ShadoCam Imaging Library

Rad-icon Imaging Corp.

P/N 1046 Rev. 2.1

1. Introduction

Thank you for purchasing the ShadoCam Imaging Library (SIL). This manual will guide you through the installation and use of this library. Our goal is to get you started on the image acquisition portion of your imaging project, and to help you develop a successful application.

Overview

The ShadoCam Imaging Library consists of a set of image processing functions that duplicate portions of Rad-icon's ShadoCam Image Acquisition application. These functions allow you to perform the same image calibrations that ShadoCam uses, in order to achieve the best possible image quality from your Shad-o-Box camera.

The ShadoCam Imaging Library does not contain any frame grabber-specific function calls. Since each frame grabber is different, we decided to keep the SIL as universal as possible by including only "generic" image processing functions. The software development kit (SDK) provided by your frame grabber manufacturer contains the function calls needed to initialize your frame grabber and acquire images.

Technical Support

Although we have attempted to make this manual as complete as possible, we realize that there are always additional unanswered questions, as well as unique situations not covered in this booklet. Rad-icon is committed to providing excellent customer service and technical support for all of our products. After all, your success is our business.

For technical assistance with the ShadoCam Imaging Library or your Shad-o-Box camera please e-mail your questions to support@rad-icon.com, or contact our customer service department (8 am to 5 pm Pacific Time) at 408-736-6000. Please be prepared to give a detailed description of your problem.

For the latest contact information, data sheets and application notes please visit our web site at http://www.rad-icon.com.

2. Installation

Installation of the ShadoCam Imaging Library is as simple as double-clicking on the installer (setup.exe) on the installation disk. The installation program will create a folder called "ScImgLib" (in your "C:\Program Files" directory by default). The following files will be copied to this folder:

Scilib21.dll	SIL run-time library
License.rtf	SIL end-user license agreement
Readme.txt	installation and release information
ScManual.pdf	this manual in PDF format
\Include\Scilib21.h	SIL C++ header file
\Lib\Scilib21.lib	SIL C++ link library

The following camera configuration files will be installed in the "\CamFiles" subdirectory:

\PXD1000\SB####PX.cam	for the Imagenation PXD1000 frame grabber
\DT3157\SB####DT.cam	for the Data Translation DT3157 frame grabber
\MV-2500\	for the µTech MV-2500 frame grabber
\X64-LVDS\	for the DALSA X64-LVDS frame grabber
\Meteor2\SB####M2.dcf	for the Matrox Meteor-II/Digital frame grabber

These camera configuration files have been tested with their respective frame grabbers and provide a starting point for configuring your own software. Please contact our customer service department if you need support for additional frame grabbers not listed above.

To start using the SIL functions, first make sure that your frame grabber SDK is installed. Check that your Shad-o-Box camera and your frame grabber are running correctly using ShadoCam or another suitable application. Make sure that the paths for the header file and link library are accessible to your compiler or development system. Finally, copy the run-time library (.DLL) file into your application directory or to the "C:\Windows\System" directory.

3. Developing an Application

This section describes the basic steps necessary to perform an imaging sequence. It is intended as an overview, from a software development perspective, of the image acquisition process. Many of the functions that are used to acquire an image into the PC are frame grabber specific and differ from one model to another. Please refer to your frame grabber SDK documentation for details.

Initialize the Frame Grabber

The first step in acquiring an image is to start up the frame grabber. The frame grabber needs to be initialized and set up for the camera connected to it – in this case a Shad-o-Box x-ray camera. The initialization typically tells the frame grabber what kind of image to expect from the camera in terms of image size, pixel size, timing and more. Your frame grabber SDK documentation contains examples for the exact sequence of steps that are required here (the calls are different for each frame grabber – see Appendix C for some examples). Many frame grabbers use camera configuration files to feed them the necessary setup information. These files can be either read from disk or programmed directly into your application using a header file.

Some camera configuration files for specific frame grabber models are included with the SIL. They can provide a starting point for setting up the frame grabber and camera for your particular application.

Acquire an Image

Once the frame grabber and memory buffers are set up, you can acquire an image. The Shad-o-Box camera is up and running as soon as you supply power. The next step is to arm the frame grabber (if required), and then provide a software or hardware trigger to start the image acquisition. The frame grabber will usually wait for the next available image from the camera and transfer it into the image buffer. Once the image has been transferred, it is available for further processing.

Perform Image Calibrations

Once the image is acquired, you can perform various image calibrations to adjust the image quality. This is where the ShadoCam Imaging Library comes in. Rad-icon has developed several image processing functions that are specially optimized to calibrate Shad-o-Box images. Depending on the application, it may not be necessary to use all of these functions, but in general they are helpful in maximizing the image quality from your Shad-o-Box camera.

Pixel Deinterlacing

Pixel deinterlacing is a required step for all Shad-o-Box cameras except for the Shad-o-Box 512. In the multi-channel Shad-o-Box cameras, the individual channels are scanned in parallel and multiplexed for transmission to the PC. This maximizes the camera's frame rate without requiring additional bulky interface cables for the extra channels. It also means that the information that arrives in the image buffer is scrambled. The pixel deinterlacing function in the ShadoCam Imaging Library quickly unscrambles the image buffer so that all the pixels are in the correct place in the image.

Offset Correction

Offset correction is an optional image processing step that corrects for small variations in the dark image from the Shad-o-Box camera. Without light or x-rays, the only signal coming from the camera is an offset voltage and dark current. Although small, both of these can vary on a pixel-by-pixel level. The offset correction algorithm requires an offset image (typically an averaged dark image) to subtract from subsequent frames. The offset image should be refreshed frequently to account for dark current changes due to temperature variations, or when changing integration (exposure) times.

Gain Correction

Gain correction is a highly recommended processing step that corrects the image for variations in the intensity of the x-ray beam and for gain variations within the Shad-o-Box camera. The gain correction algorithm normalizes each acquired image based on a stored flat-field exposure (the gain image), by dividing each pixel value by its corresponding gain

image value and then multiplying by the mean value of the gain image. Small local variations in image contrast often become visible only after the raw image has been processed with the gain correction algorithm.

The gain image should be averaged over 10-20 frames in order to minimize any increase in image noise resulting from the gain correction. If the detector position and source kVp are constant, it may be possible to reuse the gain image over many imaging sessions.

Pixel Correction

Pixel correction is an optional image processing step that can correct for missing or "dead" pixels in the image. For more detailed information about the pixel correction algorithms provided with the SIL please refer to the **ScPixelCorrection** function description.

Pixel Filtering

A new addition to the SIL is the Bright Pixel Filter function. This function selectively replaces individual pixels that are significantly brighter (or darker) than their surroundings. This can be useful for filtering pixels with variable dark current that may not be captured in the Pixel Map, or for correcting “direct hits” of x-rays in the silicon.

Camera Control Functions

Camera control functions such as exposure control, camera reset and sparse sampling (binning) can be accessed through the frame grabber interface. An on-board pulse generator or counter/timer chip is supplied with most frame grabbers to control camera exposure times. Two RS-422 I/O lines are required to control the camera reset and binning functions. Please refer to your frame grabber manual for instructions on how to utilize these controls.

When first initializing the imaging system, it is important to establish synchronization between the Shad-o-Box camera and the frame grabber. The camera will usually start up in *continuous* mode. We recommend initializing the interface by supplying a few *External Frame Sync* pulses through the frame grabber interface or the external input connector on the camera. Once the synchronization between the camera and the frame grabber has been established, you can switch back to *continuous* mode by setting the *External Frame Sync* lines high.

