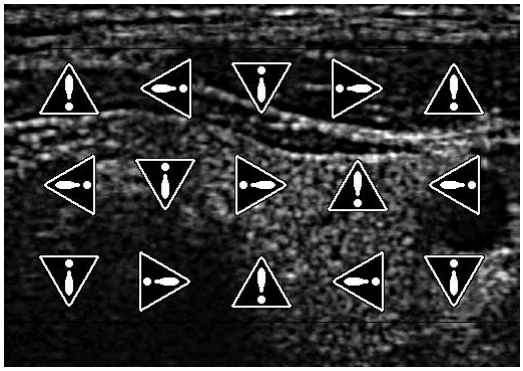
SCPP-Evaluation SDK - User Guide

This document describes the Sapheneia Software Developers Kit (SCPP SDK). The SCPP SDK includes everything needed to include Sapheneia image enhancement software into any application. This special version of the SDK is intended to be used for evaluation only. **The SDK is fully functional except that two black lines are added to the processed image.**

Note that the parameter files that are included in the SDK should only be used when integrating the processing in an application.

The SDK parameter files should not be used for evaluation of image quality or processing speed.
Warning signs are inserted in the result image to indicate this.



Parameter files for clinical use will be delivered separately as agreed separately with Sapheneia. These parameter files will not produce any warning signs in the result image.

The black lines can be replaced by another pattern by setting the SCPP_AEP_METHOD environment variable to different values:

- set SCPP_AEP_METHOD=0
  The upper and lower part of the image will contain a replication pattern.
- set SCPP_AEP_METHOD=1
  One quadrant of the result image will be covered with a replication pattern.
- set SCPP_AEP_METHOD=2
  Two lines with pixel value 0 will be added to the image. This is the default.

Version history	
Version	**Change**
V010	A few minor language corrections.
V009	Updated tuning panel description.
V008	Usage Scenario: Corrected parameters to SCPPCreateLut().
V007	Added x-ray tuning panel appendix.
V006	Updated ultrasound tuning panel appendix.
V005	Added ultrasound tuning panel appendix.
V004	Added description of new processing type SCPP_PROCESSING_LUT and the new function SCPPCreateLut(). The usage scenario is now based on C function interface since it is the recommended way to use the SDK.
V003	Added description of SCPP_AEP_METHOD=2
V002	This version described both the Windows and the Linux versions.
V001	The initial version.

Contents

1 Overview

The SCPP SDK exists in one version for Windows and one version for Linux. This User Guide describes both versions. The two versions are very similar to use. The small differences that exist are described explicitly in this User Guide.

The main component of the SCPP SDK is the SCPP library.

The library makes extensive use of the SSE SIMD instructions defined by Intel. The SSE instructions were introduced in the Intel Pentium III processor and are available in all later Intel processors. These SSE instructions are also available in the AMD processors.
The library is threaded internally which means that it will take advantage of multiple core CPUs.

A console example program, with source code, is included in the SDK. This example program shows how to use the C functions in the SCPP library.
E.g. the example program can be used to process an image with a specified parameter file:

```
Windows:
> cd bin
> SCPPTest.exe -i 50 -e -callback -p ..\par\testDestructSingleFrame.spf;default;0 \
..\im\test_u8_hxw_384x256_01.raw;hgh=384;wid=256;type=u8   ..\im\out.raw

Linux:
$ cd bin
$ LD_LIBRARY_PATH=.
$ export LD_LIBRARY_PATH
$ ./SCPPTest.exe -i 50 -e -callback -p "..\par\testDestructSingleFrame.spf;default;0" \
"..\im\test_u8_hxw_384x256_01.raw;hgh=384;wid=256;type=u8"   ..\im\out.raw
```

The contents of the SDK and the use of the example program are described further in the README.TXT file in the SDK.

2 Licensing

As described in the README.TXT in the SDK, the SCPP library needs to be enabled by installing a license key. The license key can be installed via the example program or via function calls in the library. In Windows it is also possible to use the **SCP Authorization Manager** see 2.1 below. The library calls are describe in section *3.6.15 License Handling Methods*.

Note: In Windows, the license information and logging status is stored in a common location and all users will have the same setting. In Linux, the setting is normally stored in the users home directory, i.e. all users will have their private setting. See *5 Linux storage location configuration* for more information.

A license request must be sent to Sapheneia in order to obtain a license key. The license request information can be retrieved via a function call in the library or by using the example program:

```
Windows:
> cd bin
> SCPPTest -l request
Product: SCPP-Evaluation
00-14-22-F8-8F-5F Broadcom 440x 10/100 Integrated Controller - Packet Scheduler Miniport

Linux:
$ cd bin
$ LD_LIBRARY_PATH=.
$ export LD_LIBRARY_PATH
$ ./SCPPTest -l request
Product: SCPP-Evaluation
00-14-22-F8-8F-5F Ethernet Adapter
```

The license request information should be sent to Sapheneia. The returned license key can be entered via a library function call or via the example program:

```
> SCPPTest -l CKBET-6D2CK-GPDZD-6WR66-5O4Y6-XLOO6
```

The license key is stored permanently on the computer. The current license status can be checked by a function call in the library or by using the example program:

```
> SCPPTest -l check
 Product: SCPP-Evaluation
     Key: CKBET-6D2CK-GPDZD-6WR66-5O4Y6-XLOO6
      Id: 00-14-22-F8-8F-5F
    Type: MAC
  Expire: 2018-01-01
  Status: License key is valid
```

It is also a possible to just get a list of the available host ids (MAC addresses).

```
> SCPPTest -l hids
Hid                 Type  Description
-------------------------------------------------------
00-14-22-F8-8F-5F    MAC  Broadcom 440x 10/100 Integrated Controller - Packet Scheduler Minipo
```

2.1 The SCP Authorization Manager (Windows only)

The SCP Authorization Manager program can be used for license handling as an alternative to the SCPPTest program and SCPP library function calls. This program has a graphical user interface which makes it easy to use. The program is described in section *4, Licensing with SCP Authorization Manager*.

3 Application Programmers Interface (API)

The API (C functions) are defined in SCPP.h in the SDK. The API has been designed to be both simple to use and effective to run.

There is also a C++ class interface to the library. The C function interface is however the preferred interface. See 3.5 for more information.

