AN13754

Enable PXP for OpenCV

Rev. 0 — 20 October 2022 **Application note**

Document information

Information	Content
Keywords	OpenCV, MCU, PXP optimized
Abstract	OpenCV (Open Source Computer Vision Library) is an open-source library that includes several hundreds of computer vision algorithms.

1 Introduction

OpenCV (Open Source Computer Vision Library: http://opencv.org) is an open-source library that includes hundreds of computer vision algorithms.

OpenCV has a modular structure, which means that the package includes several shared or static libraries. The following modules are available: Core functionality, Image Processing, Video Analysis, Camera Calibration, 3D reconstruction (calib3d), 2D Features Framework (features2D), Object Detection (objdetect), High-level GUI (highgui), and Video I/O (videoio).

The OpenCV has integrated multiple hardware optimizations in the Hardware Acceleration Layer (HAL), such as, SSE, NEON, OpenCL, CUDA, OpenCV4Tegra. But it does not optimize MCU platform, especially when the MCU also has an acceleration hardware, such as, PXP or even the 2D-GPU on the i.MX RT1170. This document introduces how to optimize the openCV library with PXP and then run it on our RT-Series MCU platform, such as, i.MX RT1170 EVKB board.

This article is an extension of *Run openCV on Cortex-M7 MCU* (document AN13725). We assume that you have already read AN13725 and have that environment installed on your PC.

2 PXP

The Pixel Processing Pipeline (PXP) is used to process graphics buffers or composite video and graphics data before sending to an LCD display or TV encoder. It is used to minimize the memory footprint required for the display pipeline and provide an area and performance optimized to both SDRAM-less and SRAM-based systems.

The PXP integrates several independent processing stages into a cohesive strategy to create flexible pixel pipeline.

Figure 1 shows the PXP block diagram.

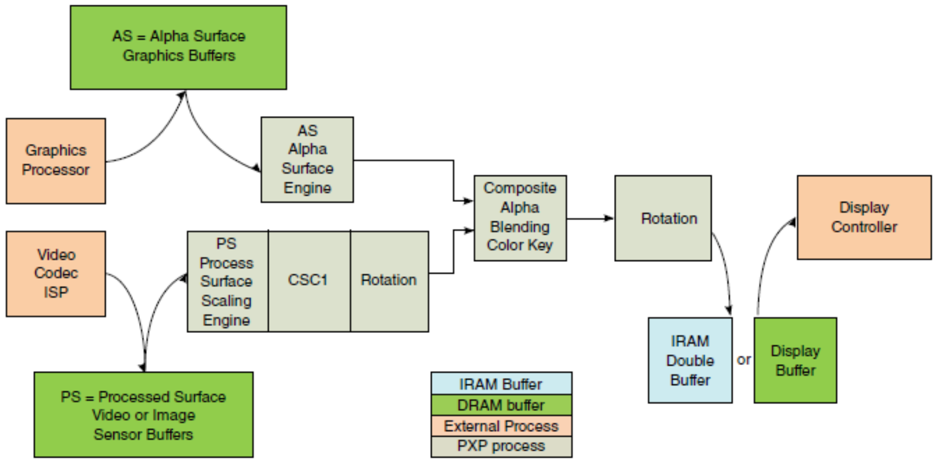


Figure 1. Block diagram

By integrating multiple blocks, remove intermediate buffer operations to external memory, reducing external memory bandwidth, power, and software control complexity. The PXP combines the following into a single processing engine:

- Scaling

- Color Space Conversion (CSC)
- Rotation

The main features of PXP include:

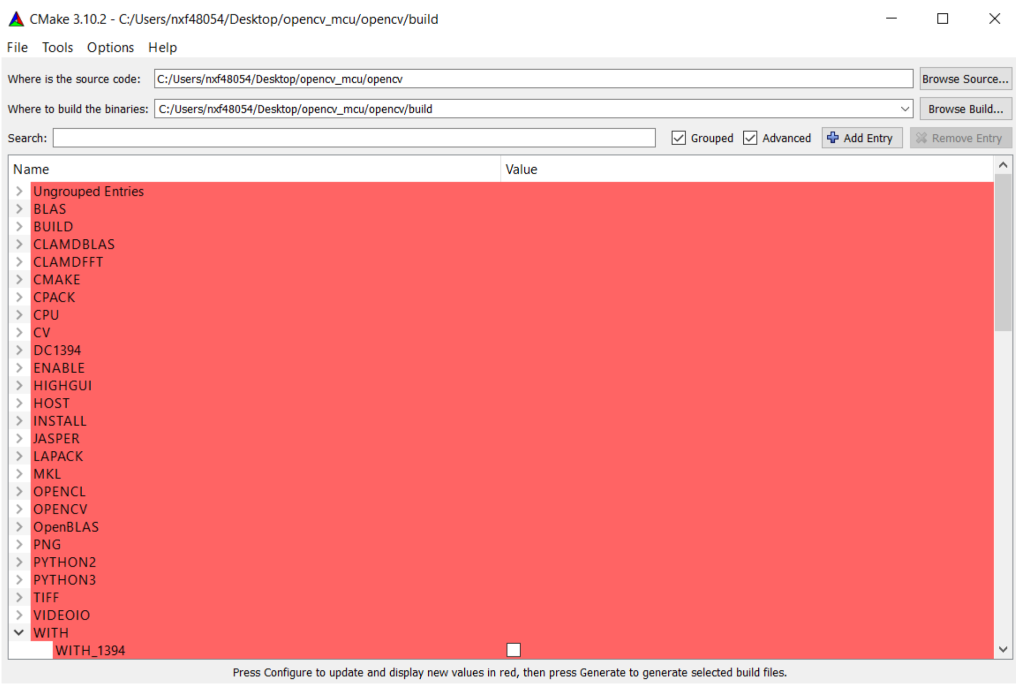
- BitBlit
- Multiple input/output format support, including YUV/RGB/Grayscale
- Supports both RGB/YUV scaling
- Supports overlay with Alpha blending
- Supports Rotation of 0, 90, 180, and 270 degrees with vertical and horizontal flip options
- Color space conversion
- Image resize
- Standard 2D-DMA operation

3 Configure the OpenCV module to add PXP support

We already know that the OpenCV has a modular structure. However, who organizes these modules and tells the compiler which module is included to the final library? The answer is CMake.

CMake is an open-source and cross-platform family of tools, designed to build, test, and package software. CMake is used to control the software compilation process using simple platform and compiler independent configuration files, and generate native makefiles and workspaces that can be used in the compiler environment of your choice. Kitware creates the suite of CMake tools in response to the need for a powerful and cross-platform build environment for open-source projects.

But How does it work? To store the building files, create a folder, named **build** for example. And from here, open the CMake-gui (which is a GUI-tool for CMake, providing an easy-to-use interface):



Where is the source code: C:/Users/nxf48054/Desktop/opencv_mcu/opencv

Where to build the binaries: C:/Users/nxf48054/Desktop/opencv_mcu/opencv/build

Search:

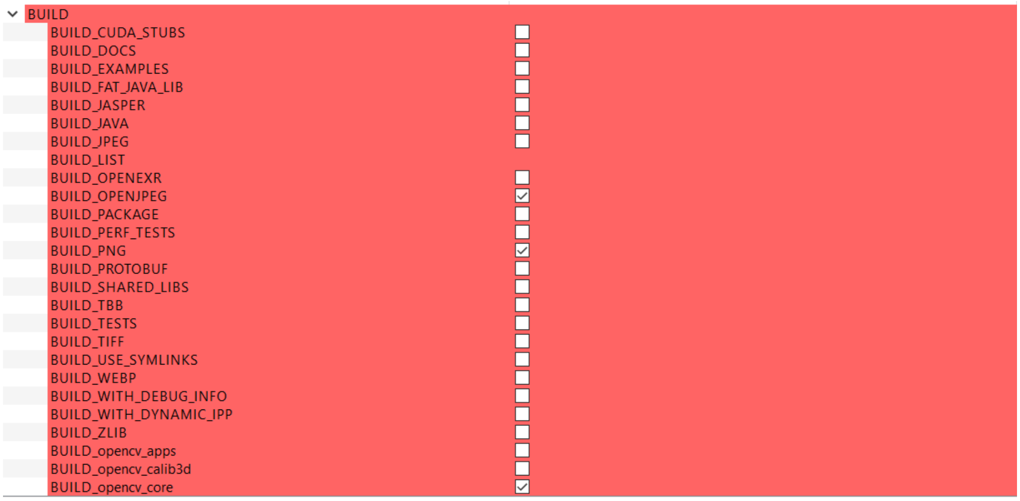
☑ Grouped ☑ Advanced

Ungrouped Entries
BLAS
BUILD
CLAMDBLAS
CLAMDFFT
CMAKE
CPACK
CPU
CV
DC1394
ENABLE
HIGHGUI
HOST
INSTALL
JASPER
LAPACK
MKL
OPENCL
OPENCV
OpenBLAS
PNG
PYTHON2
PYTHON3
TIFF
VIDEOIO
WITH
WITH_1394 ☐

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Figure 2. Default configurations

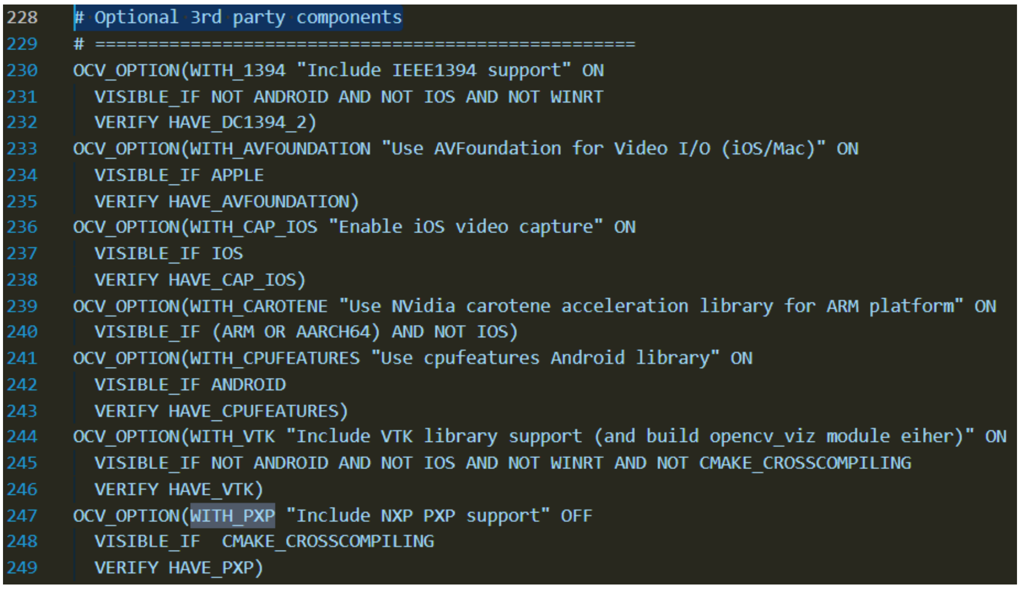
The new values are in red. We can imagine that each module has a dedicated symbol defined in a CMake file named *CMakeLists.txt*, which is the entry for a CMake project. After defining the symbol, it becomes the switch to control the code. If we define or set it as true, for example, we can find that under the build group, there are so many symbols that seem related to a module. So at last only the checked module is compiled into the final image. In this instance, the JPEG, PNG, and the *opencv_core* are compiled, but others not.



BUILD
BUILD_CUDA_STUBS ☐
BUILD_DOCS ☐
BUILD_EXAMPLES ☐
BUILD_FAT_JAVA_LIB ☐
BUILD_JASPER ☐
BUILD_JAVA ☐
BUILD_JPEG ☐
BUILD_LIST
BUILD_OPENEXR ☐
BUILD_OPENJPEG ☑
BUILD_PACKAGE ☐
BUILD_PERF_TESTS ☐
BUILD_PNG ☑
BUILD_PROTOBUF ☐
BUILD_SHARED_LIBS ☐
BUILD_TBB ☐
BUILD_TESTS ☐
BUILD_TIFF ☐
BUILD_USE_SYMLINKS ☐
BUILD_WEBP ☐
BUILD_WITH_DEBUG_INFO ☐
BUILD_WITH_DYNAMIC_IPP ☐
BUILD_ZLIB ☐
BUILD_opencv_apps ☐
BUILD_opencv_calib3d ☐
BUILD_opencv_core ☑

