ACF Architecture documentation

ABSTRACT:		
The document is a guide to the APEX driver structure and host-executed ACF framework which enables APEX-2 functionality on i.mx VSX.		
KEYWORDS:		
APEX-2, ACF, Driver, ICP, DataDescriptor, Kernel, Graph		
APPROVED:		
AUTHOR	**SIGN-OFF SIGNATURE #1**	**SIGN-OFF SIGNATURE #2**
Rostislav Hulik		

Revision History

VERSION	DATE	AUTHOR	CHANGE DESCRIPTION
0.0	24-March-14	Rostislav Hulik	First draft

Table of Contents

1 Introduction

This is an architecture documentation of the ACF framework, which consists of APEX Driver and host executed framework for enabling APEX-2 functionality on i.mx VSX hardware. In following document, each component is described from architectural and programming point of view.

1.1 Audience Description

This document is intended to the team of APEX-2/ACF Kernel developers.

References

Id	*Title*	*Location*

Table 1 References Table

1.2 Definitions, Acronyms, and Abbreviations

Term/Acronym	*Description*

1.3 Document Location

s32v234_sdk/docs/ ACF_ArchitectureGuide.docx

2 ACF Overview

The ACF Framework is the set of tools which enables the programming and use of APEX subsystem on i.mx VSX . The framework itself consists of several components which will be described separately in following sections:

APEX2 Driver

The APEX-2 Driver is the only part which accesses directly the APEX-2 memory mapped registers. It consists of kernel-space module and the user-space library which provides functions to program the APEX-2 hardware.

ICP

The ICP subcomponent is the place of data structure definition. In ACF framework, this is the place where the DataDescriptor class is defined, which encapsulates an 1D or 2D array of values (e.g. the image, filter kernel) to be passed to APEX-2.

ACF Host

ACF Host part of the ACF framework is the main tool used to program the APEX-2 subsystem. It consists of user-space functions and classes which permit the user to load a built APU kernel to the APEX-2 hardware, feed it with concrete data and execute it. The user should use this component to all operations on APEX-2.

ACF APEX

ACF framework has a part also for APU itself, which is added to the compiled APEX-2 kernel. The APEX-2 part of ACF is not the subject of this documentation, but will be described in a separate paper.

The whole concept is depicted on Figure 1. The only point the user uses for access the APEX-2 hardware is the ACF framework. Both ACF/ICP and APEX-2 Driver are using OAL functions to access OS layer to ensure portability between operating systems.