3.1 A Simple Usage Scenario

The example below shows a simple usage scenario for the C functions in the SCPP library. Note that the processing result depends on the parameter file.
Note: The parameter files included in the SDK are just example parameter files. They should only be used to test functionality and not to evaluate image quality. Sapheneia will create special parameter files for specific image types.

```
// Using C function interface
SCPPHandle scppHandle                          // A handle to a SCPP instance
SCPPError scppStatus;                          // An error code variable
int nParameterBlocks;                          // Number of parameter blocks in parameter file
const char **parameterBlockNames;              // Parameter block names in parameter file
const int *nSettings;                          // Number of settings in each parameter block
char *parameterFile = "testCombined.spf";      // Parameter file (just an example)
int hgh,wid;                                   // Image height and width
char *type="u8";                               // Image pixel type (just an example)
#define ERR_MSG_LEN 2048
char errorMsg[ERR_MSG_LEN];                    // Error Message from SCPPCreateEx()

// A handle to an instance of SCPP must be created.
// Several instances of SCPP can be created if necessary.

scppStatus = SCPPCreateEx(&scppHandle, errorMsg, ERR_MSG_LEN);
if( scppStatus != SCPP_ERR_OK ) {
  fprintf(stderr,"SCPPCreateEx: %s\n", errorMsg);
  return -1;
}

// A processing mask can be set if only a part of the image should be processed.
// Note that the height and width of the processing mask must match the image that
// is processed later.
// The mask is used by the SCPPProcessImage() and SCPPProcessImageSequenceFirst/Next()
// functions.
// The mask should only be set if needed e.g. when only a fan shaped area
// in an ultrasound image should be processed.

scppStatus = SCPPSetMaskImage(scppHandle, pMaskImage, hgh, wid, type)
if( scppStatus != SCPP_ERR_OK ) {
  fprintf(stderr,"SCPPSetMaskImage: %s\n", SCPPGetErrorMessage(scppHandle));
  return -1;
}

// The parameter file controls the image processing and it must be selected depending on
// the desired processing result. The parameter file contains a number of named parameter
// blocks and each block contains a number of settings.
// The number of parameter blocks, the parameter block names and the number of settings in
// each block is returned by SCPPReadParameterFile().

scppStatus = SCPPReadParameterFile(scppHandle, parameterFile,
                                   &nParameterBlocks, &parameterBlockNames, &nSettings);
if( scppStatus != SCPP_ERR_OK ) {
  fprintf(stderr,"SCPPReadParameterFile: %s\n", SCPPGetErrorMessage(scppHandle));
  return -1;
}

int blockNbr=0; // Use the first parameter block in this example.
                // The possible range is [0..(nParameterBlocks-1)]
int setting=0;  // Use setting 0 in this example. Possible is [0..(nSettings[blockNbr]-1)]
```

```
// We need to find out what type of processing the selected block
// and setting contains in the loaded parameter file.
TSCPPProcessingInfo processingInfo;

scppStatus = SCPPGetProcessingInfo(scppHandle, parameterBlockNames[blockNbr],
                                                setting, &processingInfo);
if( scppStatus != SCPP_ERR_OK ) {
  fprintf(stderr,"SCPPGetProcessingInfo: %s\n", SCPPGetErrorMessage(scppHandle));
  return -1;
}

switch( processingInfo.processingType) {

  case SCPP_PROCESSING_SINGLE_FRAME:{
    // The processing type is single frame. Use SCPPProcessImage() function.

    while( 1 ) {                // Loop until all images have been processed
      void *pInBuffer;          // Input image buffer
      void *pOutBuffer;         // Output image buffer (of same size as input buffer)

      // Allocate and read input image into inBuffer here.
      // Allocate output buffer of same size as input buffer.
      // Set wid, hgh and image type.
      // Then process the image with this call:
      scppStatus = SCPPProcessImage(scppHandle, pInBuffer, hgh, wid, type, pOutBuffer,
                                    parameterBlockNames[blockNbr], setting);
      if( scppStatus != SCPP_ERR_OK ) {
        fprintf(stderr,"SCPPProcessImage: %s\n", SCPPGetErrorMessage(scppHandle));
        return -1;
      }

      // Optionally: Get the min and max values in the result image
      //             if these values are needed e.g. in a DICOM header
      // Note: This operation should only be performed if necessary.
      float minPixel, maxPixel;
      scppStatus = SCPPGetImageMinMax(scppHandle, pOutBuffer,
                                      hgh, wid, type, minPixel, maxPixel);
      if( scppStatus != SCPP_ERR_OK ) {
        fprintf(stderr,"SCPPGetImageMinMax: %s\n", SCPPGetErrorMessage(scppHandle));
        return -1;
      }
      // Save the processed result from the output image buffer
    }
  }
  break;

  case SCPP_PROCESSING_SEQUENCE:{
    // The processing type is sequence. The processing of image sequences can use the
    // the information in the images 'before' (and sometimes 'after') the current image
    // to produce a better processing result.
    // The processingInfo.nInputImages value tells how many input
    // images are needed to produce one output image.
    // E.g. a value of 1 means that only one image is needed, a value of 2 means that
    // also the next image in the sequence is needed and so on.
    // Note: For processing of ultrasound/fluoro images the value is usually 1. I.e. only
    // the current image is needed.
    // The output image is always the processing result for the first input image.
    // (The number of output images is always one at the moment.)

    int nInputImages=xx;  // The number of images in the image sequence
    void *pInputImages[]; // Vector of pointers to input images (size should be
                          // processingInfo.nInputImages)
    void *pOutputImages[];// Vector of pointers to output images (size should be
                          // processingInfo.nOutputImages, i.e. 1 at the moment)

    for(int outNbr=0; outNbr < nInputImages; outNbr++) {

      for(int inNbr=outNbr; inNbr< outNbr+processingInfo.nInputImages; inNbr++) {

        int i=inNbr-outNbr;

        if(inNbr< nInputImages)
          pInputImages[i]= <set to pointer to image #inNbr in the image sequence>
        else
```

```
          pInputImages[i]=(void *)0; // No more images available. Set to 0;
      }
      pOutputImages[0] = <set pointer to output buffer>; //(just one 1 at the moment)

      if(outNbr==0) {
        // Process the first image in the sequence
        scppStatus = SCPPProcessImageSequenceFirst(scppHandle, pInputImages, hgh, wid, type,
                                                   pOutputImages,
                                                   parameterBlockNames[blockNbr],setting);
      }
      else {
        // Process the remaining images in the sequence
        scppStatus = SCPPProcessImageSequenceNext(scppHandle, pInputImages, hgh, wid, type,
                                                  pOutputImages,
                                                  parameterBlockNames[blockNbr],setting);
      }
      if( scppStatus != SCPP_ERR_OK ) {
        fprintf(stderr,"SCPPProcessImageSequence: %s\n", SCPPGetErrorMessage(scppHandle));
        return -1;
      }

      // Optionally: Get the min and max values in the result image
      //             if these values are needed e.g. in a DICOM header
      // Note: This operation should only be performed if necessary.
      float minPixel, maxPixel;
      scppStatus = SCPPGetImageMinMax(pOutputImages[0], hgh, wid, type, minPixel, maxPixel);
      if( scppStatus != SCPP_ERR_OK ) {
        fprintf(stderr,"SCPPGetImageMinMax: %s\n", SCPPGetErrorMessage(scppHandle));
        return -1;
      }
      // Save the processed result from the output image buffer
    }
  }
  break;

  case SCPP_PROCESSING_LUT:{
    // The SCPPCreateLut() function should be used to create a lookup table (LUT)
    // suitable for the supplied image. The supplied image is usually the result from
    // a SCPPProcessImage() function call

    TSCPPLutInfo lutInfo;        // A structure for the LUT information
    while( 1 ) {                 // Loop until all images have been used
      void *pInBuffer;           // Input image buffer

      // Allocate and read input image into inBuffer here. Set wid, hgh and image type.
      // Then process the image with this call:
      scppStatus = SCPPCreateLut(scppHandle, pInBuffer, hgh, wid, type,
                                 parameterBlockNames[blockNbr], setting, &lutInfo);
      if( scppStatus != SCPP_ERR_OK ) {
        fprintf(stderr,"SCPPCreateLut: %s\n", SCPPGetErrorMessage(scppHandle));
        return -1;
      }
      // Save the lutInfo.LUT with the image.
      // The lutInfo structure contains information about the created LUT for the
      // image. The LUT is intended to be used as display lut for the image, e.g. a
      // VOI LUT in the DICOM header.
      // NOTE: The lutInfo.LUT table is only valid until the next call to SCPPCreateLut()!
    }

  }
  break;
default:
fprintf(stderr,"Unknown processing type - %s.\n", processingInfo.processingName);
  return -1;
}
```

3.2 Threading

The SCPP library is safe to use in threaded applications if some simple rules are followed.

- A C function interface handle (SCPPHandle) can only be used from one thread at the time. The simplest way to follow this rule is to create one handle for each thread.
- A specific CSCPP instance can only be used from one thread at the time. The simplest way to follow this rule is to create one CSCPP instance for each thread.

3.3 Image data types

The SCPP library can handle images of different pixel data types. The pixel data type is described by a character string. The following pixel data type strings are defined:

```
"u16"          // unsigned 16-bit data (unsigned short)

"s16"          // signed 16-bit data (short)

"u8"           // unsigned 8-bit data (unsigned char)

"f32"          // 32 bit float (float)
```

The pixel data should be packed. I.e. no padding should be used anywhere.

The function descriptions in the following sections describes which data type is supported by a specific function.

3.4 C function handle

The C functions use a handle to an SCPP instance as there first parameter. The handle is created with the `SCPPCreateEx()` function and destroyed with the `SCPPDestroy()` function.

```
// C
// Create SCPPHandle. Must be called before any other function which has handle as parameter.
// Note: This function is the only one returning any error message as a parameter.
SCPPError SCPPCreateEx(SCPPHandle *handle, char *errMsg, int errMsgSize);

// Destroy SCPPHandle. Deletes a handle and any resources associated with it.
SCPPError SCPPDestroy(SCPPHandle *handle);
```

An example of how to create a handle for the C functions is shown below:

```
// C
SCPPHandle handle;
SCPPError scppStatus;
const int scppMsgSize=2048;
char scppMsg[scppMsgSize];

scppStatus = SCPPCreateEx(&handle, scppMsg, scppMsgSize);

if( scppStatus != SCPP_ERR_OK ) {
  printf("SCPPCreateEx failed: %s\n", scppMsg);
  return -1;
}
```

The SCPPTest example program in the SDK includes more examples.

3.5 C++ CSCPP object

Note: The C function interface is the preferred way to use the SCPP library. The CSCPP class interface is likely to be difficult to use. It only works without issues if the same compiler version as the library was compiled with is used.

A CSCPP instance must be created to use the CSCPP methods. There is only one constructor with no parameters.

```
// C++
CSCPP scpp;
```

3.6 Functions and Methods

This section describes all the C functions and the corresponding CSCPP methods. The example program in the SDK can be used to get additional information about how to use the functions.

3.6.1 SCPPVersion()

The function returns the version string for the library. Note that this function does not need the SCPPhandle parameter.

```
// C
const char *SCPPVersion(char *version, int versionLen);

// C++
const char *CSCPP::Version();
```

C Parameters

`version` Pointer to a caller supplied buffer to receive the version string in.

`versionLen` Size of the supplied buffer. The value should be at least 128.

C++ Parameters

`void` N/A.

3.6.2 SCPPGetErrorMessage()

Most C functions return an error code of the `SCPPError` type. An error is indicated by a return code that differs from `SCPP_ERR_OK`. The function is used to get a text description for the last error.

```
// C
const char *SCPPGetErrorMessage(SCPPHandle handle);

// C++
const char *CSCPP::GetErrorMessage();
```

C Parameters

`handle` A handle created with `SCPPCreateEx()`.

C++ Parameters

`void` N/A.

Note: The `SCPPCreateEx()` function returns any error message as a parameter. The returned error message should be used in this case.
Use SCPPGetErrorMessage() when other functions return a value that differs from `SCPP_ERR_OK`.

See *3.1 A Simple Usage Scenario* for further description.

3.6.3 SCPPSetProgressCallback()

The function makes it possible to setup a callback function which is called during the processing of an image. The progress function should return 0 if the processing should continue after the callback. The processing can also be aborted if a non-zero value is returned. The processing function will return SCPP_ERR_ABORTED if the processing was aborted due to a non-zero return value from a callback function.
NOTE: The use of a progress callback function will slow down processing. Consider not to use the progress callback function when the processing time for an image is short.

```
#ifdef _WIN32
// C Windows
SCPPError SCPPSetProgressCallback(SCPPHandle handle,
                                  int (__stdcall *fCallBack)(unsigned char percent));

// C++ Windows
SCPPError CSCPP::SetProgressCallback( int (__stdcall *fCallBack)(unsigned char percent));
#endif

#ifdef __linux
// C Linux
SCPPError SCPPSetProgressCallback(SCPPHandle handle,
                      int (__attribute__((__stdcall__)) *fCallBack)(unsigned char percent));

// C++ Linux
SCPPError CSCPP::SetProgressCallback(
                      int (__attribute__((__stdcall__)) *fCallBack)(unsigned char percent));
#endif
```

C Parameters

`handle` A handle created with `SCPPCreateEx()`.

