuffer[0], &s_ImgBuffer[0]);
```

Cortex-A7 and Cortex-M4 Communication

- Cortex-A7 side

1. Create application socket
2. Create thread to receive from CM4
3. Using spinlock for 2 thread synchronize

```
rtSocketFd = Application_Connect(rtAppComponentId);
pthread_spin_init(&update_lock, PTHREAD_PROCESS_PRIVATE);
pthread_create(&thread_id, NULL, epoll_thread, NULL);
```

4. Call send and recv functions to transfer data between Cortex-A7 and Cortex-M4

```
ssize_t bytesSent = send(rtSocketFd, &s_ImgBuffer[i * maxInterCoreBufSize],
maxInterCoreBufSize, 0);
ssize_t bytesReceived = recv(rtSocketFd, &buf[0], 5, 0);
```

- Cortex-M4 side

1. Call GetIntercoreBuffers functions to get the buffer inbound and outbound handler

2. DequeueData and EnqueueData functions are for receiving and sending data with Cortex-A7

```
GetIntercoreBuffers(&outbound, &inbound, &sharedBufSize);
DequeueData(outbound, inbound, sharedBufSize, &recvBuf[0], &recvSize);
EnqueueData(inbound, outbound, sharedBufSize, &recvBuf[0], payloadStart +
5);
```

Model Execution

With NeuroPilot-Micro SDK, model(s) can be put at the flash, which provides much bigger size than TCM but slower. Do not worry about the speed, it will be as fast as running at TCM by NeuroPilot-Micro optimization. Take person detection as example, model running at flash takes 3.3 seconds. With NeuroPilot-Micro it only takes around 1 second only, similar as running at TCM.

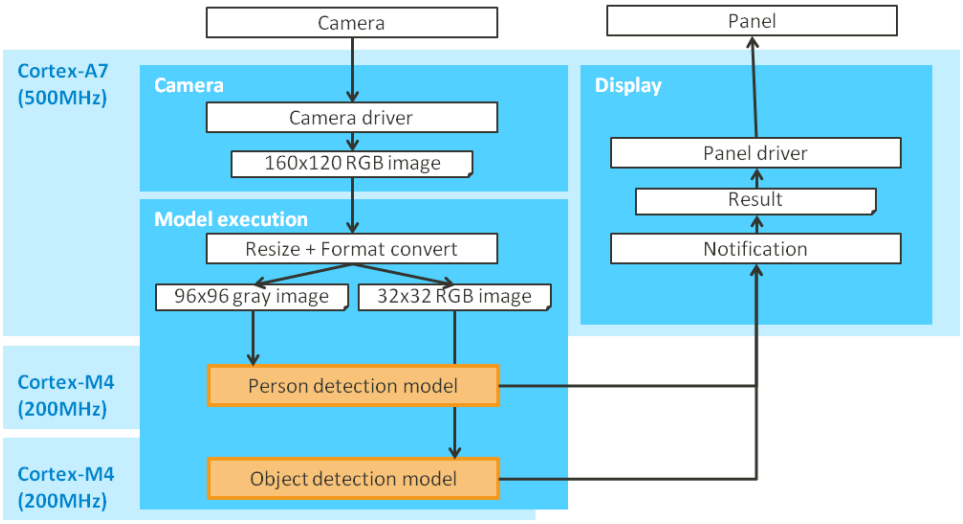
In general, model(s) and tensor arena should be at TCM for best performance. TCM is limited and may not be enough for neural network model execution. One benefit of NeuroPilot-Micro is we never need to take care about model size. All models can be placed to flash.

```
#if defined PERSON_DETECTION_DEMO
  top_index = person_detection_loop((uint8_t*)&ImgBuf[0]);
#elif defined CIFAR10_DEMO
  top_index = cifar10_invoke((uint8_t*)&ImgBuf[0]);
#endif
```

Dual Core Processing SDK User Guide

The dual core processing platform is based on the tiny vision platform. Vision is good sensing input for innovations. There are lots of vision related application, and may need to optimize with MT3620 dual core processing system.

The major benefit for dual core processing is not boosting performance by doubled 200MHz to 400MHz. It is not real and need well-scheduled models for equivalent 400MHz Cortex-M4. Dual core processing here is used for real-time. Real-time or faster model can be executed without delaying by another model(s).



For simple and real-time, the captured image will be preprocessed to two input for each Cortex-M4 core at the same time. Therefore each Cortex-M4 core can get real-time input, and will not be delayed by each other.

How to Execute Model

Cortex-A7 and Cortex-M4 Communication

- Cortex-A7 side

The communication flow between Cortex-A7 and Cortex-M4 are the same as vision demo. The only difference is that in the dual core demo, all things need to be done twice. So we need two Cortex-M4 application IDs for recognizing which Cortex-M4 core is connected. The allowed application ID is list at app_manifest.json.

```
"Capabilities": {
    "AllowedApplicationConnections": [ "6583cf17-d321-4d72-8283-
0b7c5b56442b", "4299f0b0-6d19-40f4-8f57-7ab6c4c0d172" ]
}
```

- Cortex-M4 side

The Cortex-M4 applications need to specify its application id and Cortex-A7 application id in app_manifest.json.