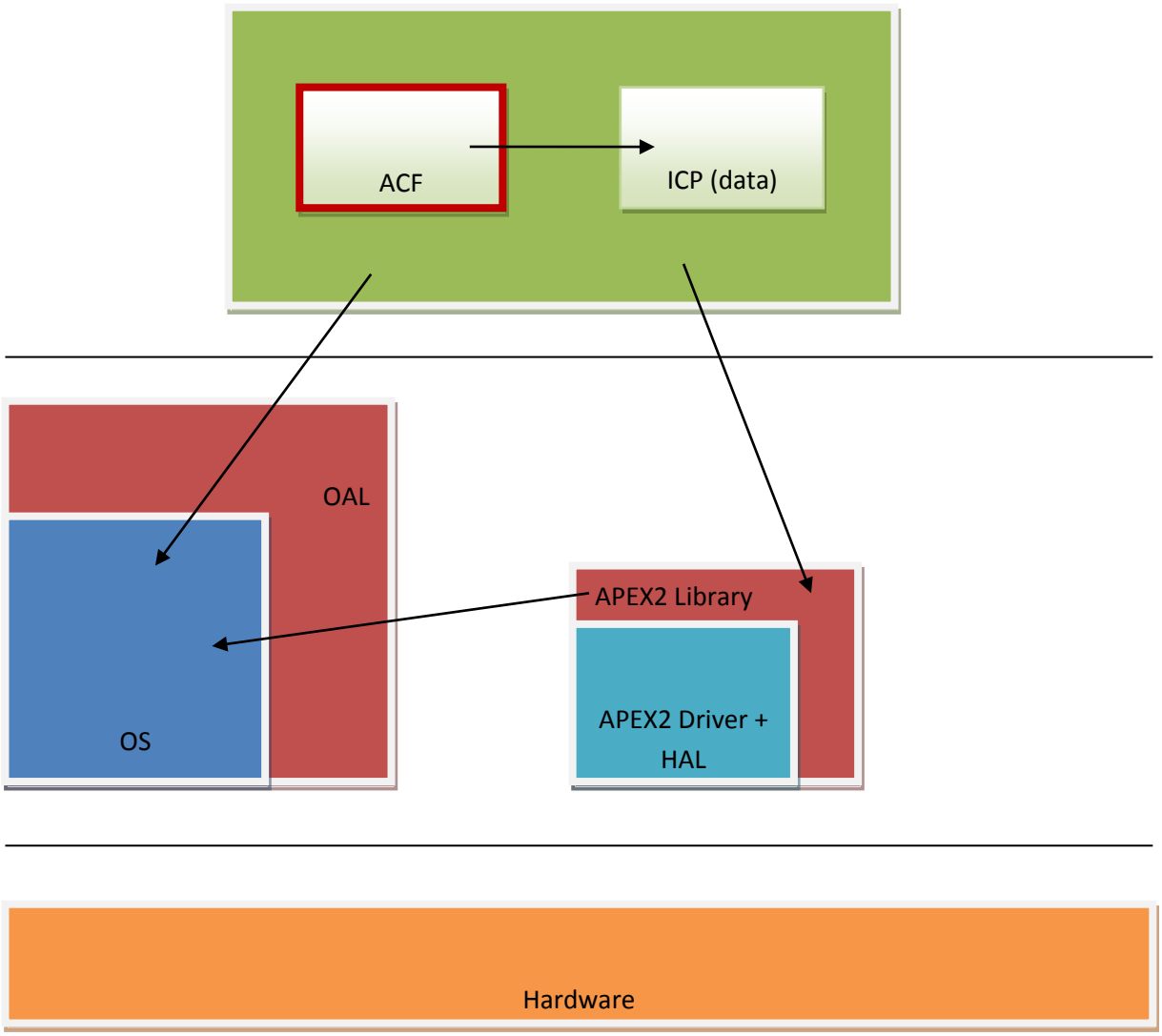

Figure 1 The ACF architecture concept

3 APEX2 Library + Driver

The Driver and user space library are the main point where the HW is accessed. All memory mapped registers are accessed via OAL memory map. The only OS-specific part is the interface between kernel space module (if present in OS concept) and user space module.

The structure of user space library is similar to the module one. The library's only function is to provide an understandable interface for user, encapsulating the whole user-kernel space communication into simple functions. The driver structure is visible on Figure 2.

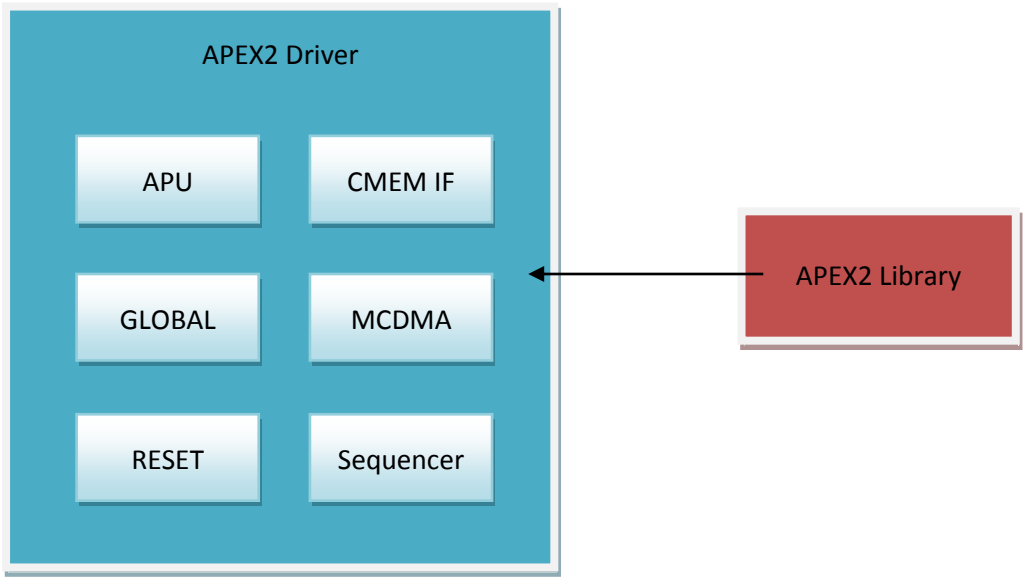

Figure 2 The Driver + Library structure

APU component

APU component contains functions which are tightly linked with APU, mainly the function for **loading the APU program** into various parts of memory (DMEM, PMEM) and communication with APU:

- `void LoadProgram (int id, const APEXDRIVER_LOAD_PROGRAM* program);`
  - The function is used for loading the compiled APU program into DMEM/PMEM
  - **id** Pararemeter specifies the APU ID (0 or 1).
- `void HostComRX (int id, int* msg);`
  - Function waits for APU message and returns it. The functionality can be used for testing the APU program, after the program load, the message should be raised. If not, consider the problem in APU program build.
  - **id** Pararemeter specifies the APU ID (0 or 1).
- `void HostComTX (int id, int msg);`

  - Function sends a specific message to the APU.
  - **id** Pararemeter specifies the APU ID (0 or 1).