C/C++ Parameters

`fCallBack` A pointer to the callback function. Passing a 0 (null) pointer disables the callback.

Callback function parameter

`percent` The value indicates the progress of the processing. The value goes from 0 to 100.

Example of how to setup a progress function callback:

```
#ifdef _WIN32
// Windows
// Define a progress callback function
int (__stdcall Progress)(unsigned char percent)
{
  printf("Progress: %3d\n", percent);
  return 0;
}

#endif

#ifdef __linux
// Linux
// Define a progress callback function
int (__attribute__((__stdcall__)) Progress)(unsigned char percent)
{
  printf("Progress: %3d\n", percent);
  return 0;
}
#endif
```

```
// Later in the code:
{

  SCPPError scppStatus;

  scppStatus = SCPPSetProgressCallback(scppHandle, Progress);

  if( scppStatus != SCPP_ERR_OK ) {
    fprintf(stderr,"SCPPSetProgressCallback: %s\n", SCPPGetErrorMessage(scppHandle));
    return -1;
  }
}
```

3.6.4 SCPPSetMaskImage()

The function sets a processing mask to be used during processing. The processing will only be applied where the mask image pixels are non zero.
The mask image must have the same size as the image that is processed.
The mask is used by the SCPPProcessImage() and SCPPProcessImageSequenceFirst/Next() functions.

```
// C
SCPPError SCPPSetMaskImage(SCPPHandle handle, void *data, int hgh, int wid, const char *type);

// C ++
SCPPError CSCPP::SetMaskImage(void *data, int hgh, int wid, const char *type);
```

C Parameters

`handle` A handle created with `SCPPCreateEx()`.

C/C++ Parameters

`data` A pointer to the mask image.
A null (0) pointer can be used to clear (unset) the mask.

`hgh, wid` Height and width of the mask image in pixels.

`type` Data type of the mask image pixels.
Possible types are “u8”, “u16, “s16” and “f32”.

Note: The bounding box of the non-zero pixels must be of a reasonable size. Usually the reasonable size of the bounding box is at least 64x64 pixels.

3.6.5 SCPPReadParameterFile()

The function loads the parameter file which controls the processing of the images.
A parameter file contains the description of a processing algorithm to be used and the parameters to the algorithm. This means that the processing result is completely controlled by the contents of the parameter file.
Each parameter file can contain one or more algorithm/parameter descriptions. Each such description is called a setting. The settings are grouped in named parameter blocks. Each such parameter block can contain one or more settings. The number of parameter blocks, the names of the blocks and the number of settings in each block is returned by the function.

```
// C
SCPPError SCPPReadParameterFile(SCPPHandle handle, const char *fileName,
                                int *nParameterBlocks, const char ***parameterBlockNames,
                                const int **nSettings);

// C++
SCPPError CSCPP::ReadParameterFile(const char *fileName,
                                   int &nParameterBlocks, const char ** &parameterBlockNames,
                                   const int * &nSettings);
```

C Parameters

`handle` A handle created with `SCPPCreateEx()`.

C/C++ Parameters

`fileName` The name of the parameter file to use. The full path must be specified.

`nParameterBlocks`
The number of parameter blocks in the file.

`parameterBlockNames`
An array containing the names of the parameter blocks.

`nSettings` An array containing the number of settings in each parameter block.

NOTE: The `SCPPReadParameterFile()` function needs some processing time to prepare for the processing. This means that it is a very good idea to load the parameter file outside the main processing loop as in the example in section *3.1, A Simple Usage Scenario*, on page 6.

3.6.6 SCPPReadParameterFromBuffer()

The functionality of `SCPPReadParameterFromBuffer()` is exactly the same as `SCPPReadParameterFile()` method except that the contents of a parameter file is passed to this function instead of the name of the parameter file.

See section *3.6.5* for further information of the functionality.

```
// C
SCPPError SCPPReadParameterFromBuffer(SCPPHandle handle, const char *fileContentsBuffer,
                                int *nParameterBlocks, const char ***parameterBlockNames,
                                const int **nSettings);

// C++
SCPPError CSCPP::ReadParameterFromBuffer(const char *fileContentsBuffer
                                   int &nParameterBlocks, const char ** &parameterBlockNames,
                                   const int * &nSettings);
```

C Parameters

`handle` A handle created with `SCPPCreateEx()`.

C/C++ Parameters

`fileContentsBuffer`
The null terminated character string containing the parameter file contents.

`nParameterBlocks`
The number of parameter blocks in loaded buffer.

`parameterBlockNames`
An array containing the names of the parameter blocks.

`nSettings` An array containing the number of settings in each parameter block.

3.6.7 SCPPWriteParameterFile ()

The function saves the currently loaded parameter file including any changes that have been made to tuning panel parameter values. See section *3.6.10, TuningPanels,* for more information on tuning panels.

```
// C
SCPPError SCPPWriteParameterFile(SCPPHandle handle, const char *fileName);

// C++
SCPPError CSCPP::WriteParameterFile(const char *fileName);
```

C Parameters

`handle` A handle created with `SCPPCreateEx()`.

C/C++ Parameters

`fileName` The full path name of the parameter file to save to.

3.6.8 SCPPGetProcessingInfo()

The function is used to get information about the processing type in the specified parameter block and setting of the loaded parameter file. This information is used to decide what type of processing should be used.
The processing type can be of some different types:

- **Single Frame images.** This means that an image is processed independently from any other images.
- **Image Sequence.** The image is considered to be part of an image sequence of some type. The image can be a slice in an image volume in which case the images 'above' and 'below' the image can be used to produce a better result. The image can also be a part of a time sequence in which case previous images in the sequence can be used to produce a better result.
- **LUT.** The image can be analyzed and a display lookup table (LUT) can be generated for it.

```
// C
SCPPError SCPPGetProcessingInfo(SCPPHandle handle,
                                const char *parameterBlockName, int setting,
                                TSCPPProcessingInfo *processingInfo);

// C++
SCPPError CSCPP::GetProcessingInfo(const char *parameterBlockName, int setting,
                                   TSCPPProcessingInfo &processingInfo);
```

C Parameters

`handle` A handle created with `SCPPCreateEx()`.

C/C++ Parameters

`parameterBlockName`
The parameter block name to use. The name should be one of the names that was returned in `parameterBlockNames[]` by `SCPPReadParameterFile()` or `SCPPReadParameterFromBuffer()`.

`setting` Parameter setting number to use. This value should be one of the settings in the selected parameter block.
I.e. if `parameterBlockName=parameterBlockNames[x]` was selected then setting should be in the range [0.. (`nSettings[x]`-1)].

`processingInfo`
The processing information is returned in this structure. The structure is defined as:

```
typedef struct TSCPPProcessingInfo {
    int processingType;       // SCPP_PROCESSING_SINGLE_FRAME,
                              // SCPP_PROCESSING_SEQUENCE, or
                              // SCPP_PROCESSING_LUT
    char processingName[15]; // A name for the above
    char filterName[10];      // Name of sequence filter
    int nInputImages;         // Number of input images
    int nOutputImages;        // Number of output images (is 1 at the moment)
} TSCPPProcessingInfo;
```

- The SCPP_PROCESSING_SINGLE_FRAME processing type should use the `SCPPProcessImage()` function described in 3.6.9.
- The SCPP_PROCESSING_SEQUENCE processing type should use the `SCPPProcessImageSequenceFirst()` /

`SCPPProcessImageSequenceNext()` functions described in 3.6.10.

- The SCPP_PROCESSING_LUT processing type should use SCPPCreateLut() function described in 3.6.11.

The `nInputImages` and `nOutputImages` specify the number of input and output images that are required for the sequence processing. The number of required input images depends on the specific parameter setting. As an example it can be 1 for a time sequence setting and typically 3 for a volume sequence setting.

See the *3.1 A Simple Usage Scenario* for further information about how to use the function in combination with `SCPPProcessImage()`, `SCPPProcessImageSequenceFirst/Next()` and `SCPPCreateLut()`.

3.6.9 SCPPProcessImage()

The function processes a single frame image using the parameter file settings loaded with the `SCPPPeadParameterFile()`.
The SCPP`GetProcessingInfo()` function, described in 3.6.8, is used to get information about the processing type of a specific parameter block and setting. The SCPPProcessImage() function should be used if the processing type for the setting is SCPP_PROCESSING_SINGLE_FRAME.

```
// C
SCPPError SCPPProcessImage(SCPPHandle handle,
                           void *indata, int hgh, int wid,
                           const char *type, void *outdata, const char *parameterBlockName,
                           int setting);

// C++
SCPPError CSCPP::ProcessImage(void *indata, int hgh, int wid,
                              const char *type,
                              void *outData, const char *parameterBlockName, int setting);
```

C Parameters

`handle` A handle created with `SCPPCreateEx()`.

C/C++ Parameters

`indata` A pointer to input image.

`hgh, wid` Height and width (in pixels) of the input and output image.

`type` Data type of the image pixels. Possible types are “u16”, “s16” and “u8”.

`outData` Pointer to the output buffer.

`parameterBlockName`
Parameter block name to use. The name should be one of the names that was returned in `parameterBlockNames[]` by `SCPPReadParameterFile()` or `SCPPReadParameterFromBuffer()`.

`setting` Parameter setting number to use. This value should be one of the settings in the selected parameter block.
I.e. if `parameterBlockName=parameterBlockNames[x]` was selected then setting should be in the range [0.. (`nSettings[x]`−1)].

See the *3.1 A Simple Usage Scenario* for further information about how to use the function .