Figure 3. The build options

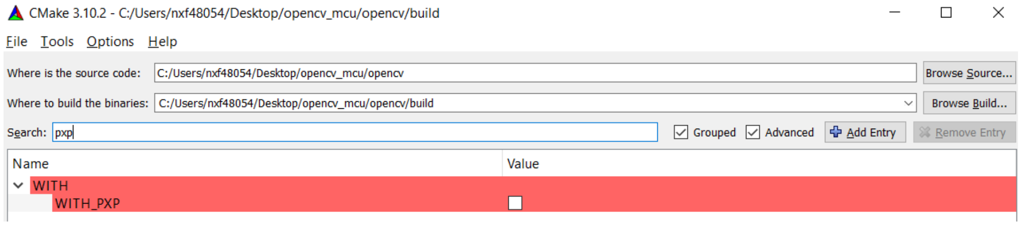
So, to add the PXP, add such a symbol as a switch to control the compiling flow. First, add the new option into *CMakeLists.txt*, like this:



```
# Optional 3rd party components
# ==================================================
OCV_OPTION(WITH_1394 "Include IEEE1394 support" ON
  VISIBLE_IF NOT ANDROID AND NOT IOS AND NOT WINRT
  VERIFY HAVE_DC1394_2)
OCV_OPTION(WITH_AVFOUNDATION "Use AVFoundation for Video I/O (iOS/Mac)" ON
  VISIBLE_IF APPLE
  VERIFY HAVE_AVFOUNDATION)
OCV_OPTION(WITH_CAP_IOS "Enable iOS video capture" ON
  VISIBLE_IF IOS
  VERIFY HAVE_CAP_IOS)
OCV_OPTION(WITH_CAROTENE "Use NVidia carotene acceleration library for ARM platform" ON
  VISIBLE_IF (ARM OR AARCH64) AND NOT IOS)
OCV_OPTION(WITH_CPUFEATURES "Use cpufeatures Android library" ON
  VISIBLE_IF ANDROID
  VERIFY HAVE_CPUFEATURES)
OCV_OPTION(WITH_VTK "Include VTK library support (and build opencv_viz module eiher)" ON
  VISIBLE_IF NOT ANDROID AND NOT IOS AND NOT WINRT AND NOT CMAKE_CROSSCOMPILING
  VERIFY HAVE_VTK)
OCV_OPTION(WITH_PXP "Include NXP PXP support" OFF
  VISIBLE_IF  CMAKE_CROSSCOMPILING
  VERIFY HAVE_PXP)
```

Figure 4. Add new entries

It is better to place the newline inside the Option 3rd party comments. Now we have defined a new symbol/variable name **WITH_PXP**. Reopen the *CMake-gui.exe* to check if it works.



CMake 3.10.2 - C:/Users/nxf48054/Desktop/opencv_mcu/opencv/build

Where is the source code: C:/Users/nxf48054/Desktop/opencv_mcu/opencv

Where to build the binaries: C:/Users/nxf48054/Desktop/opencv_mcu/opencv/build

Search: pxp

☑ Grouped ☑ Advanced

Name	Value
WITH	
WITH_PXP	☐

Figure 5. The new symbol

The new variable works. But this is not the end. The variable is only visible to CMake, and the source code cannot see it. It means that we can only use this symbol outside the code, but not as macro to do the pre-compile. So we must add another line:

```
if(WITH_PXP)
 add_definitions(-DHAVE_PXP)
endif()
```

With the *add_definitions* function of CMake, we can pass the new symbol, *HAVE_PXP*, to the compiler as a pre-defined macro which is visible inside a code. We use it like this:

```
#if HAVE_PXP
…
#else
…
#endif        Now we have defined a new symbol
```

And the `if(WITH_PXP)` checks whether the WITH_PXP symbol is checked through the CMake-gui. If not, the macro is not defined.

Now we can rebuild the OpenCV library with the **WITH_PXP** option checked.

4 Write the code to enable the PXP

Step 3 told us a way to add the pxp-symbol into the Cmake build-system. This section takes the Resize function as an example to show how to write some code to integrate the PXP.