CMEM IF component

The CMEM interface component contains functions for setting up the CMEM configuration. In current version, it contains the only function to specify the configuration of CMEM vector (32-bit for each APU or 64-bit combined).

- `void SetApuVectorCfg(int cfg);`
  - Function sets the CMEM vector configuration.
  - cfg can be equal to **CMEMIF_APU_0_CU_0_63** or **CMEMIF_APU_0_CU_0_31__APU_1_CU_32_63**

GLOBAL component

The GLOBAL component contains functions which are not specific to any part of APEX-2 hardware. Mainly, it consists of version information and specifications.

- `int GetHwVersion(void);`
  - Function returns a HW version number
- `int GetSwVersion(void);`
  - Function returns a SW version number

MCDMA component

The MCDMA component contains functions for servicing and programming the APEX-2 DMA subsystem. In current version, the interrupt notification is not available. The DMA-complete status is get by polling the interrupt register.

- `void FillWords(unsigned int w1, unsigned int w2, unsigned int w3, unsigned int w4);`
  - Function is used when DMA fill feature is used and sets the words to be filled in destination array.
  - w1 is a word for 8-bit, 16-bit and 32-bit value filling.
  - w2 is a word is used when 64-bit filling is selected (combined with w1)
  - w3 and w4 are used when 128-bit filling is selected (combined with w1 and w2)

- `void Start(unsigned int channel, APEX_MCDMA_TRANSFER transfer);`
  - The function starts a specified DMA channel with **transfer** parameters. The parameters are described further in this section.
- `void Wait(unsigned int channel);`
  - The wait function waits for **channel** done interrupt (by polling).
- `void InterruptClear(unsigned int channel);`

- The function clears all interrupts for specified **channel.**

The structure for DMA configuration specifies following fields:

32-bit source address	If fill is selected, the source address is equal to 1 << 31 | 0 << 4 | 7 << 0
32-bit destination address	
32-bit linked list address	LL can be used for defining multiple successive transfers. Please refer to CGV documentation for any details. The LL field can be null.

Config register	Settings:
	• Bit 0..1
	  ○ Dest. format
	  ○ 0 = 1D, 1 = 2D, 2 = 3D, 3 = 4D
	• Bit 2
	  ○ Dest. addressing type
	  ○ 0 = direct, 1 = indirect
	• Bit 3
	  ○ Dest transfer type
	  ○ 0 = mem based, 1 = peripheral based
	• Bit 4..5
	  ○ Source format
	  ○ 0 = 1D, 1 = 2D, 2 = 3D, 3 = 4D
	• Bit 6
	  ○ Source addressing type
	  ○ 0 = direct, 1 = indirect
	• Bit 7
	  ○ Source transfer type
	  ○ 0 = mem based, 1 = peripheral based
	• Bit 8..10
	  ○ Bus word size
	  ○ 0 = 8b, 1 = 16b, 2 = 32b, 3 = 64b, 4 = 128b
	• Bit 11..14
	  ○ DMA burst length
	  ○ 0..F beat
	• Bit 28
	  ○ When in LL, this list is last
	• Bit 29
	  ○ When in LL, loop on last
	• Bit 30
	  ○ Raise an interrupt when done
	• Bit 31
	  ○ Stop the MCDMA subsystem when done

Transfer size	Number of bytes to transfer
Span	• Bit 0..15 ○ Destination span • Bit 16..31

	○ Source span (0 if fill)
Width	When 2D transfer, this is an image width

RESET component

The reset component contains reset functions for every APEX-2 subsystem. The module consists of following functions, each of them resets, enables or disables corresponding system:

- `int All (void);`
- `int CMEM_DMA (void);`
- `int Sequencer (void);`
- `int DMA (void);`
- `int MCDMA (void);`
- `int CMEM_Interface (void);`
- `int APU (unsigned int id);`
- `int VU (unsigned int id);`
- `int DMEM (unsigned int id);`
- `int EnableAPU (unsigned int id);`
- `int DisableAPU (unsigned int id);`
- `int EnableSequencer (unsigned int id);`
- `int DisableSequencer (unsigned int id);`

Sequencer subsystem

The last subsystem of the sequencer contains the unction for sending commands to and programming of the sequencer. The functions are typically used after loading the APU program, when the user needs to start the execution of the program managed by the sequencer.

