VM_LIB MACHINE-VISION LIBRARY

User's Guide

23 May 2011

Welcome to the VM_LIB image-processing and machine-vision library. This manual describes how to use the library functions.

Before going on reading the manual, we kindly ask you to read the following

Conventions used in this manual:

INFORMATION. *This sign marks a section in the manual, which is for information only.*

ATTENTION. *This sign marks a section in the manual, which is particularly important for the general understanding of the document. Please, make sure to read this section before proceeding with next sections.*

TIPS & TRICKS. *This sign marks a Tips & Tricks section. Here you can find some practical advises on using the system or you can get a more detailed explanation of some features. Reading this section may help you in solving a particular problem and give you some ideas but is not of vital importance for understanding of the document.*

PREMISE. *This sign marks a section, which describes prerequisite condition(s) for some next operation(s).*

"1.1. About"	*Section reference. If the section is within the current manual no manual name is specified. When the section is within external manual the name of the respective manual is also included.*
NA	*Value not available or not calculated yet.*

CONTENTS

1. Introduction

The VM_LIB library is a set of C functions for image processing of data and computer vision applications. The library is intended to work on PC and intelligent cameras. You can create completely portable code, which can be compiled and tested on PC, and then compiled and executed on the camera.

The library functions are divided in the following groups (sub-libraries) according to their purpose:

Basic libraries:

Calculation and geometry library
Image processing library
Drawing library
General-purpose utility library
High-speed optimized vector function library

High-level libraries:

BLOB library
Edge-detection library
Normalized correlation (pattern matching) library
Object recognition library
Barcode recognition library

With a very few hardware-dependent exceptions, all functions have plain portable C code, therefore they can easily be ported on new hardware platforms and operating systems.

1.1. Basic philosophy of the VM_LIB library

All image-processing functions work on a gray-level video image. The video image usually is filled at the beginning of each system cycle with the captured camera picture. All drawing functions draw into a memory image. The drawing image usually is cleared at the beginning of each system cycle. The two images are usually displayed at the end of each system cycle on the camera screen as a combined gray-level/overlay picture.

Most of the library functions are created as black boxes – all input parameters and results are passed by the function call and you don't need any additional initializations to use the functions. You must however open the library once on program entry by **vm_lib_open** and close the library on program exit by **vm_lib_close**. Include one or more header files and link your project with the library.

Some VM_LIB need additional initialization. Usually you should call an open function, which allocates memory and creates handle. Pass this handle to other functions from the group and close the handle on exit. See 2.2. Using the library functions" for more details.

1.2. Example

The file **`TEMPL.C`** contains demo code, which gives an idea how to use the library functions. You should open the library by **vm_lib_open** at the beginning (this must be the first VM_LIB function call). Call **vm_lib_close** before return. Put image-processing, drawing and other functions inside.

```
/****** TEMPL.C = VM_LIB demo code = 11.2010 ****************************/
/*
* This file contains a demo code, which shows how to use VM_LIB library
* functions.
*/

#include <math.h>
#include <string.h>
#include "vm_lib.h"

    int      rc = 0;        /* return code                      */
    VD_IMAGE vd_img;        /* video image (gray-level picture) */
    DR_IMAGE dr_img;        /* drawing image (overlay picture)  */
    unsigned long time;     /* execution time (ms)              */
    char     str[80];
    int      vm_key = 12345;    /* VM_LIB license key */

/*----------------------------------------------------------------------*/
/*                               Open VM_LIB                            */
/*----------------------------------------------------------------------*/
    rc = vm_lib_open(vm_key);
    if(rc) goto done:

/* Clear video image (recommended but not obligatory) */
    ip_clear_img(&vd_img);

/*----------------------------------------------------------------------*/
/*                            Execution loop                            */
/*----------------------------------------------------------------------*/
    for(;;)
    {
      time = sys_time();

/* Clear the drawing image on each loop cycle */
      ip_clear_img(&dr_img);

/* Capture image and store it into vd_img */
      . . . . . . . . . . .
/*
* Put image-processing and drawing code here. Pass 'vd_img' to the image-
* processing functions. Pass 'dr_img' to the drawing functions. Some
* VM_LIB functions need both 'vd_img' and 'dr_img'.
*/

/* Perform image pyramid */
      ip_img_pyramid(&vd_img, &vd_img);

/* Binarize the pyramid image */
      rc = ip_img_binar (&vd_img, 0, 128, 0, 255);
      if(rc) goto done;

      time = sys_time() - time;

/* Draw text at (0,0) with fgnd and bgnd color */
      sprintf(str, "Success: Execution time = %d ms", (int)time)
      rc = dr_text(&dr_img, str, 0, 0, DR_COLOR_BRIGHT_GREEN, DR_COLOR_NONE);
      if(rc) goto done;

/* Other code, break execution loop... */
      . . . . . . . . . . .
    } /* for(;;) - end of execution loop */

/*----------------------------------------------------------------------*/
/*                               Close VM_LIB                           */
/*----------------------------------------------------------------------*/
done:
    vm_lib_close();
    return;
```

1.3. Usage

The chapter describes how to use the VM_LIB library functions. You need a prerequisite experience in C programming.

Chapter ”6. Function reference” contains detailed description of the library functions

1.4. Header files

All library header files are placed in **VL\H**. The platform-dependent definitions are contained in **cdef.h**.

Header file	Description
bc_lib.h	Barcode library header
bl_lib.h	Public BLOB library header
cdef.h	Common system-dependent definitions (mem_alloc, VD_IMAGE, DR_IMAGE)
cl_lib.h	Calculation and geometry library header
dr_lib.h	Drawing library header
ip_lib.h	Image-processing library header
lib_err.h	VM_LIB library error codes
ncor.h	Normalized correlation library header
or_lib.h	Object recognition library header
sys_lib.h	System library header
ut_lib.h	Utility library header
vf_lib.h	Optimized vector-function library header
vm_lib.h	General VM_LIB header – includes all other header files

***TIPS & TRICKS**. It is enough to include **vm_lib.h** in your code. This header file includes all necessary header files.*

***ATTENTION**. Do not use macros, type definitions and function prototypes from the header file(s), which are not documented in this manual. They are intended for internal use only and may be changed without a notice.*

1.5. Using the library functions

As already mentioned in the introduction chapter, most of the library functions behave as black boxes. All you need is to pass input arguments and to get results by the function call. Most of the functions do not need any setup, just pick a function from the library, include the necessary header(s) and link with the VM_LIB library. Some functions and high-level tools however need setup as described in the next sections.

The image-processing functions receive gray-level images via the **VD_IMAGE** structure. The drawing functions draw into a **DR_IMAGE** structure. The **VD_IMAGE** and **DR_IMAGE** structures are defined in **CDEF.H**. Currently these image structures are equivalent to the **image** structure. Warning: These structures could be changed in future library releases to support other image types (currently 1 byte == 1 pixel).

1.5.1. VM_LIB open and close

The VM_LIB library needs to be opened and closed. To assure the correct operation of the library functions, you must call:

- **vm_lib_open** – once on program entry to open the library
- **vm_lib_close** – once on program exit to close the library

1.5.2. Tool handles

Some high-level VM_LIB functions (called tools) need to keep internal states and other data. These tools must be opened before usage. The open functions return handles, which must be passed to the tool processing functions. The tools must be closed on exit to free allocated memory:

- BLOB tools
- SIN/COS generator with integer calculations
- Object recognition tools

1.5.3. Other features

The following introductory sections in the library reference chapters describe other features, which could be useful when working with the VM_LIB library:

Calculation and geometry library:

Section “*6.3.1. Segment definitions of figures*”.

Section “*6.3.2. VM_LIB angles*”.

Section “*6.3.3. Scaled integers*”.

Drawing library:

Section “*6.4.1. Coordinate system*”.

Section “*6.4.2. VM_LIB colors*”.

Section “*6.4.3. Fonts*”.

Section “*6.4.4. Drawing origin*”.

Section “*6.4.5. Clipping*”.

1.6. Limitations

The VM_LIB library has the following limitations:

- no limitations except available heap memory

1.7. Copy protection

The VM_LIB library is copy-protected and only works in combination with VRmagic imaging devices. If the function vm_lib_open is not called with a valid VM_LIB license key at program start, all functions will return errors and crashes can occur. To create a valid VM_LIB license key, you have to call the following functions which are part of the VRmUsbCam API:

- **VRmCreateVMLIBKey(VRmDWORD* vm_key)**: has to be called to use VM_LIB on a PC (Linux or Windows) or on the ARM processor of a VRmagic intelligent camera
- **VRmCreateVMLIBKeyDsp(VRmDWORD* vm_key)**: has to be called to use VM_LIB on the DSP of a VRmagic intelligent camera. The key has to be created on the ARM and handed over to the DSP. The call to vm_lib_open has to be done on the DSP.

For more details, see the example projects “vm_lib_demo” and “vm_lib_demo_davinci” in the VRmUsbCam DevKit.

1.8. Execution speed

Here are some execution times, which show the speed of the VM_LIB library functions tested on different platforms.

1. Object recognition in 640x480 image. The table below gives pure search time in the SEARCH stage.

2. Normalized correlation on program generated image:
   - Pattern image = 100x100
   - Search image = 640x480

3. BLOB analysis in 640x480 image.
   - 36 objects detected.

Platform	Algorithm						
	Object recognition	Normalized correlation		BLOB analysis			
		Normal search	Fine search	Binarization & object segmentation	Contour calculation	Calculation of equivalent ellipses	Calculation of min area rectangles
3 GHz Pentium 4	94	16	60-80	16	16	16	16
997 MHz AMD	210-220	50	220-230	16	16	16	16
500 MHz VIA	688	200	920	32-47	16	16	16-31
400 MHz TMS320C6400	460-480	52-62	139-141	13	27	10	70
400 MHz ARM	720	187-190	857-863	26	87	151	290
600 MHz Blackfin DSP	770-817	146-148	430-434	51	42	37	317
VRmDC-9 BW (ARM, VRmDR1)	762	133-135	579-587	24	17	11	95
VRmDC-9 BW (DSP, VRmDR1)	183	92-93	302-304	12	27	5	46

(*) All execution times are in ms

2. Library installation

Please follow the instructions of the camera manufacturer. In general, the library file should be copied into a project's library folder and/or the project settings must be changed by appending the library file to the list with project libraries.

3. Compiling and linking programs

Please follow the instructions of the camera manufacturer. Link with the VM_LIB library when building the project.

4. Examples

5. Function reference

This chapter contains detailed description of the VM_LIB library functions. The VM_LIB library is divided into sub-libraries, containing functions with similar purpose. An attempt has been made to keep relatively independent VM_LIB sub-libraries with the aim to distribute the libraries independently. Note that functions from one sub-library may call functions from other one.

5.1. Barcode library

The barcode library reads several popular barcode patterns.

Usage:

Use the function **search_barcode_pattern** to find the 4 corner points of the barcode pattern. Pass to this function `(x,y)` coordinates of a start search point, which should be somewhere on the barcode pattern. In case of success, search various barcode types by the different ***detect*** functions. Each function decodes specific barcode type. If the barcode type is not known in advance, you should call all detect functions one after another and stop decoding when one of these functions returns success.

Public barcode functions

Function	Description
`search_barcode_pattern`	Search barcode pattern. This function should be called before any of the next barcode detection functions.
`detect_barcode_39`	Detect code 3 of 9
`detect_barcode_25`	Detect code interleaved 2 of 5
`detect_barcode_EAN13`	Detect codes EAN13/UPC-A, EAN8, UPC-E
`detect_barcode_CODE128`	Detect CODE128 : raw Code-128 data, coded in 3 possible alphabets A, B and C.
`detect_barcode_EAN128`	Detect EAN128 : This code uses raw Code-128 code, but formats the data to identify the type of the contained information. The second byte after the start symbol should be FNC1 code, followed by appropriate information (application identifier); otherwise this function returns error, use CODE128 instead.

search_barcode_pattern = Search barcode pattern

Prototype:

```
#include “bc_lib.h”
short search_barcode_pattern ( image *img, short center_x, short center_y,
                               short result_x[4], short result_y[4])
```

Description:

The function searches the barcode pattern. The search starts from the start point **`(center_x, center_y)`**. If the search operation is successful, the function stores the x/y coordinates of the 4 corner points of the barcode pattern into the result buffers **`result_x`** and **`result_y`**.

This function should be called before any of the barcode detection functions.