Consider that the PXP driver has many NXP’s headers and files. So we do not include the PXP drivers into OpenCV, and leave this to users. For who want to use the PXP, they must import the PXP drivers and also the PXP-HAL into the project. Inside the OpenCV, it only uses the API as a external symbol. As a result, only one thing is left. Find the resize function and insert the function call to the PXP. The resize of OpenCV is inside the *resize.cpp*:

```
void cv::resize( InputArray _src, OutputArray _dst, Size dsize,
           double inv_scale_x, double inv_scale_y, int
 interpolation )
```

After checking the code, we saw that there is already an OCL optimized code with a macro to call the real-function API:

```
CV_OCL_RUN(_src.dims() <= 2 && _dst.isUMat() && _src.cols() >
 10 && _src.rows() > 10,
     ocl_resize(_src, _dst, dsize, inv_scale_x, inv_scale_y,
 interpolation))
```

So we can take it as a reference and write our code:

```
CV_PXP_RUN(_src.dims() <= 2 && _dst.isMat() && (interpolation
 == INTER_LINEAR),
     resize_pxp(_src, _dst, dsize, inv_scale_x, inv_scale_y))
```

Please note that, our PXP can only do the resize with a specified interpolation: **INTER_LINEAR**. So we do a check about that. Otherwise, it calls the default function without any optimized.

The Macro is defined in the private.hpp, and like this:

```
#ifdef HAVE_PXP
int resize_pxp(cv::InputArray _src, cv::OutputArray _dst,
 cv::Size dsize, float fx=0, float fy=0, int rotateCode=-1, int
 flipCode=-2);
#define CV_PXP_RUN_(condition, func, ...)
             \
try \
{ \
   if ((condition) && func)                \
   {
            \
     return __VA_ARGS__;
          \
   } \
} \
catch (const cv::Exception& e) \
{ \
   CV_UNUSED(e); /* TODO: Add some logging here */ \
}
```

```
#else
#define CV_PXP_RUN_(condition, func, ...)
#endif
#define CV_PXP_RUN(condition, func) CV_PXP_RUN_(condition,
 func)
```

Now the OpenCV side is done. Once you check the WITH_PXP, the OpenCV has a chance to call PXP. The next step is to write the PXP_HAL, including initialize the PXP and the function resize_pxp, create a file named nxp_pxp.cpp for example, and edit the file:

1. Define a PXP class:

```
#include “opencv2/opencv.hpp”
using namespace cv;
/
**************************************************************
                   construct the pxp class
**************************************************************/
class pxp_handler{
     public:
          pxp_handler();
          int resize(cv::InputArray _src, cv::OutputArray
 _dst, cv::Size dsize, float fx=0, float fy=0, int
 rotate_code=-1, int flip_code=-2);
};
```

2. Function implementation

```
pxp_handler::pxp_handler(){
  pxp_init();
}
static inline void* get_pxp_handler(){
  static pxp_handler  s_pxp_handler;
  return (void*)(&s_pxp_handler);
}
int resize_pxp(cv::InputArray _src, cv::OutputArray _dst,
 cv::Size dsize, float fx=0, float fy=0, int rotate_code=-1,
 int flip_code=-2){
  pxp_handler* handler = (pxp_handler*)get_pxp_handler();
  return handler-->resize(_src, _dst, dsize, fx, fy,
 rotate_code, flip_code);
}
```

   Here we define a static class instance. Once it is used, the construct function is called directly.
   And as a result, we finish the preparation for the PXP. Make sure that the *pxp_init()* only calls once.