- `int SendCmd (int id, int kernel_id, int num, const int* parameters);`
  - Function sends parameters to the sequencer. The parameters typically contains the information about I/O ports of the APU graph and links the concrete physical addresses with them.
  - The parameter structure is described further in thes section.

- `void WaitInterrupts (int mask);`
  - The function waits for sequencer done interrupt (by polling).
  - When the interrupt is raised means the APU program execution is complete.

- `void ClearInterrupts (int mask);`
  - Clears the interrupts raised by the sequencer.

The parameters passed to sequencer are following:

Number of input iterations	
Input tile width in chunks	
Reserved	-
Reserved	-
Start CU adjustment for ROI	
Output Tile Width in chunks	
Number of output iterations	
Output skip	
Last input tile width in chunks	
Last output tile width in chunks	
Reserved	-
Parameter 1..N	Parameters for each I/O port • Width in elements • Height in elements • Span in bytes • Element span • Physical address • Tile offset

4 ICP

The ICP part of the ACF framework encapsulates main structure data accessible by APEX. The main data structure – **DataDescriptor** – is a standard data array with specific image fields like width, height, span, ROI etc. (will be described below). The main difference between for example the OpenCV data structure is that the DataDescriptor is allocated in a physically contiguous region, which can be easily passed by its starting address to processing in APEX-2.

The memory allocation itself is managed by the OAL, which allocates the memory from predefined memory region (must not be mapped for OS use, typically excluded from OS visibility). The OAL consist of its own heap manager which looks after all allocations made during the program run. Please refer to OAL documentation for more information about contiguous memory allocation.

The memory allocation for DataDescriptor is done typically in the constructor, which specifies the size and element size of 1D or 2D structure. The structure has however also an empty constructor which does not initialize the data and leaves this action for later use.

The DataDescriptor class contains following information and functions (for detailed description, please refer to Doxygen generated documentation):

Constructor	○ Constructor, if used without parameters, it initialises an empty data. Otherwise, the constructor allocates the contiguous memory region for specified size. ○ If allocType is specified, the allocation can be changed to normal virtual contiguous.
Initialization functions	○ The init functions called from constructor. ○ The functions allocate and initialize the data structure.
Destructor	Descructor does not free allocated space by default. It's implemented for use when structure is passed as dereference parameter. For free the data when descriptor is called, please use SetFreeOnExit function.
ROI	Information about Left, Right, Top and Bottom ROI boundary
Data pointer	Pointer to underlying data structure
Data physical address	Address to the physical region of the underlying data structure.
Pixel dimensions	The pixel can have the width and height values specified.
Width/Height	Width and height of a structure in elements (pixels)
Element functions	Functions which provide management of the pixel dimensions, the element size queries etc.
Comparison operator	
Successful allocation query	
Fill with random values	Function for testing purposes

5 ACF Host Part

The last described part in this document is the ACF framework itself. The ACF is the main access means for APEX-2 hardware and provides functionality for its programming. Each application should inherit an ACF_Process_APU class which manages all necessary functionality. The main ACF process pipeline is depicted on Figure 3.