3.6.10 SCPPProcessImageSequenceFirst() and SCPPProcessImageSequenceNext()

These functions are used to process image sequences.
The `SCPPGetProcessingInfo()` method, described in 3.6.8, should be used to get information about the processing type of a specific parameter block and setting. The SCPPProcessImageSequenceFirst/Next() functions should be used if the processing type type for the setting is SCPP_PROCESSING_SEQUENCE.
The `SCPPProcessImageSequenceFirst()` method is used to process the first image in the image sequence and the `SCPPProcessImageSequenceNext()` method is used to process each of the rest of the images in the sequence.
See the *3.1 A Simple Usage Scenario* for further information about how to use The `SCPPGetProcessingInfo()` in combination with `SCPPProcessImage()` or `SCPPProcessImageSequenceFirst()/SCPPProcessImageSequenceNext()`.

```
// C
SCPPError SCPPProcessImageSequenceFirst(SCPPHandle handle,
                                        void **indata, int hgh, int wid, const char *type,
                                        void **outdata,
                                        const char *parameterBlockName, int setting);
SCPPError SCPPProcessImageSequenceNext(SCPPHandle handle,
                                        void **indata, int hgh, int wid, const char *type,
                                        void **outdata,
                                        const char *parameterBlockName, int setting);

// C++
SCPPError CSCPP::ProcessImageSequenceFirst(void **indata, int hgh, int wid, const char *type,
                                           void **outdata,
                                           const char *parameterBlockName, int setting);
SCPPError CSCPP::ProcessImageSequenceNext(void **indata, int hgh, int wid, const char *type,
                                          void **outdata,
                                          const char *parameterBlockName, int setting);
```

C Parameters

`handle` A handle created with `SCPPCreateEx()`.

C/C++ Parameters

`indata` An array of pointers to the input images.

`hgh, wid` Height and width (in pixels) of the input and output image.

`type` Data type of the image pixels. Possible types are “u16”, “s16” and “u8”.

`outdata` An array of pointers to the output buffers. At the moment only one output buffer is used. The output image is the processing result for the first input image.

`parameterBlockName` Parameter block name to use. The name should be one of the names that was returned in `parameterBlockNames[]` by `SCPPReadParameterFile()` or `SCPPReadParameterFromBuffer()`.

`setting` Parameter setting number to use. This value should be one of the settings in the selected parameter block.
I.e. if `parameterBlockName=parameterBlockNames[x]` was selected then setting should be in the range [0.. (`nSettings[x]`−1)].

See the 3.1 *A Simple Usage Scenario* for further information about how to use the function .

3.6.11 SCPPCreateLut()

The function creates a lookup table suitable for the display of the supplied image using the parameter file settings `loaded` with the `SCPPPeadParameterFile()`.
The `SCPPGetProcessingInfo()` function, described in 3.6.8, is used to get information about the processing type of a specific parameter block and setting. The SCPPCreateLut() function should be used if the processing type type for the setting is SCPP_PROCESSING_LUT.

```
// C
SCPPError SCPPCreateLut(SCPPHandle handle,
                        void *indata, int hgh, int wid, char *type,
                        const char *parameterBlockName, int setting,
                        TSCPPLutInfo *lutInfo);

// C++
SCPPError CSCPP::CreateLut(void *indata, int hgh, int wid, char *type,
                           const char *parameterBlockName, int setting,
                           TSCPPLutInfo &lutInfo);
```

C Parameters

`handle` A handle created with `SCPPCreateEx()`.

C/C++ Parameters

`indata` A pointer to input image.

`hgh, wid` Height and width (in pixels) of the input image.

`type` Data type of the image pixels. Possible types are “u16” and “u8”.

`parameterBlockName`
Parameter block name to use. The name should be one of the names that was returned in `parameterBlockNames[]` by `SCPPReadParameterFile()` or `SCPPReadParameterFromBuffer()`.

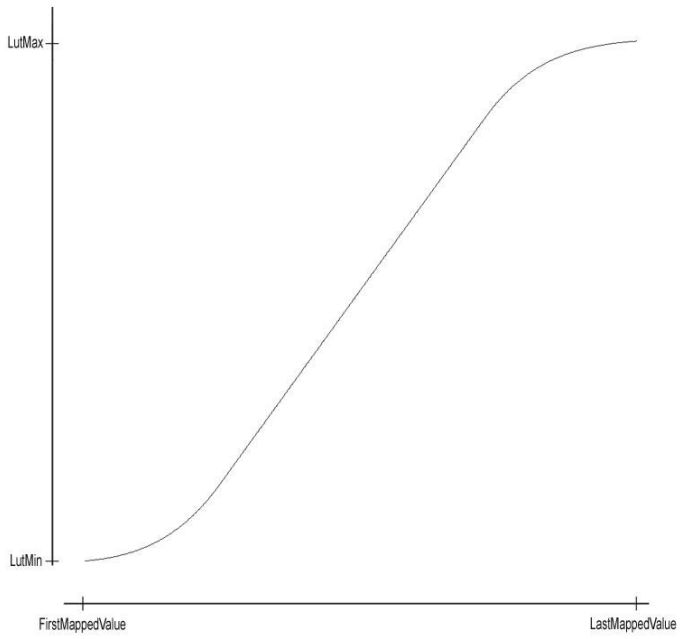
`setting` Parameter setting number to use. This value should be one of the settings in the selected parameter block.
I.e. if `parameterBlockName=parameterBlockNames[x]` was selected then setting should be in the range [0.. (`nSettings[x]`−1)].

`lutInfo` The created LUT information is returned in this structure. The structure is defined as:

```
typedef struct {
  int FirstMappedValue;   // The first mapped value
  int LastMappedValue;    // The last mapped value
  int LutNEntries;        // Number of lut entries
                          //      (==LastMappedValue-FirstMappedValue+1)
  int LutBitsStored;      // The number of bits needed to store the lut values
  int LutMin;             // Minimum value in the lut table
  int LutMax;             // Maximum value in the lut table
  char LutType[10];       // Type of lut generated, "u16" or "u8"
  void *LutPtr;           // Pointer to the lut of type LutType
// Note: that the LutPtr is not valid after the next call to SCPPCreateLut()
} TSCPPLutInfo;

// This is how to map pixel value to display value using the LUT
// Example for LutType=="u16"
unsigned short *pixelVal;                          // Points to pixelValue
unsigned short *LUT = (unsigned short *)LutPtr;    // pointer to LUT
unsigned short displayVal;

displayVal = LUT[ __max(0, __min(*pixelVal-FirstMappedValue, LutNEntries-1))];
```

The LutPtr points to the lookup table which is of type u8 (unsigned 8 bit) or u16 (unsigned 16 bit) as described by the LutType.

The first entry in the LUT is for pixel value FirstMappedValue and the last entry in the LUT is for the pixel value LastMappedValue. The LutMin and LutMax defines the minimum and maximum values present in the LUT.



Note: It is suggested the input image and the generated LUT are stored together for display as described in *Appendix A - Tuning Panel Types*, section *A.1*

3.6.12 SCPPGetImageMinMax()

This function can be used to find the minimum and maximum pixel value in an image. The method is typically used to get the minimum and maximum pixel value in the processing result. This information is sometimes needed e.g. when updating the image header in the result image.

```
// C
SCPPError SCPPGetImageMinMax(SCPPHandle handle,
                             void *indata, int hgh, int wid, const char *type,
                             float *min, float *max);
// C++
SCPPError CSCPP::GetImageMinMax(void *indata, int hgh, int wid, const char *type,
                                float &min, float &max);
```

C Parameters

`handle` A handle created with `SCPPCreateEx()`.

C/C++ Parameters

`indata` A pointer to an image.

`hgh, wid` Height and width (in pixels) of the image.

`type` Data type of the image pixels. Possible types are “u16”, “s16” and “u8”.

`min, max` The minimum and maximum pixel values are returned in these output parameters.

See the *3.1 A Simple Usage Scenario* for further information about how to use this function.

3.6.13 Tuning Panels

A parameter file setting can contain a set of user changeable parameters. The parameters can be used to modify the processing result. Such a set of parameters is called a tuning panel. The information about the tuning panel for a specific setting can be retrieved after a parameter file has been loaded into the library. The tuning panel information can be used to build a graphical user interface which enables the user to change parameters for the processing. The Windows version of SDK includes a small example of how the tuning panel information can be used to build a graphical user interface. The console example program also includes example of how to use the tuning panel functions.

The different types of tuning panels are described in *Appendix A - Tuning Panel Types*.

3.6.13.1 SCPPGetTuningPanelInfo()

The function returns information about the tuning panel for the selected parameter block and setting. The returned information includes the panel name, panel information and information about each of the available tuning panel parameters. The parameter information includes the name, type and valid ranges for the parameter values.

```
// C
SCPPError SCPPGetTuningPanelInfo(SCPPHandle handle, const char *parameterBlockName, int setting,
                                 const TSCPPTuningPanelInfo **tpInfo);

// C++
SCPPError CSCPP::GetTuningPanelInfo(const char *parameterBlockName, int setting,
                                    const TSCPPTuningPanelInfo *&tpInfo);
```

C Parameters

`handle` A handle created with `SCPPCreateEx()`.

C/C++ Parameters

`parameterBlockName`
The name of the parameter block.

`setting` Parameter setting number to use.

`tpInfo` The address/reference to a pointer to a TSCPPTuningPanelInfo structure. The pointer is set to point at the tuning panel information.

