	AMP MCU Software	
ADAS VISION	Revision <1.0>	Page 1 of 20
	AMCU_SW	

Vision SDK Programming Examples

ABSTRACT:

The document describes main programming issues and demonstrates the C++ code leading to their resolution..

KEYWORDS:

S32V234, Programming, C++, ISP, APEX

APPROVED:

AUTHOR	SIGN-OFF SIGNATURE #1	SIGN-OFF SIGNATURE #2
Rostislav Hulik		

Revision History

VERSION	DATE	AUTHOR	CHANGE DESCRIPTION
1.0	11-May-16	Rostislav Hulik	First draft

Table of Contents

Vision SDK Programming Examples **1**

1 Introduction **4**

1.1 Purpose 4
1.2 Audience Description 4

2 Hello World application **5**

2.1 BUILD.mk 5
2.2 Linked libraries 6
2.3 Application implementation 6
2.4 Build of the application 6

3 APEX Processing application **7**

3.1 APEX BUILD.mk 7
3.2 BUILD.mk 8
3.3 Application implementation 10
3.4 Build of the application 12

4 ISP camera input + DCU output **13**

4.1 BUILD.mk 13
4.2 Application implementation 15

5 ISP + APEX + DCU output **18**

5.1 APEX BUILD.mk 18
5.2 BUILD.mk 18
5.3 Application implementation 18
5.4 Build of the application 20

1 Introduction

In this document, main programming issues are presented and an example code is provided to show possible solution.

First, the hello application is described. After that, a simple APEX, ISP and OpenCV applications are provided and the text is finished by the combination of ISP, APEX and DCU output example.

1.1 Purpose

The purpose of this document is to present the S32V234 Vision SDK and help users to bring up the applications quickly.

1.2 Audience Description

This document is intended to S32V234 Vision SDK users.

2 Hello World application

The hello world application shows the easiest application to be used on S32V234. The corresponding demo to this example is available in:

s32v234_sdk/demos/other/hello

The application is supposed to print some text on UART – in this case, 1000 times.

2.1 BUILD.mk

The build info for Linux, standalone and INTEGRITY is defined in BUILD.mk file. This file consists of source definitions, linked libraries, include directories and other build related info.

```
SDK_ROOT := ../../../..

ARM_APP = hello

ARM_APP_SRCS =                                                     \
    main.cpp                                                       \

ARM_INCS =                                                         \
    -I.

######################################################################
# STANDALONE SPECIFIC INCLUDES
######################################################################

ifneq (,$(findstring -sa,$(ODIR)))

ARM_APP_LIBS +=                                                    \
    $(SDK_ROOT)/libs/startup/v234ce_standalone/$(ODIR)/libv234ce.a \
    $(SDK_ROOT)/libs/io/i2c/$(ODIR)/libi2c.a                       \
    $(SDK_ROOT)/libs/io/uartlinflex_io/$(ODIR)/liblinflex.a

endif
```

- **ARM_APP** defines the name of application (resulting name will be ARM_APP.elf)
- **ARM_APP_SRCS** contains all the sources build for the application
- **ARM_INCS** contains include paths
- **ARM_APP_LIBS** consists of libraries linked to the resulting application. Note in this case, we add v234ce specific libs for standalone build due to entry point definition.

2.2 Linked libraries

In this demo, we won't need any external libraries to be linked in, except the standalone path, which needs to link in the **libv234ce** containing entry point, **libi2c** as a I2C driver (in entry) and **liblinflex** as a linflex UART driver for printf. All those libraries are not necessary to be in Linux nor INTEGRITY builds.

2.3 Application implementation

The application itself is defined in **main.cpp** file and consists of a simple **main** function with a printf loop. All the necessary entry routines for standalone are automatically added at the beginning of the application:

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  int i = 0;
  printf("Hello World...\n");
  while (i < 1000)
  {
    printf("Iteration %i\n", i++);
  }
  return 0;
}
```

2.4 Build of the application

Execute "make" in one of build-* directories to build a specific flavor (please see the User Guide for detailed description of the build flavors)

3 APEX Processing application

The APEX Processing application demonstrates the APEX use for the simple image input (for example via input png file) and display of the processed image through DCU. The corresponding demo to this example is available in:

s32v234_sdk/demos/apex/apex_gauss5x5_cv

The application is supposed to read the input file via OpenCV function (or include the hex dump file in standalone case), execute the APEX Gauss 5x5 kernel and display the original and smoothed image through DCU.

3.1 APEX BUILD.mk

The APEX build is separated from the main build and the APEX Graphs needs to be invoked in a separate build folder prior to application build. The BUILD.mk file for APEX graphs is located in **graphs** subfolder.