4. ShadoCam Library

This section contains a complete function reference for the ShadoCam Imaging Library. The function calls have been developed to work on 32-bit Windows operating systems, and are intended for C++ applications with the provided header file.

Data Types

Image buffers for use with the SIL should be of signed short integer type (two bytes per pixel). The buffers are assumed to contain row-sequential pixel data, with the first pixel at the location indicated by the buffer pointer. Function return values (error codes) are also of signed short integer type. Other parameters vary as indicated in the function definition.

The following table gives the sizes of the various data types that are used by the SIL:

Type	*Size*
char	8 bits
short, unsigned short	16 bits
int, unsigned int	32 bits
BOOL	8 bits
all pointers (int*, char*, etc.)	32 bits

Constants

The SIL contains several defined constants:

SCMAXPIXMAPSIZE the max. number of entries in the pixel map array (10,000)
SCMAXPIXMAPFILESIZE the maximum pixel map file size (100 kB)
SCMAXERRORMESSAGE the maximum error message length (128 characters)

Additional defined constants are listed on the following pages. Please refer to the C header file if you need to see the actual definitions.

ScDeinterlace

short **ScDeinterlace(**short ***imgBuf,** unsigned int **nBufSize,** unsigned int **nWidth,**
unsigned int **nHeight,** unsigned short **CamType,** BOOL
bGapSpace);

Return Value

0 if successful; error code on failure

Parameters

imgBuf	pointer to an image buffer of 16-bit pixels
nBufSize	width of image buffer (number of pixels per row)
nWidth	width of image (number of columns)
nHeight	height of image (number of rows)
CamType	the camera type that was used to acquire the image to be deinterlaced; see Appendix A for a list of valid camera types (e.g. **SCCAMTYPE_1024A**).
bGapSpace	flag to determine whether a two-pixel gap should be inserted into the deinterlaced image to account for the space between image sections

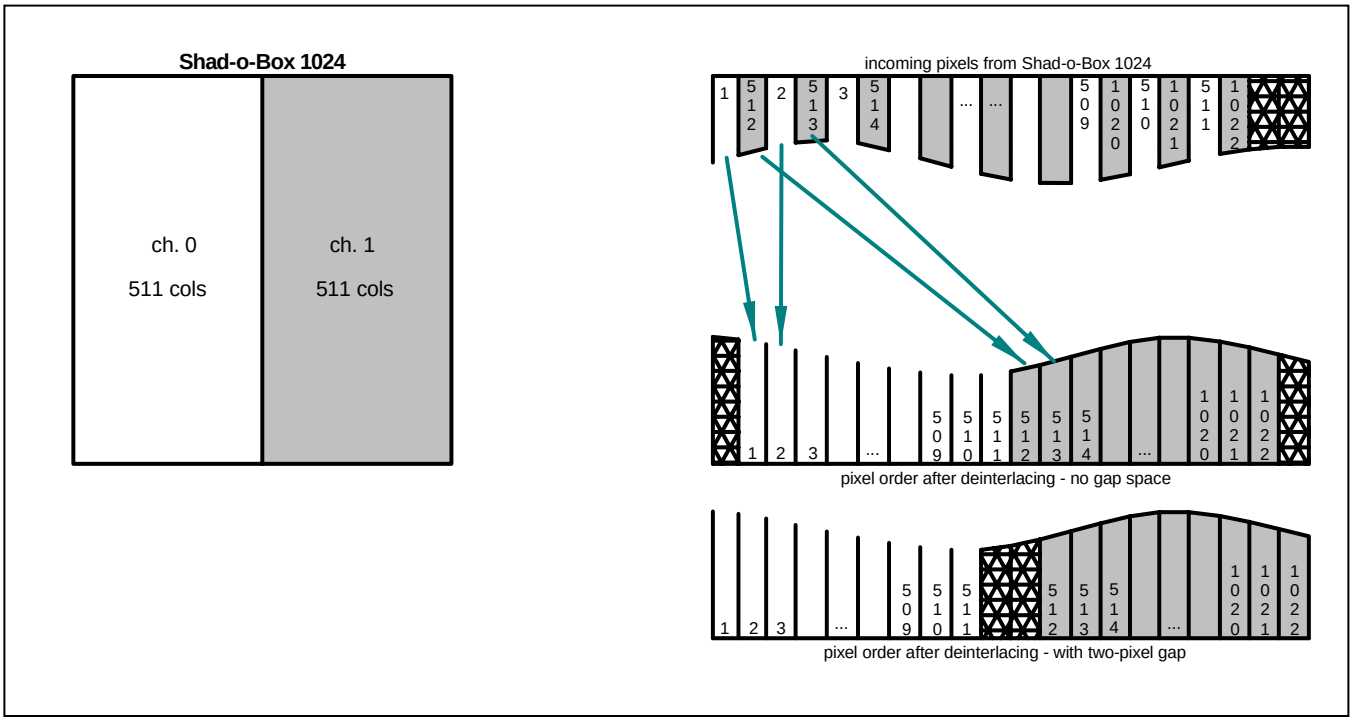
Description

Deinterlaces the pixel data in the image buffer (**imgBuf**). The **CamType** argument determines which deinterlacing algorithm is used to rearrange the pixel data. Shad-o-Snap cameras do not require deinterlacing, but this function can be used to insert a gap space into the acquired image. **nWidth** and **nHeight** specify the actual image dimensions. The parameter **nBufSize** may be larger than **nWidth** to account for additional pixel padding in the buffer at the end of each row.

Shad-o-Box 1024

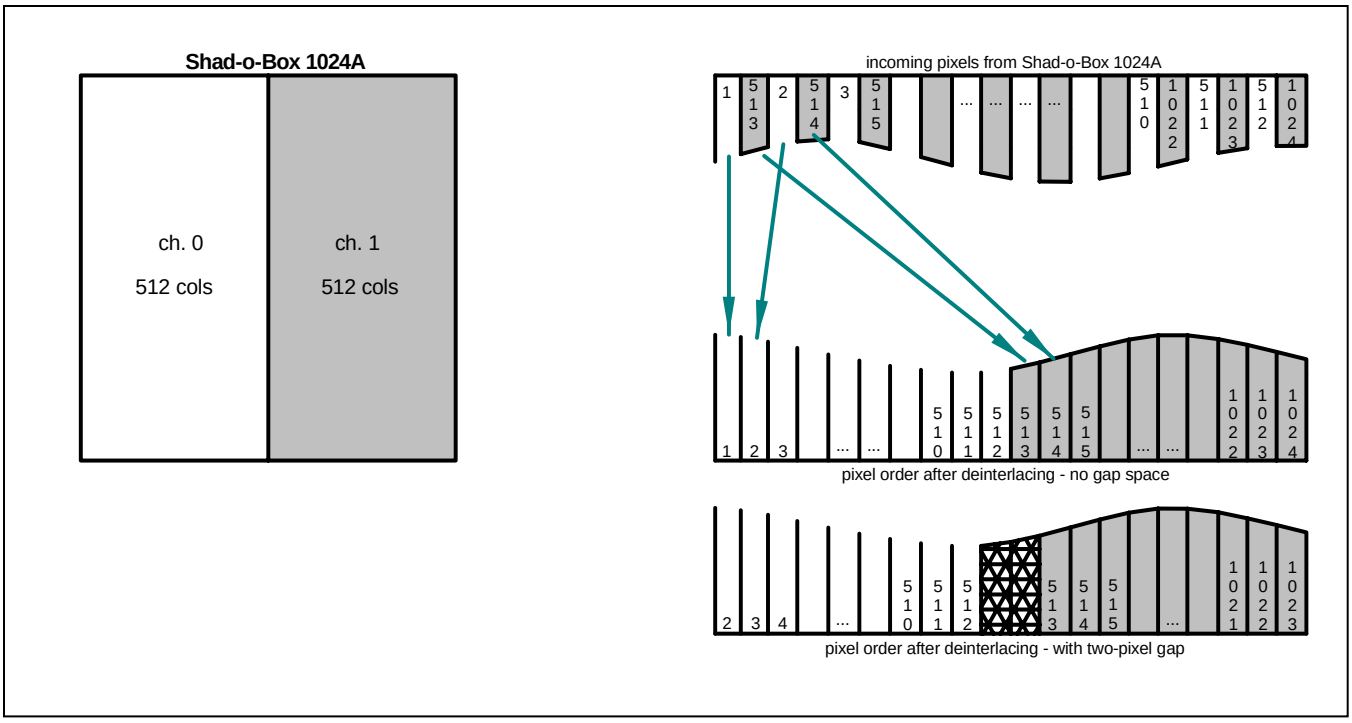
This is the original version of the Shad-o-Box camera. It contains two 511-column by 1022-row imagers with a 100-200 µm gap in between. Each row is transmitted as alternating pixels from the left and right channels of the camera. Although these cameras are no longer manufactured, the deinterlacing algorithm is included to support these older models.