The TSCPPTuningPanelInfo structure returned by GetTuningPanelInfo is described below.

```
// The tuning panel information struct
typedef struct {
              const char *panelName;                    // Name of tuning panel
              const char *panelInfo;                    // Tuning panel information
              int nParameters;                          // Number of tuning panel parameters
              TSCPPTPParameterInfo *parameterInfo;      // Linked list of TSCPPTPParameterInfo structs
} TSCPPTuningPanelInfo;

// The TSCPPTPParameterInfo struct
typedef struct TSCPPTPParameterInfo {
              const char *name;                         // Name of the parameter
              const char *type;                         // "float","int","enum","string",...
              const char *min;                          // Minimum value for type "int" and "float"
              const char *max;                          // Maximum value for type "int" and "float"
              const char *possibleValues;// Possible values for "enum", comma separated values
              TSCPPTPParameterInfo *pNext;              // Pointer to next TSCPPTPParameterInfo struct
} TSCPPTPParameterInfo;
```

NOTE: The `SCPPGetTuningPanelInfo()` function only returns a pointer to a “library internal” information structure. The internal information structure is not valid after a new call to `SCPPReadParameterFile()` or `CSCPP::ReadParameterFile()`. The tuning panel information pointer MUST be updated with a new call to `SCPPGetTuningPanelInfo()` when a new parameter file has been loaded.

3.6.13.2 SCPPSetTuningPanelValue()

The function is used to change a tuning panel parameter value.

```
// C
SCPPError SCPPSetTuningPanelValue(SCPPHandle handle, const char *parameterBlockName, int setting,
                                  const char *parameterName, const char *value);
// C++
SCPPError CSCPP::SetTuningPanelValue(const char *parameterBlockName, int setting,
                                     const char *parameterName, const char *value);
```

C Parameters

`handle` A handle created with `SCPPCreateEx()`.

C/C++ Parameters

`parameterBlockName`
The name of the parameter block.

`setting` Parameter setting number to use.

`parameterName`
The name of the tuning panel parameter.

`value` The new value for the tuning panel parameter. Note that the value should be passed as a character string regardless of the parameter type.

Any changes that have been made to tuning panel parameter values can be saved as a new parameter file. See section *3.6.7* for further information.

3.6.13.3 SCPPGetTuningPanelValue()

The function is used to get a tuning panel parameter value.

```
// C
SCPPError SCPPGetTuningPanelValue(SCPPHandle handle, const char *parameterBlockName, int setting,
                                  const char *parameterName, const char **value);
// C++
SCPPError CSCPP::GetTuningPanelValue(const char *parameterBlockName, int setting,
                                     const char *parameterName, const char *&value);
```

C Parameters

`handle` A handle created with `SCPPCreateEx()`.

C/C++ Parameters

`parameterBlockName`
The name of the parameter block.

`setting` Parameter setting number to use.

`parameterName`
The name of the tuning panel parameter.

`value` The address/reference to a character pointer which is set to point to the parameter value. Note that the character string for the parameter value is allocated by the library. The pointer is only valid until the next call to the library. This means that the value should not be used after any new calls have been made to the library.
The value is always returned as a character string regardless of the type of the tuning panel parameter. The caller has to convert the value if necessary.

3.6.14 Error Logging

The SCPP library logging can be enabled when debugging program applications. The logging should only be used during debugging since it slows down processing. Make sure to disable any logging when the debugging has been finished.

The logging status (enabled/disabled) is stored permanently in the system which means that all users' applications will be using the same logging information.

Note: In Windows, the license information and logging status is stored in a common location. I.e. all users will have the same setting. In Linux, the setting is normally stored in the users' home directory, i.e. all users will have their private setting. See *5 Linux storage location configuration* for more information.

The logging can be controlled directly with library function calls or by using the SCPPTest.exe program which is delivered with the SDK.

3.6.14.1 Get current logging status using SCPPTest

The following example shows how to display the current logging status using SCPPTest.

```
> SCPPTest -L info
           LogDirectory:
              LogErrors: 0
      LogErrorsAndCalls: 0
LogNbrOfProcessedImages: 0
```

`LogDirectory`
This is the directory in which a subdirectory with the name of the SDK is created. All logging information will be stored in this subdirectory.

`LogError`
This value is 1 if all failing function calls should be logged (set to 0 to disable).

`LogErrorsAndCalls`
This value is 1 if all function calls should be logged (set to 0 to disable).

`LogNbrOfProcessedImages`
This value defines how many processing input and output images that should be saved in the log directory (set to 0 to disable). Example: Setting the value to 5 means that the 5 latest input/output images are stored. All function calls will be logged automatically if input/output images are stored.

3.6.14.2 Enable logging using SCPPTest

The following example shows how to enable error logging of failing function calls using SCPPTest.

```
> SCPPTest -L enable;D:\Temp  (Linux: SCPPTest -L "enable;/tmp")
> SCPPTest -L info
           LogDirectory: D:\Temp
              LogErrors: 1
      LogErrorsAndCalls: 0
LogNbrOfProcessedImages: 0
```

The following example shows how to enable logging of all function calls.

```
> SCPPTest -L enable;D:\Temp;1  (Linux: SCPPTest -L "enable;/tmp;1")
> SCPPTest -L info
           LogDirectory: D:\Temp
              LogErrors: 1
      LogErrorsAndCalls: 1
LogNbrOfProcessedImages: 0
```

The following example shows how to enable logging of all function calls and enabled logging of the 3 latest input/output images.

```
> SCPPTest -L enable;D:\Temp;1;3   (Linux: SCPPTest -L "enable;/tmp;1;3")
> SCPPTest -L info
           LogDirectory: D:\Temp
              LogErrors: 1
      LogErrorsAndCalls: 1
LogNbrOfProcessedImages: 3
```

3.6.14.3 Disable logging using SCPPTest

The following example shows how to disable all logging using SCPPTest.

```
> SCPPTest -L disable
> SCPPTest -L info
           LogDirectory: D:\Temp
              LogErrors: 0
      LogErrorsAndCalls: 0
LogNbrOfProcessedImages: 0
```

The library functions for logging are described on the following pages.

3.6.14.4 SCPPSetLogging()

This function is used to set the current logging status.

```
// C
SCPPError SCPPSetLogging(SCPPHandle handle,
                         const char *logDirectory,
                         int logErrors, int logAllCalls, int logNbrOfProcessedImages);
// C++
SCPPError CSCPP::SetLogging(const char *logDirectory,
                         int logErrors, int logAllCalls, int logNbrOfProcessedImages);
```

C Parameters

`handle` A handle created with `SCPPCreateEx()`.

C/C++ Parameters

`logDirectory`

The logging directory.

`logErrors`
: Set to 1 to enable logging of failing function calls. Set to 0 to disable.

`logAllCalls`
: Set to 1 to enable logging of all function calls. Set to 0 to disable.

`logNbrOfProcessedImages`
: Set to >0 to enable storing of input and output images. Set to 0 to disable.

See sections 3.6.14.1 - 3.6.14.3 for further information about the meaning of the parameters.

3.6.14.5 SCPPGetLogging()

The function is used to get the current logging status.

```
// C
SCPPError SCPPGetLogging(SCPPHandle handle,
                         const char **logDirectory,
                         int *logErrors, int *logAllCalls, int *logNbrOfProcessedImages);
// C++
SCPPError CSCPP::GetLogging(const char *&logDirectory,
                         int &logErrors, int &logAllCalls, int &logNbrOfProcessedImages);
```

C Parameters

`handle` A handle created with `SCPPCreateEx()`.

C/C++ Parameters

`logDirectory`
: A caller supplied buffer (of size MAX_PATH) in which the current logging directory is returned.

`logErrors`
`logAllCalls`
`logNbrOfProcessedImages`
: The current values for these settings are returned in these output parameters.

See sections 3.6.14.1 - 3.6.14.3 for further information about the meaning of the parameters.

3.6.15 License Handling Methods

A number of C functions have been defined to make it easy to integrate license handling in the application using the SCPP library.

Note: In Windows, the license information and logging status is stored in a common location. I.e. all users will have the same setting. In Linux, the setting is normally stored in the users' home directory, i.e. all users will have their private setting. See *5 Linux storage location configuration* for more information.

3.6.15.1 SCPPLicenseGetRequestInfo()

The function returns the license request information which should be included in a license request that is sent to Sapheneia.

```
// C
SCPPError SCPPLicenseGetRequestInfo(SCPPHandle handle, const char **requestInfo);

// C++
SCPPError CSCPP::LicenseGetRequestInfo(const char **requestInfo);
```

C Parameters

`handle` A handle created with `SCPPCreateEx()`.

C/C++ Parameters

`requestInfo` The address to a character pointer. The character pointer points to the license request information after the call.

3.6.15.2 SCPPLicenseGetHidInfo()

The function returns information about the available host ids, their type and a short description of the hid.

```
// C
SCPPError SCPPLicenseGetHidInfo(SCPPHandle handle, TSCPPHidInfo *hidInfo);

typedef struct TSCPPHidInfo {
  char Hids[SCPP_MAX_NHIDS][SCPP_MAX_HID_LEN];                    // Array of Hids
  char HidTypes[SCPP_MAX_NHIDS][SCPP_MAX_HIDTYPE_LEN];            // Array of Hid types
  char HidDescriptions[SCPP_MAX_NHIDS][SCPP_MAX_HID_DESCR_LEN];   // Array of Hid descriptions
  int  nHids;                                                     // Number of elements in arrays
} TSCPPHidInfo;

// C++
SCPPError CSCPP::LicenseGetHidInfo(TSCPPHidInfo &hidInfo)
```

C Parameters

`handle` A handle created with `SCPPCreateEx()`.

C/C++ Parameters

`hidInfo` The structure that is filled with hid information. The *nHids* value in the structure defines how many valid entries there are in the arrays.

3.6.15.3 SCPPLicenseCheck()

The function checks the license for the specified product name. An error is returned if the license is not valid.

```
// C
SCPPPError SCPPLicenseCheck(SCPPHandle handle,
                            const char *productName, const char **licenseInfo);
// C++
SCPPError CSCPP::LicenseCheck(const char *productName, const char **licenseInfo);
```

C Parameters

`handle` A handle created with `SCPPCreateEx()`.

