VICP Signal Processing Library for DM6446, DM6441, DM647, and DM648

User's Guide

Literature Number: SPRUGJ3E
March 2010

1 Introduction 6
1.1 Release Package and Directory Structure 7
1.2 Notational Conventions 8
1.3 Dependencies With Other Libraries/Modules 8
1.4 Natural C Equivalent Code 8
1.5 Usage Examples 8
1.6 EDMA Configurations for VICP and DSP 9
1.6.1 General Configuration 9
1.6.2 DM647/648 Deadlock Issue 10
2 Using VICP Signal Processing Library 11
2.1 Including the Library in Application Code 11
2.2 Library Calling Procedure 11
2.2.1 Initialization 11
2.2.2 Function Interface 12
2.2.3 Region or Rectangle of Interest (ROI) and Processing Block Size 14
2.2.4 Synchronous and Asynchronous Execution 14
2.2.5 Wait Callback 16
2.2.6 Deinitialization 16
2.2.7 Reentrancy 16
3 API Descriptions 17
3.1 Symbolic Constants and Enumerated Data Types 17
3.1.1 CPIS_Format 17
3.1.2 CPIS_ExecType 18
3.1.3 Error Symbol Definitions 18
3.1.4 Maximum Processing BLOCKSIZE Definition 18
3.1.5 Alpha Channel Definition 19
3.2 Interface Data Structures 19
3.2.1 CPIS_Init 19
3.2.2 CPIS_Size 20
3.2.3 CPIS_Buffer 20
3.2.4 CPIS_BaseParms 21
3.3 VICP Signal Processing Library APIs 22
3.3.1 CPIS_Init 22
3.3.2 CPIS_deInit 23
3.3.3 CPIS_setWaitCB 24
3.3.4 CPIS_isBusy 25
3.3.5 CPIS_start 26
3.3.6 CPIS_wait 27
3.3.7 CPIS_updateSrcDstPtr 28
3.3.8 CPIS_reset 29
3.3.9 CPIS_delete 30
3.3.10 CPIS_colorSpcConv 31
3.3.11 CPIS_alphaBlend 34
3.3.12 CPIS_rotation 36
3.3.13 CPIS_fillMem 37

3.3.14 CPIS_arrayOp 39
3.3.15 CPIS_arrayScalarOp 41
3.3.16 CPIS_arrayCondWrite 43
3.3.17 CPIS_YCbCrPack 45
3.3.18 CPIS_YCbCrUnpack 47
3.3.19 CPIS_matMul 49
3.3.20 CPIS_sum 51
3.3.21 CPIS_sumCFA 53
3.3.22 CPIS_table_lookup 55
3.3.23 CPIS_medianFilterRow 58
3.3.24 CPIS_medianFilterCol 60
3.3.25 CPIS_filter 62
3.3.26 CPIS_RGBPack 64
3.3.27 CPIS_RGBUnpack 65
3.3.28 CPIS_blkAverage 66
3.3.29 CPIS_recursiveFilter 68
3.3.30 CPIS_loadRecursiveFilterInitialValues 71
3.3.31 CPIS_setRecursiveFilterAlphaCoef 72
3.3.32 CPIS_median2D 73
3.3.33 CPIS_sobel 74
3.3.34 CPIS_integralImage 76
3.3.35 CPIS_pyramid 78
3.3.36 CPIS_affineTransform 80
3.3.37 CPIS_affineTransformGetSize 86
3.3.38 CPIS_cfa 88
3.3.39 CPIS_sad 90
3.3.40 CPIS_setSadTemplateOffset 100

List of Figures

1 DM6446, DM6441, DM647, and DM648 VICP Signal Processing Library 7
2 ROI Description 14
3 CPIS_MatMul Description 50
4 Input ROI 82
5 Output ROI 83
6 Output ROI Displayed With params→skipOutside= 1 84
7 Output ROI Displayed With params→skipOutside= 0 84
8 Case #1 of CPIS_sad() Usage 92
9 Case #2 of CPIS_sad() Usage 93
10 Case #3 of CPIS_sad() Usage 96

List of Tables

1 Data Formats Supported by the VICP Signal Processing Library 17
2 Execution Type Enumerations 18
3 Maximum Processing Size 18
4 CPIS_colorSpcConv API Parameters 31
5 CPIS_alphaBlend API Parameters 34
6 CPIS_rotation API Parameter 36
7 CPIS_fillMem API Parameters 37
8 CPIS_arrayOp API Parameters 39
9 CPIS_arrayScalarOp API Parameters 41
10 CPIS_ArrayCondWrite API Parameters 43
11 CPIS_YCbCrPack API Parameters 46
12 CPIS_YCbCrUnPack API Parameters 48
13 CPIS_matMul API Parameters 49
14 CPIS_sum API Parameters 51
15 CPIS_sumCFA API Parameters 53
16 CPIS_table_lookup API Parameters 55
17 CPIS_medianFilterRow API Parameters 58
18 CPIS_medianFilterCol API Parameters 60
19 CPIS_filter API Parameters 62
20 CPIS_blkAverage API Parameters 66
21 CPIS_recursiveFilter API Parameters 68
22 CPIS_median2D API Parameters 73
23 CPIS_Sobel API Parameters 74
24 CPIS_integralImage API Parameters 76
25 CPIS_pyramid API Parameters 78
26 CPIS_affineTransform API Parameters 81
27 CPIS_AffineTransformOutputROI Parameters 86
28 CPIS_cfa API Parameters 88
29 CPIS_sad API Parameters 90

VICP Signal Processing Library

1 Introduction

The VICP hardware accelerator is available on DM6446, DM6441, DM647, and DM648 devices. Due to its flexible architecture, the accelerator is effective in enhancing DSP performance by taking over execution of varied computationally intensive tasks. The accelerator has a flexible control and memory interface. These are some of the generic functions that can be optimally implemented by the VICP:

- Basic Math and Signal Processing functions:
  - Operation between two arrays: A.B, A + B, A – B, |A – B|, A & B, A | B, A ^ B, min(A,B), max(A,B)
  - FIR Filtering
  - IIR Filtering
  - FFT, DCT, Wavelets, etc
  - Matrix Multiplication
  - Table lookup
- Image Processing functions:
  - Color space conversion
  - Bayer→RGB conversion
  - RGB888↔RGB555, RGB888↔RGB565
  - 3x3 Median filtering
  - Alpha-blending, bit-blt
  - Affine transform (rotation, resizing)
- Vision functions:
  - Sobel edge detection
  - Gaussian Pyramid
  - Integral Image
  - Binary Morphology

The VICP performs block based processing. The VICP handles a large input buffer by splitting it into smaller sub blocks and processing each sub block one after the other. After the processing of the entire input buffer is over, the VICP notifies the DSP by triggering an interrupt. This allows the DSP to utilize the freed MIPS effectively. The VICP implements single thread processing and cannot be preempted to switch the already started task.

The VICP signal processing library is a collection of highly tuned software algorithms that execute on the VICP hardware. The library allows the application developer to effectively utilize the VICP performance without spending significant time in developing software for the accelerator.

The signal processing library provides various system features to simplify the application design, including:

- Capability to either execute the APIs in synchronous or asynchronous mode. In synchronous mode, any call to the library API does not return until the VICP completes processing. Whereas in the asynchronous mode, any call to the library API returns immediately. The DSP is notified of the completion of the processing using an interrupt.
- The VICP signal processing library internally interfaces with the system DMA manager to service the VICP DMA requirements. This reduces the system integration complexity.
- The library also handles the on-chip cache and external memory synchronization to ensure data correctness.

- The VICP signal processing library includes C equivalent implementation of all the APIs that are supported. The C equivalent implementation can be used by the application developers to better understand the signal processing functionality implemented by each API. For each API, a reference test bench is provided. The test bench allows the user to understand the correct usage of these APIs. The test bench is built on top of the DSP-BIOS real time operating system. Thus, any of the test benches can be used as the starting point for application development using VICP.

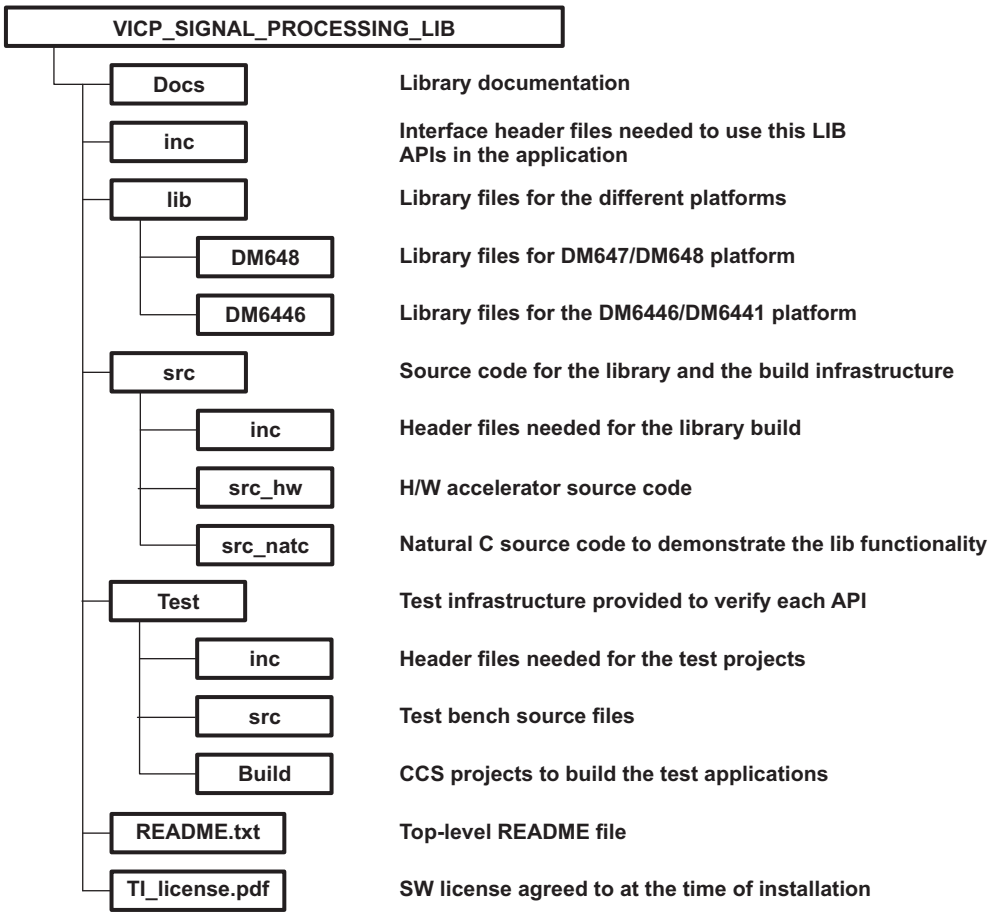
The VICP signal processing library is implemented using the VICP computation unit and VICP scheduling unit libraries. Prior knowledge about these libraries is not necessary for using the VICP signal processing library. Knowledge of these libraries is only necessary when the programmer wants to customize some existing VICP signal processing library's functions or create its own processing chain. Please refer to the *VICP Computation Unit Library and Scheduling Unit Library User's Guide* (SPRUGN1) provided inside the present release.

1.1 Release Package and Directory Structure

The VICP signal processing library can be installed at desired location using the provided SW installer. The installation does not include an uninstaller. To remove the library, simply removes the installed files from your computer.

The installation creates the directory structure shown in Figure 1:

Figure 1. DM6446, DM6441, DM647, and DM648 VICP Signal Processing Library



1.2 Notational Conventions

This document uses the following conventions:

- Program listings and program examples are shown in a `special typeface`.

  Here is a sample of C code:

```
#include <stdio.h>
main()
{    printf("hello, beautiful world\n");
}
```

- In syntax descriptions, parameters are in an *italic typeface*. Portions of a syntax that are in italics describe the type of information that should be entered.
- In code examples, parameters are enclosed in angle brackets (< >). Portions of a syntax that are in angle brackets describe the type of information that should be entered.

1.3 Dependencies With Other Libraries/Modules

This section lists the external dependencies for the VICP signal processing library. Also, we list the version of the modules that the release was verified with.

- EDMA3 LLD: The library requires that the EDMA3 LLD be installed. The release was tested with the release version 1_05_00_01. The library understands the location of the EDMA3 LLD release by looking at the global symbol TI_EDMA3LLD. Add the system environment variable TI_EDMA3LLD to point to [INSTALL_DIR]\edma3_lld_1_05_00\packages. Where INSTALL_DIR is the directory where the EDMA3 LLD software is installed.
- DSP-BIOS: The library requires that the DSP-BIOS is installed. The release was tested with the DSP-BIOS version 5_33. The library understands the location of the DSP-BIOS by looking at the global symbol BIOS_INSTALL_DIR. Ensure the system environment variable BIOS_INSTALL_DIR is defined and points to the location of the DSP-BIOS installation.

In addition to following these dependencies, to be able to use the batch build scripts batchBuild648.bat and batchBuild6446.bat, and batch execute scripts batchRun648.bat and batchRun6446.bat; you need to have active perl installed. You can download it at: http://www.activestate.com/activeperl/.

All libraries were built using code gen tools v6.1.8 and were tested in applications using DSP/BIOS v5.33.

1.4 Natural C Equivalent Code

To highlight the functionality implemented by the VICP signal processing library APIs, the release includes natural C implementation corresponding to each API. Just like the VICP signal processing library APIs, the natural C implementation of the APIs are compiled to generate a binary library that can be included in the test projects. The source code for the natural C equivalent code is available at [VICP_LIBRARY_INSTALLATION_DIR]\src\src_natc. The CCS project file to build the library is available at [VICP_LIBRARY_INSTALLATION_DIR] \src\src_natc\build. For some VICP signal processing library APIs that have support for a large number of usage options, the natural C equivalent implementation may not support all the options.

1.5 Usage Examples

For every API in the VICP signal processing library, there is a usage example provided. The intent of providing the usage example is to demonstrate the correct usage of the API. For each API, the VICP signal processing library includes a natural C implementation that highlights the API functionality. The usage examples invoke the API that executes on the VICP H/W accelerator and also the API implementation from the reference natural C library. By comparing the output from the two implementations, the usage example also ensures the correct operation of the VICP signal processing library APIs.

Each API's name starts with the prefix CPIS, which stands for coprocessor instruction set.

The CCS Project files associated with each usage example are located at [VICP_LIBRARY_INSTALLATION_DIR]\test\build. There is a separate test project provided for each API and also for each device.

To rebuild all the CCS projects in [VICP_LIBRARY_INSTALLATION_DIR]\test\build at one time, open a command window, go to the build directory and type: batchBuild648 –delete or batchBuild6446 –delete. You need to have Code Composer open with no project loaded before starting these scripts. Output of the build process is written to the file _batchBuild648.log or _batchBuild6446.log.

Likewise, the execution of each usage example can be batch processed by invoking the script batchBuild648 or batchBuild6446. Output of the execution process is written to the file _batchRun648.log or _batchRun6446.log.

The source files for the usage examples are located at [VICP_LIBRARY_INSTALLATION_DIR]\test\src. For each API, there are two source files provided. The *APIName.c* file implements the actual example. The example implementation is very generic. The example implementation receives the test parameters that decide how the actual test is run. The test parameter structure used by the example implementation is defined in the file [VICP_LIBRARY_INSTALLATION_DIR]\inc\testParams.h. The *APINameTestParams.c* file includes the various test parameters that are used for the example execution. Each example invokes the VICP library APIs multiple times based on the number of test parameters that are included in the test parameter file.

The pseudo code in Example 1 describes the implementation of the usage examples.

Example 1. Usage Example Implementation

```
Allocate continuous memory buffers in DDR using MEM_alloc function.

For each test input parameters do {
       Call nexTestParameters() to retrieve input parameters for the present test.
       Set function CPIS_<function>() 's input parameters such as pointers, region of interest
           size, etc

       Fill source buffers with random data and destination buffer with zeroes.
       Call CPIS_<function>() in asynchronous execution. This function is the function being
           tested, using hardware accelerators for computation. Since asynchronous mode is
           being tested, only hardware setup is done during the call.

       Call CPIS_start(...) to start the processing.
       Call CPIS_wait(...) to wait for the processing to complete.
       Call CPIS_reset(...) to reset processing state in preparation for next run.

       Call CPIS_start(...) to start the same processing again. We run it a second time for the
           sake of testing multiple execution of the same function.

       Call CPIS_wait(...) to wait for the processing to complete.

       Set reference function GPP_CPIS_<function>() 's input parameters
       Call GPP_CPIS_<function>(). This function is a reference function entirely implemented in
           C and not optimized.

       Compare the output of CPIS_<function> with GPP_IMPROC_<function>. If outputs match,
           display success message otherwise display failure message.

}
```

1.6 EDMA Configurations for VICP and DSP

1.6.1 General Configuration

The VICP signal processing library allocates its own EDMA channels through the EDMA3 low-level driver. These channels are programmed by the library to perform memory transfers between the VICP internal memory and other memories. Technically, it is the VICP scheduling unit library that executes the allocation and the programming of the EDMA channels. Hence, the application does not have direct control over these VICP-owned channels.

The application should allocate and program its own EDMA channels by making explicit calls to the EDMA3 low-level driver. These channels will automatically belong to the DSP.

To ensure conflict-free usage of EDMA channels from both VICP and DSP, the VICP configuration of the EDMA resources must not interfere with the DSP's. An example of the VICP's EDMA configuration is provided in the file vicp_edma3_dm64xx_cfg.c and an example of the DSP's EDMA configuration is provided in the file bios_edma3_drv_sample_dm64xx_cfg.c, both located at [VICP_LIBRARY_INSTALLATION_DIR]\test\src. These examples provide a conflict-free partition of the EDMA channels between DSP and VICP.

Be aware that bios_edma3_drv_sample_dm64xx_cfg.c 's content is different than the default one, which is provided with the EDMA3 LLD and which is used to build the edma3_drv_bios.lib. Unless you rebuild the edma3_drv_bios.lib with the updated _bios_edma3_drv_sample_dm64xx_cfg.c file, your application will not take the new configuration. To avoid rebuilding the library, you can add the configuration file to your project. The linker will take care of linking from the project's file instead of from the library.

The integration of vicp_edma3_dm64xx_cfg.c into your application is easier as it is implemented inside the dmcsl64xx_bios.lib library so you do not have to include it in your project.

In any case, if you change any of the configuration file, you will need to add it to your project for the configuration to override the default one provided by the library.

The example configurations allocate more resources than needed for the VICP. In case DSP needs more EDMA channels or param entries, update the configuration files by keeping in mind that at least the following number of resources must be reserved for the VICP:

Resources Reserved	DM64xx	DM64x
Number of EDMA channels	9	11
Number of EDMA param entries	29	31

1.6.2 DM647/648 Deadlock Issue

On DM64x chips, to avoid the deadlock issue related to advisory 1.1.5 of DM648 errata SPRZ263e, the application must force each EDMA TC (transfer controller) to perform writes to either DSP memory space or DDR2 (or EMIFA) memory space, but not to both. Consequently, the algorithm integrator must ensure that the variables VICP_EDMA3_FROM_DDR_queue and VICP_EDMA3_TO_DDR_queue in vicp_edma3_dm648_cfg.c are set in accordance with the DSP TC usage. For example, if DSP uses TC#2 for DSP→DDR transfers then VICP_EDMA3_TO_DDR_queue must be assigned to 2; and if DSP uses TC#3 for DDR→DSP transfers than VICP_EDMA3_FROM_DDR_queue must be assigned to 3. Very often, the algorithm integrator does not have any prior knowledge of which TC is used by which channel, especially when these channels are under the control of some third-party driver.

The best way to discover the TC assignment is to build and run the application with an arbitrary configuration and to inspect the DMA queue number registers starting at 0x02A0 0240 (refer to p.71 of the DM648 datasheet). Look at the bit-fields corresponding to the channels allocated for the DSP only (refer to bios_edma3_drv_sample_dm648_cfg.c to find which ones belong to DSP) and write down the associated queue number. To find out which direction path the channel transfers data, inspect the source and destination addresses in the associated param entry starting at 0x02A0 4000. From this discovery process, you will know which queue is associated with FROM DDR transfers and which queue is associated with TO DDR transfers. Since queue #n is tied to TC #n, deduce the value of VICP_EDMA3_FROM_DDR_queue and VICP_EDMA3_TO_DDR_queue.

The vicp_edma3_dm648_cfg.c file currently provides VICP_EDMA3_TO_DDR_queue= 2 and VICP_EDMA3_FROM_DDR_queue= 3, which is the TC assignment used by software provided by TI such as the video driver. So in theory, you should not need to change these values.

2 Using VICP Signal Processing Library

This section describes how the VICP signal processing library can be used in the application.

2.1 Including the Library in Application Code

To be able to use the provided APIs in the DSP application, it is first required that the libraries for the appropriate device are included in the application project. The libraries are provided in the [*VICP_LIBRARY_INSTALLATION_DIR*]\lib\[*DEVICE_NAME*] folder. Include all the libraries that are provided in the folder. Only exclude the lib file that has the GPP_ prefix. That library is generated from the natural C code provided. It is not needed in the final application.

To call the APIs from the DSP application, include the header file provided in the folder [*VICP_LIBRARY_INSTALLATION_DIR*]\inc. There is no need to include the header file with GPP_ prefix. That header file is only needed when invoking the reference APIs from the natural C library.

2.2 Library Calling Procedure

This section describes how to invoke the VICP signal processing lib APIs in the application.

2.2.1 Initialization

Prior to calling any of the VICP Signal processing library APIs, the function CPIS_init() must be called. This function performs the needed initialization before the other APIs can be invoked. This function accepts as input argument a structure of type CPIS_Init. The API definition is:

```
typedef struct {
        Uint16 maxNumProcFunc;
        void * mem;
        Uint32 memSize;
        Cache_wbInv cacheWbInv;
        Uint16 staticDmaAlloc;
} CPIS_Init;
```

The library initialization function definition is:

```
Int32 CPIS_init(CPIS_Init *init);
```

maxNumProcFunc	This variable specifies the maximum number of functions that can be pre-initialized by the library. This parameter is meaningful in the context of asynchronous execution (see Section 2.2.4). In the context of synchronous execution, setting maxNumProcFunc to 1 is sufficient. Current release only supports maxNumProcFunc=1.
mem	This is pointer to a buffer allocated by the application. The buffer size must be equal to memSize.
memSize	Obtained using memSize = CPIS_getMemSize(cpisInit.maxNumProcFunc)
cacheWbInv	Function pointer used by the VICP library to ensure cache coherency.
staticDmaAlloc	Setting to 1 allows the VICP library to statically allocate DMA channels. This reduces set up time. Four channels are allocated for input and four channels are allocated for output.

Example 2 illustrates how to initialize VICP library.

Example 2. Initializing VICP Library

```
CPIS_Init vicpInit;

    vicpInit.cacheWbInv = (Cache_wbInv) BCACHE_wb;
    vicpInit.staticDmaAlloc= 1;
    vicpInit.maxNumProcFunc= 1;
    vicpInit.memSize= CPIS_getMemSize(vicpInit.maxNumProcFunc);
    vicpInit.mem= MEM_alloc(DDR2, vicpInit.memSize, 4);

    if (CPIS_init(&vicpInit)== -1) {
        printf("\nCPIS_init error\n");
        exit(-1);
    };
```

2.2.2 Function Interface

All the APIs included with the VICP signal processing library share a similar interface. Two arguments that are passed to the APIs are parameter structures. The first argument is the base parameter structure that includes parameters common to all the APIs. The second argument is the parameter structure specific to the API. This section describes the base parameters. The API-specific parameter structure is described in Example 3.

Example 3. API-Specific Parameter Structure

```
Int32 CPIS_<functionName>(
       CPIS_Handle *handle,
       CPIS_BaseParms *baseParams,
       CPIS_<functionName>Parms *params,
       CPIS_ExecType execType
);
```

The structure definition of CPIS_BaseParms is shown in Example 4:

Example 4. CPIS_BaseParms Structure

```
typedef struct {
       CPIS_Format srcFormat[4];
       CPIS_Buffer srcBuf[4];
       CPIS_Format dstFormat[4];
       CPIS_Buffer dstBuf[4];
       CPIS_Size roiSize;
       CPIS_Size procBlockSize;
       Uint16 numInput;
       Uint16 numOutput;
} CPIS_BaseParms;
```

The pixel format of the source and destination data is specified in base→srcFormat and base→dstFormat. It can be one of the values shown in Example 5.