```
SDK_ROOT := ../../../../..
KERNELS  := $(SDK_ROOT)/kernels/apu

APU_GRAPH_LIBS =                                                      \
    $(KERNELS)/filtering_kernels/build-apu-tct-sa-d/filtering_kernels.a

APU_GRAPH_INCS = -I..

APU_GRAPHS = ../apu_gauss_5x5_apu_process_desc.hpp
```

- **APU_GRAPH_LIBS** consists of all used APEX kernel libraries – the libs where kernels used in graphs are implemented.
- **APU_GRAPH_INCS** are the APEX build include directories
- **APU_GRAPHS** specifies the kernel process description files, as described in ACF guide.

3.2 BUILD.mk

The build info for Linux, standalone and INTEGRITY is defined in BUILD.mk file. This file consists of source definitions, linked libraries, include directories and other build related info.

```
SDK_ROOT := ../../../..

ARM_APP = apex_gauss5x5_cv
ARM_APP_SRCS = main.cpp
ARM_INCS +=                                                    \
    -I$(SDK_ROOT)/libs/apex/icp/include                        \
    -I$(SDK_ROOT)/libs/apex/acf/include                        \
    -I$(SDK_ROOT)/libs/apex/drivers/user/include               \
    -I$(SDK_ROOT)/libs/io/frame_io/include                     \
    -I$(SDK_ROOT)/libs/io/dcu/include                          \
    -I$(SDK_ROOT)/libs/utils/oal/user/include                  \
    -I$(SDK_ROOT)/libs/utils/common/include                    \
    -I../graphs/build-apu-tct-sa-d                             \
    -I$(OPENCV_ROOT)/include

ARM_APP_LIBS =                                                 \
    $(SDK_ROOT)/libs/apex/acf/$(ODIR)/libacf.a                 \
    $(SDK_ROOT)/libs/apex/icp/$(ODIR)/libicp.a                 \
    $(SDK_ROOT)/libs/io/frame_io/$(ODIR)/libframe_io.a         \
    $(SDK_ROOT)/libs/apex/drivers/user/$(ODIR)/libapexdrv.a    \
    $(SDK_ROOT)/libs/utils/oal/user/$(ODIR)/liboal.a           \
    $(SDK_ROOT)/libs/utils/common/$(ODIR)/libcommon.a          \
    $(SDK_ROOT)/libs/io/dcu/$(ODIR)/libdcu.a                   \
    $(SDK_ROOT)/libs/utils/communications/$(ODIR)/lib_communications.a
```

- **ARM_APP** defines the name of application (resulting name will be ARM_APP.elf)
- **ARM_APP_SRCS** contains all the sources build for the application
- **ARM_INCS** contains include paths. Note the OPENCV_ROOT variable available for different build flavors if OpenCV path is needed to be included.
- **ARM_APP_LIBS** consists of libraries linked to the resulting application.
  - **libacf** – Library containing ACF functions, must be present when APEX is needed.
  - **libicp** – Library contains main ACF DataDescriptor class – the data structure encapsulating the image.
  - **libframe_io –** Library needed for I/O from camera to display. Here used for DCU output routine.
  - **libapexdrv** – APEX Low level driver.
  - **liboal** – OAL – The library needed for contiguous memory allocations of images for APEX processing.

- **libcommon –** Library contains some common functions, such as time computation etc.
- **libdcu –** DCU driver library, needed with combination with libframe_io
- **lib_communications –** Needed to be included with combination with APEX libraries

```
######################################################################
# STANDALONE SPECIFIC INCLUDES
######################################################################
ifneq (,$(findstring -sa,$(ODIR)))

ARM_APP_LIBS += (…)
ARM_LDOPTS += (…)

######################################################################
# INTEGRITY SPECIFIC INCLUDES
######################################################################
else
ifneq (,$(findstring -integrity,$(ODIR)))