If **bGapSpace** is **FALSE**, the image data from the two sections are placed side-by-side, and two blank columns are inserted at the left and right edges of the image. If **bGapSpace** is **TRUE**, two blank columns are inserted in the center of the image. These columns should be entered into the Pixel Map and filled in using the appropriate pixel correction algorithm (see **ScPixelCorrection**).



Shad-o-Box 1024A

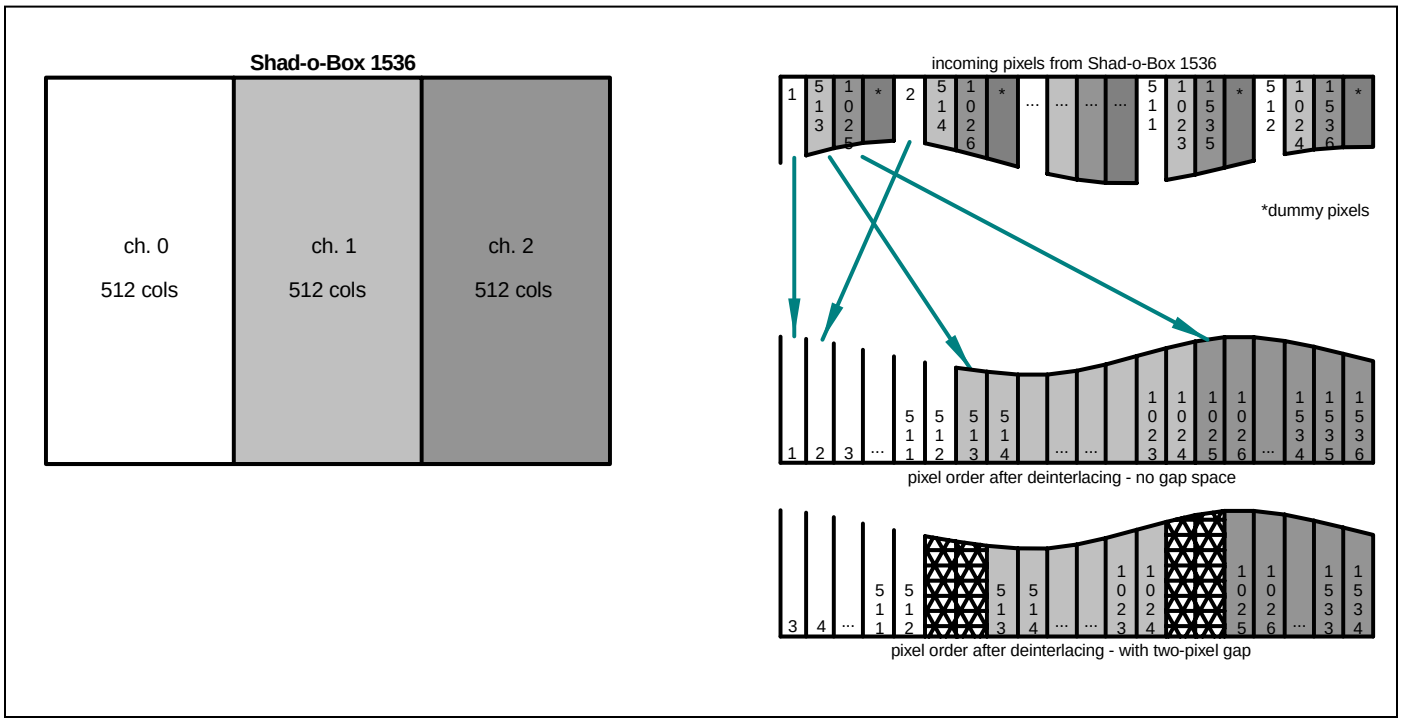
The newer version of the Shad-o-Box camera contains two 512-column by 1024-row imagers with a 100 µm gap in between. Each row is transmitted as alternating pixels from the left and right channels of the camera. If **bGapSpace** is **FALSE**, the image data from the two sections are placed side-by-side. There are no blank columns to fill in. If **bGapSpace** is **TRUE**, two blank columns are inserted in the center of the image. The left-most column of the left-hand (channel 0) device and the right-most column of the right-hand (channel 1) device are discarded in order to keep the image width constant at 1024 columns.