Example 5. Pixel Format

```
typedef enum {
       CPIS_YUV_420P=0, /* Planar symbols must be listed first */
       CPIS_YUV_422P,
       CPIS_YUV_444P,
       CPIS_YUV_411P,
       CPIS_YUV_422VP,  /* Vertical subsampling */
       CPIS_RGB_P,
       CPIS_BAYER_P,
       CPIS_YUV_422IBE,
       CPIS_YUV_422ILE,
       CPIS_RGB_555,
       CPIS_RGB_565,
       CPIS_BAYER,
       CPIS_YUV_444IBE,
       CPIS_YUV_444ILE,
       CPIS_RGB_888,
       CPIS_YUV_GRAY,
       CPIS_8BIT,
       CPIS_16BIT,
       CPIS_32BIT,
       CPIS_64BIT,
       CPIS_U8BIT,
       CPIS_U16BIT,
       CPIS_U32BIT,
       CPIS_U64BIT
} CPIS_Format;
```

These formats include YUV and RGB pixels formats as well as Bayer and regular 8-bits, 16-bits, 32-bits, 64-bits data types. Not all formats are supported by every API. The API's documentation specifies which format is supported. Some APIs handle certain formats better than others, resulting in faster processing. Such formats are called native format for that API and are listed in the API's documentation.

Pixel formats that contain the character P at the end of the symbol name refer to planar format. Planar formats break the pixel data into several planes, each plane containing a particular component. Each plane does not need to be adjacent to each other in memory since it is possible to specify up to four different source buffers and four different destination buffers as shown in CPIS_BaseParms. Source and Destination buffers are represented by the CPIS_BaseParms members srcBuf and dstBuf. These are of type CPIS_Buffers. The type is defined as:

Example 6. CPIS_Buffers Type Definition

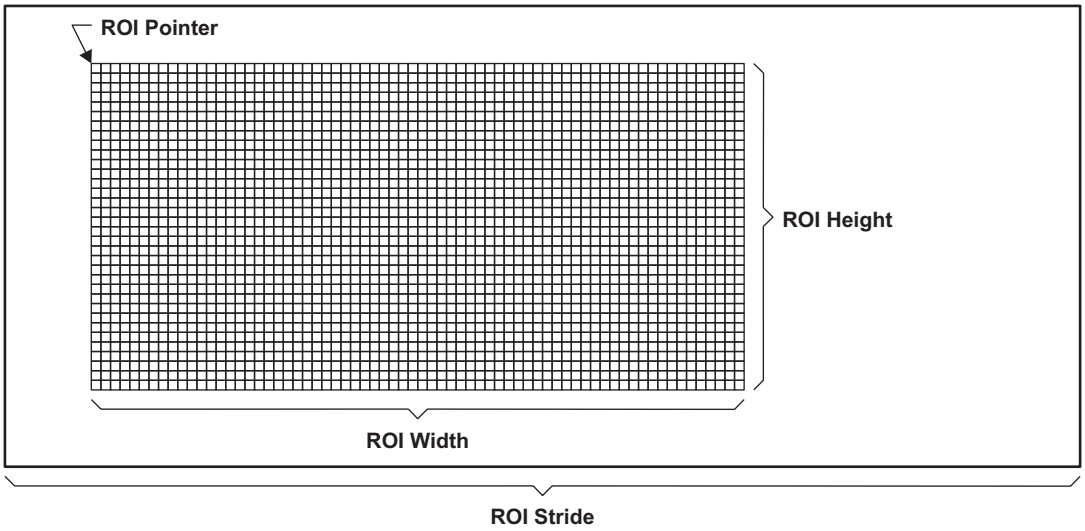
```
typedef struct {
       Uint8 *ptr;
       Uint32 stride;
} CPIS_Buffer;
```

Stride is the distance in pixels between the start of consecutive lines for the data buffer. Separate stride can be specified for every buffer, thus, every source and destination buffer can have unique stride. The pixel definition depends on the format. For ex, for YUV_422ILE, each pixel is 16-bit (Packed 8bit Chrominance and 8bit Luminance) where as for YUV_422P, each pixel is 8-bit. There can be up to four source or destination buffers in order to support RGB planar with alpha channel or RGB Bayer planar.

2.2.3 Region or Rectangle of Interest (ROI) and Processing Block Size

All APIs can operate not only on entire images but also on a part of the image. The Region of Interest or Rectangle of Interest (ROI) are rectangular areas which may be either some part of the image or the whole image. ROI of an image is defined by the size, location and the stride between each row.

Figure 2. ROI Description



Both the source and destination images can have a rectangle of interest. In such cases, the sizes of ROIs are assumed to be the same while strides may differ. The processing is then performed on data of the source ROI, and the results are written to the destination ROI. The common ROI size is provided by the member roiSize in CPIS_BaseParms.

As mentioned earlier, the VICP Signal processing library does not process the data in raster scan. It is instead done in 2-D blocks. Each API divides the ROI into a grid of processing blocks, as shown in the figure above. The size of each block must be provided to the API through the procBlockSize member of the base parameter structure. The roiSize.width and roiSize.height of the ROI must be exact multiple of procBlockSize.width and procBlockSize.height. The height and width is specified in number of pixels.

Also, for each API, a maximum size for the processing block is specified. The processing block size specified should never exceed that size MAX_*function_name*_BLOCKSIZE procBlockSize.width x procBlockSize.height <= MAX_*function_name*_BLOCKSIZE. The MAX_*function_name*_BLOCKSIZE is provided in the interface header file for the VICP signal processing library. It is recommended that the processing buffer size is kept as close as possible to the maximum processing buffer size allowed. This reduces the control overhead associated in doing the block processing.

Some functions have restrictions on procBlockSize.width and procBlockSize.height. Functions may require the width and the height for the blocks to be a multiple of 2, 4 or 8. For functions that do not have such a restriction, it is always advisable to try and keep the block width a multiple of 8. If a multiple of 8 is not possible, a multiple of 4 or 2 is preferable. For most functions, VICP performance improves if the block width is a multiple of 8.

2.2.4 Synchronous and Asynchronous Execution

The argument execType in the base parameter structure controls whether the execution of the API is synchronous or asynchronous. If execType= CPIS_SYNC, the API call is blocking. The control returns back to the caller after the entire processing is completed. Once the control returns to the application, the data would have been processed. If execType= CPIS_ASYNC, the API call only sets up the VICP for the particular operation and returns a handle of type CPIS_Handle. Later, the execution has to be initiated explicitly by calling CPIS_start() with the handle as argument. Once the processing has been started by calling the CPIS_start(), the application has few options to synchronize with VICP to understand the end of processing.

- The application can understand the completion of the processing by calling the CPIS_wait().This is a blocking call and returns only when the processing is complete. This method does not require the application to configure VICP interrupt. This method has disadvantage that DSP may spend time blocked in CPIS_wait() routine as the DSP does not know the exact time the processing is over. This method does not require the application to enable the VICP DSP interrupt.
- The application can call the API CPIS_isBusy() to understand if the VICP has completed the processing started by the last CPIS_start() call. The application can periodically poll the VICP by using the CPIS_isBusy() API call. This method also requires the DSP to spend processing resource to check for completion of the VICP processing. This method also does not require the application to enable the VICP DSP interrupt.
- The application can enable the VICP DSP interrupt. In the interrupt service routine, the application can post semaphore to indicate completion of processing. A higher priority thread can be made to pend for the semaphore. The posting of the semaphore can cause the higher priority thread to wake up and take appropriate action. This method achieves maximum efficiency as the DSP has to spend minimum resource to check for the completion of VICP processing.

Every call to CPIS_start() must have a corresponding CPIS_wait(). Also, if the same API needs to be called again, the application must call CPIS_reset() to reset some internal state related to the function before next call to CPIS_start().

The VICP signal processing library is not able to queue several CPIS_start() requests. If many calls are called in asynchronous context then the VICP signal processing library must keep context information for each call internally. The number of asynchronous functions that can be set up during run time must be set at initialization time by calling CPIS_init() by setting the maxNumProcFunc member of the initialization structure. During run time the application must make sure the number of asynchronous functions never exceeds maxNumProcFunc. It can call CPIS_delete() to ensure that this restriction is maintained.

NOTE: In the current release, the asynchronous execution is supported only for one function that has been set up since maxNumProcFunc=1 is the only supported option.

Example 7. Use of maxNumProcFunc

For instance with maxNumProcFunc = 1 , the following sequence would not work:

```
CPIS_alphaBlend(&handle1,..., CPIS_ASYNC);
CPIS_colorSpcConv(&handle2,..., CPIS_ASYNC);
CPIS_start(handle1);
CPIS_wait(handle1);
CPIS_start(handle2);
CPIS_wait(handle2);
CPIS_delete(handle1);
CPIS_delete(handle2);
```

Actually the call to CPIS_colorSpcConv would return -1 and set CPIS_errno to CPIS_MAXNUMFUNCREACHED.

The following sequence would work:

```
CPIS_alphaBlend(&handle1,..., CPIS_SYNC);
CPIS_colorSpcConv(&handle2,..., CPIS_ASYNC);
CPIS_start(handle2);
CPIS_wait(handle2);
CPIS_delete(handle2);
```

Since the first CPIS_alphaBlend() call is synchronous and the execution is completed at its return, all the context information related to that call is released. Next call to CPIS_colorSpcConv is asynchronous and only performs the hardware setup. Actual processing is initiated by calling CPIS_start(). The context information related to that function must be released by calling CPIS_delete().

Of course with maxNumProcFunc=1, you cannot really take full advantage of the asynchronous capability of the library if the application needs more than one function applied to the data one after the other.

2.2.5 Wait Callback

Once the execution of an API is started either by the API itself (in synchronous mode) or by CPIS_start() (in asynchronous mode), the application must wait for its completion. In synchronous mode, this waiting operation is automatically initiated inside the API and in asynchronous mode, CPIS_wait() must be called.

In both cases, the waiting operation is implemented inside the library. Inside the library the waiting operation is a simple busy while () loop that checks the status of hardware register. This method is pretty inefficient. This inefficiency can be alleviated by using the interrupt capability of VICP. The VICP generates interrupt to the DSP at the end of processing. The DSP can enable the interrupt to receive the synchronization event. To provide maximum flexibility to the application developer, the VICP signal processing library can accept a wait callback function from the application. This wait callback function is called by the VICP signal processing library when the wait operation is invoked and overrides the library's own busy loop.

This wait callback function must be implemented by the application and typically should pend for a semaphore or a flag that is set upon receiving the completion interrupt. Likewise the interrupt setup and service routine must be implemented by the application or provided by the OS kernel. The usage examples demonstrate a simple synchronization method based on the interrupt scheme.

The way to pass a callback function to the VICP library is through the function CPIS_setWaitCB(): Int32 CPIS_setWaitCB(Int32 (*waitCB)(void*arg), void*waitCBarg); The argument of this function is a pointer to the wait callback function implemented by the application. This wait call back function can accept an application defined argument 'arg'. The function prototype has it typecasted as (void*) for flexibility. Since the caller of the wait callback function will be the VICP signal processing library, it needs that argument beforehand. This is achieved by providing it through the parameter waitCBarg.

2.2.6 Deinitialization

Calling CPIS_deInit() frees up internal resources used by the VICP signal processing library.

2.2.7 Reentrancy

The VICP signal processing library is not reentrant. The application must ensure that calls to VICP signal processing library APIs don't overlap. In a multi-task environment, it means that calls to APIs functions must be made in a serialized way. Not following this requirement may result in incorrect operation. However, this limitation does not prevent the DSP from running concurrently with VICP. DSP is still able to run concurrently with the VICP signal processing library APIs.

3 API Descriptions

This section describes the various APIs supported by the VICP signal processing library. It also describes the various data structures and enumeration needed to interface with the library.

3.1 Symbolic Constants and Enumerated Data Types

This section summarizes all the symbolic constants specified as either #define macros and/or enumerated C data types. Described alongside the macro or enumeration is the semantics or interpretation of the same in terms of what value it stands for and what it means.

3.1.1 CPIS_Format

This enumeration type specifies pixel and data formats supported by the VICP signal processing library. In addition it is possible to add alpha channels by ORing the type with symbol CPIS_ALPHA.

Table 1. Data Formats Supported by the VICP Signal Processing Library

Symbolic Constant Name	Description or Evaluation
CPIS_YUV_420P	YUV 4:2:0 planar. Each Y, U, V plane contains an 8-bit value.
CPIS_YUV_422P	YUV 4:2:2 planar. Each Y, U, V plane contains an 8-bit value.
CPIS_YUV_444P	YUV 4:4:4 planar. Each Y, U, V plane contains an 8-bit value.
CPIS_YUV_411P	YUV 4:1:1 planar. Each Y, U, V plane contains an 8-bit value.
CPIS_YUV_422VP	YUV 4:2:2 planar subsampled in vertical direction. Each Y, U, V plane contains an 8-bit value.
CPIS_RGB_P	RGB color format planar. Each R, G, B plane contains an 8-bit value.
CPIS_BAYER_P	Bayer in planar format. Elements in a plane are 16-bit values.
CPIS_YUV_422IBE	YUV 4:2:2 interleaved (big endian). Each pixel is a 16-bit value.
CPIS_YUV_422ILE	YUV 4:2:2 interleaved (little endian). Each pixel is a 16-bit value.
CPIS_RGB_555	Packed RGB555. Each pixel is a 16-bit value.
CPIS_RGB_565	Packed RGB565. Each pixel is a 16-bit value.
CPIS_BAYER	Bayer pattern. Each pixel is a 16-bit value.
CPIS_YUV_444IBE	YUV 4:4:4 interleaved (big endian). Each pixel is a 24-bit value.
CPIS_YUV_444ILE	YUV 4:4:4 interleaved (little endian). Each pixel is a 24-bit value.
CPIS_RGB_888	Packed RGB24. Each pixel is a 24-bit value in little-endian format, with byte 0 corresponding to R.
CPIS_YUV_GRAY	Gray format. One plane of 8-bit values
CPIS_8BIT	8-bit data type
CPIS_16BIT	16-bit data type
CPIS_32BIT	32-bit data type
CPIS_64BIT	64-bit data type
CPIS_U8BIT	8-bit unsigned data type
CPIS_U16BIT	16-bit unsigned data type
CPIS_U32BIT	32-bit unsigned data type
CPIS_U64BIT	64-bit unsigned data type

3.1.2 CPIS_ExecType

This enumeration type is used to set the execType parameter for any VICP signal processing library's API. This parameter determines whether the API is executed synchronously or asynchronously. In synchronous execution, both the hardware setup and the processing occur in a single function call. In asynchronous call, only the hardware setup is done when calling the function and the processing must be separately triggered at a later time by calling CPIS_start().

Table 2. Execution Type Enumerations

Symbolic Constant Name	Description or Evaluation
CPIS_SYNC	Synchronous execution
CPIS_ASYNC	Asynchronous execution

3.1.3 Error Symbol Definitions

Below is a list of error symbols that could be held by the global variable CPIS_errno after a function returns -1.

```
/* Error symbols used by the library */
#define CPIS_INIT_ERROR            1 /* Error during initialization */
#define CPIS_NOTINIT_ERROR         2 /* Error during de-init */
#define CPIS_UNPACK_ERROR          3 /* Error during unpack operation */
#define CPIS_NOSUPPORTFORMAT_ERROR 4 /* Data format not supported by API */
#define CPIS_NOSUPPORTDIM_ERROR    5 /* Not supported interlaced mode */
#define CPIS_PACK_ERROR            6 /* Error during pack operation */
#define CPIS_MAXNUMFUNCREACHED     7 /* Maximum no functions queued */
#define CPIS_OUTOFMEM              8 /* VICP internal memory exhausted */
#define CPIS_NOSUPPORTANGLE_ERROR  9 /* Not supported angle for rotation */
#define CPIS_NOSUPPORTOP_ERROR     10 /* Not supported operation */
```

3.1.4 Maximum Processing BLOCKSIZE Definition

For each function, the input parameters must satisfy the following relation, where scale is 1,2 or 4 bytes depending on the type of the input format:

procBlockSize.width × procBlockSize.height × scale < MAX_*function_name*_BLOCKSIZE

The symbols MAX_*function_name*_BLOCKSIZE are listed in the interface header file.

Table 3. Maximum Processing Size

API	Maximum Processing Size
MAX_ALPHABLEND_GLOBAL_ALPHA_BLOCKSIZE	2048 bytes
MAX_ALPHABLEND_BLOCKSIZE	1260 bytes
MAX_COLORSPCCONV_BLOCKSIZE	744 bytes
MAX_ROTATION_BLOCKSIZE	512 bytes
MAX_FILLMEM_BLOCKSIZE	8188 bytes
MAX_ARRAYOP_BLOCKSIZE	4096 bytes
MAX_ARRAYSCALAROP_BLOCKSIZE	8192 bytes
MAX_ARRAYCONDWRITE_BLOCKSIZE	2730 bytes
MAX_YCBCRPACK_BLOCKSIZE	1364 bytes
MAX_YCBCRUNPACK_BLOCKSIZE	1364 bytes
MAX_MATMUL_BLOCKSIZE	4096 bytes
MAX_SUM_BLOCKSIZE	8188 bytes
MAX_SUMCFA_BLOCKSIZE	8176 bytes
MAX_LUT_BLOCKSIZE	8192 bytes
MAX_BLKAVERAGE_BL0CKSIZE	8188 bytes
MAX_MEDIANFILTER_ROW_BLOCKSIZE	8192 bytes
MAX_MEDIANFILTER_COL_BLOCKSIZE	8192 bytes
MAX_FILTER_BLOCKSIZE	8192 bytes

Table 3. Maximum Processing Size (continued)

API	Maximum Processing Size
MAX_RGBPACK_BLOCKSIZE	1638 bytes
MAX_RGBUNPACK_BLOCKSIZE	1638 bytes
MAX_MEDIAN2D_BLOCKSIZE	8192 bytes
MAX_SOBEL_BLOCKSIZE	4096 bytes
MAX_PYRAMID_BLOCKSIZE	8192 bytes
MAX_AFFINE_BLOCKSIZE	5000 bytes
MAX_CFA_BLOCKSIZE	2730 bytes
MAX_SAD_BLOCKSIZE	8192 bytes
MAX_SAD_TEMPLATESIZE	32768 bytes

3.1.5 Alpha Channel Definition

The symbol CPIS_ALPHA can be logically ORed with most of the values in CPIS_Format type in order to specify that an alpha channel is available.

3.2 Interface Data Structures

This section describes the various data structures that are used to interface with the VICP signal processing library. This section only describes the data structures that are common to all the APIs. The data structures that are specific to a particular API are listed in the section that describes the particular API.

3.2.1 CPIS_Init

This structure is used as an input parameter to the initialization function CPIS_init().

Example 8. Initialization Structure for the Library

```
typedef void (*Cache_wbInv) ();

typedef struct {
        Uint16 maxNumProcFunc;
        void * mem;
        Uint32 memSize;
        Cache_wbInv cacheWbInv;
        Uint16 staticDmaAlloc;
} CPIS_Init;
```

Parameter	Description
maxNumProcFunc	Maximum number of asynchronous functions that can be handed by the VICP signal processing library during run time.
*mem	Pointer to memory buffer pre-allocated by the application.
memSize	Size of buffer in number of bytes. The size must be equal to CPIS_getMemSize(cpisInit.maxNumProcFunc)
cacheWbInv	Cache write back invalidate function pointer. This function is used by the VICP signal processing library to maintain synch between cache and external memory
staticDmaAlloc	Setting to 1 allows the VICP signal processing library to statically allocate DMA channels. This reduces set up time. Four channels are allocated for input and four channels are allocated for output.

3.2.2 CPIS_Size

This structure is used to store width and height dimensions. Structures are used to convey information regarding the size of the various blocks.

```
typedef struct {
    Uint32 width;
    Uint32 height;
} CPIS_Size;
```

Parameter	Description
width	Width in number of pixels
height	Height in number of lines

3.2.3 CPIS_Buffer

This structure is used to store pointer and stride information for source and destination buffers.

```
typedef struct {

    Uint8 *ptr;
    Uint32 stride;
} CPIS_Buffer;
```

Parameter	Description
*ptr	Pointer to the buffer
stride	Stride width in number of pixels

3.2.4 CPIS_BaseParms

The structure in Example 9 contains the base parameters passed to any function of the VICP signal processing library.

Example 9. Base Parameters Common to All APIs

```
typedef struct {

    CPIS_Format srcFormat[4];
    CPIS_Buffer srcBuf[4];
    CPIS_Format dstFormat[4];
    CPIS_Buffer dstBuf[4];
    CPIS_Size roiSize;
    CPIS_Size procBlockSize;
    Uint16 numInput;
    Uint16 numOutput

} CPIS_BaseParms;
```

Parameter	Description
srcFormat	Data format of the source. Value is chosen among the enumeration type CPIS_Format and can be ORed with CPIS_ALPHA if alpha channel is present. Each element in the array corresponds to the respective buffer pointer.
srcBuf	Array of source buffer structures. Four in total to support planar format and Alpha channel. If only one plane is needed then fill srcBuf[0] only. If alpha Channel is present then srcBuf[1] points to it in case of interleaved or one Component format; otherwise srcBuf[3] is used for planar format + alpha.
dstFormat	Data format of the destination. Value is chosen among the enumeration type CPIS_Format and can be ORed with CPIS_ALPHA if alpha channel is present. Each element in the array corresponds to the respective buffer pointer.
dstBuf	Array of destination buffer structures. Four in total to support planar format and Alpha channel. If only one plane needed then fill srcBuf[0] only. If alpha Channel present then srcBuf[1] points to it in case of interleaved or one Component format; otherwise srcBuf[3] is used for planar format + alpha.
roiSize	Size of the rectangle of interest, in pixels, in which processing will take place and output will be written to. roiSize.width and roiSize.height must be a multiple of procBlockSize.width and procBlockSize.height
procBlockSize	Size in pixels of the processing subblocks composing the ROI.
numInput	Number of input buffers
numOutput	Number or output buffers

3.3 VICP Signal Processing Library APIs

This section describes the various APIs that are supported by the VICP signal processing library.

3.3.1 CPIS_Init

CPIS_Init ***Initializes Library***

Syntax

Int32 **CPIS_init**(CPIS_Init *init);

Arguments

CPIS_Init *init;

```
typedef struct {
    Uint16 maxNumProcFunc;
    void * mem;
    Uint32 memSize;
    Cache_wbInv cacheWbInv;
    Uint16 staticDmaAlloc;
} CPIS_Init;
```

maxNumProcFunc	This variable specifies the maximum number of functions that can be pre-initialized by the library. This parameter is meaningful in the context of asynchronous execution (see Section 2.2.4). In the context of synchronous execution, setting maxNumProcFunc to 1 is sufficient. *Current release only supports maxNumProcFunc=1.*
mem	This is pointer to a buffer allocated by the application. The buffer size must be equal to memSize.
memSize	memSize is obtained using: `memSize = CPIS_getMemSize(cpisInit.maxNumProcFunc)`
cacheWbInv	Cache write back invalidate function pointer. This function is used by the VICP signal processing library to maintain synch between cache and external memory.
staticDmaAlloc	Setting to 1 allows the VICP signal processing library to statically allocate DMA channels. This reduces setup time. Four channels are allocated for input and four channels are allocated for output.

Return Value

0 Success

1 Error and CPIS_errorno=CPIS_INIT_ERROR

Description

This function initializes the VICP signal processing library. A memory buffer must be pre-allocated by the application and its pointer must be passed to the CPIS_init routine by properly initializing the member *mem* of the argument structure. The size of the memory allocated must be equal to CPIS_getMemSize(cpisInit.maxNumProcFunc).

If only synchronous execution is going to be used, set cpisInit.maxNumProcFunc to 1.

Performance

NA

3.3.2 CPIS_deInit

CPIS_deInit	*Deinitializes Library*
Syntax	Int32 **CPIS_deInit**();
Arguments	None
Return Value	Always returns 0.
Description	This function de-initializes the VICP signal processing library.
Performance	NA

3.3.3 CPIS_setWaitCB

CPIS_setWaitCB ***Specifies Application Wait callback Function***

Syntax

Int32 **CPIS_setWaitCB**(Int32 (**waitCB*)(void***arg*), void* *waitCBarg*)

Arguments

Int32 (**waitCB*)(void***arg*) /* Pointer to call back function */

void* *waitCBarg* /* Pointer to callback function's argument */

Return Value

Always returns 0.