C/C++ Parameters

`productName` The name of the product for which the license should be checked. Set the value to 0 (null pointer) to check the license for the library itself.

`licenseInfo` The address to a character pointer. The character pointer points to information about the license after the call.

3.6.15.4 SCPPLicenseSetKey()

The function sets and checks a license key for the specified product. An error is returned if the license key is not valid. The license key is stored in the computer which means that the license key only needs to be set once.

```
// C
SCPPError SCPPLicenseSetKey(SCPPHandle handle,
                            const char *productName,
                            const char *key, const char **licenseInfo);
// C++
SCPPError CSCPP::LicenseSetKey(const char *productName,
                               const char *key, const char **licenseInfo);
```

C Parameters

`handle` A handle created with `SCPPCreateEx()`.

C/C++ Parameters

`productName` The name of the product for which the license key should be set. Set the value to 0 (null pointer) to set the license key for the library itself.

`key` The license key obtained from Sapheneia.

`licenseInfo` The address to a character pointer. The character pointer points to information about the license after the call.

3.6.15.5 SCPPLicenseDelete()

The function can be used to delete a license key for the specified product.

```
// C
SCPPError SCPPLicenseDelete(SCPPHandle handle, const char *productName);

// C++
SCPPError CSCPP::LicenseDelete(const char *productName);
```

C Parameters

`handle` A handle created with `SCPPCreateEx()`.

C/C++ Parameters

`productName` The name of the product for which the license key should be deleted. Set the value to 0 (null pointer) to delete the license key for the library itself.

3.7 Using Graphics Board (GPU) (Windows Only)

The Windows SDK supports processing on GPU boards via DirectX 9c. This means that the processing is not tied to a specific brand of the GPU. The processing performance depends on the selected GPU board. More processing pipelines and higher clock frequency gives higher performance. The performance on a specific GPU board has to be tested for the specific image sizes and parameter settings.

The amount of memory needed on the GPU board is dependent on the image and the selected processing setting. Small CT and ultrasound images might only need 256 MB of GPU board memory while larger XR images might require 1 GB of memory or even more.

The DirectX 9c runtime must be installed to be able to use processing in the GPU. An error message about missing DirectX dynamic load library will be returned if the DirectX runtime is not installed.

The DirectX 9 library may return a 'lost device' error while processing. As in all applications using DirectX this is something that must be handled by the application that is using the SDK. Section 3.7.4 describes how to recover from a 'lost device' error.

A number of methods are available to control the use of GPU for processing. These are described below.

3.7.1 SCPPHasGPUSupport()

This function is used to check if GPU processing is supported by the SCPP library. The function does not check if DirectX 9 is installed or if processing is possible on the GPU board.

```
// C
SCPPError SCPPHasGPUSupport(SCPPHandle handle, int *bHasGPUSupport);

// C++
SCPPError CSCPP::HasGPUSupport (int &bHasGPUSupport);
```

C Parameters

`handle` A handle created with `SCPPCreateEx()`.

C/C++ Parameters

`bHasGPUSupport`

A non-zero value is returned if GPU processing is supported by the SCPP library.

3.7.2 SCPPSetUseGPU()

The function is used to enable and disable processing in the GPU. This is the only function that has to be called to redirect the processing to the GPU.

```
// C
SCPPError SCPPSetUseGPU(SCPPHandle handle, int bUseGPU, int bForceCleanUp, int displayAdapter);

// C++
SCPPError CSCPP::SetUseGPU(int bUseGPU, int bForceCleanUp, int displayAdapter);
```

C Parameters

`handle` A handle created with `SCPPCreateEx()`.

C/C++ Parameters

`bUseGPU`
: A nonzero value enables processing in the GPU. Some initialization is performed the first time the use of the GPU is enabled.

`bForceCleanUp`
: This flag is only used when disabling the GPU processing. A nonzero value will tell the library to release all resources associated with the GPU. This means that the enabling again requires a new GPU initialization.

`displayAdapter`
: The number of the display adapter to be used. This value should be 0 for a system with only one GPU.

3.7.3 SCPPGetUseGPU()

This function is used to check if GPU processing is currently enabled.

```
// C
SCPPError SCPPGetUseGPU(SCPPHandle handle, int *bUseGPU);

// C++
SCPPError CSCPP::GetUseGPU(int &bUseGPU);
```

C Parameters

`handle` A handle created with `SCPPCreateEx()`.

C/C++ Parameters

`bUseGPU`

The return value from the function. A non zero value is returned if GPU processing is enabled.

3.7.4 How to handle 'Lost Device' error

DirectX function calls might return a 'Lost Device' error. This could happen if another application uses DirectX in full screen mode or if a screen saver is activated.

A special error code, `SCPP_ERR_GPU`, is returned from an SCPP library call if this occurs. This means that the application must recover from the situation.

3.7.4.1 Recovery when using SCPP_PROCESSING_SINGLE_FRAME

The recovery is very simple when the processing type of the selected setting is SCPP_PROCESSING_SINGLE_FRAME. The image just needs to be processed again. See code example below.

```
// SCP_ERR_GPU error handling when processing type is SCPP_PROCESSING_SINGLE_FRAME

TSCPPProcessingInfo processingInfo;

scppStatus = SCPPGetProcessingInfo(handle,parameterBlockName, setting, &processingInfo);
if ( scppStatus != SCP_ERR_OK)
  // Error handling

if( processingInfo.processingType == SCPP_PROCESSING_SINGLE_FRAME) {

  do {

    scppStatus = SCPPProcessImage(handle, … parameterBlockName, setting);

  } while ( scppError == SCP_ERR_GPU );

  if( scppStatus != SCPP_ERR_OK) {
    // Normal error handling
  }

  // Save processing result
}                                        }
```

3.7.4.2 Recovery when using SCPP_PROCESSING_SEQUENCE

The recovery becomes a little bit more complicated when the processing type is SCPP_PROCESSING_SEQUENCE since the processing of the current image depends on earlier images in the sequence. The recovery can be made in one of the following ways:

1. The simplest solution would be to reprocess the entire sequence of images when the error occurs.
2. It is often not possible to reprocess the entire sequence since all previous images might not be available anymore.
   If some previous images are available then the sequence processing can restart from a few images earlier in the sequence. Where to restart depends on the nInputImages value in the TSCPPProcessingInfo structure returned by the GetProcessingMethod() described in section 3.6.8.
   a. If the nInputImages value is larger than 1 then processing of a new sequence can start at current image - nInputImage.
   b. If the nInputImages value is 1 then processing of a new sequence can start at current image - 3 in the sequence.

If no previous images are available then the sequence processing has to restart at the current image. In this case the processing result for the current image and possibly a few more will not be exactly as if the entire sequence would have been processed without interruption.

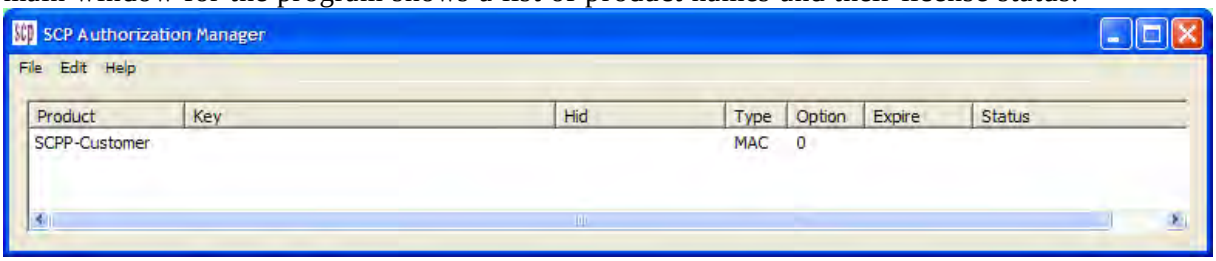
It is obviously more effective to use alternative 2 above even when the entire sequence of images is available. The pseudo code below describes how to recover.

```
// SCP_ERR_GPU error handling when processing type is SCPP_PROCESSING_SEQUENCE
TSCPPProcessingInfo processingInfo;
scppStatus = SCPPGetProcessingInfo(handle,parameterBlockName, setting, &processingInfo);
if ( scppStatus != SCP_ERR_OK)
  // Error handling

if( processingInfo.processingType == SCPP_PROCESSING_SEQUENCE) {
  int imageNbr=0;

  while( imageNbr < TotalNbrOfImagesInSequence) {
    int isFirstImageInSequence = imageNbr==0;

    do {
      // setup input image vector to contain images
      // imageNbr to imageNbr+processingInfo.nInputImages-1
      if(isFirstImageInSequence )
         scppStatus = SCPPProcessImageSequneceFirst(handle, <input vector> …
                                                    parameterBlockName, setting);
      else
         scppStatus = SCPPProcessImageSequneceNext(handle, <input vector> …
                                                    parameterBlockName, setting);
      if ( scppError == SCP_ERR_GPU ) {
        if ( earlierImagesAreAvailable) // Start a few images earlier
          imageNbr = max(0, imageNbr-(processingInfo.nInputImages==1 ? 3:
                                          processingInfo.nInputImages));
        isFirstImageInSequence=1;
      }
    } while( scppStatus == SCP_ERR_GPU ) ;

    if( scppStatus != SCPP_ERR_OK) {
      // Normal error handling
    }

    // Save result image # imageNbr
  }
}                                                    }
```

4 Licensing with SCP Authorization Manager (Windows only)

Note: The SCP Authorization Manager is only available for Windows. In Linux the license can be set as described in chapter *2 Licensing* or by using the licensing handling functions described in section *3.6.15*.

The SCP Authorization Manager program is an alternative to the licensing functions in the SCPP library. The program makes it easy to view, install and delete license keys for SCP products. The main window for the program shows a list of product names and their license status.