Arguments:

Argument	Description
`img`	[in] barcode pattern image
`center_x`	[in] x-coordinate of the start search point

`center_y`	[in] y-coordinate of the start search point
`result_x[4]`	[out] buffer with x-coordinates of detected 4 pattern corners
`result_y[4]`	[out] buffer with y-coordinates of detected 4 pattern corners

Return code:

Return code	Description
`0`	Success
`101`	Error - not enough memory for the data structure
`102`	Error - not enough memory for the edges structure
`103`	Error - not enough memory for the `contour_data` structure
`104`	Error - five contours with bad quality are found
`105`	Error - buffer overflow
`106`	Error - not enough memory for the `edges_buffer` structure
`107`	Error - the final coordinates are outside the screen area

detect_barcode_39 = Detect barcode 39 pattern

Prototype:

```
#include “bc_lib.h”
short detect_barcode_39 ( image *img, short x[4], short y[4],
                          char *result_pointer )
```

Description:

The function detects barcode **3 of 9**. The function works in two passes. In the first pass the function uses integer pixel coordinates. If the integer detection fails, the function performs 2nd pass with calculation of sub-pixel coordinates by the bilinear interpolation method.

Arguments:

Argument	Description
`img`	[in] barcode pattern image
`x[4]`	[in] buffer with x-coordinates of 4 pattern corners
`y[4]`	[in] buffer with y-coordinates of 4 pattern corners
`result_pointer`	[out] result buffer with detected barcode string

Return code:

Return code	Description
`0`	Success
`100`	Error - not enough memory for the DATA structure
`101`	Error - the three intermediate result strings are different
`102`	Error - missing stop symbol

`200`	Error - pixels buffer overflow
`201`	Error - pixels buffer overflow
`202`	Error - pixels buffer overflow
`300 (400)`	Error - unexpected value of the state variable
`301 (401)`	Error - error in the start symbol
`302 (402)`	Error - the result string is longer than the `MAX_RESULT_SIZE`
`303 (403)`	Error - a detected symbol is not present in the symbol table

detect_barcode_25 = Detect barcode I25 pattern

Prototype:

```
#include “bc_lib.h”
short detect_barcode_25 ( image *img, short x[4], short y[4],
                          char *result_pointer )
```

Description:

The function detects barcode **interleaved 2 of 5**. The function works in two passes. In the first pass the function uses integer pixel coordinates. If the integer detection fails, the function performs 2nd pass with calculation of sub-pixel coordinates by the bilinear interpolation method.

Arguments:

Argument	Description
`img`	[in] barcode pattern image
`x[4]`	[in] buffer with x-coordinates of 4 pattern corners
`y[4]`	[in] buffer with y-coordinates of 4 pattern corners
`result_pointer`	[out] result buffer with detected barcode string

Return code:

Return code	Description
`0`	Success
`100`	Error - not enough memory for the DATA structure
`101`	Error - the three intermediate result strings are different
`200`	Error - pixels buffer overflow
`201`	Error - pixels buffer overflow
`202`	Error - pixels buffer overflow
`300`	Error - unexpected value of the state variable
`301`	Error - missing start symbol
`302`	Error - the result string is longer than the `MAX_RESULT_SIZE`
`303`	Error - the 1 symbol in the current character is not in the symbol table

304	Error - the 2 symbol in the current character is not in the symbol table
305	Error - unexpected value of the state variable
306	Error - missing stop symbol

detect_barcode_EAN13 = Detect barcode EAN13 pattern

Prototype:

```
#include "bc_lib.h"
short detect_barcode_EAN13 ( image *img, short x[4], short y[4],
                             char *result_pointer, short *result_type )
```

Description:

The function detects barcodes EAN13/UPC-A, EAN8 and UPC-E.

The function works in two passes. In the first pass the function uses integer pixel coordinates. If the integer detection fails, the function performs 2nd pass with calculation of sub-pixel coordinates by the bilinear interpolation method.

Arguments:

Argument	Description
`img`	[in] barcode pattern image
`x[4]`	[in] buffer with x-coordinates of 4 pattern corners
`y[4]`	[in] buffer with y-coordinates of 4 pattern corners
`result_pointer`	[out] result buffer with detected barcode string
`result_type`	[out] type of detected code: `0` = EAN13/UPC-A `1` = EAN8 `2` = UPC-E

Return code:

Return code	Description
0	Success
100	Error - not enough memory for the DATA structure
101	Error - the three intermediate result strings are different
102	Error - missing stop symbol
200	Error - pixels buffer overflow
201	Error - pixels buffer overflow
202	Error - pixels buffer overflow
300	Error - unexpected value of the state variable
302	Error - the result string is longer than the `MAX_RESULT_SIZE`
601	Error - invalid symbol - wrong number of bars in the barcode

604	Error - unexpected end of the scan-line - forward
605	Error - unexpected end of the scan-line - reverse
606	Error - some kind of symbol error
607	Error - both sides contain font A
608	Error - unknown font pattern (EAN13-1st digit or UPC-E checksum)
609	Error - UPC-E symbol error

detect_barcode_CODE128 = Detect barcode Code-128 pattern

Prototype:

```
#include “bc_lib.h”
short detect_barcode_CODE128 ( image *img, short x[4], short y[4],
                               char *result_str, short *result_cnt )
```

Description:

The function decodes barcode Code-128 pattern, which contains raw Code-128 data, coded in 3 possible alphabets A, B or C.

The function works in two passes. In the first pass the function uses integer pixel coordinates. If the integer detection fails, the function performs 2nd pass with calculation of sub-pixel coordinates by the bilinear interpolation method.

Finally the function converts the detected codes into chars from A, B or C alphabets.

Arguments:

Argument	Description
`img`	[in] barcode pattern image
`x[4]`	[in] buffer with x-coordinates of 4 pattern corners
`y[4]`	[in] buffer with y-coordinates of 4 pattern corners
`result_str`	[out] result buffer with detected barcode string (size >= `MAX_RESULT_SIZE` bytes)
`result_cnt`	[out] number of result codes stored in **`result_str`**

Return code:

Return code	Description
0	Success
100	Error - not enough memory for the DATA structure
101	Error - the three intermediate result strings are different
102	Error - missing stop symbol
200	Error - pixels buffer overflow
201	Error - pixels buffer overflow
202	Error - pixels buffer overflow

`300`	Error - unexpected value of the state variable
`302`	Error - the result string is longer than the `MAX_RESULT_SIZE`
`501`	Error - invalid symbol - bar smaller then 2/11 or larger then 7/11
`502`	Error - invalid symbol - bar pattern not in lookup table
`503`	Error - invalid symbol - total width of bars wrong for the symbol
`504`	Error - invalid start symbol
`505`	Error - invalid stop symbol
`506`	Error - missing FNC1 symbol
`507`	Error - invalid checksum symbol
`508`	Error - unexpected special function symbol
`509`	Error - faulty data structure - data-type without following data
`510`	Error - faulty data structure - unexpected end of code data

detect_barcode_EAN128 = Detect barcode EAN128 pattern

Prototype:

```
#include “bc_lib.h”
short detect_barcode_EAN128 ( image *img, short x[4], short y[4],
                              char *result_pointer,
                              short interpreter_mode )
```

Description:

The function decodes barcode EAN128. This code uses raw Code-128 code, but formats the data to identify the type of the contained information. The second byte after the start symbol should be FNC1 code, followed by appropriate information (application identifier); otherwise the function returns error. Use the CODE128 detection subroutine for barcodes which do not contain FNC1.

The function works in two passes. In the first pass the function uses integer pixel coordinates. If the integer detection fails, the function performs 2nd pass with calculation of sub-pixel coordinates by the bilinear interpolation method.

Arguments:

Argument	Description
`img`	[in] barcode pattern image
`x[4]`	[in] buffer with x-coordinates of 4 pattern corners
`y[4]`	[in] buffer with y-coordinates of 4 pattern corners
`result_pointer`	[out] result buffer with detected barcode string
`interpreter_mode`	[in] interpreter mode: 0=raw data, 1=interpreted data

Return code:

Return code	Description
`0`	Success
`100`	Error - not enough memory for the DATA structure

`101`	Error - the three intermediate result strings are different
`102`	Error - missing stop symbol
`200`	Error - pixels buffer overflow
`201`	Error - pixels buffer overflow
`202`	Error - pixels buffer overflow
`300`	Error - unexpected value of the state variable
`302`	Error - the result string is longer than the `MAX_RESULT_SIZE`
`501`	Error - invalid symbol - bar smaller then 2/11 or larger then 7/11
`502`	Error - invalid symbol - bar pattern not in lookup table
`503`	Error - invalid symbol - total width of bars wrong for the symbol
`504`	Error - invalid start symbol
`505`	Error - invalid stop symbol
`506`	Error - missing FNC1 symbol
`507`	Error - invalid checksum symbol
`508`	Error - unexpected special function symbol
`509`	Error - faulty data structure - data-type without following data
`510`	Error - faulty data structure - unexpected end of code data

5.2. BLOB (object analysis) library

The VM_LIB library includes the basic BLOB library functions plus several high-level VM_LIB tool functions, described below. Section “6.2.1. Basic BLOB functions” provides description of basic (low-level) BLOB library functions.

Use **vm_find_blobs** to binarize rectangle image area and find all or selected BLOB objects in the area. The function returns BLOB_HND handle, which should be used by the next functions. The **vm_find_blobs** function calculates some basic features of the detected objects. To calculate all object features you should use function **vm_get_blob_info** to get the number of objects, stored in the handle. Next call the function **vm_get_blob_features** in a loop for all object indexes. This function calculates and returns in a BL_FEATURE structure the object features.

Example:

```
#include “bl_lib.h”
#include “vm_lib.h”
    int rc = 0;
    BLOB_HND hnd = NULL;
    image vd_img;
    image dr_img;

    int obj_cnt;          /* number of objects in the handle       */
    BL_FEATURE obj_ftr;   /* structure with BLOB features          */

    int cont_limit   = 0; /* contour length limit (0=don't check)  */
    int calc_contour = 1; /* calculate contour features:   0/1=N/Y */
    int calc_ellipse = 1; /* calculate ellipse features:   0/1=N/Y */
    int calc_rect    = 1; /* calculate rectangle features: 0/1=N/Y */
    int i;

/* Find BLOB objects */
    rc = vm_find_blobs(&hnd, &vd_img, &dr_img, ...);
    if(rc) goto done;

/* Get number of detected objects in obj_cnt */
    rc = vm_get_blob_info(hnd, &obj_cnt, NULL, NULL, NULL, NULL);
    if(rc) goto done;

/* Calculate all object features */
    for(i=0; i<obj_cnt; i++)
    {
      rc = vm_get_blob_features(blob_hnd, &dr_img, i,
                                  cont_limit, calc_contour, calc_ellipse,
                                  calc_rect, ...,
                                  &obj_ftr);
      if(rc) goto done;
    }

/* Close BLOB handle */
done:
    vm_blob_close(&_hnd);
    if(rc) printf(“BLOB error = %d\n”, rc);
```

The function **vm_get_blob_features** returns in the BL_FEATURE structure single-value features. Other object characteristics can be taken by the functions:

`vm_get_blob_contour`	=	Get object contour points
`vm_get_blob_convex`	=	Get object convex points

vm_find_blobs = Find BLOB objects with binarization

Prototype:

```
#include “bl_lib.h”
#include “vm_lib.h”
int vm_find_blobs ( BLOB_HND *blob_hnd, VD_IMAGE *vd_img, DR_IMAGE *dr_img,
                    int  x, int y, int dx, int dy, int pt_pos, int lo_th,
                    int up_th, int bgnd_clr, int obj_clr, int destr_mode,
                    int con_mode, int min_area, int max_area,
                    int filt_mask, int odb_size, int disp_mode,
                    int dr_mode, int dr_clr, int err_clr )
```

Description:

The function performs the first stage in the BLOB object analysis. It binarizes a rectangle area in the input/output image **vd_img**, performs object labeling and finds objects (BLOBs), which are stored into the output handle **blob_hnd**. The output BLOB handle is actually a pointer to an object data-base (ODB) buffer, where the parameters of the detected objects are stored.

Since this function performs binarization and object searching in one pass, it is faster than calling the alternative, functionally equivalent sequence:

- Generate binary image by one of the library binarization functions or by a custom binarization function.
- Call **vm_blobs()**, which works on input binary image.

If the input picture is binarized well with two fixed thresholds, you should prefer **vm_find_blobs()**. If you need special binarization method (for example binarization with dynamic threshold), you should use **vm_blobs()**.