Description

Set the wait callback function used by the VICP signal processing library to wait for the completion of a function's execution. The wait callback function is implemented by the application and usually pend on a semaphore set by the interrupt service routine that services the VICP processing completion interrupt. The wait callback function can accept an argument arg, whose pointer is passed as parameter waitCBarg. It is not mandatory to pass an argument to a wait callback function; it is usually up to the application. The argument type is a void* that can be typecasted inside the wait callback function to match the application developer's choice. It can be a pointer to a structure or a pointer to a single variable.

Wait callback function can be used for both synchronous and asynchronous execution.

Performance

NA

3.3.4 CPIS_isBusy

CPIS_isBusy ***Returns the Completion State of Started Processing***

Syntax

Int32 CPIS_isBusy(CPIS_Handle *handle*);

Arguments

CPIS_Handle *handle*;

Return Value

0 Processing done.

-1 Processing on-going.

Description

This function returns the status of the last processing started by the last CPIS_start() API call.

Busy wait can be implemented by:

```
CPIS_start(handler);
while(CPIS_isBusy(handler));
```

Performance

NA

3.3.5 CPIS_start

CPIS_start ***Starts Execution of a Function Previously Set Up for Asynchronous Execution***

Syntax

Int32 **CPIS_start**(CPIS_Handle *handle*);

Arguments

CPIS_Handle *handle* /* handle of the function to be executed */

Return Value

Always returns 0.

Description

Starts the execution of a function previously setup for asynchronous execution. The application must ensure that any previous execution started on the VICP has completed before calling CPIS_start(). This means to implement a series of asynchronous executions, an CPIS_start() must always have a corresponding CPIS_wait(). Also if the same function has already been executed using CPIS_start()/CPIS_wait() sequence, the application must call CPIS_reset() to reset some internal state related to that function before next call of CPIS_start() is applied for the same function.

For example:

```
/* Execute function pointed by handler a first time */
CPIS_start(handler1);CPIS_wait(handler1);
/* Must reset function pointed by handler1 after first execution */
CPIS_reset(handler1);
/* Execute function pointed by handler a second time */
CPIS_start(handler1);CPIS_wait(handler1);
```

Performance

NA

3.3.6 CPIS_wait

CPIS_wait	***Waits for Completion of Started Processing***
Syntax	Int32 **CPIS_wait**(CPIS_Handle *handle*);
Arguments	CPIS_Handle *handle* /* handle of the function to be executed */
Return Value	Returns 0 if callback function was not set, otherwise returns callback function's return value.
Description	Waits for the completion of the last execution initiated by CPIS_start(). If wait callback function has not been setup then busy while() loop is used as wait operation within the VICP signal processing library. Else, wait callback function overrides the internal wait method. CPIS_wait() returns the return value of the wait callback to the application.
Performance	NA

3.3.7 CPIS_updateSrcDstPtr

CPIS_updateSrcDstPtr ***Update the Source and Destination Addresses of the Processing API***

Syntax

Int32 **CPIS_updateSrcDstPtr**(CPIS_Handle *handle*, CPIS_BaseParms **base*);

Arguments

CPIS_Handle	**handle*	Handle returned in the case of asynchronous execution. In case of synchronous execution, NULL is returned.
CPIS_BaseParms	**base*	Base parameters to specify location, size of buffers

Return Value

Always returns 0.

Description

The CPIS_updateSrcDstPtr() API updates the source and destination addresses of the processing API whose handle is passed as the input argument. The address values are taken from the structure CPIS_BaseParms, base→srcBuf[.].ptr and base→dstBuf[.].ptr.

The API CPIS_reset() must be called for the new addresses to be effective the next time CPIS_start() is called. In consequence, the order of function calls should be:

CPIS_updateSrcDstPtr(…)

CPIS_reset(…)

CPIS_start(…)

CPIS_wait(…)

Performance

NA

3.3.8 CPIS_reset

CPIS_reset ***Reset Internal State of the VICP***

Syntax

Int32 **CPIS_reset**(CPIS_Handle *handle*);

Arguments

CPIS_Handle *handle* /* handle of the function to be executed */;

Return Value

Always returns 0.

Description

The CPIS_reset() API resets the internal state of the VICP. If the same processing API has to be executed more than once, the application must call CPIS_reset() after each CPIS_start()/CPIS_wait() sequence.

For example:

```
/* Execute function pointed by handler a first time */
CPIS_start(handler1);CPIS_wait(handler1);
/* Must reset function pointed by handler1 after first execution */
CPIS_reset(handler1);
/* Execute function pointed by handler a second time */
CPIS_start(handler1);CPIS_wait(handler1);
```

Performance

Execution time depends on the number of input and output buffers used by the processing API. The following table lists some configurations used in the library:

Number of Input + Output Buffers	CPU Cycles
2	6500
3	7500
4	9000

3.3.9 CPIS_delete

CPIS_delete ***Deletes All Information for Last Executed API***

Syntax Int32 **CPIS_delete**(CPIS_Handle *handle*);

Arguments CPIS_Handle *handle* /* handle of the function to be executed */;

Return Value Always returns 0.

Description The CPIS_delete() API removes all information pertaining to the API that was last executed on the VICP. The API should be called when changing the API that needs to be executed on the VICP.

Performance NA

3.3.10 CPIS_colorSpcConv

CPIS_colorSpcConv *Performs Color Space Conversions*

Syntax

Int32 **CPIS_colorSpcConv**(

CPIS_Handle	**handle*,
CPIS_BaseParms	**base*,
CPIS_colorSpcConvParms	**params*,
CPIS_ExecType	*execType*

);

Arguments

CPIS_Handle	**handle*	Handle returned in the case of asynchronous execution. In case of synchronous execution, NULL is returned.
CPIS_BaseParms	**base*	Base parameters to specify location, size of buffers
CPIS_colorSpcConvParms	**params*	The specific parameters for the colorSpcConv API are shown in Table 4.
CPIS_ExecType	*execType*	Execution type: synchronous or asynchronous

```
typedef struct {

    Int16  matrix[9];
    Uint32 qShift;
    Int16  preOffset[3];
    Int16  postOffset[3];
    Int16  signedInput[3];
    Int16  signedOutput[3];

    CPIS_ColorDsMode colorDsMode;

} CPIS_ColorSpcConvParms;
```

Table 4. CPIS_colorSpcConv API Parameters

Parameter	Description	
matrix[9];	Pointer to transformation matrix coefficients.	
qShift;	Q format or number of bits to downshift after multiplication with Matrix coefficients.	
preOffset[3];	Offset to add to each color component before matrix multiplication.	
postOffset[3];	Offset to add to each color component after matrix multiplication.	
signedInput[3];	signedInput[i] is effective only when preOffset[i] is non zero	
	0	8-bit input is unsigned
	1	8-bit input is signed
signedOutput[3];	signedOutput[i] is effective only when postOffset[i] is non zero	
	0	8-bit output is unsigned
	1	8-bit output is signed
CPIS_ColorDSMode colorDSMode	Color downsampling mode for YUV444→YUV422 conversion. Values are:	
	CPIS_DS_NONE	Default value to be used when the output format is not CPIS_YUV_422ILE.
	CPIS_DS_SKIP	Downsampling mode consisting of skipping every other U or V values; thus, keeping only the leading U or V value of each pair.
	CPIS_DS_AVERAGE	Downsampling mode consisting of averaging every pair of U or V values.
	CPIS_DS_FOR_ALPHABLEND	Special downsampling mode to be applied for image data that is going to be used as a foreground plane in a future alpha-blending processing.

Return Value

0 Success

1 Error and CPIS_errorno set to originating error.

Description

Color space conversion performs the following matrix transformation:

```
[O1         [ a b c      [ I1
 O2   =       d e f    *   I2
 O3]          g h I ]      I3 ]

O1= a*I1 + b*I2 + c*I3
O2= d*I1 + e*I2 + f*I3
O3= g*I1 + h*I2 + i*I3
```

The member matrix of the structure CPIS_ColorSpcConvParms needs to be set up as follow: params.matrix[9]={a, d, g, b, e, h, c, f, i};

The elements of the matrix are 16-bit signed.

To handle decimal number, use Q format and set params.qSfhift member to the amount to downshift.

```
For RGB->YCbCr
Real number matrix is :
[
        a= 0.299 b= 0.587 c= 0.114
        d= -0.169 e= -0.331 f= 0.5000
        g= 0.5000 h= -0.4190 i= -0.081
]
```

Integer matrix params.matrix could be Q15. In this case multiply real number matrix by 32765, convert to 16 bit integer and set qShift=15.

- RGB input must be between [0 255]
- Y output will be between [0 255]
- CbCr output will be between [-127 128]

```
For YCbCr->RGB
Real number matrix is :
[
        a= 1.0 b= -0.0009 c= 1.4017
        d= 1.0 e= -0.3437 f= -0.7142
        g= 1.0 h= 1.7722 i= 0.0010
]
```

Integer matrix params.matrix could be Q15. In this case multiply real number matrix by 32765, convert to 16 bit integer and set qShift=15.

- Y input must be between [0 255]
- CbCr input must be between [-127 128]
- RGB output will be between [0 255]

If necessary, use params.preOffset or params.postOffset to adjust input or output to desired range. In this case it is necessary to tell the function whether the input or output are signed or unsigned 8-bit value by filling params.signedInput and params.signedOutput.

When output format is set to CPIS_YUV_422ILE, it is possible to control the way the horizontal downsampling of the U and V colors is done by setting the parameter params. colorDsMode.

Constraints

- base. procBlockSize.width × base. procBlockSize.height < MAX_COLORSPC_BLOCKSIZE, where MAX_COLORSPC_BLOCKSIZE is defined in vicplib.h.
- base. procBlockSize.width must be a multiple of 4.
- Format Support
  - Native format (faster processing): CPIS_YUV_444P, CPIS_RGB_P
  - Non native format:
    - CPIS_YUV422ILE, CPIS_YUV422ILE|CPIS_ALPHA
    - CPIS_RGB888|IMPROC_ALPHA, which correspond to 32-bits packed rgb format with alpha channel, also called ARGB. The format assumes little endian where byte 0 corresponds to the alpha value.
- CPIS_RGB888 alone and other formats are not supported.

Performance

For RGB_P to YUV_444P. For input buffer size of 360x150 and processing buffer size of 36x10.

Setup Time	~130000 CPU Clocks
Processing Time	~550000 CPU Clocks

3.3.11 CPIS_alphaBlend

CPIS_alphaBlend ***Performs Alpha-Blending Between a Foreground Plane and Background Plane***

Syntax

Int32 **CPIS_alphaBlend** (

CPIS_Handle	**handle*,
CPIS_BaseParms	**base*,
CPIS_alphaBlendParms	**params*,
CPIS_ExecType	*execType*

);

Arguments

CPIS_Handle	**handle*	Handle returned in the case of asynchronous execution. In case of synchronous execution, NULL is returned.
CPIS_BaseParms	**base*	Base parameters to specify location, size of buffers
CPIS_alphaBlendParms	**params*	The specific parameters for the alphaBlend API are shown in Table 5.
CPIS_ExecType	*execType*	Execution type: synchronous or asynchronous

```
typedef struct {

    Uint16      useGlobalAlpha;
    Uint16      alphaValue;
    CPIS_Buffer background;

} CPIS_AlphaBlendParms;
```

Table 5. CPIS_alphaBlend API Parameters

Parameter	Description
useGlobalAlpha	If set to 1 then alpha value specified next is used for entire image. Otherwise if 0, use alpha plane passed as src[1].
alphaValue	Global alpha value, 0-255, 255 let see foreground, 0 let see background
background	Background buffer information

Return Value

0 Success

1 Error and CPIS_errorno set to originating error.

Description

Perform alpha-blending between a foreground plane and background plane, both in yuv422 interleaved format. Either a global alpha coefficient or an alpha plane can be used. If a global alpha coefficient is used, set params.useGlobalAlpha to 1 and params.alphaValue to the desired value between 0 and 255.

The blending used is:

output= alpha/256 × *foreground* + (255 – alpha)/256 × *background*

The way the pointers to source, background, alpha planes are passed to the function is as follow:

```
foreground plane -> base.SrcBuf[0].ptr
alpha plane -> base.srcBuf[1].ptr
background -> params.background.ptr
```

Constraints

- base. procBlockSize.width must be a multiple of 2 if global alpha is used, otherwise it must be a multiple of 4.
- base. procBlockSize.width × base. procBlockSize.height <MAX_ALPHABLEND_GLOBAL_ALPHA_BLOCKSIZE if global alpha used. Otherwise, MAX_ALPHABLEND_BLOCKSIZE. These symbols are defined in vicplib.h.
- Native (faster processing): CPIS_YUV_422ILE
- Other formats are not supported

Performance

For input buffer size of 160x100 and processing buffer size of 32x25

Setup Time	~100000 CPU Clocks
Processing Time	~100000 CPU Clocks

3.3.12 CPIS_rotation

CPIS_rotation ***Performs Rotation of the Input Buffer***

Syntax

Int32 **CPIS_rotation** (

CPIS_Handle	**handle*,
CPIS_BaseParms	**base*,
CPIS_rotationParms	**params*,
CPIS_ExecType	*execType*

);

Arguments

CPIS_Handle	**handle*	Handle returned in the case of asynchronous execution. In case of synchronous execution, NULL is returned.
CPIS_BaseParms	**base*	Base parameters to specify location, size of buffers
CPIS_rotationParms	**params*	The specific parameters for the rotation API are shown in Table 6.
CPIS_ExecType	*execType*	Execution type: synchronous or asynchronous

```
typedef struct {
    Int16 angle;
} CPIS_RotationParms;
```

Table 6. CPIS_rotation API Parameter

Parameter	Description
angle	Angle of rotation

Return Value

0 Success

1 Error and CPIS_errorno set to originating error.

Description

Perform rotation of input buffer by the specified angle.

Constraints

- base. procBlockSize.width must be a multiple of 2.
- The angles supported for rotation:
  - 0, 360
  - 90, -270
  - 180, -180
  - 270, -90
- The API supports CPIS_YUV_422ILE format for both source and data.
- Other formats are not supported.
- base. procBlockSize.width × base. procBlockSize.height <MAX_ROTATION_BLOCKSIZE
  The symbol is defined in vicplib.h.

Performance

For 422ILE. For input buffer size of 320x240 and processing buffer size of 16x16.

Setup Time	~100000 CPU Clocks
Processing Time	~1000000 CPU Clocks

3.3.13 CPIS_fillMem

CPIS_fillMem ***Fills Input Buffer With Constant Data***

Syntax

Int32 **CPIS_fillMem** (

CPIS_Handle	**handle*,
CPIS_BaseParms	**base*,
CPIS_fillMemParms	**params*,
CPIS_ExecType	*execType*

);

Arguments

CPIS_Handle	**handle*	Handle returned in the case of asynchronous execution. In case of synchronous execution, NULL is returned.
CPIS_BaseParms	**base*	Base parameters to specify location, size of buffers
CPIS_fillMemParms	**params*	The specific parameters for the fillMem API are shown in Table 7.
CPIS_ExecType	*execType*	Execution type: synchronous or asynchronous

```
typedef struct {
    Uint8 * constData;
} CPIS_FillMemParms;
```

Table 7. CPIS_fillMem API Parameters

Parameter	Description
constData	Pointer to 8-bit or 16-bit data that needs to be filled in the input buffer.

Return Value

0 Success

1 Error and CPIS_errorno set to originating error.

Description

The CPIS_fillMem API is used to fill the input buffer with a constant data. The constant data used to fill the input buffer is pointed to by the constData pointer in the API params. The data is either 1 byte for the case of CPIS_8BIT or 2bytes in the case of CPIS_16BIT.

Constraints

- The API supports CPIS_16BIT and CPIS_8BIT data format for both source and destination data.
- The source and destination data formats should be the same.
- base. procBlockSize.width × base. procBlockSize.height < MAX_FILLMEM_BLOCKSIZE

  The symbol is defined in vicplib.h. The MAX_FILLMEM_BLOCKSIZE is defined in terms of bytes. Thus, when using the data format of CPIS_16BIT, base.procBlockSize.width × base. procBlockSize.height < MAX_FILLMEM_BLOCKSIZE/2

Performance

For CPIS_16BIT. For input buffer size of 1860x330 and processing buffer size of 186x22.

Setup Time	~90000 CPU Clocks
Processing Time	~800000 CPU Clocks

For this function, most of the processing time is due to EDMA overhead as the VICP is doing very little computation in this simple algorithm. If necessary you can customize the implementation of this function by chaining a few more VICP computation functions without incurring any extra performance drop. Refer to *VICP Computation Unit Library and VICP Scheduling Unit Library for DM6446, DM6441, DM647, and DM648* (SPRUGN1) for details about this topic.

3.3.14 CPIS_arrayOp

CPIS_arrayOp ***Performs Arithmetic and Logical Operation Between Elements of Two Input Arrays***

Syntax

Int32 **CPIS_arrayOp** (

CPIS_Handle	**handle*,
CPIS_BaseParms	**base*,
CPIS_arrayOpParms	**params*,
CPIS_ExecType	*execType*

);

Arguments

CPIS_Handle	**handle*	Handle returned in the case of asynchronous execution. In case of synchronous execution, NULL is returned.
CPIS_BaseParms	**base*	Base parameters to specify location, size of buffers
CPIS_arrayOpParms	**params*	The specific parameters for the arrayOp API are shown in Table 8.
CPIS_ExecType	*execType*	Execution type: synchronous or asynchronous

```
/* The various arithmetic and logical operations supported by the library */
typedef enum {
    CPIS_OP_MPY=0,
    CPIS_OP_ABDF,
    CPIS_OP_ADD,
    CPIS_OP_SUB,
    CPIS_OP_TLU,
    CPIS_OP_AND,
    CPIS_OP_OR,
    CPIS_OP_XOR,
    CPIS_OP_MIN,
    CPIS_OP_MAX,
    CPIS_OP_MINSAD,
    CPIS_OP_MAXSAD,
    CPIS_OP_MEDIAN,
    CPIS_OP_BINLOG,
    CPIS_OP_3DLUT,
    CPIS_OP_CONDWR
} CPIS_Operation;

/* ArrayOperation API params */
typedef struct {
    Uint16         qShift;
    CPIS_Operation operation;
    Int32          sat_high;
    Int32          sat_high_set;
    Int32          sat_low;
    Int32          sat_low_set;

} CPIS_ArrayOpParms;
```

Table 8. CPIS_arrayOp API Parameters

Parameter	Description
qshift	Q format or number of bits to downshift after processing
operation	Operation to be performed between elements of the input arrays.
sat_high	High value to check for saturation
sat_high_set	Value to set the output with if high saturation
sat_low	Low value to check for saturation
sat_low_set	Value to set the output with if low saturation

Return Value

0 Success

1 Error and CPIS_errorno set to originating error.

Description

The CPIS_arrayOp API performs arithmetic and logical operation between corresponding elements of two input arrays. The result is written in the output array. Each of the two inputs and output can be either 16-bit data or 8-bit data. The API allows the resulting data to be rounded, shifted and also saturated before generating the output.

Constraints

- The API supports CPIS_16BIT, CPIS_U16BIT, CPIS_8BIT and CPIS_U8BIT data format for the input data. For the output, CPIS_8BIT and CPIS_16BIT are supported.
- base. procBlockSize.width × base. procBlockSize.height < MAX_ARRAYOP_BLOCKSIZE

  The symbol is defined in vicplib.h. The MAX_ARRAYOP_BLOCKSIZE is defined in terms of bytes. Thus, when using data format of CPIS_16BIT, base.procBlockSize.width × base. procBlockSize.height < MAX_ARRAYOP_BLOCKSIZE/2
- The following operations are supported:
  - CPIS_OP_ADD
  - CPIS_OP_SUB
  - CPIS_OP_MPY
  - CPIS_OP_ABDF
  - CPIS_OP_AND
  - CPIS_OP_OR
  - CPIS_OP_XOR
  - CPIS_OP_MIN
  - CPIS_OP_MAX

Performance

For CPIS_16BIT. For input buffer size of 640x480 and processing buffer size of 64x32.

Setup Time	~90000 CPU Clocks
Processing Time	~1140000 CPU Clocks

For this function, most of the processing time is due to EDMA overhead as the VICP is doing very little computation in this simple algorithm. If necessary you can customize the implementation of this function by chaining a few more VICP computation functions without incurring any extra performance drop. Refer to *VICP Computation Unit Library and VICP Scheduling Unit Library for DM6446, DM6441, DM647, and DM648* (SPRUGN1) for details about this topic.

3.3.15 CPIS_arrayScalarOp

CPIS_arrayScalarOp *Performs Operation Between Elements of an Input Array and a Scalar Value*

Syntax

```
Int32 CPIS_arrayScalarOp (
    CPIS_Handle              *handle,
    CPIS_BaseParms           *base,
    CPIS_arrayScalarOpParms  *params,
    CPIS_ExecType            execType
);
```

Arguments

CPIS_Handle	**handle*	Handle returned in the case of asynchronous execution. In case of synchronous execution, NULL is returned.
CPIS_BaseParms	**base*	Base parameters to specify location, size of buffers
CPIS_arrayScalarOpParms	**params*	The specific parameters for the arrayScalarOp API are shown in Table 9.
CPIS_ExecType	*execType*	Execution type: synchronous or asynchronous

```
/* The various arithmetic and logical operations supported by the library */
typedef enum {
    CPIS_OP_MPY=0,
    CPIS_OP_ABDF,
    CPIS_OP_ADD,
    CPIS_OP_SUB,
    CPIS_OP_TLU,
    CPIS_OP_AND,
    CPIS_OP_OR,
    CPIS_OP_XOR,
    CPIS_OP_MIN,
    CPIS_OP_MAX,
    CPIS_OP_MINSAD,
    CPIS_OP_MAXSAD,
    CPIS_OP_MEDIAN,
    CPIS_OP_BINLOG,
    CPIS_OP_3DLUT,
    CPIS_OP_CONDWR
} CPIS_Operation;

/* ArrayOperation API params */
typedef struct {
    Uint16         qShift;
    CPIS_Operation operation;
    Int32          sat_high;
    Int32          sat_high_set;
    Int32          sat_low;
    Int32          sat_low_set;
    Int32          mask[2][2];

} CPIS_ArrayScalarOpParms;
```

Table 9. CPIS_arrayScalarOp API Parameters

Parameter	Description
qshift	Q format or number of bits to downshift after processing
operation	Operation to be performed between elements of the input arrays.
sat_high	High value to check for saturation
sat_high_set	Value to set the output with if high saturation
sat_low	Low value to check for saturation
sat_low_set	Value to set the output with if low saturation

Table 9. CPIS_arrayScalarOp API Parameters (continued)

Parameter	Description
mask[2] [2]	The function accepts a 2×2 matrix that is tiled and the operation applied between the matrix and the input buffer.

Return Value

0 Success

1 Error and CPIS_errorno set to originating error.

Description

The CPIS_arrayScalarOp API performs arithmetic and logical operation between elements of an input array and a scalar value. The API allows the user to specify a 2x2 matrix that is used as the scalar data. The 2x2 matrix is effectively tiled and the operation applied between the matrix data and the input buffer. The result is written in the output array. The inputs and output can be either 16-bit data or 8-bit data. The API allows the resulting data to be rounded, shifted and also saturated before generating the output.

Constraints

- The API supports CPIS_16BIT, CPIS_U16BIT, CPIS_8BIT and CPIS_U8BIT data format for the input data. For the output, CPIS_8BIT and CPIS_16BIT are supported.
- base. procBlockSize.width × base. procBlockSize.height < MAX_ARRAYSCALAROP_BLOCKSIZE

  The symbol is defined in vicplib.h. The MAX_ARRAYSCALAROP_BLOCKSIZE is defined in terms of bytes. Thus, when using data format of CPIS_16BIT, base.procBlockSize.width × base. procBlockSize.height < MAX_ARRAYSCALAROP_BLOCKSIZE/2
- The following operations are supported:
  - CPIS_OP_ADD
  - CPIS_OP_SUB
  - CPIS_OP_MPY
  - CPIS_OP_ABDF
  - CPIS_OP_AND
  - CPIS_OP_OR
  - CPIS_OP_XOR
  - CPIS_OP_MIN
  - CPIS_OP_MAX

Performance

For CPIS_16BIT. For input buffer size of 1280x480 and processing buffer size of 128x32.