ARM_LDOPTS += (…)

######################################################################
# LINUX SPECIFIC INCLUDES
######################################################################
else

ARM_LDOPTS += (…)

endif
endif
```

Note the different **ARM_LDOPTS** for each platform – this is mainly due to different OpenCV build used for reading the input image. The build differs in multiple ways (OpenCV components), thus must be different in linked in libraries. The SA build also contains the SA specific libs known from Hello example.

3.3 Application implementation

The application itself is defined in **main.cpp** file and consists of a simple **main** function. The code shown here does not contain the includes and defines present in the demo file – please refer to that if a complete main.cpp file is needed.:

```
int main(int, char**)
{
  #ifndef APEX2_EMULATE
    OAL_Initialize();
    ACF_Init();
    using namespace icp;
  #endif

  #ifdef __STANDALONE__
    Mat in(256, 256, CV_8UC1, in_grey_256x256);
  #else
    Mat in = imread(INPUT_ROOT"in_grey_256x256.png", 0);
  #endif
```

The beginning of the main function consists of initialization of OAL and ACF. These two functions need to be executed every time the user wants to use ACF, since the OAL contains necessary connections to the memory allocator and ACF init provides successful connection to the ACF framework. These functions are executed only for real HW application, for emu-lib, the functions must be filtered out.

Next step is to load the input image. This is done by reading the included hex dump for standalone, the rest of builds will read the png image using OpenCV. Note both implementation will create an OpenCV Mat wrapper around the input image.

```
int lSrcWidth = in.cols;
int lSrcHeight = in.rows;
int lRetVal = 0;

icp::DataDescriptor lInput0(lSrcWidth, lSrcHeight, icp::DATATYPE_08U);
icp::DataDescriptor lOutput0(lSrcWidth, lSrcHeight, icp::DATATYPE_08U);
icp::DataDescriptor lOutput1(lSrcWidth, lSrcHeight, icp::DATATYPE_08U);

if (lInput0.IsOK() != 0 && lOutput0.IsOK() != 0 && lOutput1.IsOK() != 0)
{ …
```

After initial data are loaded, the DataDescriptor instances are created and allocated (in the constructor). The method IsOK() is used to check if all the allocations were executed correctly.

```
memcpy(lInput0.GetDataPtr(), in.data, lSrcWidth*lSrcHeight);

APU_GAUSS_5X5 process0(APEX_APEX0);
APU_GAUSS_5X5 process1(APEX_APEX1);

lRetVal |= process0.Initialize();
lRetVal |= process0.ConnectIO("INPUT_0", lInput0);
lRetVal |= process0.ConnectIO("OUTPUT_0", lOutput0);

lRetVal |= process1.Initialize();
lRetVal |= process1.ConnectIO("INPUT_0", lInput0);
lRetVal |= process1.ConnectIO("OUTPUT_0", lOutput1);

lRetVal |= process0.Start();
lRetVal |= process1.Start();

lRetVal |= process0.Wait();
lRetVal |= process1.Wait();
```

After all the data are initialized and allocated, we can continue with the processing itself. For demonstration purposes, both APEX instances are executed simultaneously computing the same algorithm.

First, we need to copy the input image (in.data) to the lInput0 DataDescriptor. **Note we cannot use the in.data, because it's not ensured it will be physically contiguous.** The APEX will DMA the data in and out APEX memory, so we need the contiguous blocks of memory – therefore, DataDescriptor is used.

The APEX_GAUSS_5X5 constructor creates process0 and process1, the ACF process instances linked with specific APEX device.

As a next step, we need to link the ACF Graph I/O ports with concrete data – **ConnectIO** functions. The input will be the same for both processes – the copied input image. The output however is a different buffer.

Last steps consists of starting both APEX instances and wait for their end. The user can execute whatever ARM code if needed, between Start and Wait functions to parallelize APEX and ARM processing.