ATTENTION. *Don’t forget to close the BLOB tool to free the allocated memory when the handle is no longer needed.*

*In "**auto size**" mode (`odb_size = 0`) the function allocates and frees 4Mbytes work memory buffer.*

BLOB usage:

```
#include “bl_lib.h”
#include “vm_lib.h”
   int rc;
   BLOB_HND blob_hnd;
   . . . . . . . . . .
   rc = vm_find_blobs(&blob_hnd,...);
   . . . . . . . . . .
   vm_blob_close(&blob_hnd);  // close created handle
```

The function calculates the following basic BLOB features, which can be derived by the **vm_get_blob_features** function:

```
  x_min  = Minimum object x-coordinate, relative to image (0=NA)
  y_min  = Minimum object y-coordinate, relative to image (0=NA)
  x_max  = Maximum object x-coordinate, relative to image (0=NA)
  y_max  = Maximum object y-coordinate, relative to image (0=NA)
  area   = Object area (0=NA)
  x_cent = X-coordinate of mass center (0=NA)
  y_cent = Y-coordinate of mass center (0=NA)
  x_cont = X-coordinate of beginning contour point
  y_cont = Y-coordinate of beginning contour point
  avg_br = Average object brightness = sum(pixels) / area
```

The output handle **blob_hnd** is used next by the function **vm_get_blob_features** to calculate and return more complex features like object contours, equivalent ellipses and so on.

Binarization

Object pixels: **`lo_th <= pixel value <= up_th`**

If **`lo_th > up_th`**, then **`up_th = 255`**.

Area object filtering

The function performs filtering of the searched objects according to their area. Objects with area outside the interval **`[min_area, max_area]`** are not stored in the output handle.

`min_area,max_area=[0,0]`	:	ignore area checking, include all objects
`min_area > max_are`a	:	ignore **`max_area`**

Boundary object filtering

The function performs filtering of the searched objects according to their position in the working rectangle. Filtered (excluded) objects are not stored in the output handle:

- Boundary objects are objects, which touch one of the borders of the working rectangle area: left, right, top or bottom border.
- Internal objects are those, which do not touch the rectangle borders.
- Use **`filt_mask = 0`** to include all object types.

Arguments:

Argument	Description
`blob_hnd`	[out] pointer to output BLOB handle (NULL: error)
`vd_img`	[i/o] video (gray-level) image
`dr_img`	[i/o] drawing (overlay) image (NULL: skip drawing)
`x`	[in] x-coordinate of rectangle point
`y`	[in] y-coordinate of rectangle point
`dx`	[in] rectangle width in pixels
`dy`	[in] rectangle height in pixels
`pt_pos`	[in] position of rectangle point (x,y): 0 = rectangle center 1 = top/left rectangle corner
`lo_th`	[in] lower binarization threshold [0,255]
`up_th`	[in] upper binarization threshold [0,255] (**`>= lo_th`**)
`bgnd_clr`	[in] background binarization color [0,255]
`obj_clr`	[in] object (foreground) binarization color [0,255] (**`bgnd_clr != obj_clr`**)
`destr_mode`	[in] destructive mode of operation: `0` = non-destructive mode : keep input image '**`vd_img`**' `1` = destructive mode : change the input image by setting object and background image pixels to **`fgnd_clr`** and **`obj_clr`** respectively (slower)
`con_mode`	[in] object connection mode: `0` = unconnected mode - all foreground pixels are treated as one single object

	`1 = search 8-connected objects (default)`
	`2 = search 4-connected objects`

`min_area`	[in] min object area for object filtering
`max_area`	[in] max object area for object filtering:

`filt_mask`	[in] object filter mask. Mask bit == 1 : exclude respective objects from the output BLOB handle (add 1,2,4,... to generate mask):
	`0     = include all objects`
	`bit 0 = exclude left boundary objects           (add 1)`
	`bit 1 = exclude right boundary objects          (add 2)`
	`bit 2 = exclude top boundary objects            (add 4)`
	`bit 3 = exclude bottom boundary objects         (add 8)`
	`bit 4 = exclude internal (non-boundary) objects (add 16)`

`odb_size`	[in] max size in bytes of the object data-base buffer, 0 = auto size: the function calculates approximate buffer size

`disp_mode`	[in] feature display mode (add 1,2 or 4, 0=no display):
	`0     = no feature drawing`
	`bit 0 = draw mass centers                     (add 1)`
	`bit 1 = draw object numbers                   (add 2)`
	`bit 2 = draw non-rotated enclosing rectangles (add 4)`

`dr_mode`	[in] drawing mode:
	`0 = draw on (always)`
	`1 = draw off (never)`
	`2 = draw on success only`
	`3 = draw on error only`

`dr_clr`	[in] drawing color on success (0=default: bright green)
`err_clr`	[in] drawing color on error (0=default: bright red)

Return code:

Return code	Description
`0`	Success
`VM_INV_IMAGE`	Invalid input video image
`VM_AREA_OUTSIDE`	The input rectangle is outside the image
`VM_ALLOC_ERROR`	Memory allocation error
Other	Returned by the called functions

vm_blobs = Find BLOB objects without binarization

Prototype:

```
#include “bl_lib.h”
#include “vm_lib.h”
int vm_blobs ( BLOB_HND *blob_hnd, VD_IMAGE *bin_img, DR_IMAGE *dr_img,
                    int x, int y, int dx, int dy, int pt_pos,
                    int bgnd_clr, int con_mode,
                    int min_area, int max_area, int filt_mask,
                    int odb_size, int disp_mode,
int dr_mode, int dr_clr, int err_clr )
```

Description:

The function performs the first phase in the BLOB object detection on input binary image **bin_img**, generated previously by one of the library binarization functions or by a custom function. It performs object labeling, finds objects (BLOBs) in a rectangle image area, defined by **x**, **y**, **dx** and **dy**, and stores the detected objects into the created output handle **`blob_hnd`**. The output BLOB handle is actually a pointer to an object data-base (**ODB**) buffer, where the parameters of the detected objects are stored.

Using of separate functions for binarization and BLOB detection is slower than calling **vm_find_blobs**, which does these operations in one pass (see the **vm_find_blob** description). This approach however gives the opportunity to use special binarization with dynamic threshold for example.

ATTENTION. *Don't forget to close the BLOB tool to free the allocated memory when the handle is no longer needed.*

*In "**auto size**" mode (**`odb_size = 0`**) the function allocates and frees 4Mbytes work memory buffer.*

*The object pixels in **`bin_img`** must have color **`!= bgnd_clr`** !*

BLOB usage:

```
#include “bl_lib.h”
#include “vm_lib.h”
   int rc;
   BLOB_HND blob_hnd;
   VD_IMAGE bin_img;
   . . . . . . . . . .
   rc = ip_img_binar(&bin_img, lo_th, up_th, bgnd_clr, obj_clr);
   if(rc) ...;
   rc = vm_blobs(&blob_hnd, &bin_img, ...);    // create handle, find BLOBs
   if(rc) ...;
   . . . . . . . . . .
   vm_blob_close(&blob_hnd);                   // close created handle
```

The function calculates the following basic BLOB features, which can be derived by the **vm_get_blob_features** function:

```
  x_min  = Minimum object x-coordinate, relative to image (0=NA)
  y_min  = Minimum object y-coordinate, relative to image (0=NA)
  x_max  = Maximum object x-coordinate, relative to image (0=NA)
  y_max  = Maximum object y-coordinate, relative to image (0=NA)
  area   = Object area (0=NA)
  x_cent = X-coordinate of mass center (0=NA)
  y_cent = Y-coordinate of mass center (0=NA)
  x_cont = X-coordinate of beginning contour point
  y_cont = Y-coordinate of beginning contour point
  avg_br = Average object brightness = sum(pixels) / area
```

The output handle **blob_hnd** is used next by the function **vm_get_blob_features** to calculate and return more complex features like object contours, equivalent ellipses and so on.

Area object filtering

The function performs filtering of the searched objects according to their area. Objects with area outside the interval **`[min_area, max_area]`** are not stored in the output handle.

`min_area,max_area=[0,0]`	:	ignore area checking, include all objects
`min_area > max_are``a`	:	ignore **`max_area`**

Boundary object filtering

The function performs filtering of the searched objects according to their position in the working rectangle. Filtered (excluded) objects are not stored in the output handle:

- Boundary objects are objects, which touch one of the borders of the working rectangle area: left, right, top or bottom border.
- Internal objects are those, which do not touch the rectangle borders.
- Use **`filt_mask = 0`** to include all object types.

Arguments:

Argument	Description
`blob_hnd`	[out] pointer to output BLOB handle (NULL: error)
`bin_img`	[in] binary image with background color **`bgnd_clr`** and object color, which is different from **`bgnd_clr`**
`dr_img`	[i/o] drawing (overlay) image (NULL: skip drawing)
`x`	[in] x-coordinate of rectangle point
`y`	[in] y-coordinate of rectangle point
`dx`	[in] rectangle width in pixels
`dy`	[in] rectangle height in pixels
`pt_pos`	[in] position of rectangle point (x,y): 0 = rectangle center 1 = top/left rectangle corner
`bgnd_clr`	[in] background (non-object) color in the input binary image **`bin_img`** [0,255]
`con_mode`	[in] object connection mode: `0 =` unconnected mode - all foreground pixels are treated as one single object `1 =` search 8-connected objects (default) `2 =` search 4-connected objects
`min_area`	[in] min object area for object filtering
`max_area`	[in] max object area for object filtering:

`filt_mask`	[in] object filter mask. Mask bit == 1 : exclude respective objects from the output BLOB handle (add 1,2,4,... to generate mask):
	`0     = include all objects`
	`bit 0 = exclude left boundary objects           (add 1)`
	`bit 1 = exclude right boundary objects          (add 2)`
	`bit 2 = exclude top boundary objects            (add 4)`
	`bit 3 = exclude bottom boundary objects         (add 8)`
	`bit 4 = exclude internal (non-boundary) objects (add 16)`

`odb_size`	[in] max size in bytes of the object data-base buffer, 0 = auto size: the function calculates approximate buffer size

`disp_mode`	[in] feature display mode (add 1,2 or 4, 0=no display):
	`0     = no feature drawing`
	`bit 0 = draw mass centers                     (add 1)`
	`bit 1 = draw object numbers                   (add 2)`
	`bit 2 = draw non-rotated enclosing rectangles (add 4)`

`dr_mode`	[in] drawing mode:
	`0 = draw on (always)`
	`1 = draw off (never)`
	`2 = draw on success only`
	`3 = draw on error only`

`dr_clr`	[in] drawing color on success (0=default: bright green)

`err_clr`	[in] drawing color on error (0=default: bright red)

Return code:

Return code	Description
`0`	Success
`VM_INV_IMAGE`	Invalid input video image
`VM_AREA_OUTSIDE`	The input rectangle is outside the image
`VM_ALLOC_ERROR`	Memory allocation error
Other	Returned by the called functions

vm_blob_close = Close BLOB handle

Prototype:

```
#include “bl_lib.h”
#include “vm_lib.h”
void vm_blob_close ( BLOB_HND *hnd )
```

Description:

The function closes (frees) the BLOB handle, allocated by **vm_find_blobs**, and sets the handle to NULL.

Arguments:

Argument	Description
`blob_hnd`	[i/o] BLOB handle

Return code:

None

vm_get_blob_info = Get BLOB info

Prototype:

```
#include “bl_lib.h”
#include “vm_lib.h”
int vm_get_blob_info ( BLOB_HND hnd, int *obj_cnt, int *odb_size,
                       int *max_odb_size, int *img_dx, int *img_dy )
```

Description:

The function reads BLOB info from the input handle. The handle contains an object data-base (ODB) buffer, where object data and the calculated object features are stored. The ODB buffer contains header with the number of the BLOB objects and other information, which is read by this function.

Use the **vm_get_blob_features** function to access BLOB objects with indexes from **`0`** to **`obj_cnt-1`**.

Arguments:

Argument	Description
hnd	[in] BLOB handle
obj_cnt	[out] number of objects stored in the handle (NULL: don't return)
odb_size	[out] actual size of ODB buffer in bytes (NULL: don't return)
max_odb_size	[out] max allocated size of ODB buffer in bytes (NULL: don't return)
img_dx	[out] width of initial video image in pixels (NULL: don't return)
img_dy	[out] height of initial video image in pixels (NULL: don't return)

Return code:

Return code	Description
0	Success
VM_BLOB_INV_HANDLE	Invalid input handle
VM_BLOB_INV_ODB_BUFFER	Invalid ODB buffer

vm_get_blob_features = Get BLOB features

Prototype:

```
#include “bl_lib.h”
#include “vm_lib.h”
int vm_get_blob_features ( BLOB_HND hnd, DR_IMAGE *dr_img,
                          unsigned int obj_id, int cont_limit,
                          int calc_contour, int calc_ellipse,
                          int calc_rect, int disp_mode, int dr_mode,
                          int dr_clr, int err_clr, BL_FEATURE *obj_ftr )
```

Description:

The function performs the second stage of the BLOB object analysis – the feature calculation. The function uses the input BLOB handle **hnd**, which should be initialized by a previous execution of the function **vm_find_blobs**. The function calculates and gets the features of the BLOB object with index **obj_id** from the handle and stores the feature values into the output structure **obj_ftr**.

The total number of detected objects and other BLOB information present in the handle can be read by the function **vm_get_blob_info**.