3. There is one thing to be considered first before we start writing the **pxp_handler::resize**. There is a limitation of the PXP to handle the input image. It can support both ARGB32, RGB565 or the YUV data, except the RGB24:

```
/*! @brief PXP process surface buffer pixel format. */
typedef enum _pxp_ps_pixel_format
{
  kPXP_PsPixelFormatRGB888    = 0x4,  /*!< 32-bit pixels
 without alpha (unpacked 24-bit format) */
  kPXP_PsPixelFormatRGB555    = 0xC,  /*!< 16-bit pixels
 without alpha. */
```

```
  kPXP_PsPixelFormatRGB444    = 0xD,  /*!< 16-bit pixels
 without alpha. */
  kPXP_PsPixelFormatRGB565    = 0xE,  /*!< 16-bit pixels
 without alpha. */
  kPXP_PsPixelFormatYUV1P444  = 0x10, /*!< 32-bit pixels (1-
plane XYUV unpacked). */
  kPXP_PsPixelFormatUYVY1P422 = 0x12, /*!< 16-bit pixels (1-
plane U0,Y0,V0,Y1 interleaved bytes) */
  kPXP_PsPixelFormatVYUY1P422 = 0x13, /*!< 16-bit pixels (1-
plane V0,Y0,U0,Y1 interleaved bytes) */
  kPXP_PsPixelFormatY8        = 0x14, /*!< 8-bit monochrome
 pixels (1-plane Y luma output) */
  kPXP_PsPixelFormatY4        = 0x15, /*!< 4-bit monochrome
 pixels (1-plane Y luma, 4 bit truncation) */
  kPXP_PsPixelFormatYUV2P422  = 0x18, /*!< 16-bit pixels (2-
plane UV interleaved bytes) */
  kPXP_PsPixelFormatYUV2P420  = 0x19, /*!< 16-bit pixels (2-
plane UV) */
  kPXP_PsPixelFormatYVU2P422  = 0x1A, /*!< 16-bit pixels (2-
plane VU interleaved bytes) */
  kPXP_PsPixelFormatYVU2P420  = 0x1B, /*!< 16-bit pixels (2-
plane VU) */
  kPXP_PsPixelFormatYVU422    = 0x1E, /*!< 16-bit pixels (3-
plane) */
  kPXP_PsPixelFormatYVU420    = 0x1F, /*!< 16-bit pixels (3-
plane) */
} pxp_ps_pixel_format_t;
```

The news is bad. The common pixel format of the OpenCV is RGB24. Each pixel has three bytes. If we want to use the PXP, a function used to convert RGB24 to RGB565 is necessary. But the challenge is that we must design the logic of the algorithm carefully to guarantee the efficiency, to avoid that the algorithm becomes a bottleneck against the PXP.After several attempts. We got the below code:

```
#define zip_v(v, bits, shift_l) ((v >> (8 - bits)) <<
 shift_l)
#define RGB2RGB565(r, g, b) \
       (zip_v(r, 5, 11) | zip_v(g, 6, 5) | zip_v(b, 5, 0))
typedef struct{
    union{
          rgb_clip_t rgb_clip[4];
          uint8_t rgb[12];
          uint32_t rgbx4[3];
    }rgb_rgb565;
}color_t;
int RGB888toRGB565_struct(uint32_t *prgb888, uint32_t
 *prgb565, uint32_t pixCnt) {
    color_t color;
    uint32_t rgb565x2[2];
    while (pixCnt >= 4) {
        memcpy(color.rgb_rgb565.rgbx4, prgb888, 12);
        prgb888 += 3;
        rgb565x2[0] = RGB2RGB565(color.rgb_rgb565.rgb[2],
 color.rgb_rgb565.rgb[1],
        color.rgb_rgb565.rgb[0]) |
        RGB2RGB565(color.rgb_rgb565.rgb[5],
 color.rgb_rgb565.rgb[4],
        color.rgb_rgb565.rgb[3]) << 16 ;
```

```
        rgb565x2[1] = RGB2RGB565(color.rgb_rgb565.rgb[8],
 color.rgb_rgb565.rgb[7],
        color.rgb_rgb565.rgb[6]) |
        RGB2RGB565(color.rgb_rgb565.rgb[11],
 color.rgb_rgb565.rgb[10],
        color.rgb_rgb565.rgb[9]) << 16 ;

        memcpy(prgb565, rgb565x2, 8);
        prgb565 += 2;
        pixCnt -= 4;
      }
      return 0;
}
void bgr2rgb565(cv::InputArray _src, cv::OutputArray _dst,
 uint32_t image_len){
     Mat src = _src.getMat();
     _dst.create(src.size(), CV_16U);
     Mat dst = _dst.getMat();
     uint16_t *dst_rgb16 = (uint16_t*)dst.data;
     uint8_t  *src_rgb8 = src.data;
     RGB888toRGB565_struct((uint32_t*)src_rgb8, (uint32_t
 *)dst_rgb16, image_len);
}
```