4.1 License Request

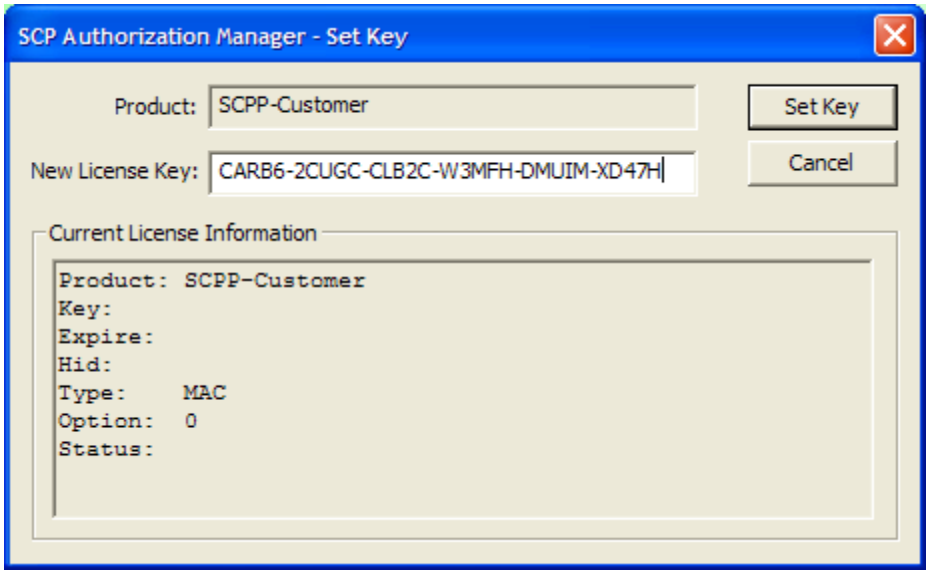
The license request information should be sent to Sapheneia to get a license key for the product. The license request information for a product can be retrieved by selecting the product name in the list and then selecting **Edit->License request…**

4.2 Setting the license key

The license key might be delivered as a character string or embedded in a license file. The installation of the key differs depending on the delivered format.

4.2.1 Installing a license key

A license key is installed in the **Set Key** window. This window is displayed by double clicking on the product name.



Enter the license key in the window and then press the **Set Key** button. The license key status is displayed in the main window when the license key has been set.

4.2.2 Installing a license key file

License key files (slf-files) can contain one or more license keys. These files can be installed by drag-and-drop. Just drag the slf-file from the file manager and drop it in the SCP Authorization Manager window.

4.3 Licensed parameter files

Parameter files can contain product names for which a license must be installed to be able to use the parameter file. It is possible to drop such parameter files on the SCP Authorization Manager to display the embedded product name in the product list.

4.4 The SCP Authorization Manager Menus

This section describes the different choices in the program menus.

File menu

`Import License File…`

Opens a file dialog window where license files (slf) and parameter files (spf) can be selected and imported. Any product names and license keys in the files are inserted in the product list. This is equivalent to drop license or parameter files on the SCP Authorization Manager.

`Save Selected License…`

The selected product names and their license keys are saved in the specified slf-file.

`Rockey4ND`

This sub-menu contains some functions to handle Rockey4ND USB dongles.

- `Import and set Rockey4ND Licenses`
  This function is used to import a license files containing multiple license keys for a number of Rockey4ND dongles. Rockey4ND dongles can then be inserted one at the time and then the license keys for the dongle is fetched from the license file and stored in the dongle.
- `Save Rockey4ND id to file`
  This function is used to store the ids for a number of Rockye4ND dongles to a file.

`Rockey2`

This sub-menu contains some functions to handle Rockey2 USB dongles.

- `Import and set Rockey2 Licenses`
  This function is used to import a license files containing multiple license keys for a number of Rockey2 dongles. Rockey2 dongles can then be inserted one at the time and then the license keys for the dongle is fetched from the license file and stored in the dongle.
- `Save Rockey2 id to file`
  This function is used to store the ids for a number of Rockye2 dongles to a file.

Edit menu

`Set License Key…`

Displays the **Set Key** window for the selected product name. This is equivalent to double-clicking on the product name.

`License Request…`

Displays the **License Request Information** window.

`Delete Selected Licenses…`

Delete the selected license keys for the selected products. The product name itself is deleted if it does not have any key.

`Add New Product…`

Display the **Add New Product** window. This window makes it possible to add a new product name by manually entering it.

Help menu

`About…` Display the version of the SCP Authorization Manager.

Product List Menu

The product list has a context menu which can be displayed by a right-click in the list. This menu includes some of the same items as the **File** and **Edit** menus.

5 Linux storage location configuration

In Linux, the license information and logging status information is normally stored in the current users' home directory as defined by the users HOME environment variable. This stored information is private for the current user. This means that each user, who will use the license, must set up the license key.

It is however possible to define another directory where the license and logging status information should be stored and shared between users. This is done by setting the SCP_HOME environment variable to the path of the selected directory. The directory must have full access (rwx) for all users.

Note: In Windows the license information and logging status information is always saved at a common location. I.e. no information is private to a specific user.

Appendix A - Tuning Panel Types

Tuning panels contain a number of user changeable parameters that controls the processing for a specific parameter setting. How to get information about a tuning panel for a specific parameter setting is described in *3.6.13 Tuning Panels*.

The tuning panels can be of different types with different sets of changeable parameters. The `panelName` parameter of the `TSCPPTuningPanelInfo` structure tells you what type of tuning panel the parameter setting has. The different types are described below.

A.1 Ultrasound Tuning Panel

For ultrasound tuning a special tuning panel type can be attached to parameter settings. The tuning panel is used to control a number of high level processing options.

This is an example of the information returned by the `SCPPGetTuningPanelInfo()` function for a parameter setting containing a typical ultrasound tuning panel.

```
Tuning panel name: Standard-A
Panel information: Created on 20-Dec-2012
-------------------------------------------------------------------------------
Parameter                  Type             Value           Range
-------------------------------------------------------------------------------
MinimumIntensity            float            0               [-32768..65535]
MaximumIntensity            float            255             [-32768..65535]
DarkFluid                   int32            40              [0..100]
I1_Position                 int32            10              [0..100]
I1_NoiseNearField           int32            0               [-100..100]
I1_SharpenNearField         int32            0               [-100..100]
I1_ContrastNearField        int32            0               [-100..100]
I1_EffectNearField          int32            85              [0..100]
I2_Position                 int32            40              [0..100]
I2_NoiseFocus               int32            0               [-100..100]
I2_SharpenFocus             int32            0               [-100..100]
I2_ContrastFocus            int32            0               [-100..100]
I2_EffectFocus              int32            80              [0..100]
I3_Position                 int32            80              [0..100]
I3_NoiseFarField            int32            0               [-100..100]
I3_SharpenFarField          int32            0               [-100..100]
I3_ContrastFarField         int32            0               [-100..100]
I3_EffectFarField           int32            70              [0..100]
-------------------------------------------------------------------------------
```

Tuning Panel Parameters

`MinimumIntensity, MaximumIntensity`
: These parameters should usually not be modified for ultrasound data.

`DarkFluid`
: Controls the amount of intensity reduction for dark areas in percent, 100 will give the maximum blackening of vessels available for the particular parameter setting at hand.

For each of the available areas (indicated by the prefix Ix, typically corresponding to NearField, Focus and FarField) there are five controls:

`Position`
: Controls where the area is assumed to be in percent from the start (could be the left or the top depending on how the data is stored). In the above example the areas are centered at 10%, 40% and 80% from the start respectively. Changing the position values moves the boundaries between the different areas.

`Noise`
: Controls the amount of smoothing relative to the default for that parameter setting. Lower values mean less noise, i.e. more smoothing.

`Sharpen`
: Controls how much higher frequencies are enhanced, this is different but not independent of noise and the two controls can to some extent take each other out.

`Contrast`
: Controls the enhancement of lower frequencies (larger structures).

`Effect`
: Controls processing strength per area. Zero turns of the processing but this might not always be totally true as there is a smooth transition between the areas.

A.2 X-Ray Tuning Panel

For x-ray images, including fluoroscopy, a special tuning panel type can be attached to parameter settings. The tuning panel is used to control a number of high level processing options.

This is an example of the information returned by the `SCPPGetTuningPanelInfo()` function for two parameter settings containing typical x-ray tuning panels.

```
Tuning panel name: Standard-A
Panel information: Created on 06-Aug-2013
-------------------------------------------------------------------------------
Parameter                    Type             Value            Range
-------------------------------------------------------------------------------
RescaleIntercept             float            0                [-32768..65535]
RescaleSlope                 float            1                [0..1000]
MinimumIntensity             float            0                [-32768..65535]
MaximumIntensity             float            65535            [-32768..65535]
Invert                       bool             false
DRC                          int32            70               [0..100]
I1_Noise                     int32            0                [-100..100]
I1_Sharpen                   int32            0                [-100..100]
I1_Contrast                  int32            0                [-100..100]
I1_Effect                    int32            76               [0..100]
-------------------------------------------------------------------------------
```

```
-------------------------------------------------------------------------------
Tuning panel name: Standard-B
Panel information: Created on 17-Mar-2014
-------------------------------------------------------------------------------
Parameter                    Type             Value            Range
-------------------------------------------------------------------------------
RescaleIntercept             float            0                [-32768..65535]
RescaleSlope                 float            1                [0..1000]
MinimumIntensity             float            0                [-32768..65535]
MaximumIntensity             float            65535            [-32768..65535]
Invert                       bool             false
DRC                          int32            70               [0..100]
I1_DeNoiseLevel              int32            0                [-100..100]
I1_DeNoiseEffect             int32            63               [0..100]
I1_Sharpen                   int32            0                [-100..100]
I1_Contrast                  int32            0                [-100..100]
I1_Effect                    int32            76               [0..100]
-------------------------------------------------------------------------------
```

Tuning Panel Parameters

There are several parameters that control how the input data is handled which can be used to adjust to different types of input data. However, typically Sapheneia will deliver parameter settings that are adjusted to the intended data and the following parameters can be ignored:

`RescaleIntercept, RescaleSlope`
: Can be used to adjust the range of the input data prior to processing, for instance to convert signed data into unsigned data. These parameters are not often used for x-ray images.