The function calculates different groups of features of the object with index **obj_id** and stores the features back into the handle. The following input flags define feature groups for calculation:

calc_contour	=	calculate contour features
calc_ellipse	=	calculate ellipse features
calc_rect	=	calculate min area rectangle features <br>

If you need basic features only calculated by **vm_find_blobs**, you should call this function with zero values of **calc_contour**, **calc_ellipse** and **calc_rect**.

Once calculated. the features of a given object are stored in the handle. Next call(s) of this function with the same **obj_id** do not re-calculate the object features, but simply return already calculated values.

The function also draws the features with graphical representations of the object **obj_id** into the drawing image **dr_img** (see **disp_mode**).

ATTENTION. *Don't forget to close the BLOB handle when no longer needed.*

Example:

```
#include “bl_lib.h”
#include “vm_lib.h”
    int rc;
    BLOB_HND hnd;
    . . . . . . . . .
    rc = vm_find_blobs(&hnd,&vd_img...); // create handle for this image
    rc = vm_get_blob_info(hnd,...);
    . . . . . . . . . . . . . . . . . .
    rc = vm_get_blob_features(hnd,...);
    rc = vm_get_blob_features(hnd,...);
    . . . . . . . . . . . . . . . . . .
    vm_blob_close(&hnd);  // close the handle
```

The function returns in the BL_FEATURE structure **obj_ftr** the calculated feature values. The BL_FEATURE structure is defined in **bl_lib.h**. All coordinates are relative to the original video image, processed in the first stage by **vm_find_blobs**.

Features, already calculated by vm_find_blobs or vm_blobs:

Feature	Description
`x_min`	Minimum object x-coordinate (0=NA)
`y_min`	Minimum object y-coordinate (0=NA)
`x_max`	Maximum object x-coordinate (0=NA)
`y_max`	Maximum object y-coordinate (0=NA)
`area`	Object area in pixels (0=NA)
`x_cent`	X-coordinate of object mass center (0=NA)
`y_cent`	Y-coordinate of object mass center (0=NA)
`x_cont`	X-coordinate of beginning contour point
`y_cont`	Y-coordinate of beginning contour point
`avg_br`	Average object brightness [0,255] = `sum(pixels) / area`

Features calculated when calc_contour = 1:

Feature	Description
`cont_len`	Length of object's external contour in pixels (>0, 0=NA)
`hole_cnt`	Number of object holes (>=0, -1=NA)

min_dist	Min distance from the mass center to a contour point (>=0, -1=NA)
max_dist	Max distance from the mass center to a contour point (>=0, -1=NA)
compactness	Object compactness (>=1, 0=NA)
circularity	Object circularity (>0, 0=NA)

Features calculated when calc_ellipse = 1:

Feature	Description
major_rad	Major (greater) radius of equivalent ellipse (>=0, -1=NA)
minor_rad	Minor (smaller) radius of equivalent ellipse (>=0, -1=NA)
ellip_angle	Rotation angle in radians [0,2PI] (-1=NA)
anisometry	Object anisometry (elongatedness) (>=1, 0=NA)
bulkiness	Object bulkiness (>=1, 0=NA)

Features calculated when calc_rect = 1:

Feature	Description
area_convex	Area of object's convex hull (>0, 0=NA)
rect_dx	Greater extent (width) of min area rectangle (>=0, -1=NA)
rect_dy	Smaller extent (height) of min area rectangle (>=0, -1=NA)
rect_angle	Rotation angle of min area rectangle in radians [0,2PI] (-1=NA)
convexity	Object convexity (how convex the object is) (>0, 0=NA)

Shape features

The shape features provide valuable information about object shape because shape classification is an important task in the machine vision applications .This chapter describes in details the shape features, which are detected by our BLOB library. The features **compactness**, **circularity**, **anisometry** and **bulkiness** show how similar an object is to a circle. The feature **convexity** shows how convex an object is. Remember that some theoretical feature values (1's for perfect circles) sometimes can't be reached due to the digital nature of the applied calculations.

Compactness

Formula: `compactness = (cont_len * cont_len) / (4 * PI * area)`

This feature describes how compact a BLOB is compared to the perfect circle:

1 = perfect circle

>1 = other shape

As it is seen from the formula, the compactness grows quadratically with the length of the contour. Objects with holes have greater compactness values because their area decreases.

Circularity

Formula: `circularity = area /(max_dist * max_dist * PI)`

This feature describes how circular a BLOB is compared to the perfect circle:

1 = perfect circle

<1 = other shapes (not a perfect circle and/or object with holes)

The circularity decreases for objects with holes because their area decreases.

Anisometry

Formula: `anisometry = major_rad / minor_rad`

This feature describes how elongated a BLOB is compared to the perfect circle:

- **`1 =`** perfect circle
- **`>1 =`** other shape

The anisometry increases for more elongated objects. If the feature is not available (for degenerate objects with `minor_rad = 0`), it receives value 0.

Bulkiness

Formula: `bulkiness = (PI * major_rad * minor_rad) / area`

This feature describes how much a BLOB bulges compared to the perfect circle or ellipse:

- **`1 =`** perfect circle or ellipse
- **`>1 =`** other shape

The bulkiness increases for objects with holes because their area decreases. It also increases for objects with more irregular contour forms. For objects with unavailable equivalent ellipse (1-pixel objects for example) the feature receives value 0.

Convexity

Formula: `convexity = area / area_convex`

This feature describes how much a BLOB differs from its convex hull:

- **`1 =`** perfect convex shape – circle or polygon
- **`<1 =`** concave shape or objects with holes

The convexity decreases for objects with holes and concavities. Non-calculated convexity features have zero values.

Arguments:

Argument	Description
`hnd`	[in] BLOB handle (NULL: error)
`dr_img`	[i/o] drawing (overlay) image (NULL: skip drawing)
`obj_id`	[in] object index in the range **`[0,obj_cnt-1]`**, where **`obj_cnt`** is returned by **vm_get_blob_info**
`cont_limit`	[in] contour length limit (0=don't check). Abort contour tracing when this limit is exceeded if **`calc_contour=1`**
`calc_contour`	[in] calculate contour features: 0=no, 1=yes
`calc_ellipse`	[in] calculate ellipse features: 0=no, 1=yes
`calc_rect`	[in] calculate min area rectangle features: 0=no, 1=yes

`disp_mode`	[in] feature display mode (add 1,2,4,... 0=no display):
	`0     = no feature drawing`
	`bit 0 = draw mass center                     (add 1)`
	`bit 1 = draw object number                   (add 2)`
	`bit 2 = draw non-rotated enclosing rectangle (add 4)`
	`bit 3 = draw contour                         (add 8)`
	`bit 4 = draw equivalent ellipse             (add 16)`
	`bit 5 = draw min area rotated rectangle      (add 32)`

`dr_mode`	[in] drawing mode:
	`0 = draw on (always)`
	`1 = draw off (never)`

	`2 = draw on success only`
	`3 = draw on error only`

`dr_clr`	[in] drawing color on success (0=default: bright green)
`err_clr`	[in] drawing color on error (0=default: bright red)
`obj_ftr`	[out] BL_FEATURE structure with calculated features of object **`obj_id`**

Return code:

Return code	Description
`0`	Success
`VM_BLOB_INV_HANDLE`	Invalid input handle
`VM_BLOB_INV_ODB_BUFFER`	Invalid ODB buffer, contained in the handle
Other	Other tool errors returned by called functions

vm_draw_blob_features = Draw BLOB features

Prototype:

```
#include “bl_lib.h”
#include “vm_lib.h”
int vm_draw_blob_features ( BLOB_HND hnd, DR_IMAGE *dr_img,
                            int *obj_list, int list_size,
                            int dr_clr, int dr_mode )
```

Description:

The function draws graphical representations of one or multiple object features into the drawing image **`dr_img`**. The input list buffer **`obj_list`** contains indexes of objects to draw (NULL: draw all object features). If an object feature is not calculated. the function skips drawing of current feature and continues with the next object from the list. On attempt to draw a non-calculated feature, the function returns with error code **`BLOB_R_FEATURE_NOCALC`** .

The **`dr_mode`** parameter specifies object feature(s) to draw. Each feature is enabled by a separate bit. Several features can be drawn at once by OR-ing the respective bits.

Arguments:

Argument	Description
`hnd`	[in] BLOB handle (NULL: error)
`dr_img`	[i/o] drawing (overlay) image (NULL: skip drawing)
`obj_list`	[in] object list buffer in format (NULL = all objects): `idx idx idx ...     (idx =` ODB object index)
`list_size`	[in] number of **`obj_list`** elements (0 = all objects)
`dr_clr`	[in] drawing color (0=default: bright green)
`dr_mode`	[in] drawing mode (OR of following bits): `bit 0  :` draw markers of mass centers `bit 1  :` draw object numbers near mass centers `bit 2  :` draw non-rotated enclosing rectangles `bit 3  :` draw contours

	`bit 4` : draw equivalent ellipses `bit 5` : draw min area rotated rectangles

Return code:

Return code	Description
`0`	Success
`VM_BLOB_INV_HANDLE`	Invalid input handle
`VM_BLOB_INV_ODB_BUFFER`	Invalid ODB buffer, contained in the handle
Other	Other errors returned by called functions

vm_get_blob_contour = Get BLOB contour

Prototype:

```
#include “bl_lib.h”
#include “vm_lib.h”
int vm_get_blob_contour ( BLOB_HND hnd, DR_IMAGE *dr_img,
                          unsigned int obj_id, int cont_limit,
                          int dr_mode, int dr_clr, int err_clr,
                          int **xy_buf, int *cont_cnt )
```

Description:

The function finds (if not calculated yet) all contours points of the object **obj_id** and stores the `(x,y)` coordinates of the contour points into the output buffer **xy_buf**. The function returns in **cont_cnt** the number of object contours. The `(x,y)` coordinates are relative to the upper left corner (0,0) of the input gray-level image.

The function allocates memory for the output x/y buffer **xy_buf**, which should be freed by the calling function. In case of error the function frees the allocated x/y buffer.

Format of the output buffer with x/y coordinates:

```
  cnt (x,y) (x,y) ... (x,y)   contour [0] (external)
  cnt (x,y) (x,y) ... (x,y)   contour [1] (hole contour...)
  . . . . . . . . . . . . .   . . . . .
  cnt (x,y) (x,y) ... (x,y)   contour [cont_cnt-1]
```

where:

`cnt` = # of points (x,y pairs) of current contour
`x,y` = point coordinates

If the contours of object **obj_id** are not calculated yet, then the function finds the object contours. The function also draws the contour points of **obj_id** in the drawing image **dr_img** (see **dr_mode**).

Example

```
int rc;
BLOB_HND hnd;
. . . . . . . . . .
rc = vm_find_blobs(&hnd,&vd_img...); // create BLOB handle for this image
rc = vm_get_blob_info(hnd,...);      // get number of objects & other info
. . . . . . . . . . . . . . . . . .
rc = vm_get_blob_contour(hnd,...);
```

```
 . . . . . . . . . . . . . . . . .
 vm_blob_close(&hnd);                 // close the handle
```

Arguments:

Argument	Description
hnd	[in] BLOB handle (NULL: error)
dr_img	[i/o] drawing (overlay) image (NULL: skip drawing)
obj_id	[in] object index in the range **[0,obj_cnt-1]**, where **obj_cnt** is returned by **vm_get_blob_info**
cont_limit	[in] contour length limit (0=don't check). Abort contour tracing when this limit is exceeded.
dr_mode	[in] contour drawing mode: 0 = draw on (always) 1 = draw off (never) 2 = draw on success only 3 = draw on error only
dr_clr	[in] drawing color on success (0=default: bright green)
err_clr	[in] drawing color on error (0=default: bright red)
xy_buf	[out] buffer with (x,y) coordinates, allocated by this function. Should be freed by the calling function.
cont_cnt	[out] number of object contours (0=error), equal to the number of object holes + 1 (the outer object contour)

Return code:

Return code	Description
0	Success
VM_BLOB_INV_HANDLE	Invalid input handle
VM_BLOB_INV_ODB_BUFFER	Invalid ODB buffer
Other	Returned by called functions

vm_get_blob_convex = Get BLOB convex

Prototype:

```
#include “bl_lib.h”
#include “vm_lib.h”
int vm_get_blob_convex ( BLOB_HND hnd, DR_IMAGE *dr_img,
                         unsigned int obj_id, int cont_limit,
                         int dr_mode, int dr_clr, int err_clr,
                         int **cvx_buf, int *cvx_cnt )
```

Description:

The function calculates the convex hull of the object **obj_id** and stores the (x,y) coordinates of the convex points into the output buffer **cvx_buf**. The function returns in **cvx_cnt** the number of convex points in **cvx_buf**. The result (x,y) coordinates are relative to the upper left corner (0,0) of the input gray-level image.

The function allocates memory for the output x/y buffer **'cvx_buf'**, which should be freed by the calling function. In case of error this function frees the allocated x/y buffer.