Now the final code of the **pxp_handler::resize** is here:

```
int pxp_handler::resize(cv::InputArray _src, cv::OutputArray
 _dst, cv::Size dsize, float fx, float fy, int rotate_code,
 int flip_code){
    Mat src = _src.getMat();
    Mat dst = _dst.getMat();
    if(src.data == dst.data || dst.data == nullptr){
        // only 90/270 need create new one
        if((rotate_code == ROTATE_90_CLOCKWISE) ||
 (rotate_code == ROTATE_90_COUNTERCLOCKWISE))
              _dst.create(Size(dsize.height, dsize.width),
 src.type());
        else
              _dst.create(Size(dsize.width, dsize.height),
 src.type());
              dst = _dst.getMat();
    }
    uint32_t src_w = src.cols, src_h = src.rows, src_c =
 src.channels(), src_ptr = (uint32_t)src.data;
    uint32_t dst_w = dst.cols, dst_h = dst.rows, dst_c =
 dst.channels(), dst_ptr = (uint32_t)dst.data;
    Mat tmp(src);
    if(src_c != 2){
        bgr2rgb565(src, tmp, src_w * src_h);
        src_ptr = (uint32_t)tmp.data;
    }
    PXP_CFG(dsize.width, dsize.height);
    PXP_SetProcessSurfaceScaler(PXP, src_w, src_h,
 dsize.width, dsize.height);
    WAIT_PXP_DONE();
    return 1;
}
```

For more details, see AN13754SW.

Now we have both the OpenCV code which has integrated the PXP acceleration and the PXP_HAL. The next step is to create a project to invalidate the new code.

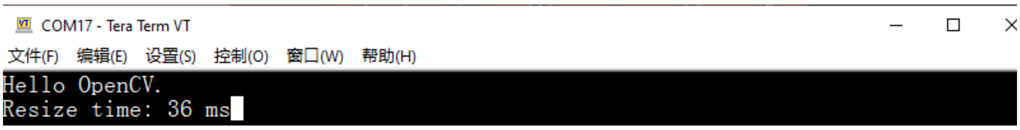
5 Deploy the optimized OpenCV on MIMXRT1170 EVK

Next step is to deploy the new library on the board. We make an example to show that the library can work on our board. You can find the project from the attachment. Copy all the libraries into the "**source/library**" folder within the project and test images into the "**source/data**" folder at the same time. Here we also use the famous *lena.jpg* as the input:

Figure 6. The famous Lena as test data

The picture is a decoded *jpg* image with a shape (512, 512). Then we call the function to resize the image to a new shape (320, 240) with different OpenCV library. One is the default and the other is optimized with PXP. After building and downloading the project into board, boot up the EVK board and start debugging.

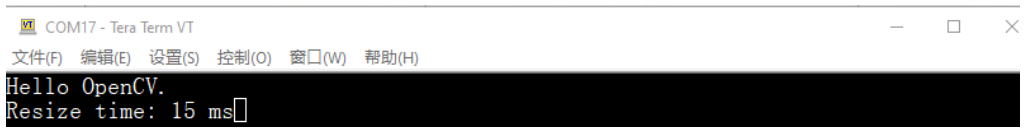
Figure 7 shows the resize time of the default CV.



```
Hello OpenCV.
Resize time: 36 ms
```

Figure 7. Default time

Figure 8 shows the resize time of the optimized CV.



```
Hello OpenCV.
Resize time: 15 ms
```

Figure 8. Time after optimized

With the PXP, the optimized function can save 21 ms each in this test case (from a (512, 512) to (320, 240)), and the performance boosts about 58 %.

6 Reference

The file mentioned in the article is shipped in the attachments.

7 Revision history

Rev.	Date	Description
0	20 October 2022	Initial release

8 Legal information

8.1 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.

8.2 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.

8.3 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 Introduction 2
2 PXP 2
3 Configure the OpenCV module to add PXP support 3
4 Write the code to enable the PXP 6
5 Deploy the optimized OpenCV on MIMXRT1170 EVK 10
6 Reference 11
7 Revision history 11
8 Legal information 12

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 20 October 2022
Document identifier: AN13754