Shad-o-Box 1536

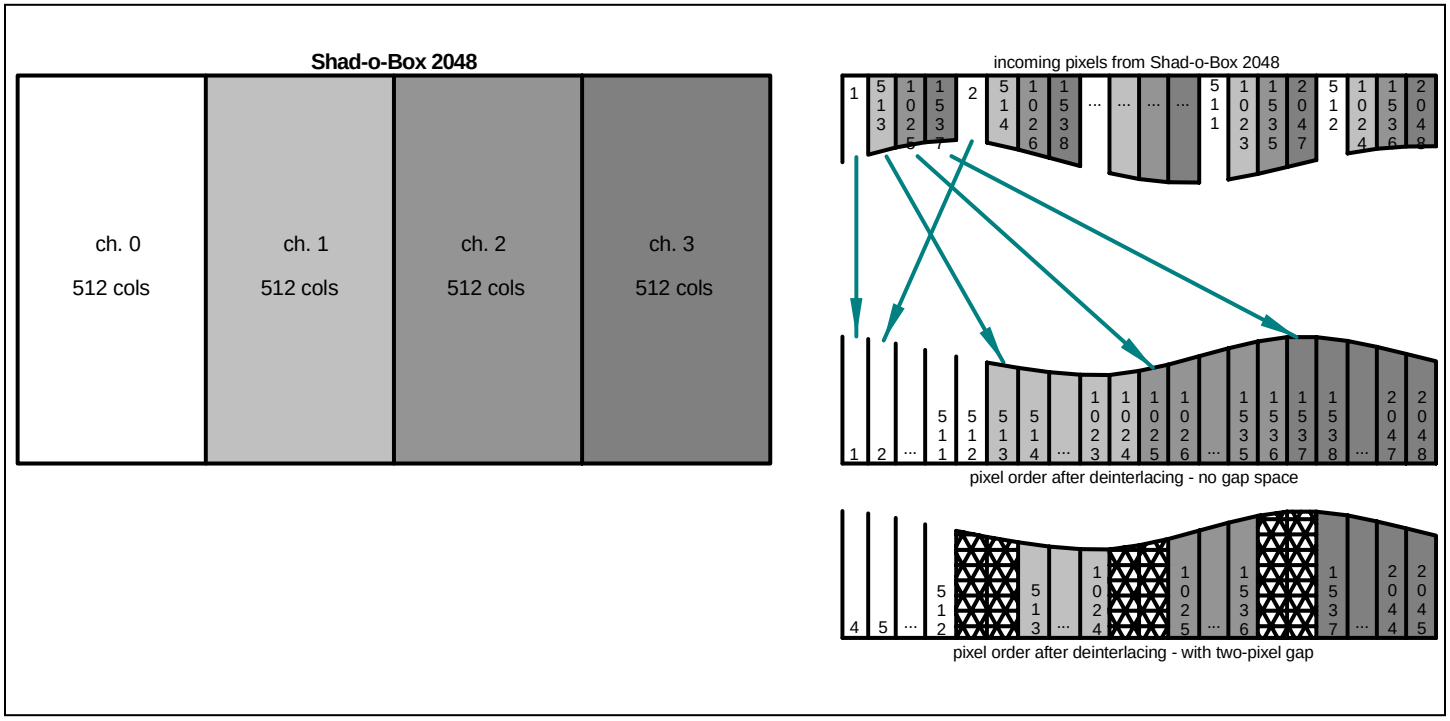
The Shad-o-Box 1536 camera contains three 512-column by 1024-row imagers separated by two 100 µm gaps. The data from the three camera channels are interlaced together with a fourth dummy channel. If **bGapSpace** is **FALSE**, the image data from the three sections are placed side-by-side. There are no blank columns to fill in. If **bGapSpace** is **TRUE**, two blank columns are inserted between each of the image sections. The two left-most columns of the left-hand (channel 0) device and the two right-most columns of the right-hand (channel 2) device are discarded in order to keep the image width at 1536 columns.

Note that the image buffer for the Shad-o-Box 1536 must be at least 2048 pixels wide.



Shad-o-Box 2048

The Shad-o-Box 2048 camera contains four 512-column by 1024-row imagers separated by three 100 µm gaps. The data from the four camera channels are interlaced together in the same format as for the Shad-o-Box 1536. If **bGapSpace** is **FALSE**, the image data from the four sections are placed side-by-side. There are no blank columns to fill in. If **bGapSpace** is **TRUE**, two blank columns are inserted between each of the image sections. The three left-most columns of the left-hand (channel 0) device and the three right-most columns of the right-hand (channel 3) device are discarded in order to keep the image width at 2048 columns.



Shad-o-Box 4K

The Shad-o-Box 4K camera contains eight 512-column by 1024-row sensors arranged in a 2x4 matrix. The imagers are separated by one horizontal and three vertical 100 µm gaps. The data from the eight camera channels are interlaced together as shown in the figure below. Note that the upper row of sensors (ch. 4-7) reads out in the opposite direction from the bottom row.

If **bGapSpace** is **FALSE**, the image data from the eight sections are placed side-by-side. There are no blank rows or columns to fill in. If **bGapSpace** is **TRUE**, two blank columns or rows are inserted between each of the image sections. The three left-most columns of the left-hand (channel 0) device and the three right-most columns of the right-hand (channel 3) device are discarded in order to keep the image width at 2048 columns.

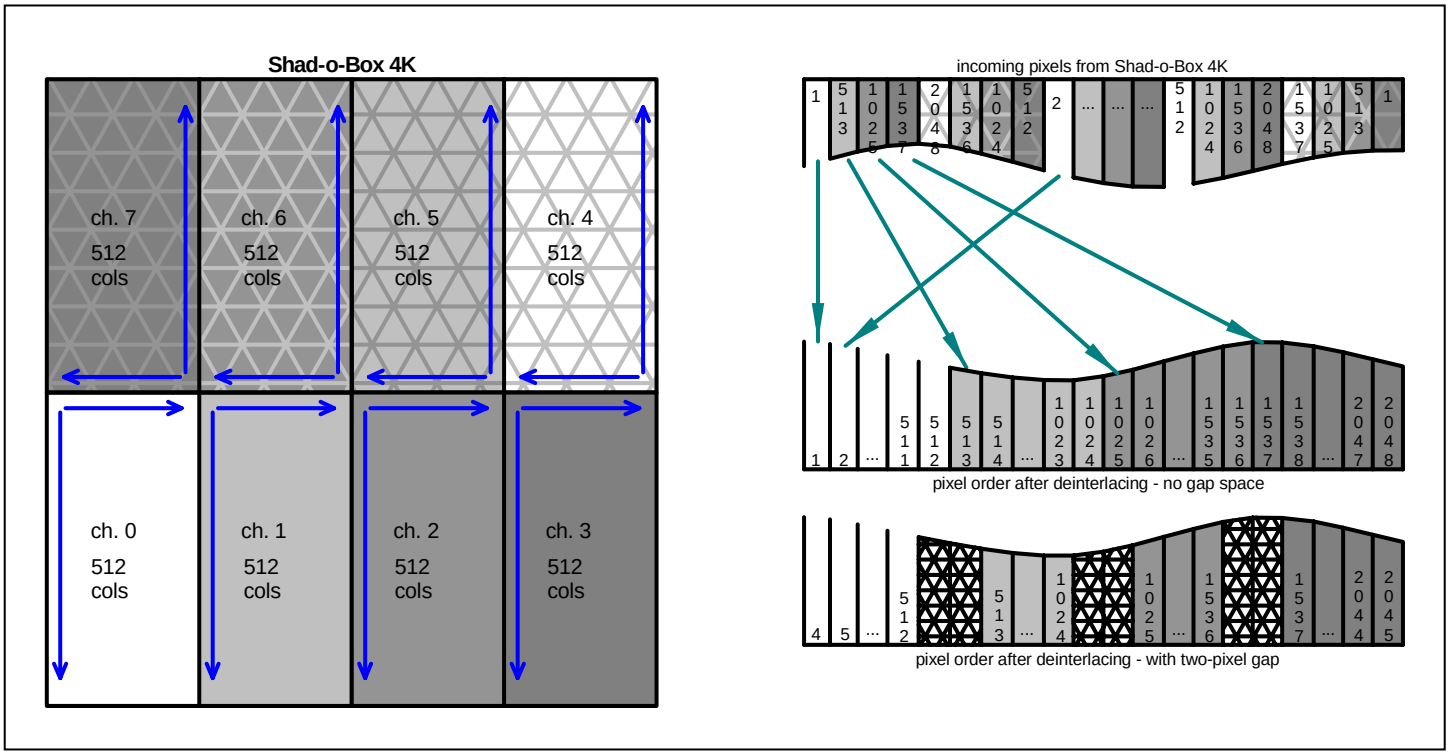


Image Size

The size of the image buffer must be at least **nBufSize** * **nHeight** * 2 bytes. No overflow checking is performed. If the buffer size is set too small, the memory space will be contaminated and unpredictable results may occur. The deinterlacing algorithm assumes a standard, row-sequential image buffer with **nBufSize** * 2 bytes per row. The image buffer can be padded with dummy pixels by specifying a **nBufSize** parameter that is larger than **nWidth**.