Format of the output buffer **cvx_buf** with x/y coordinates:

```
  (x,y) (x,y) ... (x,y)
```

If the contours of object **obj_id** are not calculated yet, then the function finds the object contour points, which are needed to find the convex hull. The function also draws the convex points in the drawing image **dr_img** (see **dr_mode**).

Example

```
int rc;
BLOB_HND hnd;
. . . . . . . . . .
rc = vm_find_blobs(&hnd,&vd_img...); // create BLOB handle for this image
rc = vm_get_blob_info(hnd,...);      // get number of objects & other info
. . . . . . . . . . . . . . . . . . .
rc = vm_get_blob_convex(hnd,...);
. . . . . . . . . . . . . . . . . . .
vm_blob_close(&hnd);                 // close the handle
```

Arguments:

Argument	Description
hnd	[in] BLOB handle (NULL: error)
dr_img	[i/o] drawing (overlay) image (NULL: skip drawing)
obj_id	[in] object index in the range **[0,obj_cnt-1]**, where **obj_cnt** is returned by **vm_get_blob_info**
cont_limit	[in] contour length limit (0=don't check). Abort contour tracing when this limit is exceeded.

dr_mode	[in] convex drawing mode:
	`0 = draw on (always)`
	`1 = draw off (never)`
	`2 = draw on success only`
	`3 = draw on error only`

dr_clr	[in] drawing color on success (0=default: bright green)
err_clr	[in] drawing color on error (0=default: bright red)
cvx_buf	[out] buffer with (x,y) coordinates of convex points, allocated by this function. Should be freed by the calling function.
cvx_cnt	[out] number of convex points (x.y pairs) stored in **cvx_buf**

Return code:

Return code	Description
0	Success
VM_ALLOC_ERROR	Memory allocation error

`VM_INV_ARG`	Invalid argument
Other	Returned by called functions

vm_get_convex_obj = Get convex object

Prototype:

```
#include “bl_lib.h”
#include “vm_lib.h”
int vm_get_convex_obj ( BLOB_HND blob_hnd, image *vd_img, image *dr_img,
                        int obj_id, int pix_fill, int dr_mode,
                        int dr_clr, int err_clr, image *obj_img,
                        int *min_val, int *max_val )
```

Description:

The function calculates the convex hull of the object **obj_id** and stores the image pixels, enclosed in the convex hull, into the output image **obj_img**. Additionally the function finds the min and max values of the pixels inside the convex hull. The output image **obj_img** is allocated by this function and should be freed by the calling function(s). In case of error this function frees the image.

ATTENTION. *At present the convex points are not stored in the handle and the function calculates the convex hull each time when called.*

Arguments:

Argument	Description
`blob_hnd`	[in] BLOB handle (NULL: error)
`vd_img`	[in] gray-level image
`dr_img`	[i/o] drawing (overlay) image (NULL: skip drawing)
`obj_id`	[in] object index in the range **`[0,obj_cnt-1]`**, where **`obj_cnt`** is returned by **vm_get_blob_info**
`pix_fill`	[in] pixel value, filled in the result image outside the convex hull area

`dr_mode`	[in] drawing mode:
	`0 = draw on (always)`
	`1 = draw off (never)`
	`2 = draw on success only`
	`3 = draw on error only`

`dr_clr`	[in] drawing color on success (0=default: bright green)
`err_clr`	[in] drawing color on error (0=default: bright red)
`obj_img`	[out] image with object pixels, enclosed by the object's convex hull - allocated by THIS function!
`min_val`	[out] minimum object pixel value (NULL: don't return)
`max_val`	[out] maximum object pixel value (NULL: don't return)

Return code:

Return code	Description
`0`	Success
`VM_ALLOC_ERROR`	Memory allocation error
`VM_INTERNAL_ERR`	Internal error
Other	Returned by called functions

5.2.1. Basic BLOB functions

This section describes the basic (low-level) BLOB functions, which are called by the high-level functions.

bl_img_binar = Binarize image

Prototype:

```
int bl_img_binar ( image *img, image *img_bin, BL_UINT32 lo_th,
                   BL_UINT32 up_th, int bgnd_clr, int fgnd_clr )
```

Description:

The function binarizes the input image **`img`** and stores the result binary image into **`img_bin`**. The dimensions of the output image should be equal or greater than the dimensions of the input image, otherwise the function returns with error. The binary data is stored in the upper/left corner of the output image. The parameters of the output image are not changed if the image dimensions differ. The function works faster in place: **`img`** == **`img_bin`**

Algorithm:

The function checks each pixel in the input image ***in_pix*** if it belongs to an object area or a background area. The function generates output pixel value ***out_pix*** according to the following formula:

```
  if(lo_th <= in_pix <= up_th) then  out_pix = fgnd_clr;  // object pixel
                               else  out_pix = bgnd_clr;  // bgnd pixel
```

where:

`lo_th` = lower binarization threshold `[0,255]` `(lo_th <= up_th)`
`up_th` = upper binarization threshold `[0,255]` (if `lo_th > up_th` then `up_th = 255`)

As seen from the formula above, object pixels in the output binary image are set to `fgnd_clr` and background pixels are set to `bgnd_clr`. The two values should be different. The function uses lookup table for faster image processing.

Parameters:

`img`	Input image
`img_bin`	Output binary image: `bgnd_clr :` background pixels `fgnd_clr :` object pixels
`lo_th`	Input lower binarization threshold `[0,255]`
`up_th`	Input upper binarization threshold `[0,255] (>= lo_th).` If `lo_th > up_th,` then `up_th = 255.`
`bgnd_clr`	Input background binarization color [0,255]

`fgnd_clr`	Input foreground binarization color [0,255]

Return code:

`0`	Success
`BLOB_R_IMG_OVF`	Small output image

bl_get_objects = Get objects

Prototype:

```
int bl_get_objects ( image *imgb, int bgnd_clr, char *odb_buf,
                     BL_UINT32 max_odb_size, int con_mode,
                     BL_UINT32 min_area, BL_UINT32 max_area,
                     int filt_mask )
```

Description:

The function performs the first phase of the object feature calculations. It converts an input binary image to a RLC buffer, performs segmentation (object labeling) inside the RLC buffer and finally generates an object database (ODB) buffer. The ODB buffer is used as input/output buffer in the next calculation stages - getting of object contours, calculation of equivalent ellipses and so on.

***ATTENTION**. The ODB buffer must be allocated by the calling function and its size in bytes must be passed to the function in the **`max_odb_size`** argument.*

The function performs object filtering. Objects with area outside a given range **`[min_area,max_area]`** can be excluded from ODB. Objects with given position in the image (boundary or internal) can be excluded from ODB. Boundary objects are those, which touch one of the image borders - left, right, top or bottom. Internal objects are those, which do not touch image borders. Use **`filt_mask`** = **`BLOB_OBJ_ALL`** to include all types of objects.

The function calculates and saves into ODB some basic object features. Uncalculated or not available (NA) features have zero values. Pixel coordinates are relative to the input image.

In unconnected mode the function treats all foreground image pixels as one object. The basic features described below are valid for this unconnected object. Other features are not available.

***ATTENTION**. Currently the library does not support calculation of more sophisticated contour, ellipse and rectangle features in unconnected mode. Note that some features have no sense for unconnected objects. The object filtering is disabled in unconnected mode.*

Calculated features:

`x_min`	=	Minimum object x-coordinate (0=NA)
`y_min`	=	Minimum object y-coordinate (0=NA)
`x_max`	=	Maximum object x-coordinate (0=NA)
`y_max`	=	Maximum object y-coordinate (0=NA)
`area`	=	Object area (0=NA)
`x_cent`	=	X-coordinate of mass center (0=NA)
`y_cent`	=	Y-coordinate of mass center (0=NA)

Parameters:

`imgb`	Input binary image: `bgnd_clr :` background pixels

	`other      :` object pixels (`!=bgnd_clr`)
`bgnd_clr`	Input color of background pixels in `imgb`
`odb_buf`	Output object database buffer
`max_odb_size`	Input max size of `odb_buf` in bytes
`con_mode`	Input connection mode: • **`0`** `:` unconnected mode - all foreground pixels are treated as one single object. Disables object filtering. • **`1`** `:` search 8-connected objects (default) • **`2`** `:` search 4-connected objects
`min_area`	Input min object area for object filtering
`max_area`	Input max object area for object filtering: `min_area,max_area=[0,0] :` ignore area filtering `min_area > max_area      :` ignore `max_area`
`filt_mask`	Input object filter mask = OR of next bits. Mask bit equal to 1 excludes respective objects from ODB buffer (the bit macros are defined in BL_LIB.H): `BLOB_OBJ_LEFT     :` exclude left boundary objects `BLOB_OBJ_RIGHT    :` exclude right boundary objects `BLOB_OBJ_TOP      :` exclude top boundary objects `BLOB_OBJ_BOT      :` exclude bottom boundary objects `BLOB_OBJ_INTERNAL :` exclude internal objects

Return code:

`0`	Success
`BLOB_R_NO_MEMORY`	Memory allocation error
Other	Returned by the called functions

bl_gen_odb = Generate ODB

Prototype:

```
int bl_gen_odb ( image *img, BL_UINT32 lo_th, BL_UINT32 up_th,
                 int  bgnd_clr, int fgnd_clr, int destructive,
                 char *odb_buf, BL_UINT32 max_odb_size, int con_mode,
                 BL_UINT32 min_area, BL_UINT32 max_area, int filt_mask )
```

Description:

The function performs image binarization and ODB generation in one pass, which is faster than the separate stage execution. It can be used instead of the function sequence:

`bl_img_binar()`

`bl_get_objects()`

The function converts an input gray-level image to a RLC buffer, performs segmentation (object labeling) inside the RLC buffer and finally generates an object database (ODB) buffer. The ODB buffer is used as input/output buffer in the next calculation stages - getting of object contours, calculation of equivalent ellipses and so on.

***ATTENTION**. The ODB buffer must be allocated by the calling function and its size in bytes must be passed to the function in the **`max_odb_size`** argument.*

The function performs object filtering. Objects with area outside a given range **`[min_area,max_area]`** can be excluded from ODB. Objects with given position in the image (boundary or internal) can be excluded from ODB. Boundary objects are those, which touch one of the image borders - left, right, top or bottom. Internal objects are those, which do not touch image borders. Use **`filt_mask`** = **`BLOB_OBJ_ALL`** to include all types of objects.

The function calculates and saves into ODB some basic object features. Uncalculated or not available (NA) features have zero values. Pixel coordinates are relative to the input image.

In unconnected mode the function treats all foreground image pixels as one object. The basic features described below are valid for this unconnected object. Other features are not available.

ATTENTION. *Currently the library does not support calculation of more sophisticated contour, ellipse and rectangle features in unconnected mode. Note that some features have no sense for unconnected objects. The object filtering is disabled in unconnected mode.*

Calculated features:

`x_min`	=	Minimum object x-coordinate (0=NA)
`y_min`	=	Minimum object y-coordinate (0=NA)
`x_max`	=	Maximum object x-coordinate (0=NA)
`y_max`	=	Maximum object y-coordinate (0=NA)
`area`	=	Object area (0=NA)
`x_cent`	=	X-coordinate of mass center (0=NA)
`y_cent`	=	Y-coordinate of mass center (0=NA)
`avg_br`	=	Average object brightness (sum of pixels / area)

Parameters:

`img`	Input/output gray-level image
`lo_th`	Input lower binarization threshold `[0,255]`
`up_th`	Input upper binarization threshold `[0,255] (>= lo_th).` If `lo_th > up_th,` then `up_th = 255.`
`bgnd_clr`	Input background binarization color [0,255] (used in destructive mode)
`fgnd_clr`	Input foreground binarization color [0,255] (used in destructive mode)
`destructive`	Input destructive mode: • **`0 :`** non-destructive mode, keep input image • **`1 :`** destructive mode - change input image by setting object and background image pixels to `fgnd_clr` and `bgnd_clr`. This mode increases the execution time.
`odb_buf`	Output object database buffer
`max_odb_size`	Input max size of `odb_buf` in bytes
`con_mode`	Input connection mode: • **`0`** `:` unconnected mode - all foreground pixels are treated as one single object. Disables object filtering. • **`1`** `:` search 8-connected objects (default) • **`2`** `:` search 4-connected objects
`min_area`	Input min object area for object filtering
`max_area`	Input max object area for object filtering: `min_area,max_area=[0,0]` : ignore area filtering `min_area > max_area` : ignore `max_area`

`filt_mask`	Input object filter mask = OR of next bits. Mask bit equal to 1 excludes respective objects from ODB buffer (the bit macros are defined in BL_LIB.H): `BLOB_OBJ_LEFT` : exclude left boundary objects `BLOB_OBJ_RIGHT` : exclude right boundary objects `BLOB_OBJ_TOP` : exclude top boundary objects `BLOB_OBJ_BOT` : exclude bottom boundary objects `BLOB_OBJ_INTERNAL` : exclude internal objects

Return code:

`0`	Success
`BLOB_R_NO_MEMORY`	Memory allocation error
Other	Returned by the called functions

bl_get_contours = Get contours

Prototype:

```
int bl_get_contours ( char *odb_buf, BL_UINT32 *obj_list,
                      BL_UINT32 list_size, int calc_mode, int cont_limit )
```

Description:

The function finds object contours and contour-related features and stores the results into ODB. The function finds the outer and all internal object contours. The input buffer **`obj_list`** selects indexes of ODB objects, the contours of which are searched. The function skips processing of objects, the contour features of which are already calculated.