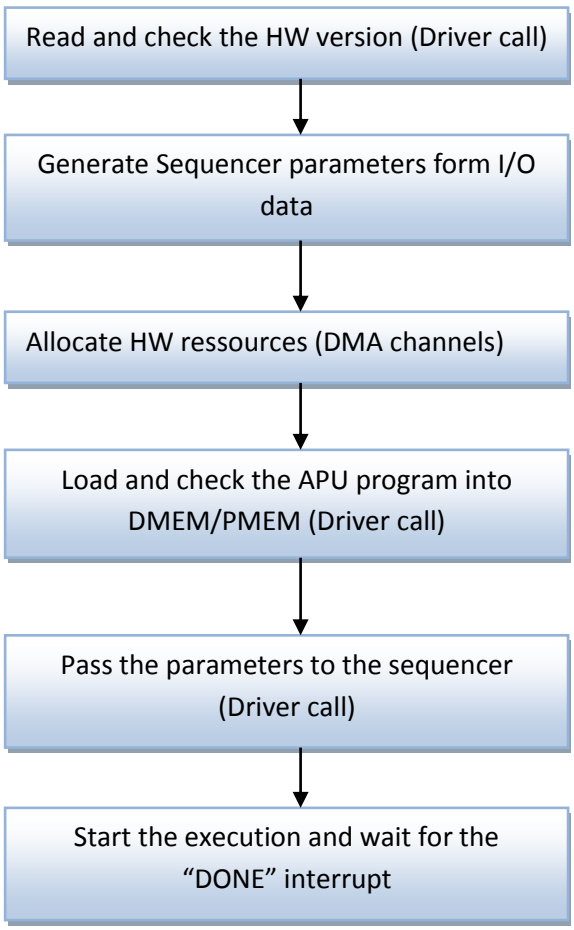

Figure 3 ACF process

Application structure

The application which is supposed to execute the APEX-2 program is supposed use the ACF for programming the HW. The standard application structure is described on Figure 4.

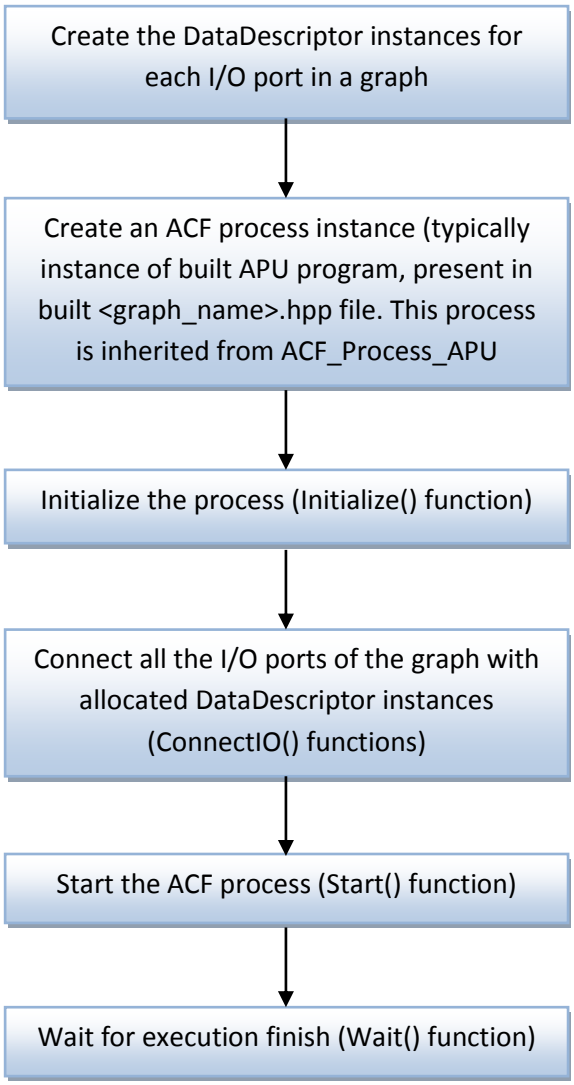

Figure 4 ACF application structure

Application itself must be supplied with following entities:

- Pre-compiled APU kernel/graph in the header file format (<name>.hpp and <name>_APU_LOAD.h files)
- Input data (connected to the process in DataDescriptor structures)