`MinimumIntensity, MaximumIntensity`
: These parameters control the input range for the processing, values outside the range will be saturated before processing. These values should be adapted to the input range, for example MaximumIntensity=4095 for 12bit data.

`Invert`
: Determines if the data is to be inverted (using MinimumIntensity and MaximumIntensity) before processing. This can be used if the same parameter setting

is to be used on data stored using Monochrome1 and Monochrome2 conventions.

In addition to the data handling parameters above there are several control parameters to be used to fine-tune the processing. Please note that not all controls are available in all parameter files:

`DRC`
: Controls the amount of dynamic range compression, sometimes also called latitude compression or equalization. Higher values result in more homogenous background.

`I1_DeNoiseLevel`
: Controls the level (what kind of variation is considered noise) of de-noising relative to the default for that parameter setting. Lower values mean less noise reduction.

`I1_DeNoiseEffect`
: Controls the amount of de-noising from 0 to 100%.

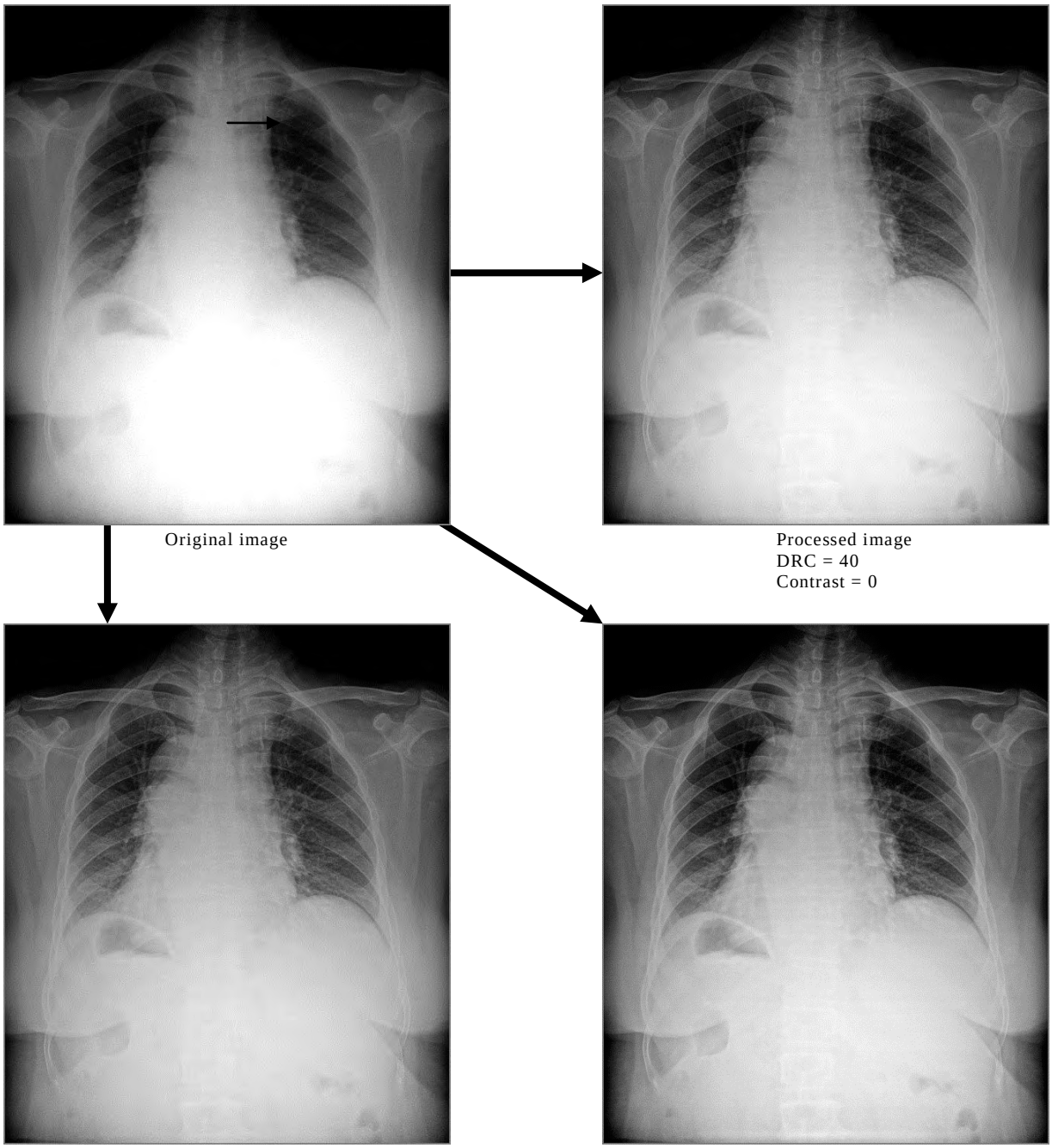
`I1_Noise`
: Controls the amount of smoothing relative to the default for that parameter setting. Lower values mean less noise, i.e. more smoothing.

`I1_Sharpen`
: Controls how much higher frequencies are enhanced, this is different but not independent of noise and the two controls can to some extent take each other out.

`I1_Contrast`
: Controls the enhancement of lower frequencies (larger structures) resulting in increased or decreased contrast, again relative to the default used in the particular setting.

`I1_Effect`
: Controls processing strength ranging from 0 to 100% where zero turns off the processing.

The example below illustrates the behavior of the DRC and contrast processing controls:



Original image

Processed image
DRC = 40
Contrast = 0

Processed image
DRC = 70
Contrast = 0

Processed image
DRC = 70
Contrast = 80

A.3 Lut-A Tuning Panel

This tuning panel type is only attached to parameter settings with processing type SCPP_PROCESSING_LUT to be used with the `SCPPCreateLut()` function. The tuning panel is used to control how the generated LUT is created.

This is an example of the information returned by the `SCPPGetTuningPanelInfo()` function for a parameter setting containing a Lut-A tuning panel.

```
Tuning panel name: Lut-A
Panel information:
--------------------------------------------------------------------------------
Parameter                    Type             Value            Range
--------------------------------------------------------------------------------
Mode                         enum             Auto             [Auto,Manual]
ROIX                         float            0.4              [0..1]
ROIY                         float            0.4              [0..1]
ROIWidth                     float            0.2              [0..1]
ROIHeight                    float            0.2              [0..1]
SCenter                      float            0.5              [0..1]
SWidth                       float            0.5              [0..1]
ContrastValue                float            1.5              [0.1..1000]
--------------------------------------------------------------------------------
```

The information above was retrieved using the command: `SCPPTest -p ..\par\testLut.spf;default;0;tuningpanel`

Tuning Panel Parameters

`Mode`

This parameter can have one of two values:

- Auto – The SCPPCreateLut() function will create a LUT using the specified region of interest (ROI) and ContrastValue. The SCenter and SWidth parameters are updated to the new values of the generated LUT.
- Manual – The SCPPCreateLut() function will use the specified SCenter, SWidth and ContrastValues to generate a LUT.

`ROIX, ROIY, ROIWidth, ROIHeight`

Defines the region of interest when using Mode=Auto. The values are defined between 0 and 1.

`SCenter, SWidth`

These values are generated when using Mode=Auto.

Note: The values have to be retrieved using the `SCPPGetTuningPanelValue()` function after the call to `SCPPCreateLut()` to get the generated values.

`ContrastValue`

This value can be used in both auto and manual mode to fine tune the LUT.

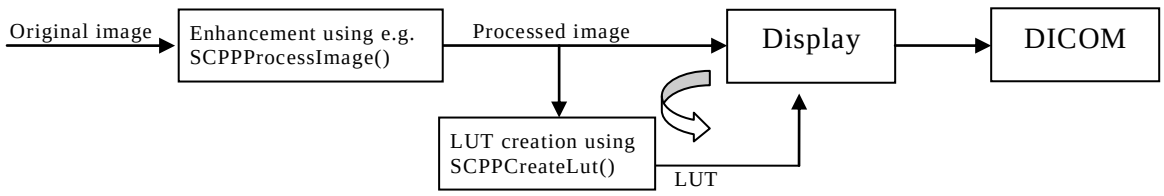
How it works

The idea is to use the Mode=Auto to generate a LUT using the specified region of interest (ROI) in the image. In the example above the ROI is located in the middle of the image. The ROI should contain examples of the important structures that the LUT should make visible. The ROI can be changed to another position and size to create a better LUT for the image. The ContrastValue can also be changed to fine-tune the LUT.

Using Mode=Manual can be used after using Mode=Auto when a satisfactory LUT was not generated. In the manual mode the auto generated SCenter and SWidth values can be changed to adjust brightness and contrast in the generated LUT. The ContrastValue can be changed in both manual and auto mode. The ROI is not used in the manual mode.

Suggested usage scheme

1. The original image is processed using anatomy specific parameter file setting.
2. A LUT is generated using Mode=Auto in the Lut-A tuning panel.
3. The processed image is displayed using the LUT.
4. If necessary, a new ROI is defined and step 2 and 3 are repeated.
5. The processed image is stored as DICOM, including the generated LUT in the header.



Using the suggested scheme makes it possible to use alternative display methods for the processed image since the processed pixel data is always available.