The function counts the detected contour points in each traced contour (external or internal) and aborts with error **`BLOB_R_CONTOUR_LIMIT`** when the limit **`cont_limit`** is exceeded. This check is enabled by non-zero value of **`cont_limit`**. In unconnected mode the function returns error **`BLOB_R_INV_FEATURE`**.

PREMISE. *The function needs valid input ODB buffer created by previous call to* ***`bl_get_objects`***.

When accumulating the contour length, the function adds 1 for horizontal or vertical pixels and sqrt(2) for diagonal pixels. The argument **`calc_mode`** specifies contour length calculation method:

- `0 :` by measuring the length of the outer object's boundary – the line which lies between contour points and background (default, more precise)
- `1 :` by measuring the length of the contour itself.

Calculated features:

`cont_len`	=	Length of outer object's contour (0=NA)
`hole_cnt`	=	Number of object holes (>=0, -1=NA)
`min_dist`	=	Min distance from the mass center to a contour point
`max_dist`	=	Max distance from the mass center to a contour point
`compactness`	=	Object compactness
`circularity`	=	Object circularity

Features of degenerate 1-pixel objects:

`cont_len`	=	`4.0` (total border length of a square pixel)

```
hole_cnt    = 0
min_dist    = 0
max_dist    = 0
circularity = 0
```

Parameters:

`odb_buf`	Input/output object database buffer
`obj_list`	Input object list buffer in format (NULL = all objects): `idx idx idx ...` (`idx =` ODB object index)
`list_size`	Input number of "`obj_list`" elements (0 = find contours of all objects)
`calc_mode`	Input contour length calculation mode (see description above)
`cont_limit`	Input contour limit = max number of detected contour points in one contour before the function aborts (0=don't check).

Return code:

`0`	Success
`BLOB_R_INVALID_ODB`	Invalid input ODB buffer
`BLOB_R_INV_FEATURE`	Attempt to get features in unconnected mode
`BLOB_R_CONTOUR_LIMIT`	Contour limit exceeded
Other	Returned by the called functions

bl_get_ellipses = Get ellipses

Prototype:

```
int bl_get_ellipses ( char *odb_buf, BL_UINT32 *obj_list,
                      BL_UINT32  list_size )
```

Description:

The function finds parameters of object’s equivalent ellipses and ellipse-related shape features. The results are stored into ODB. The input buffer **`obj_list`** selects indexes of ODB objects for which calculations should be done. The function skips processing of objects, the ellipse features of which are already calculated. In unconnected mode the function returns error **`BLOB_R_INV_FEATURE`**.

PREMISE.

1. *The function needs valid input ODB buffer created by previous call to **`bl_get_objects`**.*
2. *The function has all necessary information to calculate all ellipse features and related shape features with one exception:*
   - *If an object contour is not calculated by previous call to **`bl_get_contours`**, or the contour does not contain orientation data (1-pixel object for example), then **`ellip_angle`** is in the range [0,PI] and **`bl_get_ellipses`** returns with error code **`BLOB_R_NO_ORIENT`**.*
   - *In case of valid contour data **`ellip_angle`** is in the range [0,2PI].*

Calculated features:

`major_radius` = Greater radius of equivalent ellipse
`minor_radius` = Smaller radius of equivalent ellipse

`ellip_angle`	= Rotation angle of equivalent ellipse [0,2PI]. Needs object contours, else the angle is in the range [0,PI].
`anisometry`	= Object anisometry (elongatedness): `1` : perfect circle `>1` : other shapes - greater values for more elongated objects
`bulkiness`	= Object bulkiness. It describes how much the object bulges compared to perfect circle or ellipse: `1` : perfect circle or ellipse `>1` : other shapes

Features of degenerate 1-pixel objects:

```
major_rad   = 0.0
minor_rad   = 0.0
ellip_angle = 0.0
anisometry  = 0.0  (not available)
bulkiness   = 0.0  (not available)
```

Parameters:

`odb_buf`	Input/output object database buffer
`obj_list`	Input object list buffer in format (NULL = all objects): `idx idx idx ...` (`idx` = ODB object index)
`list_size`	Input number of "`obj_list`" elements (0 = all objects)

Return code:

`0`	Success
`BLOB_R_INVALID_ODB`	Invalid input ODB buffer
`BLOB_R_INV_FEATURE`	Attempt to get features in unconnected mode
`BLOB_R_NO_MEMORY`	Memory allocation error
`BLOB_R_FEATURE_NOCALC`	Prerequisite feature not calculated (contour data not present)
`BLOB_R_NO_ORIENT`	Insufficient orientation data
Other	Returned by the called functions

bl_get_rects = Get rectangles

Prototype:

```
int bl_get_rects ( char *odb_buf, BL_UINT32 *obj_list,
                   BL_UINT32  list_size )
```

Description:

The function finds convex hulls, minimum area rectangles, which enclose the objects, and convex-related shape features. The results are stored into ODB. The input buffer **`obj_list`** selects indexes of ODB objects for which calculations should be done. The function skips processing of objects, the rectangle features of which are already calculated. In unconnected mode the function returns error **`BLOB_R_INV_FEATURE`**.

PREMISE.

*1. The function needs valid input ODB buffer created by previous call to **`bl_get_objects`**.*

2. *The function needs contours calculated by previous call to **`bl_get_contour`**:*
   - *If the contour does not contain enough orientation data (1-pixel object for example), then **`rect_angle`** is in the range [0,PI] and **`bl_get_rects`** returns with error code **`BLOB_R_NO_ORIENT`**.*
   - *In case of present contour orientation data **`rect_angle`** is in the range [0,2PI].*

Calculated features:

`area_convex`	=	Area of object's convex hull (0=NA)
`rect_dx`	=	Greater extent (width) of min area rectangle (-1=NA)
`rect_dy`	=	Smaller extent (height) of min area rectangle (-1=NA)
`rect_angle`	=	Rotation angle of minimum area rectangle [0,2PI] (-1=NA) Note; The angle is in [0,PI] if there is no contour orientation data.
`convexity`	=	Object convexity (how convex the object is) (>0, 0=NA): `1:` perfect convex shape `<1:` blob with holes or concave shape

Features of degenerate 1-pixel objects:

```
area_convex  = 1.0
rect_dx      = 0.0
rect_dy      = 0.0
rect_angle   = 0.0
convexity    = 1.0
```

Parameters:

`odb_buf`	Input/output object database buffer
`obj_list`	Input object list buffer in format (NULL = all objects): `idx idx idx ...` (`idx =` ODB object index)
`list_size`	Input number of "`obj_list`" elements (0 = all objects)

Return code:

`0`	Success
`BLOB_R_INVALID_ODB`	Invalid input ODB buffer
`BLOB_R_INV_FEATURE`	Attempt to get features in unconnected mode
`BLOB_R_FEATURE_NOCALC`	Prerequisite feature not calculated (contour data not present)
`BLOB_R_NO_ORIENT`	Insufficient orientation data
Other	Returned by the called functions

bl_read_odb_par = Read ODB parameters

Prototype:

```
void bl_read_odb_par ( char *odb_buf, ODB_PAR *odb_par )
```

Description:

The function reads ODB header parameters into the output structure **`odb_par`**. Use this function to get the number of objects, contained in ODB, as well as other ODB parameters (see the definition of the **`ODB_PAR`** structure in BL_LIB.H).

Parameters:

`odb_buf`	Input object database buffer
`odb_par`	Output structure with ODB parameters

Return code:

None

bl_read_obj_features = Read objects features from ODB

Prototype:

```
int bl_read_obj_features ( char *odb_buf, BL_UINT32  obj_id,
                           BL_FEATURE *obj_ftr )
```

Description:

The function extracts from ODB the features of the object **`obj_id`** and stores their values into members of the output **`BL_FEATURE`** structure **`obj_ftr`** (see BL_LIB.H).

PREMISE. *Remember that you must call feature calculation functions before you read features values. Uncalculated or not available (NA) features have zero or negative values (see description of **`obj_ftr`** features below).*

The function returns the following feature values in **`obj_ftr`**:

Feature	Description	Calculated by:
`x_min`	Minimum object x-coordinate, relative to image (0=NA)	**`bl_get_objects`** **`bl_gen_odb`**
`y_min`	Minimum object y-coordinate, relative to image (0=NA)	
`x_max`	Maximum object x-coordinate, relative to image (0=NA)	
`y_max`	Maximum object y-coordinate, relative to image (0=NA)	
`area`	Object area (0=NA)	
`x_cent`	X-coordinate of mass center (0=NA)	
`y_cent`	Y-coordinate of mass center (0=NA)	
`avg_br`	Average object brightness [0,255] – by **`bl_gen_odb`** only	
`cont_len`	Length of outer object's contour (0=NA)	**`bl_get_contours`**
`hole_cnt`	Number of object holes (>=0, -1=NA)	
`min_dist`	Min distance from the mass center to a contour point (>=0, -1=NA)	
`max_dist`	Max distance from the mass center to a contour point (>=0, -1=NA)	
`compactness`	Object compactness (>=1, 0=NA)	
`circularity`	Object circularity (>0, 0=NA)	
`major_rad`	Major (greater) radius of equivalent ellipse (>=0, -1=NA)	**`bl_get_ellipses`**
`minor_rad`	Minor (smaller) radius of equivalent ellipse (>=0, -1=NA)	

`ellip_angle`	Rotation angle of equivalent ellipse [0,2PI] (-1=NA)	
`anisometry`	Object anisometry (elongatedness) (>=1, 0=NA)	
`bulkiness`	Object bulkiness (>=1, 0=NA)	
`area_convex`	Area of object's convex hull (0=NA)	**`bl_get_rects`**
`rect_dx`	Greater extent (width) of min area rectangle (-1=NA)	
`rect_dy`	Smaller extent (height) of min area rectangle (-1=NA)	
`rect_angle`	Rotation angle of minimum area enclosing rectangle [0,2PI] (-1=NA)	
`convexity`	Object convexity (>0, 0=NA)	

Parameters:

`odb_buf`	Input object database buffer
`obj_id`	Input object index
`obj_ftr`	Output structure with values of object features

Return code:

`0`	Success
`BLOB_R_INV_OBJ_IDX`	Invalid object index `obj_id`
`BLOB_R_DELETED_OBJ`	Deleted object
Other	Returned by the called functions

bl_read_obj_contours = Read object contours from ODB

Prototype:

```
int bl_read_obj_contours ( char *odb_buf, BL_UINT32  obj_id,  int **xy_buf,
                           int  *cont_cnt )
```

Description:

The function reads all contours of the object **`obj_id`** and stores the `(x,y)` coordinates of the contour points into the output buffer **`xy_buf`**. The function returns in **`cont_cnt`** the number of object contours. The `(x.y)` coordinates are relative to the upper left corner of the input image, which has coordinates `(0,0)`.

***ATTENTION**. The function allocates memory for the output x/y buffer **`xy_buf`**, which should be freed by the calling function. In case of error the function frees the allocated x/y buffer.*

Format of the output buffer **`xy_buf`**:

```
   cnt  (x,y) (x,y) ... (x,y)    /* contour 0 : external  */
   cnt  (x,y) (x,y) ... (x,y)    /* contour 1             */
   . . . . . . . . . . . . .     . . . . . . . . . . . . . .
   cnt  (x,y) (x,y) ... (x,y)    /* contour (cont_cnt-1)  */
```

where:

`cnt` = # of points (`x,y` pairs) of current contour

`x,y`	=	point coordinates
`cont_cnt`	=	number of object contours

***PREMISE**. Remember that you must have called the contour detection function **`bl_get_contours`** before you read object contours, otherwise the function will return with error **`BLOB_R_FEATURE_NOCALC`**.*

Parameters:

`odb_buf`	Input object database buffer
`obj_id`	Input object index
`xy_buf`	Output buffer with (x,y) coordinates
`cont_cnt`	Output number of object contours (0=error)

Return code:

`0`	Success
`BLOB_R_INVALID_ODB`	Invalid input ODB buffer
`BLOB_R_INV_OBJ_IDX`	Invalid object index `obj_id`
`BLOB_R_NO_MEMORY`	Memory allocation error
`BLOB_R_FEATURE_NOCALC`	Prerequisite feature not calculated (contours NA)
`BLOB_R_INTERNAL_ERR`	Internal error
`Other`	Returned by the called functions

bl_draw_features = Draw features

Prototype:

```
int  bl_draw_features ( image *dr_img, char *odb_buf, BL_UINT32 *obj_list,
                        BL_UINT32  list_size, int x0, int y0,
                        int clr, int  dr_mode )
```

Description:

The function draws the graphical representations of the object features into the drawing image **`dr_img`**. The input list buffer **`obj_list`** selects indexes of objects to draw (NULL: draw all object features). If an object feature is not calculated. the function skips drawing of current feature and continues with the next object in the list. Finally the function may return with error code **`BLOB_R_FEATURE_NOCALC`** on attempt to draw non-calculated feature.