Setup Time	~80000 CPU Clocks
Processing Time	~1450000 CPU Clocks

For this function, most of the processing time is due to EDMA overhead as the VICP is doing very little computation in this simple algorithm. If necessary you can customize the implementation of this function by chaining a few more VICP computation functions without incurring any extra performance drop. Refer to *VICP Computation Unit Library and VICP Scheduling Unit Library for DM6446, DM6441, DM647, and DM648* (SPRUGN1) for details about this topic.

3.3.16 CPIS_arrayCondWrite

CPIS_arrayCondWrite *Conditionally Write Data to an Array*

Syntax

Int32 **CPIS_arrayCondWrite** (

CPIS_Handle	**handle*,
CPIS_BaseParms	**base*,
CPIS_arrayCondWriteParms	**params*,
CPIS_ExecType	*execType*

);

Arguments

CPIS_Handle	**handle*	Handle returned in the case of asynchronous execution. In case of synchronous execution, NULL is returned.
CPIS_BaseParms	**base*	Base parameters to specify location, size of buffers
CPIS_arrayCondWriteParms	**params*	The specific parameters for the arrayCondWrite API are shown in Table 10.
CPIS_ExecType	*execType*	Execution type: synchronous or asynchronous

```
/* The conditions supported by the array conditional write API */
typedef enum {
    CPIS_WR_ZERO=0,    /* Write if Zero */
    CPIS_WR_NOTZERO,   /* Write if Not Zero */
    CPIS_WR_SAT,       /* Write if Saturate */
    CPIS_WR_NOTSAT     /* Write if Not Saturate */
} CPIS_WriteMode;

/* ArrayConditionalWrite API params */
typedef struct {
    Uint16         qShift;
    CPIS_WriteMode writeMode;
    Int32          sat_high;
    Int32          sat_high_set;
    Int32          sat_low;
    Int32          sat_low_set;

} CPIS_ArrayCondWriteParms;
```

Table 10. CPIS_ArrayCondWrite API Parameters

Parameter	Description
qshift	Q format or number of bits to downshift after processing
writeMode	Conditional Write Mode
sat_high	High value to check for saturation
sat_high_set	Value to set the output with if high saturation
sat_low	Low value to check for saturation
sat_low_set	Value to set the output with if low saturation

Return Value

0 Success

1 Error and CPIS_errorno set to originating error.

Description

The Array Conditional Write API accepts as input three arrays; input1, input2 and input3. The API checks every element of the input2 array to see if the element satisfies the conditions specified by the writeMode. If the condition is met, the corresponding element in the output array is set equal to the corresponding element in the input1 array. If the condition is not met, the output element is set equal to the corresponding element in the input3. The inputs and output can be either 16-bit data or 8-bit data. The API allows the resulting data to be rounded, shifted and also saturated before generating the output.

Constraints

- The API supports CPIS_16BIT, CPIS_U16BIT, CPIS_8BIT and CPIS_U8BIT data format for the input data. For the output, CPIS_8BIT and CPIS_16BIT are supported.
- base. procBlockSize.width × base. procBlockSize.height < MAX_ARRAYCONDWRITE_BLOCKSIZE

  The symbol is defined in vicplib.h. The MAX_ARRAYCONDWRITE_BLOCKSIZE is defined in terms of bytes. Thus, when using data format of CPIS_16BIT, base.procBlockSize.width × base. procBlockSize.height < MAX_ARRAYCONDWRITE_BLOCKSIZE/2

Performance

For CPIS_16BIT. For input buffer size of 1700x120 and processing buffer size of 170x8.

Setup Time	~99000 CPU Clocks
Processing Time	~1060000 CPU Clocks

For this function, most of the processing time is due to EDMA overhead as the VICP is doing very little computation in this simple algorithm. If necessary you can customize the implementation of this function by chaining a few more VICP computation functions without incurring any extra performance drop. Refer to *VICP Computation Unit Library and VICP Scheduling Unit Library for DM6446, DM6441, DM647, and DM648* (SPRUGN1) for details about this topic.

3.3.17 CPIS_YCbCrPack

CPIS_YCbCrPack ***Packs Input Planar YCbCr Data to Generate Interleaved YCbCr Data***

Syntax

```
Int32 CPIS_YCbCrPack (
    CPIS_Handle          *handle,
    CPIS_BaseParms       *base,
    CPIS_YCbCrPackParms  *params,
    CPIS_ExecType        execType
);
```

Arguments

CPIS_Handle	**handle*	Handle returned in the case of asynchronous execution. In case of synchronous execution, NULL is returned.
CPIS_BaseParms	**base*	Base parameters to specify location, size of buffers
CPIS_YCbCrPackParms	**params*	The enums shown in Table 11 represent the various color space types that are supported by the YCbCrPack API.
CPIS_ExecType	*execType*	Execution type: synchronous or asynchronous

```
/*  These enums represent the various color space types that
    are supported by the YCbCrPack routine                  */

typedef enum {
    CPIS_444_16BIT_TO_422_8BIT=0,
    CPIS_422_16BIT_TO_422_8BIT,
    CPIS_420_16BIT_TO_422_8BIT,
    CPIS_422V_16BIT_TO_422_8BIT,
    CPIS_444_16BIT_TO_444_8BIT,
    CPIS_422_16BIT_TO_444_8BIT,
    CPIS_420_16BIT_TO_444_8BIT,
    CPIS_422V_16BIT_TO_444_8BIT,

    CPIS_444_8BIT_TO_422_8BIT=0x4000,
    CPIS_422_8BIT_TO_422_8BIT,
    CPIS_420_8BIT_TO_422_8BIT,
    CPIS_422V_8BIT_TO_422_8BIT,
    CPIS_444_8BIT_TO_444_8BIT,
    CPIS_422_8BIT_TO_444_8BIT,
    CPIS_420_8BIT_TO_444_8BIT,
    CPIS_422V_8BIT_TO_444_8BIT,

    CPIS_444_16BIT_TO_422_16BIT=0x8000,
    CPIS_422_16BIT_TO_422_16BIT,
    CPIS_420_16BIT_TO_422_16BIT,
    CPIS_422V_16BIT_TO_422_16BIT,
    CPIS_444_16BIT_TO_444_16BIT,
    CPIS_422_16BIT_TO_444_16BIT,
    CPIS_420_16BIT_TO_444_16BIT,
    CPIS_422V_16BIT_TO_444_16BIT,

    CPIS_444_8BIT_TO_422_16BIT=0xC000,
    CPIS_422_8BIT_TO_422_16BIT,
    CPIS_420_8BIT_TO_422_16BIT,
    CPIS_422V_8BIT_TO_422_16BIT,
    CPIS_444_8BIT_TO_444_16BIT,
    CPIS_422_8BIT_TO_444_16BIT,
    CPIS_420_8BIT_TO_444_16BIT,
    CPIS_422V_8BIT_TO_444_16BIT

} CPIS_YCbCrPack;

/* YCbCrPack API params */
typedef struct {
```

```
    Uint16              qShift;
    CPIS_ColorSpacePack colorSpace;
    Int32               sat_high;
    Int32               sat_high_set;
    Int32               sat_low;
    Int32               sat_low_set;
    Int16               scale;

} CPIS_YCbCrPackParms;
```

Table 11. CPIS_YCbCrPack API Parameters

Parameter	Description
qshift	Q format or number of bits to downshift after processing
colorSpace	Enum that decides the pack operation
sat_high	High value to check for saturation
sat_high_set	Value to set the output with if high saturation
sat_low	Low value to check for saturation
sat_low_set	Value to set the output with if low saturation
scale	Scaling that is applied to the input data

Return Value

0 Success

1 Error and CPIS_errorno set to originating error.

Description

The YCbCrPack API takes as input planar YCbCr data and packs the data to generate interleaved YCbCr data. The API also allows the data to be scaled, shifted and saturated before generating the output. The API also supports changing the subsampling format on the fly. Thus, the input can be 444 planar and the output can be 422 or 420. In this case, the chroma is simply subsampled. Similarly, the input can be 420 planar and the output can be 444 or 422 interleaved. In this case, the data is simply replicated to generate the needed output format.

Constraints

- The API supports CPIS_YUV_420P, CPIS_YUV_422P and CPIS_YUV_444P data formats for the input data. The output data can be in CPIS_YUV_422ILE or CPIS_YUV_444ILE data format.
- base. procBlockSize.width × base. procBlockSize.height < MAX_YCBCRPACK_BLOCKSIZE

  The symbol is defined in vicplib.h.
- The following packing operations are supported:
  - CPIS_444_8BIT_TO_422_8BIT
  - CPIS_422_8BIT_TO_422_8BIT
  - CPIS_420_8BIT_TO_422_8BIT
  - CPIS_444_8BIT_TO_444_8BIT
  - CPIS_422_8BIT_TO_444_8BIT
  - CPIS_420_8BIT_TO_444_8BIT

Performance

For CPIS_444_8BIT_TO_444_8BIT. For input buffer size of 1360x150 and processing buffer size of 136x10

Setup Time	~100000 CPU Clocks
Processing Time	~1300000 CPU Clocks

3.3.18 CPIS_YCbCrUnpack

CPIS_YCbCrUnpack *Unpacks Input Interleaved YCbCr Data to Generate Planar YCbCr Data*

Syntax

```
Int32 CPIS_YCbCrUnpack (
    CPIS_Handle              *handle,
    CPIS_BaseParms           *base,
    CPIS_YCbCrUnpackParms    *params,
    CPIS_ExecType            execType
);
```

Arguments

CPIS_Handle	**handle*	Handle returned in the case of asynchronous execution. In case of synchronous execution, NULL is returned.
CPIS_BaseParms	**base*	Base parameters to specify location, size of buffers
CPIS_YCbCrUnPackParms	**params*	The enums shown in Table 12 represent the various color space types that are supported by the YCbCrUnPack API.
CPIS_ExecType	*execType*	Execution type: synchronous or asynchronous

```
/*  These enums represent the various color space types that
    are supported by the YCbCrUnPack routine                */

typedef enum {
    CPIS_422_TO_444_8BIT=0,
    CPIS_422_TO_422_8BIT,
    CPIS_422_TO_420_8BIT,
    CPIS_444_TO_444_8BIT,
    CPIS_444_TO_422_8BIT,
    CPIS_444_TO_420_8BIT,

    CPIS_422_TO_444_16BIT=0x8000,
    CPIS_422_TO_422_16BIT,
    CPIS_422_TO_420_16BIT,
    CPIS_444_TO_444_16BIT,
    CPIS_444_TO_422_16BIT,
    CPIS_444_TO_420_16BIT

} CPIS_YCbCrUnpack;

/* YCbCrUnPack API params */

typedef struct {
    Uint16                qShift;
    CPIS_ColorSpaceUnpack colorSpace;
    Int32                 sat_high;
    Int32                 sat_high_set;
    Int32                 sat_low;
    Int32                 sat_low_set;
    Int32                 scale;

} CPIS_YCbCrUnpackParms;
```

Table 12. CPIS_YCbCrUnPack API Parameters

Parameter	Description
qshift	Q format or number of bits to downshift after processing
colorSpace	Enum that decides the unpack operation
sat_high	High value to check for saturation
sat_high_set	Value to set the output with if high saturation
sat_low	Low value to check for saturation
sat_low_set	Value to set the output with if low saturation
scale	Scaling that is applied to the input data

Return Value

0 Success

1 Error and CPIS_errorno set to originating error.

Description

The YCbCrUnPack API takes as input interleaved YCbCr data and unpacks the data to generate planar YCbCr data. The API also allows the data to be scaled, shifted and saturated before generating the output. The API also supports changing the subsampling format on the fly. Thus, the input can be 444 interleaved and the output can be 422 or 420 planar. In this case, the chroma is simply subsampled. Similarly, the input can be 422 interleaved and the output can be 444. In this case, the data is simply replicated to generate the needed output format.

Constraints

- The API supports CPIS_YUV_422ILE and CPIS_YUV_444ILE data formats for the input data. The output data can be in CPIS_YUV_420P, CPIS_YUV_422P or CPIS_YUV_444P data format.
- base. procBlockSize.width × base. procBlockSize.height < MAX_YCBCRUNPACK_BLOCKSIZE

  The symbol is defined in vicplib.h.
- The following packing operations are supported:
  - CPIS_444_TO_422_8BIT
  - CPIS_422_TO_422_8BIT
  - CPIS_420_TO_422_8BIT
  - CPIS_444_TO_444_8BIT
  - CPIS_422_TO_444_8BIT
  - CPIS_420_TO_444_8BIT

Performance

For CPIS_444_TO_444_8BIT. For input buffer size of 1360x150 and processing buffer size of 136x10.

Setup Time	~100000 CPU Clocks
Processing Time	~1180000 CPU Clocks

3.3.19 CPIS_matMul

CPIS_matMul ***Performs Matrix Multiplication Between Two Matrices***

Syntax

```
Int32 CPIS_matMul (
    CPIS_Handle      *handle,
    CPIS_BaseParms   *base,
    CPIS_MatMulParms *params,
    CPIS_ExecType    execType
);
```

Arguments

CPIS_Handle	**handle*	Handle returned in the case of asynchronous execution. In case of synchronous execution, NULL is returned.
CPIS_BaseParms	**base*	Base parameters to specify location, size of buffers
CPIS_MatMulParms	**params*	The specific parameters for the matMul API are shown in Table 13.
CPIS_ExecType	*execType*	Execution type: synchronous or asynchronous

```
typedef struct {
    Uint16      qShift;

    Int32       sat_high;
    Int32       sat_high_set;
    Int32       sat_low;
    Int32       sat_low_set;

    Int16       matWidth;
    Int16       matHeight;
    void        * matPtr;
    CPIS_Format matFormat;

} CPIS_MatMulParms;
```

Table 13. CPIS_matMul API Parameters

Parameter	Description
qshift	Q format or number of bits to downshift after processing
sat_high	High value to check for saturation
sat_high_set	Value to set the output with if high saturation
sat_low	Low value to check for saturation
sat_low_set	Value to set the output with if low saturation
matWidth	Matrix width
matHeight	Matrix height
matPtr	Pointer to matrix data arranged in row order
matFormat	Data format for the matrix coefficients

Return Value

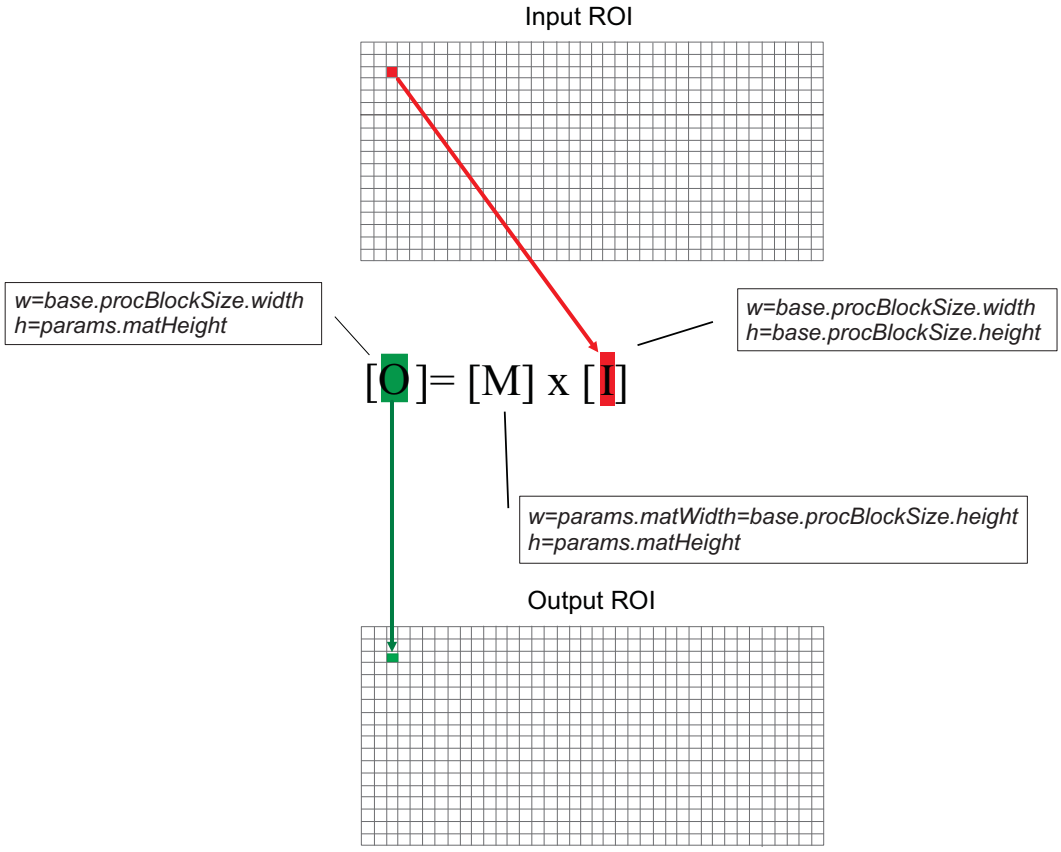
0 Success

1 Error and CPIS_errorno set to originating error.

Description

The matMul API performs matrix multiplication. Each processing block of the ROI grid is considered as a matrix. The matMul API multiplies the matrix *M* passed as a parameter in CPIS_MatMulParms with each processing block of the input ROI. The matrix *M* does not have to be square; its width can be different than its height. The only constraint is that its width must be equal to the processing block's height: base. procBlockSize.height = params.matWidth. The output ROI grid is made of blocks of size: base. procBlockSize.width × params.matHeight.

Figure 3. CPIS_MatMul Description



Constraints

- The API supports CPIS_16BIT, CPIS_U16BIT, CPIS_8BIT and CPIS_U8BIT data format for the input data. The matrix format and the input format should be the same. For the output, CPIS_8BIT, CPIS_16BIT, and CPIS_32BIT are supported.
- base. procBlockSize.width × base. procBlockSize.height < MAX_MATMUL_BLOCKSIZE

  The symbol is defined in vicplib.h. The MAX_ MATMUL _BLOCKSIZE is defined in terms of bytes. Thus, when using data format of CPIS_16BIT, base.procBlockSize.width × base. procBlockSize.height < MAX_ MATMUL _BLOCKSIZE/2.
- base. procBlockSize.height = CPIS_matMulParms.matWidth

Performance

For CPIS_16BIT. For input buffer size of 2020x10 and processing buffer size of 202*10. For Matrix size of 10x10.

Setup Time	~88000 CPU Clocks
Processing Time	~240000 CPU Clocks

3.3.20 CPIS_sum

CPIS_sum ***Sums the Elements in a Block of Data***

Syntax

```
Int32 CPIS_sum (
    CPIS_Handle     *handle,
    CPIS_BaseParms  *base,
    CPIS_SumParms   *params,
    CPIS_ExecType   execType
);
```

Arguments

CPIS_Handle	**handle*	Handle returned in the case of asynchronous execution. In case of synchronous execution, NULL is returned.
CPIS_BaseParms	**base*	Base parameters to specify location, size of buffers
CPIS_SumParms	**params*	The specific parameters for the sum API are shown in Table 14.
CPIS_ExecType	*execType*	Execution type: synchronous or asynchronous

```
typedef struct {
    Uint16      qShift;

    Int32       sat_high;
    Int32       sat_high_set;
    Int32       sat_low;
    Int32       sat_low_set;

    Int16       * scalarPtr;
    CPIS_Format scalarFormat;

} CPIS_SumParms;
```

Table 14. CPIS_sum API Parameters

Parameter	Description
qshift	Q format or number of bits to downshift after processing
sat_high	High value to check for saturation
sat_high_set	Value to set the output with if high saturation
sat_low	Low value to check for saturation
sat_low_set	Value to set the output with if low saturation
scalarPtr	Pointer to scalar data. Scale applied to input before summation.
scalarFormat	Data format for the scalar

Return Value

0 Success

1 Error and CPIS_errorno set to originating error.

Description

The sum API performs sum of the elements in a block of data. Each of the input samples is scaled before the summation. The size of the input block that is used for summation is specified by the procBlockSize field in the CPIS_BaseParms structure. The SUM API effectively breaks the input data into sub-blocks each of size as specified by the procBlockSize. Summation is performed for each sub-block and the resulting sum is placed in the output buffer. Thus, the number of values written in the output buffer will equal the number of subBlocks the input buffer is split into. To find out the sum of the entire input buffer, the resulting partial sub-block sums should be added up.

Constraints

- The API supports CPIS_16BIT, CPIS_U16BIT, CPIS_8BIT and CPIS_U8BIT data format for the input data. For the output, CPIS_8BIT and CPIS_16BIT are supported.
- base. procBlockSize.width × base. procBlockSize.height < MAX_SUM_BLOCKSIZE
  The symbol is defined in vicplib.h. The MAX_SUM_BLOCKSIZE is defined in terms of bytes. Thus, when using a data format of CPIS_16BIT, base.procBlockSize.width × base. procBlockSize.height < MAX SUM_BLOCKSIZE/2.
- The base. procBlockSize.height must be < 256.
- The base.procBlockSize.width and base. procBlockSize.height must be a multiple of 2.

Performance

For CPIS_16BIT. For input buffer size of 1360x600 and processing buffer size of 136x30.

Setup Time	~90000 CPU Clocks
Processing Time	~1080000 CPU Clocks

For this function, most of the processing time is due to EDMA overhead as the VICP is doing very little computation in this simple algorithm. If necessary you can customize the implementation of this function by chaining a few more VICP computation functions without incurring any extra performance drop. Refer to *VICP Computation Unit Library and VICP Scheduling Unit Library for DM6446, DM6441, DM647, and DM648* (SPRUGN1) for details about this topic.

3.3.21 CPIS_sumCFA

CPIS_sumCFA ***Sums the Elements in a Block of Data Following CFA Pattern***

Syntax

```
Int32 CPIS_sumCFA (
    CPIS_Handle       *handle,
    CPIS_BaseParms    *base,
    CPIS_SumCFAParms  *params,
    CPIS_ExecType     execType
);
```

Arguments

CPIS_Handle	**handle*	Handle returned in the case of asynchronous execution. In case of synchronous execution, NULL is returned.
CPIS_BaseParms	**base*	Base parameters to specify location, size of buffers
CPIS_SumCFAParms	**params*	The specific parameters for the sumCFA API are shown in Table 15.
CPIS_ExecType	*execType*	Execution type: synchronous or asynchronous

```
typedef struct {
    Uint16      qShift;

    Int32       sat_high;
    Int32       sat_high_set;
    Int32       sat_low;
    Int32       sat_low_set;

    Int16       * scalarPtr;
    CPIS_Format scalarFormat;

} CPIS_SumCFAParms;
```

Table 15. CPIS_sumCFA API Parameters

Parameter	Description
qshift	Q format or number of bits to downshift after processing
sat_high_set	Value to set the output with if high saturation
sat_low	Low value to check for saturation
sat_low_set	Value to set the output with if low saturation
scalarPtr	Pointer to scalar data. Scale applied to input before summation.
scalarFormat	Data format for the scalar

Return Value

0 Success

1 Error and CPIS_errorno set to originating error.

Description

The sumCFA API performs sum of the elements in a block of data. A partial sum is obtained for each element of a 2x2 tile (CFA Pattern). Each of the input samples is scaled before the summation. The size of the input block that is used for summation is specified by the procBlockSize field in the CPIS_BaseParms structure. The SUMCFA API effectively breaks the input data into sub-blocks each of size as specified by the procBlockSize. Summation is performed for each sub-block, for each CFA element and the resulting sums (four Values) are placed in the output buffer. The order of appearance in the output is:

1. Sum of the top left component of the tiles
2. Sum of the top right component of the tiles
3. Sum of the bottom left component of the tiles
4. Sum of the bottom right component of the tiles

Thus, the number of values written in the output buffer will equal four times the number of subBlocks the input buffer is split into. To find out the sum of the entire input buffer, the resulting partial sub-block sums should be added up.