If the buffer width **nBufSize** is less than the image **nWidth**, the function will return with an **SCERROR_INVALIDBUFFERWIDTH** error. If either **nWidth** or **nHeight** are zero, the function will return with an **SCERROR_INVALIDIMAGESIZE** error.

The deinterlacing algorithm expects the image width to be the standard image width defined for each camera type (see Appendix A). If a different-sized image width is passed to the **ScDeinterlace** function, it will still process the image but return with error code **SCERROR_INVALIDIMAGEWIDTH**. Depending on the value passed, the resulting image may not be properly deinterlaced.

ScOffsetCorrection

short **ScOffsetCorrection(**short ***imgBuf,** short ***ofstBuf,** unsigned int **nWidth,**
unsigned int **nHeight);**

Return Value

0 if successful; error code on failure

Parameters

imgBuf	pointer to an image buffer of 16-bit pixels
ofstBuf	pointer to an image buffer that contains the offset image data
nWidth	width of image buffer (number of columns)
nHeight	height of image buffer (number of rows)

Description

Performs an offset correction on the pixel data in the image buffer (**imgBuf**). For each pixel value in the image buffer, the corresponding pixel value in the offset image buffer (**ofstBuf**) is subtracted. The result, which may contain negative pixel values, is returned to the image buffer.

The size of both image buffers must be at least **nWidth** * **nHeight** * 2 bytes. No overflow checking is performed. If the buffer size is set too small, the memory space will be contaminated and unpredictable results may occur. If either **nWidth** or **nHeight** are zero, the function will return with an **SCERROR_INVALIDIMAGESIZE** error.

ScGainCorrection

short **ScGainCorrection(**short ***imgBuf,** short ***gainBuf,** unsigned int **nWidth,**
unsigned int **nHeight,** short **nMean);**

Return Value

0 if successful; error code on failure

Parameters

imgBuf	pointer to an image buffer of 16-bit pixels
gainBuf	pointer to an image buffer that contains the gain image data
nWidth	width of image buffer (number of columns)
nHeight	height of image buffer (number of rows)
nMean	mean value to normalize image to; if zero, mean value is calculated from the gain image data

Description

Performs a gain correction on the pixel data in the image buffer (**imgBuf**). Each pixel value in the image buffer is divided by the corresponding pixel value in the gain image buffer (**gainBuf**), and then multiplied by **nMean**. The result, rounded to the nearest integer value, is returned to the image buffer.

If the normalization constant **nMean** is zero, the function will calculate its own constant by computing the average of all the pixels in the gain image. This will add processing overhead and should only be done if the computation time is not critical.

A zero pixel value in the gain image will result in a zero pixel value in the returned image. Very small pixel values in the gain image may result in large, possibly negative values in the returned image.

The size of both image buffers must be at least **nWidth** * **nHeight** * 2 bytes. No overflow checking is performed. If the buffer size is set too small, the memory space will be contaminated and unpredictable results may occur. If either **nWidth** or **nHeight** are zero, the function will return with an **SCERROR_INVALIDIMAGESIZE** error.

ScPixelCorrection

short **ScPixelCorrection(**short ***imgBuf,** unsigned int **nWidth,** unsigned int **nHeight,**
PIXMAPENTRY ***pixMap,** unsigned short **nPixMapCount,**
unsigned short **pcMethod);**

Return Value

0 if successful; error code on failure

Parameters

imgBuf	pointer to an image buffer of 16-bit pixels
nWidth	width of image buffer (number of columns)
nHeight	height of image buffer (number of rows)
pixMap	pointer to the pixel map (an array of **PIXMAPENTRY** data structures)
nPixMapCount	number of entries in the pixel map array
pcMethod	pixel correction method to be applied to the correction:
	SCMETHOD_MEAN simple mean of adjacent pixels
	SCMETHOD_INTERPOLATE interpolate across line defects
	SCMETHOD_GRADIENT interpolate along minimum gradient

Description

Applies pixel correction to the data in the image buffer (**imgBuf**), based on the entries in the pixel map array. An interpolation algorithm is used to estimate the missing pixel value for each pixel, row or column identified in the pixel map. The pixel is then replaced with the new value in the image buffer.

Single pixel defects are repaired by replacing the pixel value with the average of the surrounding pixels. The **pcMethod** flag identifies which pixel correction method to use for correcting row and column defects. The **INTERPOLATE** method uses straight interpolation between adjacent pixels in a direction perpendicular to the row or column to be repaired. The **GRADIENT** method, on the other hand, identifies the direction along which the image gradient (slope in gray levels) is at a minimum, and then interpolates along that direction.

Please refer to our application note AN03 "Guide to Image Quality and Pixel Correction Methods" for more details on pixel correction methods.

The size of the image buffer must be at least **nWidth** * **nHeight** * 2 bytes. No overflow checking is performed. If the buffer size is set too small, the memory space will be contaminated and unpredictable results may occur. Pixel map entries for pixel positions outside the image buffer space are ignored. If either **nWidth** or **nHeight** are zero, the function will return with an **SCERROR_INVALIDIMAGESIZE** error. If the **pcMethod** parameter does not match one of the pixel correction methods listed above, the function will return with an **SCERROR_INVALIDPCMETHOD** error.

See the **ScReadPixMap** function for a description of the pixel map data array and the **PIXMAPENTRY** data structure.

ScFixPixel

void **ScFixPixel(**short ***imgBuf,** unsigned int **nWidth,** unsigned int **nHeight,** int **nCol,**
int **nRow,** int **nMethod,** int **nMask);**

Return Value

none

Parameters

imgBuf	pointer to an image buffer of 16-bit pixels	
nWidth	width of image buffer (number of columns)	
nHeight	height of image buffer (number of rows)	
nCol	column number of pixel to be repaired; must be greater than or equal to 0 and less than **nWidth**	
nRow	row number of pixel to be repaired; must be greater than or equal to 0 and less than **nHeight**	
nMethod	pixel correction method to be applied:	
	SCMETHOD_MEAN	simple mean of adjacent pixels
	SCMETHOD_INT_HOR	interpolate horizontally across defect
	SCMETHOD_INT_VERT	interpolate vertically across defect
	SCMETHOD_GRAD_HOR	interpolate horizontally along min. gradient
	SCMETHOD_GRAD_VERT	interpolate vertically along min. gradient
nMask	defect mask of surrounding pixels; only the least significant eight bits are used	

Description

Repairs a specific pixel in an image using one of five correction methods. The function returns without processing any image data if the pixel specified by **nCol** and **nRow** is not within the image dimensions given by **nWidth** and **nHeight**, or if **nMethod** is not one of the correction methods listed above. It does not return any error codes.

This function is called by **ScPixelCorrection** to perform the actual pixel repair. It is included here only for completeness and typically wouldn't be called directly.

ScReadPixMap

short **ScReadPixMap(**char ***filePath,** PIXMAPENTRY ***pixMap,** int ***numEntries);**

Return Value

0 if successful; error code on failure

Parameters

filePath	pointer to a null-terminated string that specifies the file name of the pixel map file to open
pixMap	pointer to the pixel map (an array of **PIXMAPENTRY** data structures)
numEntries	variable to receive the number of pixel map entries read from the file

Description

Reads a "ShadoCam Pixel Map File" from disk. This file is shipped with the Shad-o-Box camera and contains the factory-calibrated pixel map that identifies any defective pixels, rows or columns in the image. The file data is transferred to the pixel map array, which is used in subsequent calls to **ScPixelCorrection** to repair these pixels.