The **`dr_mode`** parameter specifies object feature(s) to draw. Each feature is enabled by a separate bit. Several features can be drawn at once by OR-ing of the respective bits. Macros for the drawing mode bits are defined in BL_LIB.H.

Drawings are relative to an origin point (x0,y0), which should correspond to the screen coordinates of the top/left corner of the initial input image. The drawing image **`dr_img`** should be a full-screen overlay image.

Parameters:

`dr_img`	Input/output drawing image

`odb_buf`	Input object database buffer
`obj_list`	Input object list buffer in format (NULL = all objects): `idx idx idx ...` (`idx =` ODB object index)
`list_size`	Input number of "`obj_list`" elements (0 = all objects)
`x0`	Input x-coordinate of drawing origin point, relative to `dr_img`
`y0`	Input y-coordinate of drawing origin point, relative to `dr_img`
`clr`	Input drawing color
`dr_mode`	Input drawing mode (OR of following bits): `BL_DRAW_CENTER` : draw markers of mass centers `BL_DRAW_OBJ_NUM` : draw object numbers near mass centers `BL_DRAW_NONR_RECT` : draw non-rotated enclosing rectangles `BL_DRAW_CONTOUR` : draw contours `BL_DRAW_ELLIPSE` : draw equivalent ellipses `BL_DRAW_MIN_RECT` : draw min area rotated rectangles

Return code:

`0`	Success
`BLOB_R_FEATURE_NOCALC`	Prerequisite feature not calculated
Other	Returned by the called functions

bl_set_max_rlc_size = Set max RLC buffer size

Prototype:

```
void bl_set_max_rlc_size ( int max_rlc_size )
```

Description:

The function sets max size of the work RLC buffer, used by the BLOB detection functions:

- bl_get_objects
- bl_gen_odb
- vm_find_blobs
- vm_blobs

Call this function before a BLOB detection function, which returns one of the following errors:

`9102 = BLOB_R_RLCBUF_OVF`

`9120 = BLOB_R_NO_RLC_MEMORY`

In auto mode (`max_rlc_size = 0`) or if this function is not called, the BLOB library uses approximate max size of the RLC buffer, calculated for the current source image and limited to 4 Mbytes.

Parameters:

`max_rlc_size`	[in] max RLC buffer size (0=auto)

Return code:

None

5.3. Calculation and geometry library

This chapter contains detailed description of the calculation and geometry library functions. Please read the introductory sections, which present general information about the library. This information could be valuable to understand the operation of some library functions.

5.3.1. Segment definitions of figures

Some library functions use the segment-definition structure **`SEG_DEF`**, declared in **cl_lib.h**:

```
typedef struct
{
    int *seg_buf;   /* segment buffer (x1,x2) (x1,x2) ...       */
    int  y_min;     /* minimum y-coordinate (top pixel row)     */
    int  dy;        /* number of segments (x1,x2) in seg_buf    */
                    /* (figure height in pixels)                */
} SEG_DEF;
```

This structure defines convex figures like rotated rectangles and ellipses by pixel line segments. The **`seg_buf`** buffer contains the start and the end x-coordinates of each pixel line segment, which belongs to the figure:

```
                        seg_buf[2*i]  seg_buf[2*i+1]
                        ------------  -------------
 y = y_min          :      x1            x2
 y = y_min + 1      :      x1            x2
 . . .  . . . . . . . . . . . . . . . . . . .
 y = y_min + dy - 1 :      x1            x2
```

where:

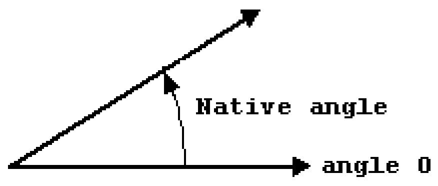
x1 = min x-coordinate of current segment (CL_SEG_NONE: N/A)

x2 = max x-coordinate of current segment (CL_SEG_NONE: N/A)

The size of **`seg_buf`** is **`dy*2`** integer words.

5.3.2. VM_LIB angles

The VM_LIB library and the BLOB library work with float angles in radians in the range **`[0,2PI]`**. These angles are called ***native angles***. The native angles increase in counter-clockwise direction and angle 0 is at 3 o'clock:



5.3.3. Scaled integers

The VM_LIB library uses the so called ***scaled*N integers*** to present fractional quantities by integer numbers. N is a power of 10 (1000 for example). Conversions between scaled integer numbers **`si`** and respective float numbers **`fn`** are done by the formulas:

```
si = (int)(fn*N);
fn = (float)si/N;
```

Example of scaled*1000 number:

```
int x = 12650;
```

Actual scaled*1000 **x** value = `12.650`

Pixel x/y coordinates are usually expressed with 1/1000 accuracy, which is sufficient in many coordinate calculations. Using of integer calculations increases significantly the execution speed. Remember that a signed 32-bit integer can hold maximum 9 full-precision decimal digits (nine '**9**'s), so you must take care about overflows in scaled integer multiplications.

INFORMATION. *Some functions (SC generator) may use scaled integers with other scale values != 1000.*

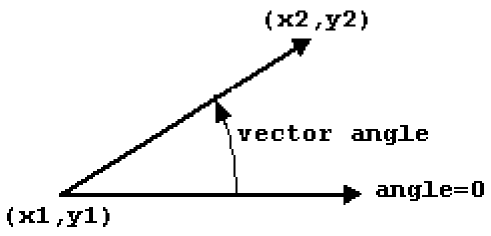
cl_vect_angle = Calculate vector angle

Prototype:

```
#include "cl_lib.h"
float cl_vect_angle ( float x1, float y1, float x2, float y2 )
```

Description:

The function calculates the angle between the input vector **`(x1,y1) (x2,y2)`** and the vector with angle 0, which is at 3 o'clock. The angle increases in counter-clockwise direction.



Arguments:

Argument	Description
x1	[in] x-coordinate of beginning vector point
y1	[in] y-coordinate of beginning vector point
x2	[in] x-coordinate of end vector point (vector tip)
y2	[in] y-coordinate rd of end vector point (vector tip)

Return code:

Return code	Description
`>=0`	Vector angle in radians in the range [0,2PI]
`-1`	Not available, degenerate vector: **`x1==x2 && y1==y2`**

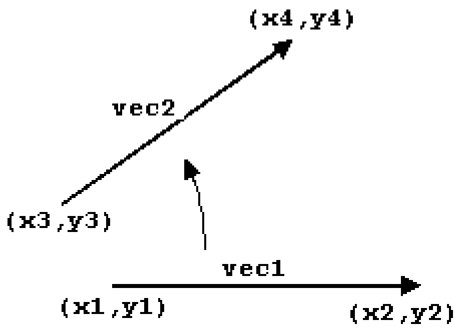
cl_vect_angle2 = Calculate angle between two vectors

Prototype:

```
float cl_vect_angle2 ( float vec1[4], float vec2[4] )
```

Description:

The function calculates the angle between the two vectors **`vec1`** and **`vec2`**, each specified by beginning and end points. The result angle is in radians in the range **`[0,2*PI]`**. The angle is measured from **`vec1`** to **`vec2`** in counterclockwise direction.



Arguments:

Argument	Description
`vec1`	[in] 4-element buffer with 1st vector points: **`(x1,y1) (x2,y2)`**
`vec2`	[in] 4-element buffer with 2nd vector points: **`(x3,y3) (x4,y4)`**

Return code:

Angle between two vectors in radians.

cl_rot_pnt = Rotate point

Prototype:

```
#include "cl_lib.h"
void cl_rot_pnt ( float *x, float *y, float ang )
```

Description:

The function rotates a point **`(x,y)`** around the origin of the coordinate system **`(0,0)`** by an angle in radians in the range [0,2*PI]. Angle 0 is at 3 o'clock and the angle increases counter clockwise.

Arguments:

Argument	Description
x	[i/o] point x-coordinate
y	[i/o] point y-coordinate
ang	[in] rotation angle in radians

Return code:

None

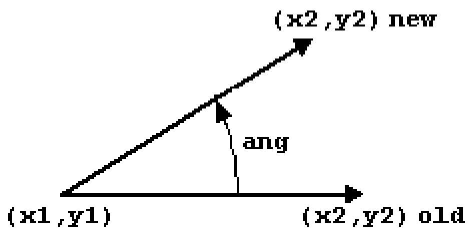
cl_rot_vect = Rotate vector

Prototype:

```
#include “cl_lib.h”
void cl_rot_vect ( float x1, float y1, float *x2, float *y2, float ang )
```

Description:

The function rotates a vector, defined by two points **(x1,y1)** and **(x2,y2)**, around the first point **(x1,y1)** by an angle in radians in the range **[0,2*PI]**. The function works in place - the results are stored into the second vector point **(x2,y2)**. The angle increases in counter-clockwise direction.



Arguments:

Argument	Description
x1	[in] x-coordinate of beginning vector point
y1	[in] y-coordinate of beginning vector point
x2	[i/o] x-coordinate of end vector point
y2	[i/o] y-coordinate of end vector point
ang	[in] rotation angle in radians

Return code:

None

cl_rot_rect = Rotate rectangle

Prototype:

```
#include “cl_lib.h”
void cl_rot_rect ( float rect[8], float x0, float y0, float ang )
```

Description:

The function rotates the input/output rectangle **rect** around the rotation point **(x0,y0)**. The rotation angle is in radians in the range **[0,2*PI]** and increases counter-clockwise. Te rotated corners are stored back in **rect**.

Arguments:

Argument	Description
rect	[i/o] buffer with 4 rectangle corners: `(x1,y1) (x2,y2) (x3,y3) (x4,y4)`
x0	[in] x-coordinate of rotation point
y0	[in] y-coordinate of rotation point
ang	[in] rotation angle in radians

Return code:

None

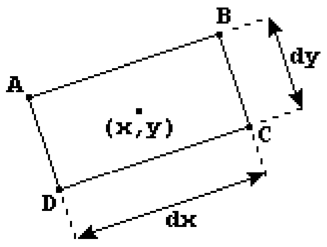
cl_rect_corners = Calculate rectangle corners

Prototype:

```
#include “cl_lib.h”
void cl_rect_corners ( int x, int y, int dx, int dy, int pt_pos,
                       float ang, float rect[8] )
```

Description:

The function calculates the x/y coordinates of the 4 corners of a rotated rectangle, defined by a rectangle-origin point **(x,y)**, width **dx**, height **dy** and rotation angle **ang** around the origin point:



The x/y coordinates of the 4 rectangle corners A, B, C, D are stored into the 8-element destination buffer **rect**:

```
  A = rect[0,1] = (x1,y1)
  B = rect[2,3] = (x2,y2)
  C = rect[4,5] = (x3,y3)
  D = rect[6,7] = (x4,y4)
```

The rotation angle is in radians in the range **[0,2*PI]** and increases in counter-clockwise direction.

Arguments:

Argument	Description
x	[in] x-coordinate of rectangle-origin point
y	[in] y-coordinate of rectangle-origin point
dx	[in] rectangle width in pixels
dy	[in] rectangle height in pixels
pt_pos	[in] position of rectangle-origin point **(x,y)**: `0` = at rectangle center `1` = at top/left rectangle corner

`ang`	[in] rotation angle in radians [0,2PI] around the origin point
`rect`	[out] destination buffer with 4 rectangle corners: `(x1,y1) (x2,y2) (x3,y3) (x4,y4)`

Return code:
None

cl_round = Round float to integer

Prototype:
```
#include “cl_lib.h”
int cl_round ( float x )
```

Description:
The function rounds the input float number to nearest integer and returns rounded integer value.

Arguments:

Argument	Description
x	[in] float number to round

Return code:
Rounded integer value

si_round = Round scaled*1000 integer to integer

Prototype:
```
#include “cl_lib.h”
int si_round ( int x )
```

Description:
The function rounds the input scaled*1000 integer number to nearest integer and returns the rounded value.