6 Application example

On following lines, an example application code is present to demonstrate the use of ACF framework. The application will find the descriptors in the input image and outputs the binary image with found feature points using FAST9 algorithm.

```
int main(int argc, char* argv[])
{
  /*
   * HW and resources initialization
   * The APEX will reset all components and the OAL will initialize the
   * memory space for contiguous memory allocation. The position of the heap
   * is specified in OAL.h file.
   */
  apex::reset::All();
  apex::resources::Init();
  OAL_MemoryAllocInit();

  /*
   * Create input and output image.
   * Suppose we have the dimensions specified outside this code, both images
   * will be 8-bit greyscale. Also both will be contiguously allocated
   * (default)
   */
  icp::DataDescriptor lInput0(lSrcWidth, lSrcHeight, icp::DATATYPE_08U);
  icp::DataDescriptor lOutput0(lSrcWidth, lSrcHeight, icp::DATATYPE_08U);

  /*
   * Check if the allocation passed through.
   */
  if (lInput0.IsOK() != 0 && lOutput0.IsOK() != 0)
  {
    /*
     * Load the data into lInput0 file using
     * unsigned char *data = (unsigned char *)lInput0.GetDataPtr(); pointer
     * ...
     * ...
     */

    /*
     * Create an instance of the ACF process. (The APU_FAST9 is in the
     * APU_FAST9.hpp header file created by the chess compiler and it's
     * an instance of the ACF_Process_APU class.
     */
    APU_FAST9 process;
```

```
    /*
     * Initialize the process.
     */
    lRetVal |= process.Initialize();

    /*
     * Connect the input and output data with graph ports. The APU_FAST9
     * process has two ports, INPUT_0 and OUTPUT_0.
     * This action associates the data with virtual ports.
     */
    lRetVal |= process.ConnectIO("INPUT_0", lInput0);
    lRetVal |= process.ConnectIO("OUTPUT_0", lOutput0);

    /*
     * Start the execution on APEX-2
     */
    lRetVal |= process.Start();

    /*
     * Wait for finish (for now, only polling for interrupt is available)
     */
    lRetVal |= process.Wait();
  }
  else
  {
    lRetVal = 1;
  }

  /*
   * At this point, if the lRetVal == 0, the execution finished without a
   * problem and the lOutput0 contains the output data.
   */

  /*
   * At the end, the DataDescriptor must be set to free on exit. Otherwise
   * the memory is not freed.
   */
  lInput0.SetFreeOnExit(true);
  lOutput0.SetFreeOnExit(true);
  return 0;
}
```

7 Two APEX Core application example

On following lines, an example application code is present to demonstrate the use of ACF framework. The application will execute the parallel addition of two images and outputs two identical images.

```
int main(int argc, char* argv[])
{
  apex::reset::All();
  apex::resources::Init();
  OAL_MemoryAllocInit();

  /*
   * Create input and output image.
   */
  icp::DataDescriptor lInput0(lSrcWidth, lSrcHeight, icp::DATATYPE_08U);
  icp::DataDescriptor lInput1(lSrcWidth, lSrcHeight, icp::DATATYPE_08U);
  icp::DataDescriptor lOutput0_ACF1(lSrcWidth, lSrcHeight,icp::DATATYPE_16U);
  icp::DataDescriptor lOutput0_ACF2(lSrcWidth, lSrcHeight,icp::DATATYPE_16U);
  icp::DataDescriptor lOutput0_REF(lSrcWidth, lSrcHeight, icp::DATATYPE_16U);

  /*
   * Check if the allocation passed through.
   */
  if (lInput0.IsOK() != 0 && lInput1.IsOK() != 0 && lOutput0_ACF1.IsOK() != 0
      && lOutput0_ACF2.IsOK() != 0 && lOutput0_REF.IsOK() != 0)
  {
     /*
      * Fill the inputs randomly
      */
      lInput0.FillRandom();
      lInput1.FillRandom();

     /*
      * Initialize the process on APEX 1
      */
      APU_PIXEL_A1 process_apex1(APEX_APEX1);
      lRetVal |= process_apex1.Initialize();
      lRetVal |= process_apex1.ConnectIO("INPUT_0", lInput0);
      lRetVal |= process_apex1.ConnectIO("INPUT_1", lInput1);
      lRetVal |= process_apex1.ConnectIO("OUTPUT_0", lOutput0_ACF1);

     /*
      * Initialize the same process on APEX 2
      */
      APU_PIXEL_A1 process_apex2(APEX_APEX2);
      lRetVal |= process_apex2.Initialize();
      lRetVal |= process_apex2.ConnectIO("INPUT_0", lInput0);
```



```
      lRetVal |= process_apex2.ConnectIO("INPUT_1", lInput1);
      lRetVal |= process_apex2.ConnectIO("OUTPUT_0", lOutput0_ACF2);

     /*
      * Compute reference software solution
      */
      lRetVal |= pixel_a1_ref(lInput0, lInput1, lOutput0_REF);

     /*
      * Start both APEX cores
      */
      lRetVal |= process_apex1.Start();
      lRetVal |= process_apex2.Start();

     /*
      * Wait for both done
      */
      lRetVal |= process_apex1.Wait();
      lRetVal |= process_apex2.Wait();

     /*
      * Compare the outputs
      */
      if (0 == lRetVal)
      {
        if (lOutput0_ACF1 == lOutput0_REF && lOutput0_ACF1 == lOutput0_ACF2)
          lRetVal = 0;
        else
          lRetVal += 1;
      }
  }
  else
  {
    lRetVal = 1;
  }

  return 0;
}
```