Constraints

- The API supports CPIS_16BIT, CPIS_U16BIT, CPIS_8BIT and CPIS_U8BIT data format for the input data. For the output, CPIS_8BIT and CPIS_16BIT are supported.
- base. procBlockSize.width × base. procBlockSize.height < MAX_SUMCFA_BLOCKSIZE

  The symbol is defined in vicplib.h. The MAX_ SUMCFA _BLOCKSIZE is defined in terms of bytes. Thus, when using data format of CPIS_16BIT, base.procBlockSize.width × base. procBlockSize.height < MAX_ SMUCFA _BLOCKSIZE/2.
- The base. procBlockSize.height and base. procBlockSize.width must be <= 512
- The base.procBlockSize.width and base. procBlockSize.height must be a multiple of 2

Performance

For CPIS_16BIT. For input buffer size of 1360x600 and processing buffer size of 136x30.

Setup Time	~80000 CPU Clocks
Processing Time	~1080000 CPU Clocks

3.3.22 CPIS_table_lookup

CPIS_table_lookup ***Performs Look Up Operation for the Input Buffer Data***

Syntax

Int32 **CPIS_table_lookup** (

CPIS_Handle	**handle*,
CPIS_BaseParms	**base*,
CPIS_LUTParms	**params*,
CPIS_ExecType	*execType*

);

Arguments

CPIS_Handle	**handle*	Handle returned in the case of asynchronous execution. In case of synchronous execution, NULL is returned.
CPIS_BaseParms	**base*	Base parameters to specify location, size of buffers
CPIS_LUTParms	**params*	The specific parameters for the table lookup API are shown in Table 16.
CPIS_ExecType	*execType*	Execution type: synchronous or asynchronous

```
typedef struct {
    Uint16      qShift;

    Int32       sat_high;
    Int32       sat_high_set;
    Int32       sat_low;
    Int32       sat_low_set;

    Int16       * lutPtr;
    CPIS_Format lutFormat;
    Int16       numLUT;
    Int16       LUTSize;

    CPIS_Format scalarFormat;

} CPIS_LUTParms;
```

Table 16. CPIS_table_lookup API Parameters

Parameter	Description
qshift	Q format or number of bits to downshift after processing
sat_high_set	Value to set the output with if high saturation
sat_low	Low value to check for saturation
sat_low_set	Value to set the output with if low saturation
lutPtr	Pointer to LUT data.
lutFormat	Data format for the lookup table
numLUT	Number of interleaved lookup tables (1, 2 or 4)
LUTsize	Size for LUT data. Maximum size MAX_LUT_SIZE bytes.

Return Value

0 Success

1 Error and CPIS_errorno set to originating error.

Description

The LUT API performs look up operation for the data provided in the input buffer. The input array contains the indexes of the output elements in the lookup table. This API will simply fetch the corresponding elements in the lookup table and write them in the output array. It is possible to perform 1, 2 or 4 independent lookups. Number of LUTs is specified by the numLUT field in the params structure. Below is the layout of the input array and table array for different values of numLUT:

- **numLUT=1**: There is a single table in the lookup table array and all elements of the input array are indexes in this unique table.
- **numLUT=2**: There are two tables interleaved in the lookup table array. If those tables are called T1 and T2, then the memory layout of the lookup table array pointed by lutPtr must look like the following, where ti_j is the jth element of table Tj:
  - If lutFormat is 8 bit:

```
lutPtr [ ]={
t1_1, t1_2, t1_3, t1_4, t1_5, t1_6, t1_7, t1_8,
t2_1, t2_2, t2_3, t2_4, t2_5, t2_6, t2_7, t2_8,
t1_9, t1_10, t1_11, t1_12, t1_13, t1_14, t1_15, t1_16,
t2_9, t2_10, t2_11, t2_12, t2_13, t2_14, t2_15, t2_16,
...
}
```

  - If lutFormat is 16 bit:

```
lutPtr [ ]={
t1_1, t1_2, t1_3, t1_4,
t2_1, t2_2, t2_3, t2_4,
t1_5, t1_6, t1_7, t1_8,
t2_5, t2_6, t2_7, t2_8,
...
}
```

  The input array contains indexes into the lookup tables, arranged in a cyclic manner, where Ii_j an index of element num j in LUT Ti:

  Input_data[ ]={I1_1, I2_2, I1_3, I2_4, I1_5, I2_6, I1_7, …} (1)

- **numLUT=4**: There are four tables interleaved in the Lookup table array. If we call T1, T2 , T3, T4 , those four tables, then the memory layout of the lookup table pointed by lutPtr must look like the following, where ti_j is the jth element of table Tj:
  - If lutFormat is 8 bit

```
lutPtr [ ]={
t1_1, t1_2, t1_3, t1_4,
t2_1, t2_2, t2_3, t2_4,
t3_1, t3_2, t3_3, t3_4,
t4_1, t4_2, t4_3, t4_4,
t3_5, t3_6, t3_7, t3_8,
t4_5, t4_6, t4_7, t4_8,
...
}
```

  - If lutFormat is 16 bit:

```
lutPtr [ ]={
t1_1, t1_2,
t2_1, t2_2,
t3_1, t3_2,
t4_1, t4_2,
...
}
```

  The input array contains indexes into the lookup tables, arranged in a cyclic manner, where Ii_j an index of element num j in LUT Ti:

  Input_data[ ]={I1_1, I2_2, I1_3, I2_4, I1_5, I2_6, I1_7, …}

  Each input data point is rounded and saturated prior to lookup. Negative indexing is permitted when the input data array's elements are signed. In this case, the lookup table's base address points to the element whose index is 0. The natural C equivalent code provided for the LUT API only implements numLUT=1 case.

Constraints

- The API supports CPIS_16BIT, CPIS_U16BIT, CPIS_8BIT and CPIS_U8BIT data format for the input data and the LUT data. For the output, CPIS_8BIT and CPIS_16BIT are supported.
- base. procBlockSize.width × base. procBlockSize.height < MAX_LUT_BLOCKSIZE
  The symbol is defined in vicplib.h. The MAX_ LUT _BLOCKSIZE is defined in terms of bytes. Thus, when using the data format of CPIS_16BIT, base.procBlockSize.width × base. procBlockSize.height < MAX_ LUT _BLOCKSIZE/2.
- The LUTSize represents the total size of the LUT data (and not for individual LUT). The size should be less than MAX_LUT_SIZE.
- The numLUT can take values of 1,2 or 4.
- The base.procBlockSize.width and base. procBlockSize.height must be a multiple of 4.

Performance

For CPIS_16BIT. For input buffer size of 1280x320 and processing buffer size of 10x10.

Setup Time	~97000 CPU Clocks
Processing Time	~980000 CPU Clocks

For this function, most of the processing time is due to EDMA overhead as the VICP is doing very little computation in this simple algorithm. If necessary you can customize the implementation of this function by chaining a few more VICP computation functions without incurring any extra performance drop. Refer to *VICP Computation Unit Library and VICP Scheduling Unit Library for DM6446, DM6441, DM647, and DM648* (SPRUGN1) for details about this topic.

3.3.23 CPIS_medianFilterRow

CPIS_medianFilterRow *Performs Median Filtering Along Input Buffer Rows*

Syntax

```
Int32 CPIS_medianFilterRow (

    CPIS_Handle              *handle,
    CPIS_BaseParms           *base,
    CPIS_MedianFilterRowParms *params,
    CPIS_ExecType            execType
    );
```

Arguments

CPIS_Handle	*handle	Handle returned in the case of asynchronous execution. In case of synchronous execution, NULL is returned.
CPIS_BaseParms	*base	Base parameters to specify location, size of buffers
CPIS_MedianFilterRowParms	*params	The specific parameters for the median filter row API are shown in Table 17.
CPIS_ExecType	execType	Execution type: synchronous or asynchronous

```
typedef struct {
    Uint16 qShift;

    Int32  sat_high;
    Int32  sat_high_set;
    Int32  sat_low;
    Int32  sat_low_set;

    Int16  median_size;

} CPIS_MedianFilterRowParms;
```

Table 17. CPIS_medianFilterRow API Parameters

Parameter	Description
qshift	Q format or number of bits to downshift after processing
sat_high	High value to check for saturation
sat_high_set	Value to set the output with if high saturation
sat_low	Low value to check for saturation
sat_low_set	Value to set the output with if low saturation
median_size	Median Size: 3 or 5

Return Value

0 Success

1 Error and CPIS_errorno set to originating error.

Description

This function performs a median filtering operation along the rows of an input buffer. The size of the median filter can be 3-tap or 5-tap. At each output location, the median of the values in a window of size three or five (depending on the median filter size) is written to the output. Like other filtering operations, a border of one pixel for a 3-tap filter and a border of two pixels for a 5-tap filter must be present on the input data to obtain the correct output size.

Constraints

- The API supports CPIS_16BIT, CPIS_U16BIT, CPIS_8BIT and CPIS_U8BIT data format for the input data. For the output, CPIS_8BIT and CPIS_16BIT are supported.
- (base. procBlockSize.width + params.median_size - 1) × base. procBlockSize.height < MAX_MEDIANFILTER_ROW_BLOCKSIZE

  The symbol is defined in vicplib.h. The MAX_ MEDIANFILTER_ROW _BLOCKSIZE is defined in terms of bytes. Thus, when using the data format of CPIS_16BIT, (base.procBlockSize.width + params.median_size - 1) × base. procBlockSize.height < MAX_ MEDIANFILTER_ROW _BLOCKSIZE/2.

Performance

For CPIS_16BIT and median filter for 5 taps, for input buffer size of 960x320 and processing buffer size of 96x32

Setup Time	~80000 CPU Clocks
Processing Time	~790000 CPU Clocks

3.3.24 CPIS_medianFilterCol

CPIS_medianFilterCol *Performs Median Filtering Along Input Buffer Columns*

Syntax

```
Int32 CPIS_medianFilterCol (

    CPIS_Handle              *handle,
    CPIS_BaseParms           *base,
    CPIS_MedianFilterColParms *params,
    CPIS_ExecType            execType
    );
```

Arguments

CPIS_Handle	*handle	Handle returned in the case of asynchronous execution. In case of synchronous execution, NULL is returned.
CPIS_BaseParms	*base	Base parameters to specify location, size of buffers
CPIS_MedianFilterColParms	*params	The specific parameters for the median filter column API are shown in Table 18.
CPIS_ExecType	execType	Execution type: synchronous or asynchronous

```
typedef struct {
    Uint16 qShift;

    Int32  sat_high;
    Int32  sat_high_set;
    Int32  sat_low;
    Int32  sat_low_set;

    Int16  median_size;

} CPIS_MedianFilterColParms;
```

Table 18. CPIS_medianFilterCol API Parameters

Parameter	Description
qshift	Q format or number of bits to downshift after processing
sat_high	High value to check for saturation
sat_high_set	Value to set the output with if high saturation
sat_low	Low value to check for saturation
sat_low_set	Value to set the output with if low saturation
median_size	Median Size: 3 or 5

Return Value

0 Success

1 Error and CPIS_errorno set to originating error.

Description

This function performs a median filtering operation along the cols of an input buffer. The size of the median filter can be 3-tap or 5-tap. At each output location, the median of the values in a window of size three or five (depending on the median filter size) is written to the output. Like other filtering operations, a border of one pixel for a 3-tap filter and a border of two pixels for a 5-tap filter must be present on the input data to obtain the correct output size.

Constraints

- The API supports CPIS_16BIT, CPIS_U16BIT, CPIS_8BIT and CPIS_U8BIT data format for the input data. For the output, CPIS_8BIT and CPIS_16BIT are supported.
- base. procBlockSize.width × (base. procBlockSize.height + params.median_size - 1) < MAX_MEDIANFILTER_COL_BLOCKSIZE

  The symbol is defined in vicplib.h. The MAX_ MEDIANFILTER_COL_BLOCKSIZE is defined in terms of bytes. Thus, when using the data format of CPIS_16BIT, base.procBlockSize.width × (base. procBlockSize.height + params.median_size - 1) < MAX_ MEDIANFILTER_COL _BLOCKSIZE/2.

Performance

For CPIS_16BIT and median filter for 5 taps, for input buffer size of 960x320 and processing buffer size of 96x32:

Setup Time	~80000 CPU Clocks
Processing Time	~807000 CPU Clocks

3.3.25 CPIS_filter

CPIS_filter ***Performs FIR Filtering Operation***

Syntax

```
Int32 CPIS_filter (
    CPIS_Handle     *handle,
    CPIS_BaseParms  *base,
    CPIS_FilterParms *params,
    CPIS_ExecType   execType
);
```

Arguments

CPIS_Handle	**handle*	Handle returned in the case of asynchronous execution. In case of synchronous execution, NULL is returned.
CPIS_BaseParms	**base*	Base parameters to specify location, size of buffers
CPIS_FilterParms	**params*	The specific parameters for the FIR filtering API are shown in Table 19.
CPIS_ExecType	*execType*	Execution type: synchronous or asynchronous

```
typedef struct {
    Uint16      qShift;

    Int32       sat_high;
    Int32       sat_high_set;
    Int32       sat_low;
    Int32       sat_low_set;

    Int16       coeff_width;
    Int16       coeff_height;
    Int16       * coeffPtr;
    CPIS_Format coeffFormat;

} CPIS_FilterParms;
```

Table 19. CPIS_filter API Parameters

Parameter	Description
qshift	Q format or number of bits to downshift after processing
sat_high	High value to check for saturation
sat_high_set	Value to set the output with if high saturation
sat_low	Low value to check for saturation
sat_low_set	Value to set the output with if low saturation
coeff_width	Width of the coefficient buffer
coeff_height	Height of the coefficient buffer
coeffPtr	Pointer to the coefficient data
coeffFormat	Data format of the coefficient data

Return Value

0 Success

1 Error and CPIS_errorno set to originating error.

Description

This function can perform one of the following FIR filtering operations on the input data:

- 2-D filtering
- 1-D row filtering
- 1-D column filtering

For the case of 1-D row filtering, set the coeff_width to 1, and for 1-D column filtering, set coeff_height to 1. To ensure that the boundary pixels are correctly calculated, the input frame must be padded with border pixel data. If the dimensions of the output frame is WIDTH × HEIGHT then the dimensions of the input frame must at least be (WIDTH + coeff_width -1) × (HEIGHT + coeff_height -1).

The filtering is performed by using correlation instead of convolution operation. 2-D correlation is related to 2-D convolution by a 180 degrees rotation of the coefficient matrix.

Constraints

- The API supports CPIS_16BIT, CPIS_U16BIT, CPIS_8BIT and CPIS_U8BIT data format for the input data and the filter coefficients. For the output, CPIS_8BIT and CPIS_16BIT are supported.
- (base. procBlockSize.width + params.coeff_width - 1) × (base. procBlockSize.height + param.coeff_height - 1) < MAX_FILTER_BLOCKSIZE

  The symbol is defined in vicplib.h. The MAX_ FILTER_BLOCKSIZE is defined in terms of bytes. Thus, when using the data format of CPIS_16BIT, (base. procBlockSize.width + params.coeff_width - 1) × (base. procBlockSize.height + params.coeff_height - 1) < MAX_ FILTER_BLOCKSIZE/2.
- The base. procBlockSize.width must be a multiple of 8.
- The procBlockSize.height must be <= 256

Performance

For CPIS_16BIT, filter size of 16x8, for input buffer size of 960*280 and processing buffer size of 96*28:

Setup Time	~80000 CPU Clocks
Processing Time	~8800000 CPU Clocks

For this function, most of the processing time is due to EDMA overhead as the VICP is doing very little computation in this simple algorithm. If necessary you can customize the implementation of this function by chaining a few more VICP computation functions without incurring any extra performance drop. Refer to *VICP Computation Unit Library and VICP Scheduling Unit Library for DM6446, DM6441, DM647, and DM648* (SPRUGN1) for details about this topic.

3.3.26 CPIS_RGBPack

CPIS_RGBPack ***Packs R, G, B Separated Planes into 16 bpp RGB555 or RGB565 Data***

Syntax

Int32 **CPIS_RGBPack** (

CPIS_Handle	**handle*,
CPIS_BaseParms	**base*,
CPIS_RGBPackParms	**params*,
CPIS_ExecType	*execType*

);

Arguments

CPIS_Handle	**handle*	Handle returned in the case of asynchronous execution. In case of synchronous execution, NULL is returned.
CPIS_BaseParms	**base*	Base parameters to specify location, size of buffers
CPIS_RGBPackParms	**params*	The specific parameters for the RGBPack API.
CPIS_ExecType	*execType*	Execution type: synchronous or asynchronous

```
typedef struct {
    /* No user param is supported by the API */
    Int8 reserved;
} CPIS_RGBPackParms;
```

Return Value

0 Success

1 Error and CPIS_errorno set to originating error.

Description

This function packs R, G, B separated planes into 16 bpp RGB555 or RGB565 data. This function takes each of the 8-bit R, G, B planes of a bitmap image and pack the components together. Each element in the input R, G, B, planes is 8 bits wide. The output format is 16 bit per pixel and is either RGB555 or RGB565. The first pixel in the output is made of the first R, G, B values of the input planes, the second is pixel in the output is made of the second R, G, B value of the input planes and so on.

All the needed input for the API, input data pointers, output pointer, output data format, input size etc are provided using the base parameters. There are no user-specified API specific parameters.

Constraints

- The API supports CPIS_RGB_P data format for the input data and CPIS_RGB_555 or CPIS_RGB_565 for the output data.
- base. procBlockSize.width × base. procBlockSize.height < MAX_RGBPACK_BLOCKSIZE

  The symbol is defined in vicplib.h. The MAX_ RGBPACK_BLOCKSIZE is defined in terms of element size
- The base. procBlockSize.width must be a multiple of 8.

Performance

For input buffer size of 400*600 and processing buffer size of 40*40:

Setup Time	~100000 CPU Clocks
Processing Time	~900000 CPU Clocks

3.3.27 CPIS_RGBUnpack

CPIS_RGBUnpack ***Unpacks 16 bpp RGB555 or RGB565 Data into R, G, B Separated Planes***

Syntax

Int32 **CPIS_RGBunpack** (

CPIS_Handle	**handle*,
CPIS_BaseParms	**base*,
CPIS_RGBunpackParms	**params*,
CPIS_ExecType	*execType*

);

Arguments

CPIS_Handle	**handle*	Handle returned in the case of asynchronous execution. In case of synchronous execution, NULL is returned.
CPIS_BaseParms	**base*	Base parameters to specify location, size of buffers
CPIS_RGBPackParms	**params*	The specific parameters for the RGBUnpack API.
CPIS_ExecType	*execType*	Execution type: synchronous or asynchronous

```
typedef struct {
    /* No user param is supported by the API */
    Int8 reserved;
} CPIS_RGBUnpackParms;
```

Return Value

0 Success

1 Error and CPIS_errorno set to originating error.

Description

This function unpacks 16 bpp RGB555 or RGB565 data into R, G, B separated planes. This function takes 16 bits-per-pixel RGB format RGB555 or RGB565 as input and unpacks it into 3 color planes: R, G, B.

All the needed input for the API, input data pointer, output pointers, input data format, input size etc are provided using the base parameters. There are no user-specified API specific parameters.

Constraints

- The API supports CPIS_RGB_555 or CPIS_RGB_565 for the input data and CPIS_RGB_P data format for the output data.
- base. procBlockSize.width × base. procBlockSize.height < MAX_RGBUNPACK_BLOCKSIZE

  The symbol is defined in vicplib.h. The MAX_ RGBUNPACK_BLOCKSIZE is defined in terms of element size
- The base. procBlockSize.width must be a multiple of 8.

Performance

For input buffer size of 400*600 and processing buffer size of 40*40:

Setup Time	~100000 CPU Clocks
Processing Time	~900000 CPU Clocks

3.3.28 CPIS_blkAverage

CPIS_blkAverage ***Calculates the Average Value of the Elements in a Data Block***

Syntax

Int32 **CPIS_blkAverage** (

CPIS_Handle	**handle*,
CPIS_BaseParms	**base*,
CPIS_BlkAverageParms	**params*,
CPIS_ExecType	*execType*

);

Arguments

CPIS_Handle	**handle*	Handle returned in the case of asynchronous execution. In case of synchronous execution, NULL is returned.
CPIS_BaseParms	**base*	Base parameters to specify location, size of buffers
CPIS_BlkAverageParms	**params*	The specific parameters for the block averaging API are shown in Table 20.
CPIS_ExecType	*execType*	Execution type: synchronous or asynchronous

```
typedef struct {
    Uint16 qShift;

    Int32  sat_high;
    Int32  sat_high_set;
    Int32  sat_low;
    Int32  sat_low_set;

} CPIS_BlkAverageParms;
```

Table 20. CPIS_blkAverage API Parameters

Parameter	Description
qshift	Q format or number of bits to downshift after processing
sat_high	High value to check for saturation
sat_high_set	Value to set the output with if high saturation
sat_low	Low value to check for saturation
sat_low_set	Value to set the output with if low saturation

Return Value

0 Success

1 Error and CPIS_errorno set to originating error.

Description

The blkAverage API calculates the average value of the elements in a block of data. The kernel adds all the elements and then applies a user specified shift to the resulting sum. The size of the input block that is used for summation is specified by the procBlockSize field in the CPIS_BaseParms structure. The blkAverage API effectively breaks the input data into sub-blocks each of size as specified by the procBlockSize. Block Average is calculated for each sub-block and the resulting average value is placed in the output buffer. Thus, the number of values written in the output buffer will equal the number of subBlocks the input buffer is split into. Shift can be selected using the following equation:

$$qShift = \log2(base.\ procBlockSize.width \times base.\ procBlockSize.height) \quad (2)$$

Constraints

- The API supports CPIS_16BIT, CPIS_U16BIT, CPIS_8BIT and CPIS_U8BIT data format for the input data. For the output, CPIS_8BIT and CPIS_16BIT are supported.
- base. procBlockSize.width × base. procBlockSize.height < MAX_BLKAVERAGE_BLOCKSIZE

  The symbol is defined in vicplib.h. The MAX_ BLKAVERAGE_BLOCKSIZE is defined in terms of bytes. Thus, when using the data format of CPIS_16BIT, base.procBlockSize.width × base. procBlockSize.height < MAX_ BLKAVERAGE_BLOCKSIZE/2.
- The base. procBlockSize.width must be a multiple of 8 and <=2048.
- The procBlockSize.height must be <= 256

Performance

For CPIS_16BIT, for input buffer size of 2720*510 and processing buffer size of 136*30:

Setup Time	~80000 CPU Clocks
Processing Time	~1800000 CPU Clocks

3.3.29 CPIS_recursiveFilter

CPIS_recursiveFilter *Applies First Order Recursive Filtering on a 2-D Source Frame*

Syntax

```
Int32 CPIS_recursiveFilter (

    CPIS_Handle             *handle,
    CPIS_BaseParms          *base,
    CPIS_RecursiveFilterParms *params,
    CPIS_ExecType           execType
    );
```

Arguments

CPIS_Handle	*handle	Handle returned in the case of asynchronous execution. In case of synchronous execution, NULL is returned.
CPIS_BaseParms	*base	Base parameters to specify location, size of buffers
CPIS_RecursiveFilterParms	*params	The specific parameters for the recursive filtering API are shown in Table 21.
CPIS_ExecType	execType	Execution type: synchronous or asynchronous

```
typedef struct {
    CPIS_FilterDir direction;

    CPIS_FilterInitialMode initialMode;

    CPIS_Buffer initialValues;

    Uint16 alpha;

    Uint16 qShift;

    Int32  sat_high;
    Int32  sat_high_set;
    Int32  sat_low;
    Int32  sat_low_set;

} CPIS_RecursiveFilterParms;
```

Table 21. CPIS_recursiveFilter API Parameters

Parameter	Description	
direction	Propagation direction of the recursive filtering. Use enumeration type CPIS_FilterDir to set this parameter:	
	CPIS_TOP2BOTTOM	Filter is vertical and propagates from top row to bottom row
	CPIS_BOTTOM2TOP	Filter is vertical and propagates from bottom row to top row
	CPIS_LEFT2RIGHT	Filter is horizontal and propagates from left column to right column
	CPIS_RIGHT2LEFT	Filter is horizontal and propagates from right column to left column
initialMode	Sets how the initial state of the recursive filter is managed. Use enumeration type CPIS_FilterInitialMode to set this parameter:	
	CPIS_USE_BOUNDARY	Use boundary pixels as initial state of the recursive filter. When filter direction is:
		CPIS_TOP2BOTTOM: The boundary pixels equal to the ROI's top-most row
		CPIS_BOTTOM2TOP: The boundary pixels equal to the ROI's bottom-most row.
		CPIS_LEFT2RIGHT: The boundary pixels equal to the ROI's left-most column.
		CPIS_RIGHT2LEFT: The boundary pixels equal to the ROI's right-most column.
	CPIS_USE_PASSED_VALUES	Use values passed through the member initialValues as initial state of the recursive filter.