The pixel map is stored as an array of **PIXMAPENTRY** structures, defined as follows:

```
typedef struct {
        char type;              // defect type: 'P'ixel, 'R'ow or 'C'olumn
        unsigned short x1;      // x (column) coordinate or start of column range
        unsigned short x2;      // end of column range
        unsigned short y1;      // y (row) coordinate or start of row range
        unsigned short y2;      // end of row range
        unsigned short mask;    // correction mask to identify surrounding defects
        char flag;              // preferred correction method
} PIXMAPENTRY;
```

The **type** field identifies whether the pixel map entry refers to a single pixel, a row or a column. The coordinates of the pixel, row or column are given by the **x1**and **y1** fields. In addition, the **x2** and **y2** fields are used to identify the range of a partial row or column defect. The **mask** field specifies the correction mask for surrounding pixels (see **ScCalcPixMapMask** for more details). A **flag** may be used to specify a particular correction method.

Prior to calling **ScReadPixMap**, you need to initialize the **pixMap** pointer and allocate enough memory to hold the pixel map array. You can use the malloc() or GlobalAlloc() functions to do this:

```
pixMap = (PIXMAPENTRY*)malloc( SCMAXPIXMAPSIZE * sizeof(PIXMAPENTRY) );
```

The maximum size of the pixel map array is about 24 kB, which holds up to 2000 pixel map entries. You can allocate less memory if you know for sure that your pixel map has less than 2000 entries. However, the **ScReadPixMap** function will keep writing to the pixel map array up to the **SCMAXPIXMAPSIZE** limit if it finds additional entries in the file. Remember to free the heap memory when you no longer need it.

After reading the pixel map file, **ScReadPixMap** calls the **ScCalcPixMapMask** function to fill in the **mask** field for each **PIXMAPENTRY** structure in the pixel map array. The total number of pixel map entries that were read from the pixel map file is placed into the **numEntries** variable before the function returns.

If it is unable to open the specified pixel map file, the function will return with an **SCERROR_INVALIDFILEPATH** error. If the file size of the specified pixel map file is either zero or greater than **SCMAXPIXMAPFILESIZE**, or if certain keywords in the file were not found, the function will return with an **SCERROR_INVALIDPIXMAPFILE** error. An **SCERROR_PIXMAPSYNTAX** error will be returned if there was a syntax error detected in reading the pixel map file. The lines containing any errors are ignored.

ScCalcPixMapMask

short **ScCalcPixMapMask(**PIXMAPENTRY ***pixMap,** unsigned short **nPixMapCount);**

Return Value

0 if successful; error code on failure

Parameters

pixMap pointer to the pixel map (an array of **PIXMAPENTRY** data structures)

nPixMapCount number of entries in the pixel map array

Description

Calculates the **mask** field for each entry in the pixel map array. The **mask** field identifies which, if any, of the pixels surrounding the one being repaired have been identified as defective. Only "good" pixels are used in the pixel correction algorithms.

The **mask** field is an 8-bit register in which each bit corresponds to one of the nearest neighbor pixels, starting at the upper left with the least significant bit and proceeding in a counter-clockwise direction:

bit0	bit7	bit6
bit1	X	bit5
bit2	bit3	bit4

For example, if the pixel to the left is defective, the mask value would be 2 (0x02). If the entire column to the right is defective, the mask value would be 112 (0x70). For row and column entries, the **mask** field identifies adjacent defective rows or columns in the pixel map. Individual pixel entries should always be listed first in the pixel map so that they are repaired before the rows and columns are processed. The row and column correction algorithms can then assume that all adjacent pixels are "good".

For the pixels along the edge of an image, the "missing" pixels outside the image boundary are not included in the **mask** field. The image boundary checking is done instead in the **ScPixelCorrection** function.

ScImageFilter

short **ScImageFilter(**short ***imgBuf,** int **nWidth,** int **nHeight,** short **nValue**, int **nLimits**);

Return Value

Returns the number of pixels that were selected for replacement.

Parameters

imgBuf	pointer to an image buffer of 16-bit pixels
nWidth	width of image buffer (number of columns)
nHeight	height of image buffer (number of rows)
nValue	replacement value of filtered pixels (median value of surrounding pixels is used if **nValue** is zero)
nLimits	maximum percent deviation (±) from surrounding average

Description

Applies a "Bright Pixel Filter" to the data in the image buffer (**imgBuf**). Each pixel in the image is compared to the average value of its nine nearest neighbors (fewer neighbors are used for edge or corner pixels). The pixel is replaced with **nValue** if the percentage difference between its current value and the surrounding average is greater than **nLimits**. If **nValue** is zero, the pixel is replaced with the median value of its neighbors.

Use this function to filter out individual pixels that are significantly brighter (or darker) than their surrounding neighbors. Because of the inherently low pixel-to-pixel contrast of most x-ray images, a deviation greater than 10% is almost certainly caused by either a "direct hit" of an x-ray absorbed in the silicon (instead of the scintillator), or by a faulty signal from the sensor itself. This function can filter out statistically defective pixels that vary in location from image to image, or are otherwise not listed in the pixel map.

The size of the image buffer must be at least **nWidth** * **nHeight** * 2 bytes. No overflow checking is performed. If the buffer size is set too small, the memory space will be contaminated and unpredictable results may occur.

ScPixelFilter

short **ScFixPixel**(short* **pList**, int **nElements**, int **nLimits**);

Return Value

1 if pixel was replaced; otherwise 0

Parameters

pList array of short pixel values; first element is the test pixel itself

nElements size of the pList array (number of entries)

nLimits maximum percentage deviation (±) of acceptable pixel values

Description

Calculates the average of the **pList** pixel value array, excluding the first element. If the value of the first element exceeds the calculated average by more than **nLimits** percent, the function replaces the first element in **pList** by the median value of the array and returns 1. Otherwise, it leaves **pList** unchanged and returns 0.

This function is called by **ScImageFilter** to perform the actual filter calculation. It is included here only for completeness and typically wouldn't be called directly.

ScErrorMessage

short **ScErrorMessage(**short **errCode,** char ***errMessage);**

Return Value

error code; 99 if error code is not valid

Parameters

errCode the error code to be deciphered

errMessage pointer to a character buffer to receive the error message text

Description

Looks up the error code provided and returns the corresponding text error message as a null-terminated string. The maximum length of the error message is defined by the **SCMAXERRORMESSAGE** parameter. See Appendix B for a listing of error messages.