```
#ifdef __STANDALONE__
  io::FrameOutputDCU output(1280, 720,  io::IO_DATA_DEPTH_08, CHNL_CNT);
#else
  io::FrameOutputV234Fb output(1280, 720, io::IO_DATA_DEPTH_08, CHNL_CNT);
#endif

void *lp_buffer = OAL_MemoryAllocFlag(
                  1280*720*3,
                  OAL_MEMORY_FLAG_ALIGN(ALIGN2_CACHELINE) |
                  OAL_MEMORY_FLAG_CONTIGUOUS);
memset(lp_buffer, 0, 1280*720*3);
```

The DCU initialization must be different too for STANDALONE and for Linux (INTEGRITY does not support the DCU output). The difference is in io::FrameOutputDCU/io::FrameOutputV234Fb initialization, further use is common for both builds.

The **lp_buffer** needs to be allocated (OAL_MemoryAllocFlag) as contiguous – this memory will represent the back buffer, where we paint the display output. The back buffer will be set to “0”, i.e. black color.

```
cv::Mat output_mat = cv::Mat(720, 1280, CV_8UC3, lp_buffer);

cvtColor(out0, out0, CV_GRAY2RGB);
cvtColor(out1, out1, CV_GRAY2RGB);
cvtColor(in, in, CV_GRAY2RGB);

in.copyTo(output_mat(cv::Rect(0, 232, 256, 256)));
out0.copyTo(output_mat(cv::Rect(300, 232, 256, 256)));
out1.copyTo(output_mat(cv::Rect(600, 232, 256, 256)));

output.PutFrame(output_mat.data);
OAL_MemoryFree(lp_buffer);
```

The output buffer organization is done via OpenCV functions. First, the helper **output_mat** is created – the OpenCV wrapper around the **lp_buffer**. All three displayed images (input, output from APEX0 and APEX1) are converted to RGB (the output buffer is RGB).

The OpenCV copyTo function is used to copy the images to the specified region.

Final step consists of **PutFrame** function, which will draw the buffer to the screen.

3.4 Build of the application

Execute “make allsub” in the graphs/build-apu-tct-sa-d directory and then “make” in one of build-* directories to build a specific flavor (please see the User Guide for detailed description of the build flavors)

4 ISP camera input + DCU output

The ISP Camera input and DCU output application demonstrates a simple use of ISP and DCU output. The application runs in infinite loop and shows the direct camera input to the screen. The corresponding demo to this example is available in:

s32v234_sdk/demos/isp/isp_csi_dcu

The application is supposed to read the camera input via SDI, convert it to RGB (during ISP grabbing graph) and display it to screen.

4.1 BUILD.mk

The build info for Linux, standalone is defined in BUILD.mk file. This file consists of source definitions, linked libraries, include directories and other build related info.

```
ARM_APP = isp_csi_dcu
ISP_GRAPH = mipi_simple

VPATH = ../src/lib

ARM_APP_SRCS += main.cpp

ARM_APP_LIBS +=                                                       \
    $(SDK_ROOT)/libs/io/frame_io/$(ODIR)/libframe_io.a               \
    $(SDK_ROOT)/libs/io/sdi/$(ODIR)/libsdi.a                         \
    $(SDK_ROOT)/libs/io/gdi/$(ODIR)/libgdi.a                         \
    $(SDK_ROOT)/libs/isp/csi/user/$(ODIR)/libcsidrv.a                \
    $(SDK_ROOT)/libs/utils/log/$(ODIR)/liblog.a                      \
    $(SDK_ROOT)/libs/isp/sequencer/user/$(ODIR)/libseqdrv.a          \
    $(SDK_ROOT)/libs/isp/fdma/user/$(ODIR)/libfdmadrv.a              \
    $(SDK_ROOT)/libs/utils/oal/user/$(ODIR)/liboal.a                 \
    $(SDK_ROOT)/libs/isp/sram/user/$(ODIR)/libsramdrv.a              \
    $(SDK_ROOT)/isp/firmware/$(ODIR)/sequencer.a                     \
    $(SDK_ROOT)/isp/graphs/$(ISP_GRAPH)/$(ODIR)/$(ISP_GRAPH).a       \
    $(SDK_ROOT)/libs/utils/common/$(ODIR)/libcommon.a
```

Along with some defines already described in the previous sections, there are some specific for the ISP processing:

- **ISP_GRAPH** defines the ISP graph used in the application. This variable is used also in linked in libs.
- **ARM_APP_LIBS** consists of libraries linked to the resulting application. Only the libs not mentioned in the
    - **libsdi** –

- **libgdi** –
- **libcsidrv** –
- **liblog** –
- **libseqdrv** –
- **libfdmadrv** –
- **libsramdrv** –
- **sequencer** –
- **$(ISP_GRAPH)** –

Note the different **ARM_LDOPTS** for each platform.