Table 21. CPIS_recursiveFilter API Parameters (continued)

Parameter	Description
initialValues	Member of type CPIS_Buffer. • When initialMode == CPIS_USE_PASSED_VALUES, this member supplies the location and stride of the initial values, whose type is the same as the source buffer. initialValues.ptr points to the first element of the initial values. initialValues.stride represents the stride in number of elements between each initial values. Set it to 1 if values are sequentially ordered. • When initialMode == CPIS_USE_BOUNDARY, this member is ignored.
alpha	Alpha coefficient
qshift	Q format or number of bits to downshift after processing
sat_high	High value to check for saturation
sat_high_set	Value to set the output with if high saturation
sat_low	Low value to check for saturation
sat_low_set	Value to set the output with if low saturation

Return Value

0 Success

-1 Error and CPIS_errorno set to originating error.

Description

This function applies 1st order recursive filtering on a 2-D source frame. The propagation direction can be either vertical: from top to bottom, bottom to top, or horizontal: from left right or right to left.

Mathematically the operation can be expressed as follow: if we let x[i, j] be the input array and y[i,j] the output array where i is the column index and j the row index, then for a given column C, each term y[C, j] is computed using the previous output y[C, j-1] and the present input x[C, j]:

$$y[C, j]= (1 – \alpha) * x[C, j] + \alpha * y[C, j – 1] \text{ , for } j > 0 \quad (3)$$

For top to bottom propagation, j would index the rows in downward fashion, meaning j=0 would represent the top-most row and j=1, the row just below it. For bottom to top propagation, j would index the rows in upward fashion, meaning j=0 would represent the bottom-most row and j=1, the row just above it. For left to right propagation, j=0 would represent the left-most column and j=1, the column just right to it. For left to right propagation, j=0 would represent the right-most column and j=1, the column just left to it.

To calculate the terms of the first row j=0, since previous values belonging to the previous row are not available, two options are offered:

- If initialMode== CPIS_USE_BOUNDARY, the initial values simply equal the first row j=0.
- If initialMode== CPIS_USE_PASSED_VALUES, the values pointed by initialValues.ptr are used. If the computation is initiated more than one time with CPIS_start(), the application must call the function CPIS_loadRecursiveFilterInitialValues() to load these initial values, every time before CPIS_start().

Constraints

- The API supports CPIS_16BIT, CPIS_U16BIT, CPIS_8BIT and CPIS_U8BIT data format for the input data. The output format must be set equal to the input format.
- base. procBlockSize.width × base. procBlockSize.height + 1 < 8192 for 8-bit source format or < 4096 for 16-bit source format.
- The base. procBlockSize.width must be a multiple of 8 and <=2048.

Performance Tips

For vertical filtering, base.procBlockSize.width should be as large as possible for optimal performance. If possible set base.procBlockSize.width= base. roiSize.width.

For horizontal filtering, base.procBlockSize.height should be as large as possible for optimal performance. If possible set base.procBlockSize.height= base. roiSize.height.

Performance

- For CPIS_TOP2BOTTOM or CPIS_BOTTOM2TOP direction, CPIS_U16BIT input format, input buffer size of 320*240 and processing buffer size of 64*60:

Setup Time	~196763 CPU Clocks
Processing Time	~193395 CPU Clocks

- For CPIS_TOP2BOTTOM or CPIS_BOTTOM2TOP direction, CPIS_U8BIT input format, input buffer size of 320*240 and processing buffer size of 320*10:

Setup Time	~203207 CPU Clocks
Processing Time	~120587 CPU Clocks

- For CPIS_LEFT2RIGHT or CPIS_RIGHT2LEFT direction, CPIS_U16BIT input format, input buffer size of 320*240 and processing buffer size of 320*10:

Setup Time	~210935 CPU Clocks
Processing Time	~492213 CPU Clocks

- For CPIS_LEFT2RIGHT or CPIS_RIGHT2LEFT direction, CPIS_U8BIT input format, input buffer size of 320*240 and processing buffer size of 320*10:

Setup Time	~210935 CPU Clocks
Processing Time	~492213 CPU Clocks

3.3.30 CPIS_loadRecursiveFilterInitialValues

CPIS_loadRecursiveFilterInitialValues *Loads Initial Values for Recursive Filter*

Syntax

```
Int32 CPIS_loadRecursiveFilterInitialValues (
    CPIS_Handle     *handle,
    void            *src,
    CPIS_Format     *format,
    Uint32          stride
);
```

Arguments

CPIS_Handle	**handle*	Handle corresponding to the filter operation previously initialized and for which we need to pass new initial values.
void	**src*	Pointer to first element making up the initial values
CPIS_Format	**format*	The format of the initial values. Can be CPIS_U8BIT, CPIS_8BIT, CPIS_U16BIT, or CPIS_16BIT.
Uint32	*stride*	Stride between elements, unit is whatever specified by the argument format.

Return Value

0 Success

-1 Error and CPIS_errorno set to originating error.

Description

This function is used after CPIS_recursiveFilter(…, …., CPIS_ASYNC, …) and before each call to CPIS_start(), to load the initial values.

3.3.31 CPIS_setRecursiveFilterAlphaCoef

CPIS_setRecursiveFilterAlphaCoef *Updates Alpha Coefficient Value*

Syntax

Int32 **CPIS_setRecursiveFilterAlphaCoef** (

CPIS_Handle	**handle*,
Uint16	*alpha*

);

Arguments

CPIS_Handle	**handle*	Handle corresponding to the recursive filter operation previously initialized and for which we need to pass new initial values.
Uint16	*alpha*	Pointer to first element making up the range function's table.

Return Value

0 Success

-1 Error and CPIS_errorno set to originating error.

Description

This function is used after CPIS_recursiveFilter(…, …., CPIS_ASYNC, …) and before each call to CPIS_start(), to update the alpha coefficient value, if the algorithm needs to have it changed.

3.3.32 CPIS_median2D

CPIS_median2D ***Calculates the Median Value of the Elements in a 2-D Data Block***

Syntax

```
Int32 CPIS_median2D (
    CPIS_Handle          *handle,
    CPIS_BaseParms       *base,
    CPIS_Median2DParms   *params,
    CPIS_ExecType        execType
);
```

Arguments

CPIS_Handle	**handle*	Handle returned in the case of asynchronous execution. In case of synchronous execution, NULL is returned.
CPIS_BaseParms	**base*	Base parameters to specify location, size of buffers
CPIS_Median2DParms	**params*	The specific parameters are shown in Table 22.
CPIS_ExecType	*execType*	Execution type: synchronous or asynchronous

```
typedef struct {
    Int16 filterWidth;
    Int16 filterHeight;
} CPIS_Median2DParms;
```

Table 22. CPIS_median2D API Parameters

Parameter	Description
filterWidth	Median filter width
filterHeight	Median filter height

Return Value

0 Success

-1 Error and CPIS_errorno set to originating error.

Description

This function performs a 2-D median filtering operation of all the data point in the input buffer. Currently only 3x3 median filter is supported. At each output location, the median of the values in a window of size 3x3 is written to the output. Like other filtering operations, a border of one pixel for 3x3 filter must be present on the input data to obtain the correct output size.

Constraints

- The API supports CPIS_16BIT, CPIS_U16BIT, CPIS_8BIT and CPIS_U8BIT data format for the input data. For the output, CPIS_8BIT and CPIS_16BIT are supported and must match the input. For example a combination of input data format of CPIS_16BIT and output data format of CPIS_8BIT would not be supported.
- (4 × base. procBlockSize.width + 2) × (base. procBlockSize.height + 2) < MAX_MEDIAN2D_BLOCKSIZE. The symbol is defined in vicplib.h. The MAX_MEDIAN2D_BLOCKSIZE is defined in terms of bytes. The function will return -1 and sets CPIS_errorno to CPIS_NOSUPPORTDIM_ERROR in case the constraint is not respected.

Performance

For CPIS_16BIT and median filter for 3 taps, for input buffer size of 320*240 and processing buffer size of 32*24:

Setup Time	~84070 CPU Clocks
Processing Time	~475406 CPU Clocks

3.3.33 CPIS_sobel

CPIS_sobel ***Calculates Horizontal and Vertical Gradient Approximations of Data***

Syntax

Int32 **CPIS_sobel** (

CPIS_Handle	**handle*,
CPIS_BaseParms	**base*,
CPIS_SobelParms	**params*,
CPIS_ExecType	*execType*

);

Arguments

CPIS_Handle	**handle*	Handle returned in the case of asynchronous execution. In case of synchronous execution, NULL is returned.
CPIS_BaseParms	**base*	Base parameters to specify location, size of buffers
CPIS_SobelParms	**params*	The specific parameters are shown in Table 23.
CPIS_ExecType	*execType*	Execution type: synchronous or asynchronous

```
typedef struct {
    Uint16 qShift;

    Int32 sat_high;
    Int32 sat_high_set;
    Int32 sat_low;
    Int32 sat_low_set;

} CPIS_SobelParms;
```

Table 23. CPIS_Sobel API Parameters

Parameter	Description
qshift	Q format or number of bits to downshift after processing
sat_high	High value to check for saturation
sat_high_set	Value to set the output with if high saturation
sat_low	Low value to check for saturation
sat_low_set	Value to set the output with if low saturation

Return Value

0 Success

-1 Error and CPIS_errorno set to originating error.

Description

This function calculates both the horizontal and the vertical gradient approximations on a frame of data. The horizontal and vertical components of each pixel's gradient are interleaved in the output. Kernel sizes are [3 x 3] and have the following coefficients:

$$\text{Coeff\_x} = \begin{matrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{matrix} \qquad \text{Coeff\_y} = \begin{matrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{matrix}$$

If both input and output data have the same bit-width, the recommended value for qShift is 3, because the absolute sums of filter coefficients are 8. Choosing the value of 3 guarantees that saturation will not occur, while rounding is minimal.

Constraints

- The API supports CPIS_16BIT, CPIS_U16BIT, CPIS_8BIT and CPIS_U8BIT data format for the input data. For the output, CPIS_8BIT and CPIS_16BIT are supported.
- (base. procBlockSize.width + 2) × (base. procBlockSize.height + 2) < MAX_SOBEL_BLOCKSIZE. The symbol is defined in vicplib.h. MAX_SOBEL_BLOCKSIZE is defined in terms of bytes. Thus, when using a data format of CPIS_16BIT, (base.procBlockSize.width + 2) × (base. procBlockSize.height + 2) < MAX_SOBEL_BLOCKSIZE /2
- base. procBlockSize.width must be multiple of 8.
- procBlockSize.height must be <= 256.

Performance

For CPIS_8BIT and frame size of 400*400:

Setup Time	~68500 CPU Clocks
Processing Time	~55700 CPU Clocks

For CPIS_16BIT and frame size of 400*400:

Setup Time	~68500 CPU Clocks
Processing Time	~84800 CPU Clocks

For this function, most of the processing time is due to EDMA overhead as the VICP is doing very little computation in this simple algorithm. If necessary you can customize the implementation of this function by chaining a few more VICP computation functions without incurring any extra performance drop. Refer to *VICP Computation Unit Library and VICP Scheduling Unit Library for DM6446, DM6441, DM647, and DM648* (SPRUGN1) for details about this topic.

3.3.34 CPIS_integralImage

CPIS_integralImage ***Calculates the Integral Image of a Frame***

Syntax

```
Int32 CPIS_integralImage (

    CPIS_Handle              *handle,
    CPIS_BaseParms           *base,
    CPIS_IntegralImageParms  *params,
    CPIS_ExecType            execType
    );
```

Arguments

CPIS_Handle	**handle*	Handle returned in the case of asynchronous execution. In case of synchronous execution, NULL is returned.
CPIS_BaseParms	**base*	Base parameters to specify location, size of buffers
CPIS_IntegralImageParms	**params*	The specific parameters are shown in Table 24.
CPIS_ExecType	*execType*	Execution type: synchronous or asynchronous

```
typedef struct {
    Uint16 *scratch;
    Uint16 stage;

} CPIS_IntegralImageParms;
```

Table 24. CPIS_integralImage API Parameters

Parameter	Description
*scratch	Points to a region in memory of size 2×2×base→roiSize.width × base→roiSize.height bytes
stage	Processing stage of the integral image to execute, can be 0 or 1

Return Value

0 Success

-1 Error and CPIS_errorno set to originating error.

Description

This function calculates the integral image of a frame. The input frame can be made of either 8 bits or 16 bits unsigned values. The output generated is always made of 32 bits unsigned values. The execution of the function must be carried out in two stages:

- Stage 0: The cumulative sum along the vertical direction is calculated for the input frame pointed by base→srcBuf[0[.ptr. The resulting frame is transposed and stored in the region pointed by params→scratch.
- Stage 1: The cumulative sum along the vertical direction is calculated for the frame pointed by params→scratch. The resulting frame is transposed and stored in the region pointed by base→dstBuf[0].ptr.

After both stage 0 and stage 1 are executed in order, the resulting frame pointed by base→dstBuf[0[.ptr correspond to the integral image of the frame pointed by base→srcBuf[0].ptr.

To execute stage 1 after stage 0, CPIS_integralImage() must be called two times with the argument params.stage set accordingly:

```
params.stage= 0;
CPIS_integralImage(
            &handle,
            &base,
            &params,
            CPIS_ASYNC
            );
CPIS_start(handle);
CPIS_wait(handle);
CPIS_delete(handle);

params.stage= 1;
CPIS_integralImage(
            &handle,
            &base,
            &params,
            CPIS_ASYNC
            );
CPIS_start(handle);
CPIS_wait(handle);
CPIS_delete(handle);
```

For this function, the caller does not have to set base.procBlockSize.width or base. procBlockSize.height. These values are internally set by the integral image function.

Constraints

The API supports CPIS_U16BIT and CPIS_U8BIT data format for the input data. For the output, CPIS_U32BIT is automatically forced by the function.

Performance

For CPIS_U8BIT or CPIS_U16BIT and frame size of 640*480 points:

Setup Time or stage 0	~157000 CPU Clocks
Processing Time for stage 0	~1890000 CPU Clocks
Setup Time or stage 1	~179000 CPU Clocks
Processing Time for stage 1	~2120000 CPU Clocks

3.3.35 CPIS_pyramid

CPIS_pyramid ***Calculates the Values of the Next Gaussian Pyramid Level***

Syntax

Int32 **CPIS_pyramid** (

CPIS_Handle	**handle*,
CPIS_BaseParms	**base*,
CPIS_PyramidParms	**params*,
CPIS_ExecType	*execType*

);

Arguments

CPIS_Handle	**handle*	Handle returned in the case of asynchronous execution. In case of synchronous execution, NULL is returned.
CPIS_BaseParms	**base*	Base parameters to specify location, size of buffers
CPIS_PyramidParms	**params*	The specific parameters are shown in Table 25.
CPIS_ExecType	*execType*	Execution type: synchronous or asynchronous

```
typedef struct {
    Uint16 gaussFilterSize;

    Uint16 qShift;

    Int32  sat_high;
    Int32  sat_high_set;
    Int32  sat_low;
    Int32  sat_low_set;

} CPIS_PyramidParms;
```

Table 25. CPIS_pyramid API Parameters

Parameter	Description
gaussFilterSize	Size of the Gaussian filter: 3, 5 or 7
qshift	Q format or number of bits to downshift after processing
sat_high	High value to check for saturation
sat_high_set	Value to set the output with if high saturation
sat_low	Low value to check for saturation
sat_low_set	Value to set the output with if low saturation

Return Value

0 Success

-1 Error and CPIS_errorno set to originating error.

Description

This function calculates the values of the next level in the Gaussian pyramid using the Gaussian kernel corresponding to the filter size params.gaussFilterSize:

$$\text{coeff\_3x3} = \begin{matrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{matrix}$$

$$\text{coeff\_5x5} = \begin{matrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{matrix}$$

Width of input accessed for this operation is (base. procBlockSize.width + params.gaussFilterSize - 1) and height of input accessed is (base. procBlockSize. height + params.gaussFilterSize - 1).

If both input and output data have the same bit-width, the recommended value for round_shift is 4, 8, or 12 for respective filter size of 3x3, 5x5, or 7x7. Choosing these values guarantees that saturation will not occur, while rounding is minimal.

Constraints

- The API supports CPIS_16BIT, CPIS_U16BIT, CPIS_8BIT and CPIS_U8BIT data format for the input data and the filter coefficients. For the output, CPIS_8BIT and CPIS_16BIT are supported.
- (base. procBlockSize.width + params.gaussFilterSize - 1) × (base. procBlockSize.height + params.gaussFilterSize - 1) < MAX_PYRAMID_BLOCKSIZE. The symbol is defined in vicplib.h. The MAX_PYRAMID_BLOCKSIZE is defined in terms of bytes. Thus, when using a data format of CPIS_16BIT, (base.procBlockSize.width + params.gaussFilterSize - 1) × (base. procBlockSize.height + params.gaussFilterSize - 1) < MAX_PYRAMID_BLOCKSIZE /2
- base. procBlockSize.width should be multiple of 8.
- procBlockSize.height must be <= 256

Performance

For CPIS_8BIT, params.gaussFilterSize = 3, for input buffer size of 960*380 and processing buffer size of 96*38:

Setup Time	~64000 CPU Clocks
Processing Time	~436000 CPU Clocks

For CPIS_16BIT, params.gaussFilterSize=3 , for input buffer size of 960*380 and processing buffer size of 96*38:

Setup Time	~64000 CPU Clocks
Processing Time	~682000 CPU Clocks

3.3.36 CPIS_affineTransform

CPIS_affineTransform *Applies Affine Transformation on an Input Region*

Syntax

Int32 **CPIS_affineTransform** (

CPIS_Handle	**handle*,
CPIS_BaseParms	**base*,
CPIS_AffineTransformParms	**params*,
CPIS_ExecType	*execType*

);

Arguments

CPIS_Handle	**handle*	Handle returned in the case of asynchronous execution. In case of synchronous execution, NULL is returned.
CPIS_BaseParms	**base*	Base parameters to specify location, size of buffers
CPIS_AffineTransformParms	**params*	The specific parameters are shown in Table 26.
CPIS_ExecType	*execType*	Execution type: synchronous or asynchronous

```
typedef struct {

    Uint8 *privateVars;
    Uint8 *scratch;
    Uint32 scratchSize;
    Int16 skipOutside;

    Int16 m0;
    Int16 m1;
    Int16 m2;
    Int16 m3;

    Int16 tx;
    Int16 ty;

    Int16 m0inv;
    Int16 m1inv;
    Int16 m2inv;
    Int16 m3inv;

    Int16 txinv;
    Int16 tyinv;

    Uint16 qShift;

    Int32 sat_high;
    Int32 sat_high_set;
    Int32 sat_low;
    Int32 sat_low_set;

} CPIS_AffineTransformParms;
```

Table 26. CPIS_affineTransform API Parameters

Parameter	Description
*privateVars	Pointer to a region of size CPIS_AFFINE_PRIVATE_VAR_SIZE
*scratch	Pointer to a scratch memory. Size of which can be obtained at runtime after executing CPIS_affineTransformGetSize().
scratchSize	Size of scratch buffer in byte. Depends on the affine transformation settings. Can be obtained at runtime after executing CPIS_affineTransformGetSize().
skipOutside	Enable it to optimize performance. Skip the processing of the region that would fall outside of the ROI by replacing outside data with '0' value. See API description for more details.
m0, m1, m2, m3	Coefficients of the forward 2x2 linear transformation matrix [m0 m1; m2 m3]. They are 16-bit value so if your original numbers are float numbers, they must be converted by multiplying with (1<<qShift) and rounded to the nearest 16-bit integer.
tx, ty	Coefficients of the forward translation vectors [tx ty]. They are 16-bit value so if your original numbers are float numbers, they must be converted by multiplying with (1<<qShift) and rounded to the nearest 16-bit integer.
m0inv, m1inv, m2inv, m3inv	Coefficients of the inverse of the 2x2 linear transformation matrix [m0inv m1inv; m2inv m3inv]= inverse([m0 m1; m2 m3]). They are 16-bit values so if your original numbers are float numbers, they must be converted by multiplying with (1<<qShift) and rounded to the nearest 16-bit integer.
txinv, tyinv	Coefficients of the inverse translation vectors [txinv tyinv]= -[tx ty]. They are generally equal to -tx and -ty.
qShift	Q format or number of fractional bits in the coefficients m0, m1, m2, m3, tx, ty; and m0inv, m1inv, m2inv, m3inv, txinv, tyinv. The greater qShift is, the more precise the bilinear interpolation gets but the smaller the ROI must be in order to ensure that saturation does not occur. In general, a value of qShift= 3 should work. Above that, artifacts can appear.
sat_high	High value to check for saturation
sat_high_set	Value to set the output with if high saturation
sat_low	Low value to check for saturation
sat_low_set	Value to set the output with if low saturation

Return Value

0 Success

-1 Error and CPIS_errorno set to originating error.

Description

This function applies affine transformation on the input region of interest belonging to a monochrome 8-bit or 16-bit image. The type of interpolation used is bilinear.

Any combination of rotation, resizing, translation in the 2-D plane can be mathematically expressed in form of an affine transformation. An affine transformation is separated into a linear transformation, described by a 2x2 matrix and a translation described by a 2x1 vector.

Both the forward transformation and the inverse transformation's matrices must be provided as the function doesn't calculate the inverse. Since the VICP performs all the calculation in 16-bit arithmetic, the application must take care of converting the matrices' floating point numbers into 16-bits Q-format numbers. The following example code shows how an application would set up the matrix's parameters for an affine transform composed of:

- Rotation of arbitrary angle. The center of rotation being the middle of the input ROI.
- Resize of scale scale_x along the horizontal axis and scale_y along the vertical axis.
- Translation of tx along the horizontal axis and ty along the vertical axis.

```
params.m0= (Int16)(round(scale_x * cos(angle) * exp2((double)params.qShift)));
params.m1= (Int16)(round(-scale_y * sin(angle) * exp2((double)params.qShift)));
params.m2= (Int16)(round(scale_x * sin(angle) * exp2((double)params.qShift)));
params.m3= (Int16)(round(scale_y * cos(angle) * exp2((double)params.qShift)));
params.tx= (Int16)(round(t_x * exp2((double)params.qShift)));
params.ty= (Int16)(round(t_y * exp2((double)params.qShift)));

params.m0inv= (Int16)(round((cos(angle)/scale_x) * exp2((double)params.qShift)));
params.m1inv= (Int16)(round((sin(angle)/scale_x) * exp2((double)params.qShift)));
params.m2inv= (Int16)(round((-sin(angle)/scale_y) * exp2((double)params.qShift)));
```

```
params.m3inv= (Int16)(round((cos(angle)/scale_y) * exp2((double)params.qShift)));
params.txinv= (Int16)(round(-t_x * exp2((double)params.qShift)));
params.tyinv= (Int16)(round(-t_y * exp2((double)params.qShift)));
```

The parameters params.qShift is used to magnify the floating numbers in order to keep some fractional bits before rounding occurs. params.qShift with be later used by the VICP to get rid of the fractional part. The greater qShift is, the more precise the calculations gets but the smaller the ROI must be in order to ensure that saturation doesn't occur. In general a value of qShift= 3 should work. Above that, artifacts can appear.

The center of rotation is always the middle of the input region of interest. This is not an issue since it is possible to mathematically express a rotation with an arbitrary center into a combination of centered rotation followed or preceded by a translation.

The input region of interest is defined by:

- Its upper left corner's pointer base.srcBuf[0].ptr
- Its stride base.srcBuf[0].stride
- Its data format base.srcFormat[0]
- Its dimensions base.roiSize.width and base.roiSize.height

As such, the input region of interest is always a rectangle whose edges are either horizontal or vertical.

After affine transformation, the input region of interest would map to an output region of interest, still rectangular but whose edge can be tilted if rotation was involved in the transformation. Figure 4 and Figure 5 illustrate the spatial arrangement of the region of interests before and after an affine transformation.