Appendices

A. Camera Types

The following camera types are defined in the ShadoCam Imaging Library:

Camera Type	*no. of channels*	*default image width*	*image buffer width*	*default image height*
SCCAMTYPE_512	one	512	512	1024
SCCAMTYPE_1024	two	1024	1024	1022
SCCAMTYPE_1024A	two	1024	1024	1024
SCCAMTYPE_1536	three	1536	2048	1024
SCCAMTYPE_2048	four	2048	2048	1024
SCCAMTYPE_4K	eight	2048	4096	2000
SCCAMTYPE_512HS	two	512	512	512
SCCAMTYPE_1024HS	four	1024	1024	512
SCCAMTYPE_SKIA8	eight	2048	4096	2000
SCCAMTYPE_SKIA10	ten	2560	5120	2000
SCCAMTYPE_SNAP1024	two	1024	1024	1000
SCCAMTYPE_SNAP2048	four	2048	2048	1000
SCCAMTYPE_SNAP1K	eight	1024	1024	1000
SCCAMTYPE_SNAP4K	eight	2048	2048	2000
SCCAMTYPE_REMOTE1	one	512	512	1024
SCCAMTYPE_REMOTE2	two	1024	1024	1024
SCCAMTYPE_REMOTEHR	one	1200	1200	1600
SCCAMTYPE_REMOTE200	two	1024	1024	1000
SCCAMTYPE_CUSTOM	one	n/a	n/a	n/a

B. Error Codes

The following error codes are defined in the ShadoCam Imaging Library:

SCERROR_NOERROR	No error has occurred.
SCERROR_INVALIDIMAGESIZE	The image size (**nWidth** x **nHeight**) is invalid. Typically this means that either the **nWidth** or **nHeight** argument passed to the function was 0.
SCERROR_INVALIDCAMTYPE	The **CamType** argument passed to the function is not one of the supported camera types.
SCERROR_INVALIDIMAGEWIDTH	The **nWidth** argument passed to the **ScDeinterlace** function does not match the default image width for the camera type given by the **CamType** argument.
SCERROR_INSUFFICIENTMEMORY	There is not enough memory available to perform the requested function.
SCERROR_INVALIDBUFFERWIDTH	The image buffer width (**nBufSize**) argument passed to the **ScDeinterlace** function is less than the image width parameter **nWidth**.
SCERROR_INVALIDFILEPATH	An error occurred while opening the pixel map file specified by the **filePath** argument. This may be due to a sharing violation, or because the path name does not point to a valid file.
SCERROR_INVALIDPIXMAPFILE	The file specified by the **filePath** argument is not a valid "ShadoCam Pixel Map File". This could be because certain keywords are misspelled or missing, or because the file is either empty or too large.
SCERROR_PIXMAPSYNTAX	There was a syntax error detected in reading the pixel map file. The lines containing any errors are ignored.
SCERROR_INVALIDPCMETHOD	The **pcMethod** argument passed to the function is not one of the supported pixel correction methods.
SCERROR_INVALIDERRCODE	The error code passed to the **ScErrorMessage** function does not match one of the error codes listed in this appendix.

C. Program Listings

The following is a sample program listing for a Win32 console application to initialize the PXD1000 frame grabber, acquire a single image and save it to disk:

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include "pxd.h"
#include "iframe.h"
#include "scimglib.h"

main()
{
PXD pxd;
FRAMELIB framelib;
long hFG;
short* lpFrameBase;
FRAME* pFrame;
CAMERA_TYPE* camType;

int i;
const long len = 65536;
unsigned short nLUT[len];
char filename[] = "TestPXD.raw";
char camFile[]="C:\\pxd\\Cameras\\SB1024PX.CAM";
unsigned int nWidth = 1024;  // width of Shad-o-Box image (columns)
unsigned int nHeight = 1024; // height of Shad-o-Box image (rows)

// initialize the Imagenation libraries
if ( !imagenation_OpenLibrary("pxd_32.dll", &pxd, sizeof(PXD)) )  {
       MessageBox(NULL, "Frame grabber library not loaded.", "TestPXD",
                  MB_ICONERROR);
       return 0;  }
if ( !imagenation_OpenLibrary("frame_32.dll", &framelib, sizeof(FRAMELIB)) )  {
       MessageBox(NULL, "Frame library not loaded.", "TestPXD",
                  MB_ICONERROR);
       return 0;  }

// request access to frame grabber
if ( !(hFG = pxd.AllocateFG(-1)) )  {
       MessageBox(NULL, "PXD frame grabber not found.", "TestPXD",
                  MB_ICONERROR);
       imagenation_CloseLibrary(&framelib);
       imagenation_CloseLibrary(&pxd);
       return 0;  }

// initialize camera configuration
if ( !(camType = pxd.LoadConfig(camFile)) )  {
       MessageBox(NULL, "Camera configuration not loaded.", "TestPXD",
                  MB_ICONERROR);
       pxd.FreeFG(hFG);
       imagenation_CloseLibrary(&framelib);
       imagenation_CloseLibrary(&pxd);
```

```
       return 0;  }
pxd.SetCameraConfig(hFG, camType);
pxd.ContinuousStrobes(hFG, TRUE);  // turn on camera frame sync

// initialize input LUT to shift image data down by two bits
for (i = 0; i < len; i++) nLUT[i] = i>>2;
pxd.SetInputLUT(hFG, 16, 0, 0, len, nLUT);

// set up image destination buffer
if ( !(pFrame = pxd.AllocateBuffer (pxd.GetWidth(hFG), pxd.GetHeight(hFG),
                                    PBITS_Y16)) )  {
       MessageBox(NULL, "Unable to create image buffer.", "TestPXD",
                  MB_ICONERROR);
       pxd.FreeFG(hFG);
       imagenation_CloseLibrary(&framelib);
       imagenation_CloseLibrary(&pxd);
       return 0;  }

// create pointer to image buffer
// note: the configuration file sets up the image buffer to contain
//       an extra column to the left and right of the actual image
lpFrameBase = (short *)framelib.FrameBuffer(pFrame);
lpFrameBase++; // point to first pixel in image

// wait for user input, then grab next available image
printf("Press any key to start image acquisition...");
while (!kbhit());
pxd.Grab(hFG, pFrame, 0);

// deinterlace image and save to disk
ScDeinterlace(lpFrameBase, framelib.FrameWidth(pFrame), nWidth, nHeight,
              SCCAMTYPE_1024A, FALSE);
framelib.WriteBin(pFrame, filename, 1);

// release frame grabber resources
framelib.FreeFrame(pFrame);
pxd.FreeConfig(camType);
pxd.FreeFG(hFG);
imagenation_CloseLibrary(&framelib);
imagenation_CloseLibrary(&pxd);