```
"ComponentId": "6583cf17-d321-4d72-8283-0b7c5b56442b"
"Capabilities": {
    "AllowedApplicationConnections": [ "8c48a678-97d3-4fef-a41a-
78feb57a43ae" ]
}
```

Model Execution

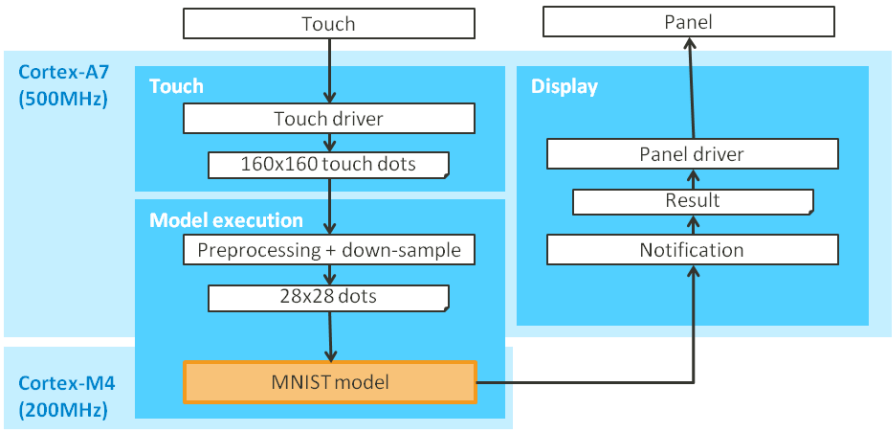
In dual core demo, one core is person detection and another core is Cifar-10 object detection. For person detection, please refer to Tiny Vision SDK User Guide. For Cifar-10 object detection, it directly call cmsis-nn inference function due to the memory restriction. The detail please refer to ARM CMSIS-NN CIFAR-10.

Handwriting Recognition SDK User Guide

The input of handwriting platform is the touch panel. For easy developing, touch driver and related processing are at Shpere OS (Cortex-A7). The processed input will be send to Cortex-M4 for neural network model inference. And result will be back to Shpere OS for display. MNIST is used for neural network model.

The following sections includes:

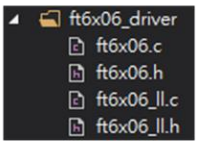
- How to use touch panel
- How to execute model
- How to display with LCD



How to Use Touch Panel

Touch Panel API

- hlcore/ft6x06_driver folder contains the lower level APIs and settings for controlling touch panel



- The touch panel API is relative simple, following is the APIs. Before using touch panel, it must call ft6x06_init() to do initialization. Then the ft6x06_detect_touch() return larger than zero when a valid touch, apply ft6x06_get_xy() to get the touch position.

```
void ft6x06_init(void);
uint8_t ft6x06_detect_touch(void);
void ft6x06_get_xy(uint16_t* p_x, uint16_t* p_y);
```

Get Touch Input

In this demo, a timer event is created for checking touch periodically (20ms). Then we using SM_IDLE, SM_DRAWING, and SM_DONE three states to identify input. Initial state is SM_IDLE, enter SM_DRAWING when detect a touch. When no input over 400ms enter SM_DONE state and regard as touch done. Clean screen after touch done 1s and return

to SM_IDLE state.

```
if (workState == SM_IDLE) {
	if (checkTouchAndDrawPoint() == VALID_TOUCH) {
		workState = SM_DRAWING;
		done_count = DONE_TO;
	}
} else if (workState == SM_DRAWING) {
	uint8_t ret = checkTouchAndDrawPoint();
	if ((ret == NO_TOUCH) || (ret == INVALID_TOUCH)) {
		done_count--;
		if (done_count == 0) {
			workState = SM_DONE;
		}
	} else {
		done_count = DONE_TO;
	}
} else if (workState == SM_DONE) {
	clean_count--;
	if (clean_count == 0) {
		workState = SM_IDLE;
		clean_count = CLEAN_TO;
	}
}
```

How to Execute Model

Preprocess and Resize

After touch done, it need to be preprocessed and resized before send data to Cortex-M4. The resize() function is to enlarge touch point by R=6 and then down sample to 28*28.

```
resize(&frameBuffer[0], &mnistBuffer[0]);

ssize_t bytesSent = send(rtSocketFd, &mnistBuffer[0], MINST_BUF_SIZE, 0);
if (bytesSent < 0) {
	Log_Debug("ERROR: Unable to send message: %d (%s)\r\n", errno,
strerror(errno));
} else if (bytesSent != MINST_BUF_SIZE) {
	Log_Debug("ERROR: Write %d bytes, expect %d bytes\r\n", bytesSent,
MINST_BUF_SIZE);
}
```

Send Input to Cortex-M4

The inter-core communication function is the same as person detection demo. After down sample to 28*28, send result buffer to Cortex-M4 by send() function.

```
ssize_t bytesSent = send(rtSocketFd, &mnistBuffer[0], MINST_BUF_SIZE, 0);
```

Model Inference

After Cortex-M4 receive data from Cortex-A7, it call mnist_invoke() function to do the inference work.

```
predic_label = mnist_invoke((uint8_t*)&recvBuffer[INTERBUFOVERHEAD]);
```

How to Display with LCD

LCD driver is the same as vision demo, the only difference is it directly call display ili9341_fillCircle() function instead of feeding image. Following is the sample code of display the handwriting content.

```
if (ft6x06_detect_touch() > 0) {
	ft6x06_get_xy(&x, &y);

	if ((x - R > SQ_LEFTUP_X) && (x + R < SQ_RIGHTDOWN_X) && (y - R >
SQ_LEFTUP_Y) && (y + R < SQ_RIGHTDOWN_Y)) {

		_x_ = x - SQ_LEFTUP_X - R;
		_y_ = y - SQ_LEFTUP_Y - R;

		for (uint8_t row = 0; row < (2 * R + 1); row++) {
			memset(&frameBuffer[SQ_SIDE * (_y_ + row) + _x_], 235,
(2 * R + 1));
		}

		ili9341_fillCircle(x, y, R, BLACK);

		return VALID_TOUCH;
	}
}
```