Figure 4. Input ROI

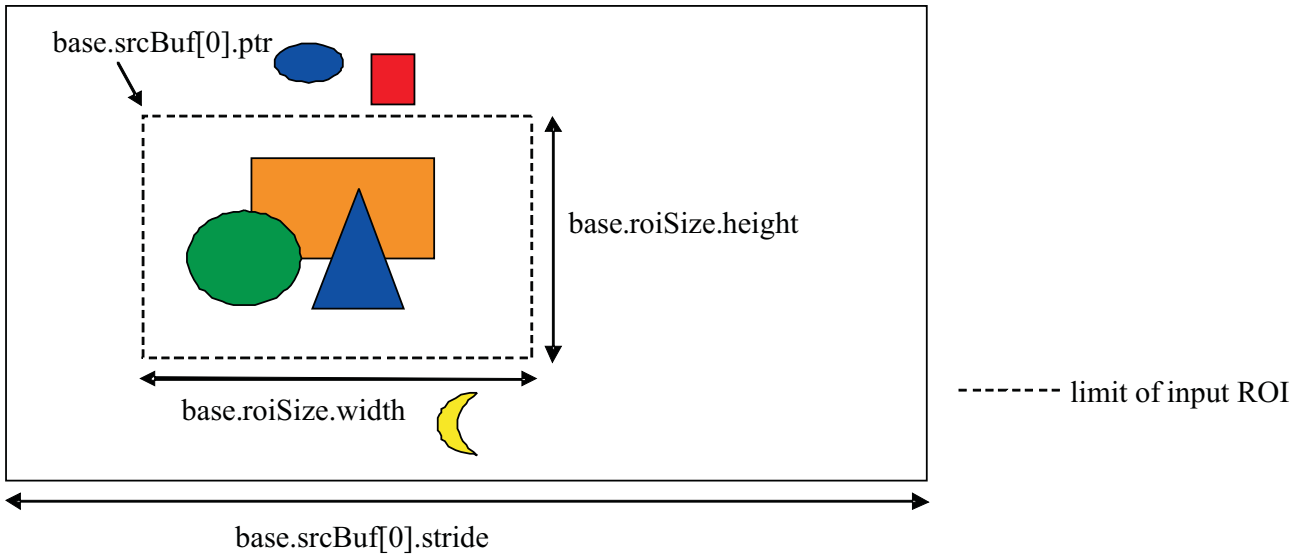
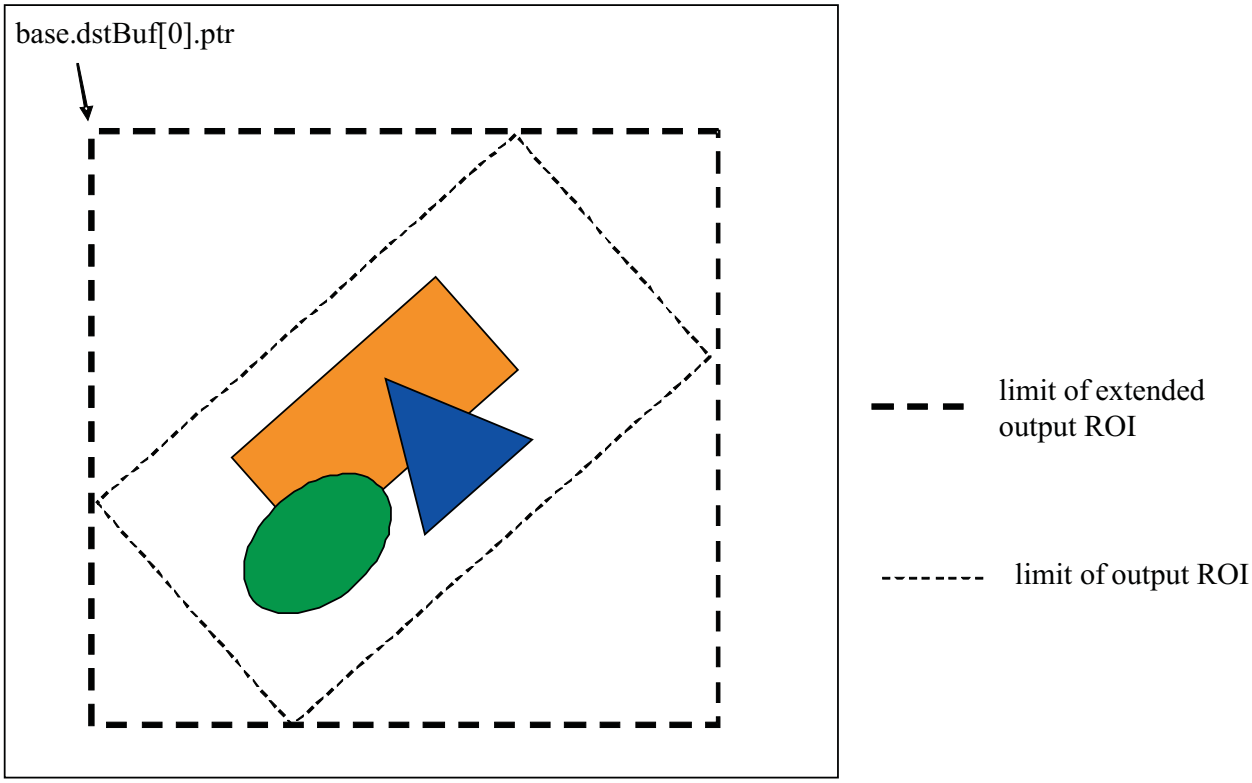


Figure 5. Output ROI



Since the VICP is only able to write data in a raster scan format, more data than desired is written to the output buffer. The rectangle that encloses the output of the affine transform is called the extended output ROI. The application can choose to output data belonging to the entire extended output ROI or data belonging to the output ROI only. The choice is made by setting params→skipOutside. If set to 1, the regions that falls outside of the output ROI are filled with zero, resulting in Figure 6. If set to 0, these same regions are nevertheless processed resulting to something similar to Figure 7. Setting params→skipOutside to 1 will always speed up the processing.

Figure 6. Output ROI Displayed With params→skipOutside= 1

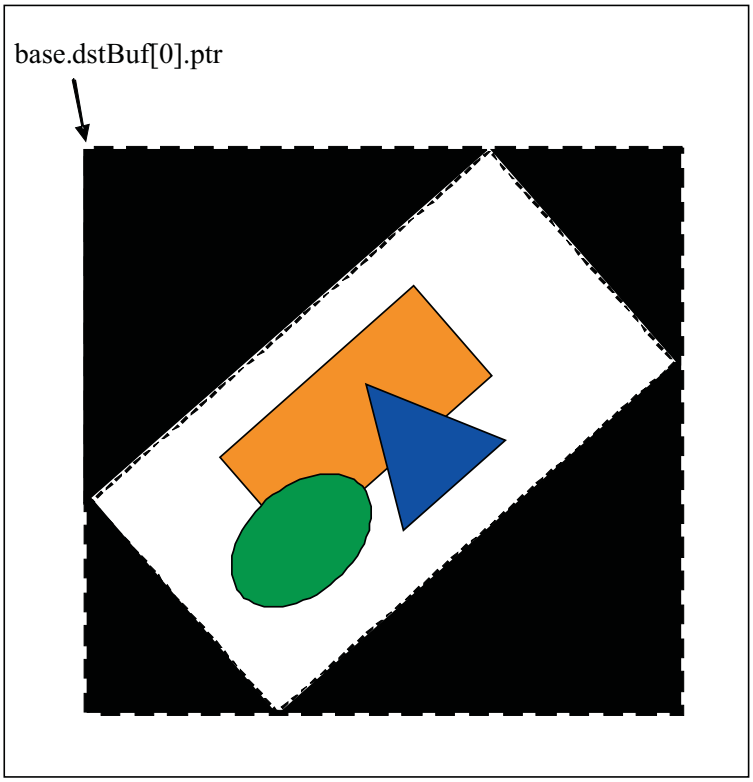
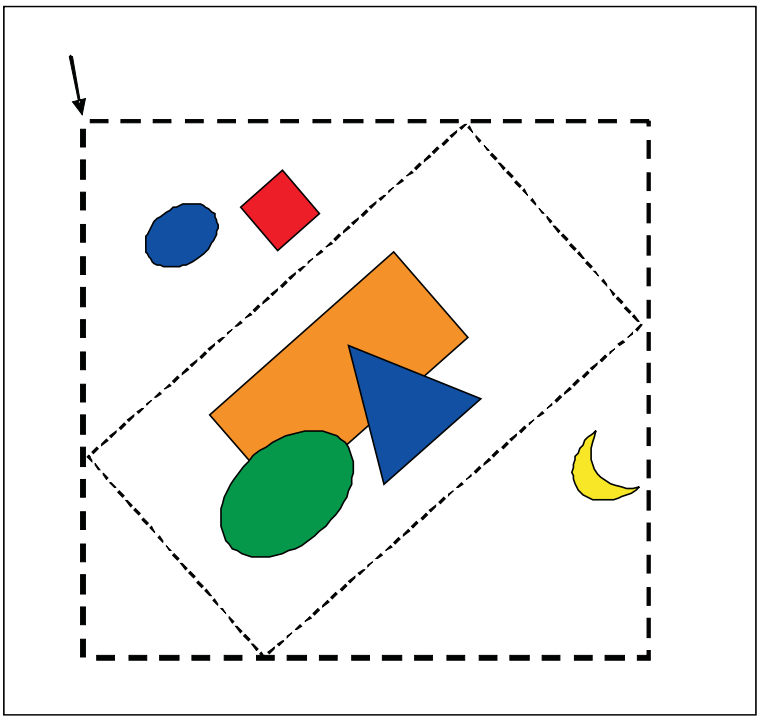


Figure 7. Output ROI Displayed With params→skipOutside= 0



Observe that when params→skipOutside=0, the extended ROI contains part of the image frames that where outside of the original input ROI. In figure 6, this is illustrates by the presence of the blue ellipse, red square and yellow crescent moon. Although these objects were not part of input ROI, the affine transform was applied to them because params→skipOutside= 0.

When there is no rotation involved, the extended output ROI matches the output ROI.

To help the application implementer to determine in advance the size of the extended output ROI, the function CPIS_affineTransformGetSize(CPIS_BaseParms *base, CPIS_AffineTransformParms *params, CPIS_AffineTransformOutputROI *outputROI) can be used. This function uses the affine transformation mathematical descriptions provided in the structure params, as well as the ROI width and height provided by base→roiSize.width and base→roiSize.height to fill the structure outputROI with the following useful information:

- outputROI→width and outputROI→height are the width and height of the extended output ROI
- outputROI→stride is the stride of the extended output ROI. Due to some internal constrain, outputROI→width does not necessarily equal to outputROI→stride. In general, it is smaller, resulting in some padding data appended at the end of each row.
- outputROI→blockWidth and output→blockHeight are the dimensions of the processing blocks. These dimensions have to be later assigned to base.procBlockSize.width and base.procBlockSize.height before calling CPIS_affineTransform().

Constraints

- The API supports CPIS_U16BIT and CPIS_U8BIT for input and output data format.
- base.procBlockSize.width and base.procBlockSize.height should be obtained by calling CPIS_affineTransformGetSize().

Performance

Setup time varies between 175000 and 215000 CPU cycles.

Processing time varies between 20 CPU cycles/point and 38 CPU cycles/point. Best processing time is achieved for params.skipOutsidet= 1.

3.3.37 CPIS_affineTransformGetSize

CPIS_affineTransformGetSize *Determines Size of Output ROI or Scratch Buffer*

Syntax

Int32 **CPIS_affineTransformGetSize** (

CPIS_BaseParms	**base*,
CPIS_AffineTransformParms	**params*,
CPIS_AffineTransformOutputROI	**outputROI*

);

Arguments

CPIS_BaseParms	**base*	Base parameters to specify location, size of buffers
CPIS_AffineTransformParms	**params*	The specific parameters are shown in Table 26.
CPIS_AffineTransformOutputROI	**outputROI*	See Table 27.

Table 27. CPIS_AffineTransformOutputROI Parameters

Parameter	Description
width	Width of the output ROI
height	Height of the output ROI
stride	Stride of the output ROI
blockWidth	Processing block width
blockHeight	Processing block height

Return Value

0 Success

-1 Error and CPIS_errorno set to originating error.

Description

This function should be used by the application to:

- Determine the size of the extended output ROI. to serve this purpose, CPIS_affineTransformGetSize() fills the structure outputROI passed as an input argument. This structure of type CPIS_AffineTransformOutputROI is given below:

```
typedef struct {
    Uint16 width;
    Uint16 height;
    Uint16 stride;
    Uint16 blockWidth;
    Uint16 blockHeight;
} CPIS_AffineTransformOutputROI;
```

- Determine the size of the scratch buffer that needs to be allocated before CPIS_affineTransform() is called. Indeed CPIS_affineTransformGetSize() fills params→scratchSize so the application can allocate the buffer.

CPIS_affineTransformGetSize() needs the following input parameters to be initialized prior to the call:

- base→srcFormat()
- base→roiSize.width
- base→roiSize.height
- params→privateVars. Must point to an area of size CPIS_AFFINE_PRIVATE_VAR_SIZE.

- params→m0, params→m1, params→m2, params→m3, params→tx, params→ty, params→m0inv, params→m1inv, params→m2inv, params→m3inv, params→m0inv, params→m1inv, params→m2inv, params→m3inv, params→txinv, params→tyinv, params→qShift.

  Please refer to the description of CPIS_affineTransform () in Section 3.3.36 for more details on the usage of CPIS_affineTransformGetSize().

Performance 28000 CPU clocks

3.3.38 CPIS_cfa

CPIS_cfa ***Performs Debayering Using a Color Filter Array Interpolation Technique***

Syntax

Int32 **CPIS_cfa** (

CPIS_Handle **handle*,
CPIS_BaseParms **base*,
CPIS_CFAParms **params*,
CPIS_ExecType *execType*
);

Arguments

CPIS_Handle	**handle*	Handle returned in the case of asynchronous execution. In case of synchronous execution, NULL is returned.
CPIS_BaseParms	**base*	Base parameters to specify location, size of buffers
CPIS_CFAParms	**params*	The specific parameters are shown in Table 28.
CPIS_ExecType	*execType*	Execution type: synchronous or asynchronous

```
typedef struct {

    Uint16 qShift;

    Int32 sat_high;
    Int32 sat_high_set;
    Int32 sat_low;
    Int32 sat_low_set;

} CPIS_CFAParms;
```

Table 28. CPIS_cfa API Parameters

Parameter	Description
qShift	Q format or number of bits to downshift after processing.
sat_high	High value to check for saturation
sat_high_set	Value to set the output with if high saturation
sat_low	Low value to check for saturation
sat_low_set	Value to set the output with if low saturation

Return Value

0 Success

-1 Error and CPIS_errorno set to originating error.

Description

This function performs debayering using color filter array interpolation technique. The interpolation used is a simple bilinear interpolation through a 3x3 filter. Three data planes are produced corresponding to the red, green, blue colors, respectively pointed by base.dstBuf[0].ptr, base.dstBuf[1].ptr and base.dstBuf[2].ptr.

Constraints

- The API supports CPIS_16BIT, CPIS_U16BIT data format for the input data. For the output, CPIS_U16BIT and CPIS_16BIT are supported in the current version.
- The input ROI must include a border of 1 pixel to ensure that the filter computes valid data at the boundary of the image. Consequently, applying CPIS_cfa() on a ROI of size WIDTH × HEIGHT will produce R, G, B planes of size (WIDTH-2) × (HEIGHT-2).
- base. procBlockSize.width × base. procBlockSize.height < MAX_CFA_BLOCKSIZE. The symbol is defined in vicplib.h. The MAX_CFA_BLOCKSIZE is defined in terms of bytes. The function returns -1 and sets CPIS_errorno to CPIS_NOSUPPORTDIM_ERROR in case the constraint is not respected.

Performance

For CPIS_16BIT, for input buffer size of 748*480 and processing buffer size of 24*40:

Setup Time	~95330 CPU Clocks
Processing Time	~3319861 CPU Clocks or 9.3 CPU cycles/pixel

3.3.39 CPIS_sad

CPIS_sad ***Template Matching Using Sum of Absolute Difference***

Syntax

Int32 **CPIS_sad** (

CPIS_Handle	**handle*,
CPIS_BaseParms	**base*,
CPIS_SadParms	**params*,
CPIS_ExecType	*execType*

);

Arguments

CPIS_Handle	**handle*	Handle returned in the case of asynchronous execution. In case of synchronous execution, NULL is returned.
CPIS_BaseParms	**base*	Base parameters to specify location, size of buffers
CPIS_SadParms	**params*	The specific parameters are shown in Table 29.
CPIS_ExecType	*execType*	Execution type: synchronous or asynchronous

Table 29. CPIS_sad API Parameters

Parameter	Description
qShift	Q format or number of bits to downshift after processing.
sat_high	High value to check for saturation
sat_high_set	Value to set the output with if high saturation
sat_low	Low value to check for saturation
sat_low_set	Value to set the output with if low saturation
templatePtr	Pointer to template. Generally input parameter except when loadTemplate= 0. In this case CPIS_sad() initializes templatePtr with the location in VICP coefficient memory where the template should be written.
templateFormat	Format of each data element composing the template. Can be CPIS_U8BIT, CPIS_8BIT, CPIS_16BIT, CPIS_U16BIT.
loadTemplate	Control pre-loading of template into coefficient memory. 0 Disable. The application is responsible of loading the template into the coefficient memory at the location returned in templatePtr by CPIS_sad(). 1 Enable
templateRoiSize	Specify the width of the template's region of interest in number of data points and the height of the template's ROI in number of lines.
templateSize	Specify the line stride of the template in number of data points and the number of line.
templateStartOfst	Offset from location pointed by templatePtr to the first data point of the template

Return Value

0 Success

-1 Error and CPIS_errorno set to originating error.

Description

This function performs template matching using sum of absolute difference between a template and a search image. The resulting output image is a map of SAD values corresponding to all center positions of the template with respect to the search image. The position corresponding to the lowest SAD value would be where the template matches best the search image.

The technique used is similar to a FIR filtering, except that the pixel operation used is absolute difference instead of multiply.

If the dimensions of the output frame is WIDTH × HEIGHT then the dimensions of the input frame must at least be (WIDTH + templateRoiSize.width – 1) × (HEIGHT + templateRoiSize.height – 1).

The template must be pre-loaded into the VICP coefficient memory prior to the start of the processing. In case of synchronous call (parameter execType= CPIS_SYNC), the processing occurs within the function call whereas in case of asynchronous call (parameter execType= CPIS_ASYNC), the processing happens with a call to CPIS_start(). For synchronous call, set params.loadTemplate=1 and params.templatePtr accordingly to have CPIS_sad() load the template automatically. For asynchronous call, the application has the option to disable the automatic loading by setting params.loadTemplate to 0 and use other means of transfers such as EDMA, IDMA, DSP. The location to where the template must be transferred to is returned by CPIS_sad() into params.templatePtr. Hence, when params.loadTemplate= 0, params.templatePtr is an input parameter whereas when params.loadTemplate=1, it becomes an output parameter, which is passed to the application by CPIS_sad(). Consequently, when params.loadTemplate= 0, the application can load the template only after CPIS_sad() returns.

The memory footprint of the template is calculated as follow: templateSize= params.templateSize.width × params.templateSize.Height and must be smaller than MAX_SAD_TEMPLATESIZE, which is 32kb (size of the VICP coefficient memory). However, the effective ROI of the template used for the sad operation is of size templateRoiSize= params.templateRoiSize.width x params.templateRoiSize.height and must be smaller than the input block size inBlockSize= (base. procBlockSize.width + params.templateRoiSize.width -1) × (base. procBlockSize.height + params.templateRoiSize.height -1) , which itself is smaller than MAX_SAD_BLOCKSIZE, which is 8kb.

The following constraints must be simultaneously respected:

- templateRoiSize < inBlockSize ≤ MAX_SAD_BLOCKSIZE
  with:
  templateRoiSize= params.templateRoiSize.width × params.templateRoiSize.height
  inBlockSize= (base. procBlockSize.width + params.templateRoiSize.Width -1) × (base. procBlockSize.height + params.templateRoiSize.height -1) × max(sizeof(INPUT_TYPE), sizeof(OUTPUT_TYPE))
- templateRoiSize ≤ templateSize ≤ MAX_SAD_TEMPLATESIZE
  with:
  templateSize= params.templateSize.width × params.templateSize.height

From these two constraints, arise three use cases for CPIS_sad().

Case 1 templateRoiSize = templateSize < inBlockSize ≤ MAX_SAD_BLOCKSIZE

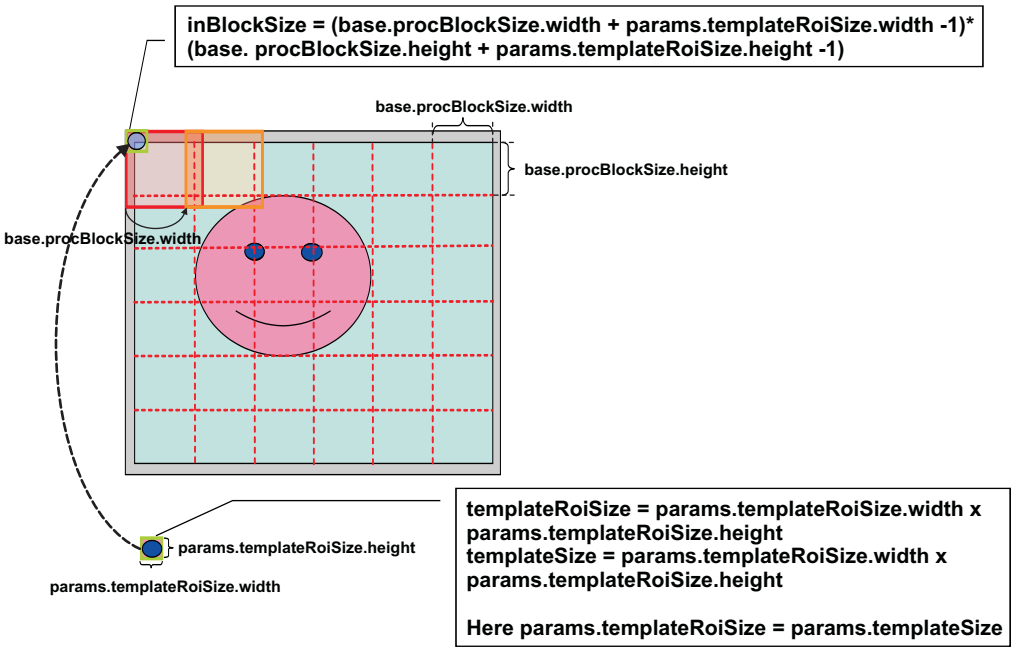
This is the most simple use case to deal with and it is illustrated in Figure 8.

Like in a typical block-based processing, the ROI of the search image is divided in processing blocks of dimensions base.procBlockSize.width x base.procBlockSize.height. Each input block's size is bigger than the processing block size in order to account for the block's border pixels and is fetched by the VICP from left to right and top to bottom, in a raster-scan fashion. Also each input block has an overlapping region with its neighbor blocks.

Since the SAD calculation with the template occurs within each input block fetched, we see why we must have templateRoiSize ≤ inBlocksSize.

In this simple case, the SAD map between the blue ellipse template and the search image can be generated in one processing shot by CPIS_sad(). To generate an output map of dimensions WIDTH × HEIGHT points, the input search image must be (WIDTH + templateRoiSize.width – 1) × (HEIGHT + templateRoiSize.height – 1). Thus, the grey border in Figure 8.