return 1;
}
```

The following is a sample program listing for a Win32 console application to initialize the DT3157 frame grabber, acquire a single image and save it to disk:

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include "olwintyp.h"
#include "olfgapi.h"
#include "olimgapi.h"
#include "3157api.h"
#include "scimglib.h"

main()
{
OLT_APISTATUS Status;
HGLOBAL hDevList;
LPOLT_IMGDEVINFO lpDevList;
OLT_IMG_DEV_ID DtDevID;
OLT_FG_FRAME_ID DtFrameID;
OLT_FG_FRAME_INFO DtFrameInfo;
short* lpFrameBase;

HANDLE hFile;
char szAppName[] = "TestDT";
char filename[] = "TestDT.raw";
unsigned long nBytesWritten;
unsigned long nBytesToWrite;
int nNumDev, nOldVal, nFlag;
unsigned short nSource = 0;
unsigned int nActExp;
unsigned int nWidth = 1024;  // width of Shad-o-Box image (columns)
unsigned int nHeight = 1024; // height of Shad-o-Box image (rows)
unsigned int nFirstCol = 4;  // DT3157 skips 4 clocks at start of row
unsigned int nFirstRow = 0;

// Get the number of installed DT devices
printf("Searching for DT3157 frame grabber...");
Status = OlImgGetDeviceCount(&nNumDev);
if ( (Status != OLC_STS_NORMAL) || (nNumDev == 0) )  {
       MessageBox(NULL, "DT frame grabber not found.", szAppName, MB_ICONERROR);
       return 0;  }

// Get an array of device info structures
hDevList = GlobalAlloc(GHND, nNumDev * sizeof(OLT_IMGDEVINFO));
if ( !hDevList )  {
       MessageBox(NULL, "Not enough memory for device list.", szAppName, MB_ICONERROR);
       return 0;  }
lpDevList = (LPOLT_IMGDEVINFO)GlobalLock(hDevList);
if ( !lpDevList)  {
       GlobalFree(hDevList);
       MessageBox(NULL, "Can't lock memory for device list.", szAppName, MB_ICONERROR);
       return 0;  }

lpDevList->StructSize = sizeof(OLT_IMGDEVINFO);
Status = OlImgGetDeviceInfo(lpDevList, nNumDev * sizeof(OLT_IMGDEVINFO));
```

```
if ( Status != OLC_STS_NORMAL )  {
       GlobalUnlock(hDevList);
       GlobalFree(hDevList);
       MessageBox(NULL, "Unable to get frame grabber info.", szAppName, MB_ICONERROR);
       return 0;  }

// We'll assume the first device in the list is the DT3157 frame grabber. If more than
// one DT board is installed, we'd have to check the list for the correct device alias.

// Open the device and set it to 14-bit digital camera mode
printf("\nInitializing DT3157 frame grabber...");
Status = OlImgOpenDevice(lpDevList[0].Alias, &DtDevID);
if ( Status != OLC_STS_NORMAL )  {
       GlobalUnlock(hDevList);
       GlobalFree(hDevList);
       MessageBox(NULL, "Unable to initialize frame grabber.", szAppName, MB_ICONERROR);
       return 0;  }
Dt3157SetDigitalCameraType(DtDevID, nSource, DT3157_DIGCAM_14BIT_INPUT);

// Set IO lines 4 & 5 high (camera CTRL1 and CTRL2 inputs)
Dt3157SetDigitalIOConfiguration(DtDevID, 0x30);
Dt3157SetDigitalIO(DtDevID, 0x30);

// Turn off sync sentinel
OlFgSetInputControlValue(DtDevID, nSource, OLC_FG_CTL_SYNC_SENTINEL, FALSE, &nOldVal);

// Get pixel clock flags and set to latch on falling edge
OlFgQueryInputControlValue(DtDevID, nSource, OLC_FG_CTL_CLOCK_FLAGS, &nOldVal);
nFlag = nOldVal & (0xFFFFFFFF ^ OLC_FG_CLOCK_EXT_ON_LO_TO_HI);
OlFgSetInputControlValue(DtDevID, nSource, OLC_FG_CTL_CLOCK_FLAGS, nFlag, &nOldVal);

// Get variable scan flags and set line enable to latch on rising edge
OlFgQueryInputControlValue(DtDevID, nSource, OLC_FG_CTL_VARSCAN_FLAGS, &nOldVal);
nFlag = nOldVal | OLC_FG_VS_LINE_ON_LO_TO_HI;
OlFgSetInputControlValue(DtDevID, nSource, OLC_FG_CTL_VARSCAN_FLAGS, nFlag, &nOldVal);

// Get variable scan flags and set frame enable to latch on rising edge
OlFgQueryInputControlValue(DtDevID, nSource, OLC_FG_CTL_VARSCAN_FLAGS, &nOldVal);
nFlag = nOldVal | OLC_FG_VS_FIELD_ON_LO_TO_HI;
OlFgSetInputControlValue(DtDevID, nSource, OLC_FG_CTL_VARSCAN_FLAGS, nFlag, &nOldVal);

// Set frame dimensions
OlFgSetInputControlValue(DtDevID, nSource, OLC_FG_CTL_FIRST_ACTIVE_PIXEL, nFirstCol,
                         &nOldVal);
OlFgSetInputControlValue(DtDevID, nSource, OLC_FG_CTL_ACTIVE_PIXEL_COUNT, nWidth,
                         &nOldVal);
OlFgSetInputControlValue(DtDevID, nSource, OLC_FG_CTL_TOTAL_PIX_PER_LINE, nWidth,
                         &nOldVal);
OlFgSetInputControlValue(DtDevID, nSource, OLC_FG_CTL_FIRST_ACTIVE_LINE, nFirstRow,
                         &nOldVal);
OlFgSetInputControlValue(DtDevID, nSource, OLC_FG_CTL_ACTIVE_LINE_COUNT, nHeight,
                         &nOldVal);
OlFgSetInputControlValue(DtDevID, nSource, OLC_FG_CTL_TOTAL_LINES_PER_FLD, nHeight,
                         &nOldVal);
OlFgSetInputControlValue(DtDevID, nSource, OLC_FG_CTL_FRAME_WIDTH, nWidth, &nOldVal);
OlFgSetInputControlValue(DtDevID, nSource, OLC_FG_CTL_FRAME_HEIGHT, nHeight, &nOldVal);
```

```
// Allocate frame buffer and get pointer
Status = OlFgAllocateBuiltInFrame(DtDevID, OLC_FG_DEV_MEM_VOLATILE,
OLC_FG_NEXT_FRAME, &DtFrameID);
if ( Status != OLC_STS_NORMAL )  {
       GlobalUnlock(hDevList);
       GlobalFree(hDevList);
       MessageBox(NULL, "Unable to allocate image memory.", szAppName, MB_ICONERROR);
       return 0;  }
OlFgMapFrame(DtDevID, DtFrameID, &DtFrameInfo);
lpFrameBase = (short*)DtFrameInfo.BaseAddress;

// Set frame period timer and acquire one frame to activate. This establishes
// the initial synchronization between the camera and the frame grabber.
OlFgSetInputControlValue(DtDevID, nSource, OLC_FG_CTL_CLOCK_FREQ, 5000000, &nOldVal);
Dt3157EnableExposureMode(DtDevID, nSource, TRUE);
Dt3157SetExposure(DtDevID, nSource, 500000, 1, &nActExp); // 500ms exposure
OlImgSetTimeoutPeriod(DtDevID, 1, NULL);
OlFgAcquireFrameToDevice(DtDevID, DtFrameID);

// Turn frame period timer off and set timeout period to 15 seconds
Dt3157EnableExposureMode(DtDevID, nSource, FALSE);
OlImgSetTimeoutPeriod(DtDevID, 15, NULL);

// Wait for user input, then grab & deinterlace next available image
printf("\n\nPress any key to start image acquisition...");
while (!kbhit());
OlFgAcquireFrameToDevice(DtDevID, DtFrameID);
ScDeinterlace(lpFrameBase, nWidth, nWidth, nHeight, SCCAMTYPE_1024A, FALSE);

// Write image buffer to disk
nBytesToWrite = nWidth * nHeight * 2;
hFile = CreateFile(filename, GENERIC_WRITE, FILE_SHARE_READ, NULL, CREATE_ALWAYS,
                   FILE_ATTRIBUTE_NORMAL, NULL);
if ( hFile == INVALID_HANDLE_VALUE )  {
       GlobalUnlock(hDevList);
       GlobalFree(hDevList);
       MessageBox(NULL, "Error writing image to file.", szAppName, MB_ICONERROR);
       return 0;  }
WriteFile(hFile, lpFrameBase, nBytesToWrite, &nBytesWritten, NULL);
CloseHandle(hFile);

// Release frame grabber resources
GlobalUnlock(hDevList);
GlobalFree(hDevList);

return 1;
}
```