```
######################################################################
# STANDALONE SPECIFIC INCLUDES
######################################################################
ifneq (,$(findstring -sa,$(ODIR)))

ARM_APP_LIBS +=                                                      \
    $(SDK_ROOT)/libs/startup/v234ce_standalone/$(ODIR)/libv234ce.a   \
    $(SDK_ROOT)/libs/io/i2c/$(ODIR)/libi2c.a                         \
    $(SDK_ROOT)/libs/io/semihost/$(ODIR)/libSemihost.a               \
    $(SDK_ROOT)/libs/io/uartlinflex_io/$(ODIR)/liblinflex.a          \
    $(SDK_ROOT)/libs/io/dcu/$(ODIR)/libdcu.a

######################################################################
# LINUX SPECIFIC INCLUDES
######################################################################
else

ARM_APP_LIBS +=                                                      \
    $(SDK_ROOT)/libs/isp/jpegdec/user/$(ODIR)/libjpegdecdrv.a        \
    $(SDK_ROOT)/libs/isp/h264enc/user/$(ODIR)/libh264encdrv.a        \
    $(SDK_ROOT)/libs/isp/viu/user/$(ODIR)/libviudrv.a

endif
```

4.2 Application implementation

The application itself is defined in **main.cpp** file and consists of a simple **main** function. The code shown here does not contain the includes and defines present in the demo file – please refer to that if a complete main.cpp file is needed.:

```
//*** Init DCU Output ***
#ifdef __STANDALONE__
  io::FrameOutputDCU lDcuOutput(WIDTH,
                                HEIGHT,
                                io::IO_DATA_DEPTH_08,
                                CHNL_CNT);
#else
  // setup Ctrl+C handler
  if(SigintSetup() != SEQ_LIB_SUCCESS)
  {
    VDB_LOG_ERROR("Failed to register Ctrl+C signal handler.");
    return -1;
  }

  printf("Press Ctrl+C to terminate the demo.\n");
  io::FrameOutputV234Fb lDcuOutput(WIDTH,
                                   HEIGHT,
                                   io::IO_DATA_DEPTH_08,
                                   CHNL_CNT);
#endif
```

TODO

```
lRes = sdi::Initialize(0);

sdi_grabber *lpGrabber = new(sdi_grabber);
lpGrabber->ProcessSet(gpGraph, &gGraphMetadata);

int32_t lCallbackUserData = 12345;
lpGrabber->SeqEventCallBackInstall(&SeqEventCallBack,
&lCallbackUserData);

sdi_FdmaIO *lpFdma = (sdi_FdmaIO*)lpGrabber->IoGet(SEQ_OTHRIX_FDMA);

GDI_ImageDescriptor lFrmDesc = GDI_ImageDescriptor(WIDTH, HEIGHT,
RGB888);
lpFdma->DdrBufferDescSet(0, lFrmDesc);

lpFdma->DdrBuffersAlloc(0, DDR_BUFFER_CNT);

lpGrabber->PreStart();

SDI_Frame lFrame;

lpGrabber->Start();
```

TODO

```
for(;;)
{
  lFrame = lpGrabber->FramePop();
  if(lFrame.mImage.mData == NULL)
  {
    printf("Failed to grab image number %u\n", lFrmCnt);
    break;
  }

  lDcuOutput.PutFrame((void*)(uintptr_t)lFrame.mImage.mData);
  if(lpGrabber->FramePush(lFrame) != LIB_SUCCESS)
  {
    break;
  }

  lFrmCnt++;

  #ifndef __STANDALONE__
    if(sStop)
    {
      break; // break if Ctrl+C pressed
    }
  #endif //#ifndef __STANDALONE__
}
```

TODO

```
if(lpGrabber)
{
  lpGrabber->Stop();
  lpGrabber->Release();
  delete(lpGrabber);
} // if grabber exists