Figure 8. Case #1 of CPIS_sad() Usage



Case 2

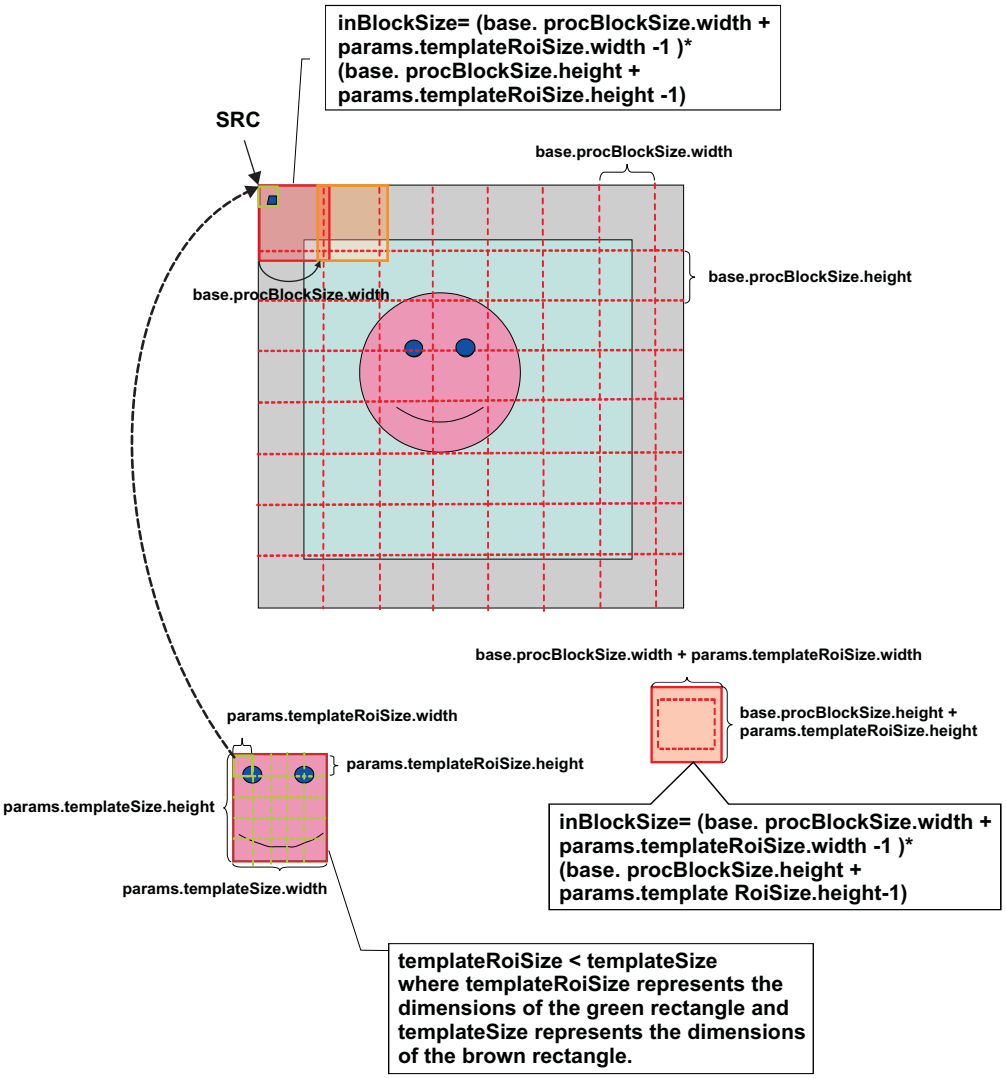
templateRoiSize < inBlockSize ≤ MAX_SAD_BLOCKSIZE < templateSize ≤ MAX_SAD_TEMPLATESIZE

The case illustrated in Figure 9 is more complicated to handle.

Since templateSize > inBlockSize the entire template is too big to be matched to any subregion #defineof the input processing block. To work around this issue, we need to implement a divide and conquer approach by performing the template matching between sub-blocks of the template and the input block. In Figure 9, the template is divided into a grid of subblocks (green colored blocks). A sub-block is of size templateRoiSize params.templateRoiSize.width × params.templateSize.height and a call to CPIS_sad() matches it with all possible positions within the input search image.

To match the entire template several CPIS_sad() executions are needed, each one matching a different sub-block of the template. To minimize overhead, the function CPIS_setSadTemplateOffset() is used to quickly change the offset of the template's ROI to the the next sub-block, avoiding the need to call CPIS_sad() each time. Also the function CPIS_updateSrcDstPtr() must be used to update the source pointer to the search image, as it moves along with the sub-block. In this case, the entire search image must be of dimensions (WIDTH + params.templateSize.width – 1) × (HEIGHT + params.templateSize.height – 1) to produce an output map of dimensions WIDTH × HEIGHT. This is different from the previous case for which the search image's dimensions needed to be (WIDTH + templateRoiSize.width – 1) × (HEIGHT + templateRoiSize.height – 1).

Figure 9. Case #2 of CPIS_sad() Usage



In conclusion, only one call to CPIS_sad() is needed in order to pass the template dimensions and load the template into the VICP internal memory. The rest of the template matching is carried out by repeatedly calling CPIS_setSadTemplateOffset(),CPIS_updateSrcDstPtr(), CPIS_reset(),CPIS_CPIS_start() and CPIS_wait(). One iteration produces the SAD map corresponding to one template's sub-block. To get the final SAD map, all SAD maps must be added together.

The following example code illustrates the whole process of handling a template whose size is bigger than the input block.

```
/* templateSize= TEMPLATE_WIDTH*TEMPLATE_HEIGHT= 14450 < MAX_SAD_TEMPLATESIZE= 32kb */
#define TEMPLATE_WIDTH 170
#define TEMPLATE_HEIGHT 85

/* The templateRoiSize= TEMPLATE_ROI_WIDTH * TEMPLATE_ROI_HEIGHT */
#define TEMPLATE_ROI_WIDTH 17
#define TEMPLATE_ROI_HEIGHT 17

#define PROC_BLOCK_WIDTH 48 /* need to be multiple of 8 for max perf */
#define PROC_BLOCK_HEIGHT 34

/* We must have INPUT_BLOCK_WIDTH * INPUT_BLOCK_HEIGHT * 2 < MAX_SAD_BLOCKSIZE= 8192
```

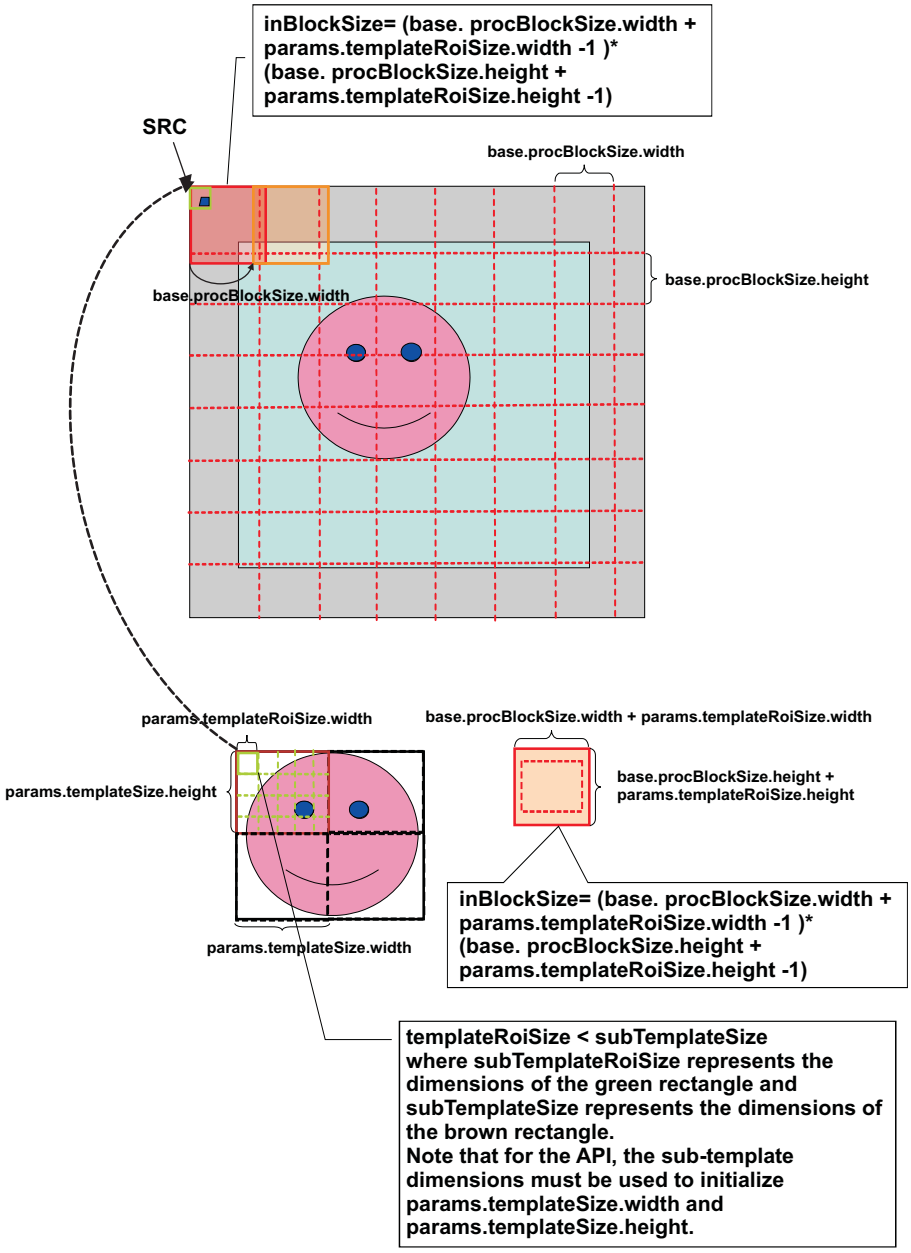
```
The multiply factor 2 is required because our output SAD map will be in CPIS_U16BIT */
#define INPUT_BLOCK_WIDTH (PROC_BLOCK_WIDTH + TEMPLATE_ROI_WIDTH -1)
#define INPUT_BLOCK_HEIGHT (PROC_BLOCK_HEIGHT + TEMPLATE_ROI_HEIGHT -1)

#define OUT_WIDTH 640
#define OUT_HEIGHT 480

#define IN_WIDTH (OUT_WIDTH + TEMPLATE_WIDTH -1)
#define IN_HEIGHT (OUT_HEIGHT + TEMPLATE_HEIGHT -1)

/* Variables declaration
CPIS_BaseParms base;
CPIS_BaseParms baseRef;
CPIS_SadParms params;
CPIS_Handle handle;
In32 row, col, r, w;

/* finalSAD points to the final SAD map */
Uint16 *finalSAD= (Uint16*)FINAL_SAD_PTR;

/* Initialize SAD parameters */
/* TEMPLATE_PTR points to the template bitmap in external memory */
params.templatePtr= (Uint8*)TEMPLATE_PTR;
params.templateFormat= CPIS_U8BIT;
params.templateSize.width= TEMPLATE_WIDTH;
params.templateSize.height= TEMPLATE_HEIGHT;
params.templateRoiSize.width= TEMPLATE_ROI_WIDTH;
params.templateRoiSize.height= TEMPLATE_ROI_HEIGHT;
params.templateStartOfst= 0;
params.loadTemplate= 1;
params.sat_high= 65535;
params.sat_high_set= 65535;
params.sat_low= 0;
params.sat_low_set= 0;
params.qShift= 0;

/* Initialize the search image's pointer and dimension */
base.srcBuf[0].ptr= (Uint8*)SRC;
base.srcBuf[0].stride= IN_WIDTH;
base.srcFormat[0]= CPIS_U8BIT;

/* Initialize the destination map's pointer and dimension */
base.dstBuf[0].ptr= (Uint8*)DST;
base.dstBuf[0].stride= OUT_WIDTH;
base.dstFormat[0]= CPIS_U16BIT;

/* Initialize the ROI dimension */
base.roiSize.width= OUT_WIDTH;
base.roiSize.height= OUT_HEIGHT;

/* Initialize the processing block's dimensions */
base.procBlockSize.width= PROC_BLOCK_WIDTH;
base.procBlockSize.height= PROC_BLOCK_HEIGHT;

/* Call to CPIS_sad() in asynchronous mode just to set-up the function and load the template into the
VICP coefficient memory */

if (CPIS_sad(
            &handle,
            &base,
            &params,
            CPIS_ASYNC
            )== -1) {
            printf("\nCPIS_sad() error %d\n", CPIS_errno);
            exit(-1);
       };
```

```
/* The final SAD map is made of the sum of all of the SAD maps generated within the loop. At the
beginning it gets zeroed-out */

finalSAD= (Uint16*)FINAL_SAD_PTR;

for(r=0;r< base.roiSize.height; r++)
  for((c=0;c< base.roiSize.width; c++)
    *finalSAD++ += 0;

/* Iterate through each sub-block of the template. Each iteration produces a SAD map that represents the
SAD values of all the template's matched positions with the search image */
for (row= 0; row < params.templateSize.height; row += params.templateRoiSize.height)
  for (col= 0; col < params.templateSize.width; col += params.templateRoiSize.width) {

        Uint16 *prevDstPtr;

/* Update templateStartOfst to point to the next sub-block in the template */
params.templateStartOfst= col + row * params.templateSize.width;
CPIS_setSadTemplateOffset(handle, params.templateStartOfst);

/* Update the pointer to the search image so it "follows" the template */
base.srcBuf[0].ptr= (Uint8*)SRC + col + row * base.srcBuf[0].stride;

/* Save pointer to previous SAD map , will need it to add to the final map */
prevDstPtr= (Uint16*)base.dstBuf[0].ptr;

/* Update the destination pointer for the next SAD map, note that the increment is in number of bytes,
thus the multiply factor of 2 since the output type is CPIS_UBIT 16*/
base.dstBuf[0].ptr+= base.roiSize.width* base.roiSize.height * 2;

/* call CPIS_updateSrcDstPtr so the new source and destination pointer gets updated into the framework*/
CPIS_updateSrcDstPtr(handle, &base),

/* Mandatory before CPIS_start() */
CPIS_reset(handle)

    /* Execute SAD processing to generate the SAD map corresponding to one sub-block of the template */
CPIS_start(handle);
```

Case 3 The entire template cannot fit within MAX_SAD_TEMPLATESIZE.

In the previous case, the entire template could fit in the VICP coefficient memory. Only the processing input block size was too small to allow the completion of the template matching in a single pass. However for this case, the template itself is too large to wholly fit in the VICP coefficient memory. The previous divide and conquer approach must be further “divided” to deal with this issue. Indeed the template is divided into sub-templates (represented by the brown rectangle in the figure below). Each of these sub-templates is processed independently as a regular template using the same approach as in case #2. Consequently, we will add another loop around the previous loop. The application will have to upload the next sub-template into the VICP coefficient memory at each iteration of this new loop. A difference with the previous case is that the loadTemplate parameter is disabled so the application can read the returned templatePtr, which becomes the write location of sub-templates.

Figure 10. Case #3 of CPIS_sad() Usage



The following example code illustrates the whole process of handling a template whose size is both bigger than the input block and the VICP coefficient memory.

```
/* templateSize= TEMPLATE_WIDTH*TEMPLATE_HEIGHT= 57800 > MAX_SAD_TEMPLATESIZE= 32kb */
#define TEMPLATE_WIDTH 340
#define TEMPLATE_HEIGHT 170

/* subTemplateSize= SUBTEMPLATE_WIDTH*SUBTEMPLATE_HEIGHT= 14450 < MAX_SAD_TEMPLATESIZE= 32kb */
#define SUBTEMPLATE_WIDTH 170
#define SUBTEMPLATE_HEIGHT 85

/* The templateRoiSize= TEMPLATE_ROI_WIDTH * TEMPLATE_ROI_HEIGHT */
#define TEMPLATE_ROI_WIDTH 17 /* Must be odd */
#define TEMPLATE_ROI_HEIGHT 17 /* Must be odd */

#define PROC_BLOCK_WIDTH 72 /* need to be multiple of 8 for max perf */
```

```
#define PROC_BLOCK_HEIGHT 24

/* We must have INPUT_BLOCK_WIDTH * INPUT_BLOCK_HEIGHT * 2 < MAX_SAD_BLOCKSIZE= 8192 */
#define INPUT_BLOCK_WIDTH (PROC_BLOCK_WIDTH + TEMPLATE_ROI_WIDTH -1)
#define INPUT_BLOCK_HEIGHT (PROC_BLOCK_HEIGHT + TEMPLATE_ROI_HEIGHT -1)

#define OUT_WIDTH 640
#define OUT_HEIGHT 480

#define IN_WIDTH (OUT_WIDTH + TEMPLATE_WIDTH -1)
#define IN_HEIGHT (OUT_HEIGHT + TEMPLATE_HEIGHT -1)

/* Variables declaration
CPIS_BaseParms base;
CPIS_BaseParms baseRef;
CPIS_SadParms params;
CPIS_Handle handle;
In32 row, col, r, w;

/* finalSAD points to the final SAD map */
Uint16 *finalSAD= (Uint16*)FINAL_SAD_PTR;

/* Initialize SAD parameters */
/* TEMPLATE_PTR points to the template bitmap in external memory */
params.templatePtr= (Uint8*)TEMPLATE_PTR;
params.templateFormat= CPIS_U8BIT;
params.templateSize.width= SUBTEMPLATE_WIDTH;
params.templateSize.height= SUBTEMPLATE_HEIGHT;
params.templateRoiSize.width= TEMPLATE_ROI_WIDTH;
params.templateRoiSize.height= TEMPLATE_ROI_HEIGHT;
params.templateStartOfst= 0;
params.loadTemplate= 0;
params.sat_high= 65535;
params.sat_high_set= 65535;
params.sat_low= 0;
params.sat_low_set= 0;
params.qShift= 0;

/* Initialize the search image's pointer and dimension */
base.srcBuf[0].ptr= (Uint8*)SRC;
base.srcBuf[0].stride= IN_WIDTH;
base.srcFormat[0]= CPIS_U8BIT;

/* Initialize the destination map's pointer and dimension */
base.dstBuf[0].ptr= (Uint8*)DST;
base.dstBuf[0].stride= OUT_WIDTH;
base.dstFormat[0]= CPIS_U16BIT;

/* Initialize the ROI dimension */
base.roiSize.width= OUT_WIDTH;
base.roiSize.height= OUT_HEIGHT;

/* Initialize the processing block's dimensions */
base.procBlockSize.width= PROC_BLOCK_WIDTH;
base.procBlockSize.height= PROC_BLOCK_HEIGHT;

/* Call to CPIS_sad() in asynchronous mode just to set-up the function and load the template into the
VICP coefficient memory */

if (CPIS_sad(
            &handle,
            &base,
            &params,
            CPIS_ASYNC
            )== -1) {
            printf("\nCPIS_sad() error %d\n", CPIS_errno);
```

```
            exit(-1);
        };

src= SRC;

/* The final SAD map is made of the sum of all of the SAD maps generated within the loop. At the
beginning it gets zeroed-out */

finalSAD= (Uint16*)FINAL_SAD_PTR;

for(r=0;r< base.roiSize.height; r++)
  for((c=0;c< base.roiSize.width; c++)
    *finalSAD++ += 0;

for (templateRow= 0; templateRow < TEMPLATE_HEIGHT; templateRow+= SUBTEMPLATE_HEIGHT)
for (templateCol= 0; templateCol < TEMPLATE_WIDTH; templateCol+= SUBTEMPLATE_WIDTH){

/* Copy the sub-template from DDR into the VICP coefficient memory
 Use a customized memcpy function that accepts line stride input, set to TEMPLATE_WIDTH here.
*/
  customMemcpy(params.templatePtr, TEMPLATE_PTR + templateCol +    templateRow*SUBTEMPLATE_WIDTH,
SUBTEMPLATE_WIDTH, SUBTEMPLATE_HEIGHT , TEMPLATE_WIDTH);

  src+= templateCol + templateRow * base.srcBuf[0].stride;

/* Iterate through each sub-block of the template. Each iteration produces a SAD map that represents the
SAD values of all the template's matched positions with the search image */
  for (row= 0; row < params.templateSize.height; row += params.templateRoiSize.height)
    for (col= 0; col < params.templateSize.width; col += params.templateRoiSize.width){

        Uint16 *prevDstPtr;

/* Update templateStartOfst to point to the next sub-block in the template */
params.templateStartOfst= col + row * params.templateSize.width;
CPIS_setSadTemplateOffset(handle, params.templateStartOfst);

/* Update the pointer to the search image so it "follows" the template */
base.srcBuf[0].ptr= (Uint8*)src + col + row * base.srcBuf[0].stride;

/* Save pointer to previous SAD map , will need it to add to the final map */
prevDstPtr= (Uint16*)base.dstBuf[0].ptr;

/* Update the destination pointer for the next SAD map, note that the increment is in number of bytes,
thus the multiply factor of 2 since the output type is CPIS_UBIT 16*/
base.dstBuf[0].ptr+= base.roiSize.width* base.roiSize.height * 2;

/* call CPIS_updateSrcDstPtr so the new source and destination pointer gets updated into the framework*/
CPIS_updateSrcDstPtr(handle, &base),

/* Mandatory before CPIS_start() */
CPIS_reset(handle)

    /* Execute SAD processing to generate the SAD map corresponding to one sub-block of the template */
CPIS_start(handle);

/* The final SAD map is made of the sum of all of the SAD maps generated within the loop. While VICP is
calculating the current SAD map, we can use the CPU to calculate the cumulative sum of the SAD maps. */

        if (row!= 0 || col!= 0) {

         finalSAD= (Uint16*)FINAL_SAD_PTR;

      for(r=0;r< base.roiSize.height; r++)
        for((c=0;c< base.roiSize.width; c++)
          *finalSAD++ += *prevDstPtr++
    }
```

```
CPIS_wait(handle);
}

  /* Do not forget to add the last SAD map to the final SAD map */
  finalSAD= (Uint16*)FINAL_SAD_PTR;

  for(r=0;r< base.roiSize.height; r++)
    for((c=0;c< base.roiSize.width; c++)
      *finalSAD++ += *prevDstPtr++
}

/* Once we are finished with the CPIS_sad() function, we delete it */
CPIS_delete(handle);
```

The CPSI_sad() function is not suitable for motion-estimation algorithms used in video encoding because the search step size is fixed to one pixel and cannot be parameterized. For video encoding, the search step size is generally equal to the macroblock's width of 4, 8 or 16. A possible customization is to change the implementation to support motion estimation type of template matching using the VICP computation unit library's function: imxenc_sum_abs_diff(), which requires the template's width to be multiple of 8 for maximum performance. The current implementation uses imxenc_filter_op(), which does not impose any width constrain on the template.

Constraints

- The API supports CPIS_16BIT, CPIS_U16BIT, CPIS_8BIT and CPIS_U8BIT data format for the input data and the filter coefficients. For the output, CPIS_U8BIT and CPIS_U16BIT are supported.
- templateRoiSize < inBlockSize ≤ MAX_SAD_BLOCKSIZE
  with:
  templateRoiSize= params.templateRoiSize.width × params.templateRoiSize.height
  inBlockSize= (base. procBlockSize.width + params.templateRoiSize.Width -1) × (base. procBlockSize.height + params.templateRoiSize.height -1) × max(sizeof(INPUT_TYPE), sizeof(OUTPUT_TYPE))
- templateRoiSize ≤ templateSize ≤ MAX_SAD_TEMPLATESIZE
  with:
  templateSize= params.templateSize.width × params.templateSize.height
- base. procBlockSize.width must be multiple of 8.
- procBlockSize.height must be ≤ 256.

Performance

For CPIS_U8BIT input, template size of 12×12, for input buffer size of 960*280 and processing buffer size of 96*28:

Setup Time	~73000 CPU Clocks
Processing Time	~9,870,000 CPU Clocks or 37 CPU cycles/point

3.3.40 CPIS_setSadTemplateOffset

CPIS_setSadTemplateOffset *Moves the Start of Template to Next Template's Sub-block*

Syntax

```
Int32 CPIS_setSadTemplateOffset (
    CPIS_Handle     *handle,
    Uint16          templateStartOfst
);
```

Arguments

CPIS_Handle	*handle	Handle corresponding to the recursive filter operation previously initialized and for which we need to pass new initial values.
Uint16	templateStartOfst	Offset in number of bytes to the first element of the template.

Return Value

0 Success

-1 Error and CPIS_errorno set to originating error.

Description

This function is used by the application to move the start offset of the template to the next template's sub-block.

See Section 3.3.39 for more details on the usage of CPIS_setSadTemplateOffset ().

Performance

925 CPU cycles

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Communications and Telecom	www.ti.com/communications
DSP	dsp.ti.com	Computers and Peripherals	www.ti.com/computers
Clocks and Timers	www.ti.com/clocks	Consumer Electronics	www.ti.com/consumer-apps
Interface	interface.ti.com	Energy	www.ti.com/energy
Logic	logic.ti.com	Industrial	www.ti.com/industrial
Power Mgmt	power.ti.com	Medical	www.ti.com/medical
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
RFID	www.ti-rfid.com	Space, Avionics & Defense	www.ti.com/space-avionics-defense
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Video and Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless-apps

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265