#ifdef __STANDALONE__
  for(;;);  // *** don't return ***
#endif

lRes = sdi::Close(0);
if(lRes != LIB_SUCCESS)
{
  lRet = LIB_FAILURE;
}
```

5 ISP + APEX + DCU output

The ISP+APEX+DCU Processing application demonstrates the full pipeline starting with camera grabbed image by the ISP, APEX processing of the frame and DCU display to the screen. The corresponding demo to this example is available in:

s32v234_sdk/demos/apex/apex_isp_fast9

The application is supposed to read the camera via ISP, execute the APEX FAST9 corner detector and display the highlighted corners to the screen. This will run in an infinite frame loop.

5.1 APEX BUILD.mk

Please refer to section 3 for the detailed description of the APEX BUILD.mk.

5.2 BUILD.mk

The build info for Linux, standalone and INTEGRITY is defined in BUILD.mk file. This file consists of source definitions, linked libraries, include directories and other build related info. Please see the section 3 and 4 for the detailed description of the linked in libraries for ISP and APEX.

5.3 Application implementation

The application itself is defined in **main.cpp** file and consists of a simple **main** function. The code shown here does not contain the includes and defines present in the demo file – please refer to that if a complete main.cpp file is needed.

The whole ISP init is similar to the one described in the section 4. This chapter will describe only the differences used for APEX processing of the input image.

```
DataDescriptor dataThreshold(1, 1, DATATYPE_08U);
DataDescriptor dataMarkColorChannel(1, 1, DATATYPE_08U);
DataDescriptor dataOut(WIDTH, HEIGHT, DATATYPE_08U, 3, 1);
DataDescriptor dataIn;

((uint8_t*)dataThreshold.GetDataPtr())[0] = 10;
((uint8_t*)dataMarkColorChannel.GetDataPtr())[0] = 1;

APU_FAST9_COLOR process;

int lRetVal = 0;
lRetVal |= process.Initialize();
lRetVal |= process.ConnectIO("THRESHOLD", dataThreshold);
lRetVal |= process.ConnectIO("MARK_COLOR_CHANNEL", dataMarkColorChannel);
lRetVal |= process.ConnectIO("OUTPUT", dataOut);
```

Right before the main loop is initiated, the I/O data buffers need to be initialized. **dataThreshold**, **dataMarkColorChannel**, **dataOut** and **dataIn** DataDescriptors are created. All but **dataIn** are also allocated. The dataIn won't be allocated here, because we use it subsequentially as a wrapper class for the ISP camera buffer. The rest is set up as usual.

Also the APEX Fast9 process is created and the I/O ports are connected. Note that input port is **not connected** because it will be initialized after grabbing the camera image and it's pointer will change. Everytime the pointer to the data structure changes, the ConnectIO on that structure must be called.

```
while(1)
{
  lFrame = lGrabber.FramePop();
  if(lFrame.mImage.mData == NULL)
  {
    break;
  }

  dataIn.InitManual(WIDTH, HEIGHT,
                    lFrame.mImage.mData,
                    OAL_MemoryReturnAddress(lFrame.mImage.mData, ACCESS_PHY),
                    DATATYPE_08U, 3, 1);

  lRetVal |= process.ConnectIO("INPUT", dataIn);

  lRetVal |= process.Start();
  lRetVal |= process.Wait();

  lDcuOutput.PutFrame(dataOut.GetDataPtr(), false);

  if(lGrabber.FramePush(lFrame) != LIB_SUCCESS)
  {
    break;
  }
```

In the main loop, the camera frame is grabbed as in the section 4. After that, the **dataIn** structure is constructed on top of that (the pointers are passed instead of allocating the new DataDescriptor).

The **dataIn** is then connected to the process. Note this must be done for each frame because the input is multi-buffered, so the pointers of input change every frame.

After the setup, the APEX processing is started and we wait for the end. When the APEX processing ends, the PutFrame DCU function is called to push the altered image to the screen. Note no copy is between ISP input and DCU output, everything is done on site.

The FramePush function of lGrabber must be called at the end of while loop to synchronize the frames.

5.4 Build of the application

Execute “make allsub” in the graphs/build-apu-tct-sa-d directory and then “make” in one of build-* directories to build a specific flavor (please see the User Guide for detailed description of the build flavors)