Examples:

X = 12650	X = 12499
Actual X value = 12.650	Actual X value = 12.499
The function returns 13	The function returns 12

Arguments:

Argument	Description
x	[in] scaled*1000 integer number

Return code:
Rounded integer value.

si_round_100 = Round scaled*100 integer to integer

Prototype:
```
#include “cl_lib.h”
int si_round_100 ( int x )
```

Description:
The function rounds the input scaled*100 integer number to nearest integer and returns the rounded value.

Examples:

X = 1265	X = 1249
Actual X value = 12.65	Actual X value = 12.49
The function returns 13	The function returns 12

Arguments:

Argument	Description
x	[in] scaled*1000 integer number

Return code:
Rounded integer value.

cl_gen_line_equ = Generate line equation

Prototype:
```
#include “cl_lib.h”
void cl_gen_line_equ ( float x1, float y1, float  x2, float y2,
                       float coef[3] )
```

Description:
The function calculates the coefficients A, B and C of the equation:

```
    A*x + B*y + C = 0
```

of a line, defined by two points **(x1,y1)** and **(x2,y2)**.

Arguments:

Argument	Description
x1	[in] x-coordinate of 1st line point

y1	[in] y-coordinate of 1st line point
x2	[in] x-coordinate of 2nd line point
y2	[in] y-coordinate of 2nd line point
coef	[out] buffer with 3 line equation coefficients A, B and C

Return code:
None

cl_parallel_line = Calculate parallel line

Prototype:
```
#include "cl_lib.h"
int cl_parallel_line ( float coef[3], float x, float y )
```

Description:
The function calculates the coefficients A, B and C of a line, which passes through a given point **(x,y)** and is parallel to another line. The equation of the input/output line is:
```
    A*x + B*y + C = 0
```
The equation of the input line must be in format, produced by the function **cl_gen_line_equ**:
```
    B = -1.0 or B = 0.0.
```

Arguments:

Argument	Description
coef	[i/o] 3-element line equation buffer (A,B,C)
x	[in] x-coordinate of a point, which defines (lies on) the output line
y	[in] y-coordinate of a point, which defines (lies on) the output line

Return code:

Return code	Description
0	OK
1	Invalid equation of input line: `B!=-1 && B!=0`

cl_2pt_perp_line = Calculate 2-pt perpendicular line

Prototype:
```
#include "cl_lib.h"
int cl_2pt_perp_line ( float line1[4], float line2[4], float dist )
```

Description:
The function finds two points on a line, which is perpendicular to the input line **line1** and goes through the beginning **line1** point **(x1,y1)**. The two result points **(xl,yl)** and **(xr,yr)** are

located on the result line left and right from **`(x1,y1)`**at distance **`dist`**. The input and the result line coordinates are float.

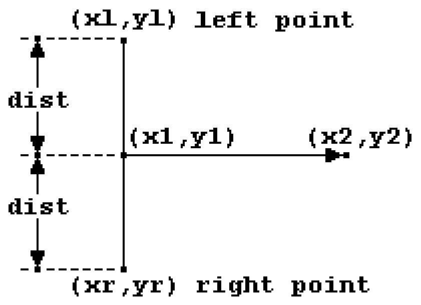
The function finds oriented result points, relative to the direction of the input line **`(x1,y1)`** **`->`** **`(x2,y2)`**. The function stores the result points into the **`line2`** buffer in format:

```
  (xl,yl) (xr,yr)
```

where:

`(xl,yl)`	=	left point on the perpendicular line
`(xr,yr)`	=	right point on the perpendicular line

(see the picture below)



Arguments:

Argument	Description
`line1`	[in] 4-element line buffer: `(x1,y1) (x2,y2)`
`line2`	[out] 4-element perpendicular line buffer: `(xl,yl) (xr,yr)`
`dist`	[in] distance from `(x1,y1)` to each of the result points `(xl,yl)` and `(xr,yr)`

Return code:

Return code	Description
`0`	OK
`CL_INVALID_ARG`	Invalid input line `(dx=0 && dy=0)`

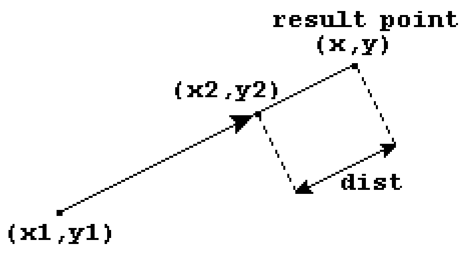
cl_prolong_line = Prolong line

Prototype:

```
#include "cl_lib.h"
void cl_prolong_line ( int x1, int  y1, int  x2, int  y2, int dist,
                       int *x, int *y )
```

Description:

The function prolongs the input line, defined by the points **`(x1,y1)`** and **`(x2,y2)`**, to the direction of the second point by **`dist`** pixels, and returns the coordinates of the new end point **`(x,y)`** of the prolonged line.



Arguments:

Argument	Description
x1	[in] x-coordinate of 1st line point
y1	[in] y-coordinate of 1st line point
x2	[in] x-coordinate of 2nd line point
y2	[in] y-coordinate of 2nd line point
dist	[in] prolongation distance in pixels
x	[out] x-coordinate of result point
y	[out] y-coordinate of result point

Return code:
None

cl_dist = Calculate distance between points

Prototype:
```
#include “cl_lib.h”
float cl_dist ( float x1, float y1, float x2, float y2 )
```

Description:
The function calculates the Euclidean distance between two points **(x1,y1)** and **(x2,y2)**.

Arguments:

Argument	Description
x1	[in] x-coordinate of 1st point
y1	[in] y-coordinate of 1st point
x2	[in] x-coordinate of 2nd point
y2	[in] y-coordinate of 2nd point

Return code:
Euclidean distance b/n points

cl_pnt_in_convex = Check for point inside convex polygon

Prototype:

```
#include “cl_lib.h”
int cl_pnt_in_convex ( int x, int y, int *pbuf, int cnt )
```

Description:

The function checks if the input point **(x,y)** is inside a convex polygon, defined by its corners.

Algorithm:

The function calculates the signed distance from the point to each polygon line. If all distances have equivalent signs, then the point is inside the polygon.

Arguments:

Argument	Description
x	[in] x-coordinate of tested point
y	[in] y-coordinate of tested point
pbuf	[in] buffer with x/y coordinates of **cnt** sequential polygon corner points: `(x1,y1) (x2,y2) (x3,y3) ...`
cnt	[in] number of polygon corners

Return code:

Return code	Description
0	Point outside polygon
1	Point inside polygon

cl_pnt_in_box_xy = Check for point inside box (X/Y)

Prototype:

```
#include “cl_lib.h”
int cl_pnt_in_box_xy ( int x, int y, int box_x1, int box_y1,
                       int box_x2, int box_y2 )
```

Description:

The function checks if the point **(x,y)** is inside a box, defined by its top/left and bottom/right corners.

Arguments:

Argument	Description
x	[in] x-coordinate of tested point
y	[in] y-coordinate of tested point
box_x1	[in] x-coordinate of top/left box corner

box_y1	[in] y-coordinate of top/left box corner
box_x2	[in] x-coordinate of bottom/right box corner
box_y2	[in] y-coordinate of bottom/right box corner

Return code:

Return code	Description
0	Point outside box
1	Point inside box

cl_pnt_in_box = Check for point inside box

Prototype:

```
#include “cl_lib.h”
int cl_pnt_in_box ( int x, int y, int box_x, int box_y, int dx, int dy )
```

Description:

The function checks if the point **(x,y)** is inside a box, defined by top/left corner, width and height.

Arguments:

Argument	Description
x	[in] x-coordinate of tested point
y	[in] y-coordinate of tested point
box_x	[in] x-coordinate of top/left box corner
box_y	[in] y-coordinate of top/left box corner
dx	[in] box width in pixels
dy	[in] box height in pixels

Return code:

Return code	Description
0	Point outside box
1	Point inside box

pt_inbox = Check for point inside VM box

Prototype:

```
#include “cl_lib.h”
int pt_inbox ( int x, int y, VM_BOX *box )
```

Description:
The function checks if the point **`(x,y)`** is inside an input box of type VM_BOX.

Arguments:

Argument	Description
`x`	[in] x-coordinate of tested point
`y`	[in] y-coordinate of tested point
`box`	[in] VM_BOX structure **`x,y,dx,dy`**

Return code:

Return code	Description
`0`	Point outside box
`1`	Point inside box

cl_sin = Calculate sine

Prototype:
```
#include “cl_lib.h”
float cl_sin ( float ang )
```

Description:
The function calculates the sine function with exact result values for angles **`k*PI/2`**.

Arguments:

Argument	Description
`ang`	[in] function argument in radians

Return code:
None

cl_cos = Calculate cosine

Prototype:
```
#include “cl_lib.h”
float cl_cos ( float ang )
```

Description:
The function calculates the cosine function with exact result values for angles **`k*PI/2`**.

Arguments:

Argument	Description
ang	[in] function argument in radians

Return code:
None

cl_check_angle = Check angle

Prototype:

```
#include “cl_lib.h”
int cl_check_angle ( float ang, float eps )
```

Description:
The function checks if the input angle **ang** is approximately equal to **k*PI/2** and returns **k**.

Arguments:

Argument	Description
ang	[in] angle in radians
eps	[in] allowed angle tolerance in radians = max difference between **ang** and **k*PI/2**

Return code:

Return code	Description
-1	Angle not equal to k*PI/2
0	Angle equal to 0 (2*PI)
1	Angle equal to PI/2
2	Angle equal to PI (2*PI/2)
3	Angle equal to 3*PI/2

cl_line_cross = Calculate cross point of 2 lines (by equations)

Prototype:

```
#include “cl_lib.h”
int cl_line_cross ( float coef1[3], float coef2[3], float *x, float *y )
```

Description:
The function calculates the coordinates of the cross point of two lines, defined by their equations:

```
    A1*x + B1*y + C1 = 0
    A2*x + B2*y + C2 = 0
```

The line equations must be generated by **cl_gen_line_equ**.

Arguments:

Argument	Description
coef1	[in] 3-element equation buffer of 1st line (A1,B1,C1)
coef2	[in] 3-element equation buffer of 2nd line (A2,B2,C2)
x	[out] x-coordinate of result cross point
y	[out] y-coordinate of result cross point

Return code:

Return code	Description
0	OK
1	No result (parallel lines)

cl_gen_line = Generate line (X/Y coordinates)

Prototype:

```
#include “cl_lib.h”
int cl_gen_line ( int x1, int y1, int x2, int y2, int **xy_buf )
```

Description:

The function generates a line and stores the x/y coordinates of the line points in the output buffer **xy_buf**. The line starts at **(x1,y1)** and ends at **(x2,y2)**. The code is based on the Bresenham's algorithm. To achieve higher speed the code is divided into partial cases for horizontal, vertical, 45-degree and arbitrary lines.

The output buffer **xy_buf** is allocated by this function and must be freed by the calling function. The size of the allocated buffer is:

```
   (max(abs(x2 - x1),abs(y2 - y1)) + 1) points,
```

each point with size `2*sizeof(int)` bytes.

Arguments:

Argument	Description
x1	[in] x-coordinate of 1st line point
y1	[in] y-coordinate of 1st line point
x2	[in] x-coordinate of 2nd line point
y2	[in] y-coordinate of 2nd line point
xy_buf	[out] buffer with coordinates of line points in format: `(x,y) (x,y) (x,y) ...` (NULL= memory allocation error)

Return code:

Return code	Description

>=0	Number of points (x,y), stored in **xy_buf**
-1	**xy_buf** allocation error

cl_gen_ellip = Generate ellipse (X/Y coordinates)

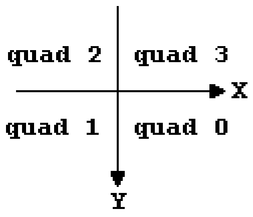
Prototype:

```
#include “cl_lib.h”
int cl_gen_ellip ( int x0, int y0, unsigned int r1, unsigned int r2,
                   int quad, int **xy_buf )
```

Description:

The function generates an ellipse and stores the x/y coordinates of the ellipse points in the output buffer **xy_buf**. The code is based on the Bresenham's algorithm. The output buffer **xy_buf** is allocated by this function and must be freed by the calling function. The size of the allocated buffer depends on the ellipse size and the **quad** parameter: 1, 2 or 4 quadrants (full ellipse).

The ellipse quadrants are:



Arguments:

Argument	Description
x0	[in] x-coordinate of ellipse center
y0	[in] y-coordinate of ellipse center
r1	[in] horizontal ellipse radius
r2	[in] vertical ellipse radius
quad	[in] number ellipse quadrants to generate: 1 : generate quadrant 0 2 : generate quadrants 0 and 1 4 : generate all four quadrants 0-3 (full ellipse)
xy_buf	[out] buffer with coordinates of ellipse points in format: `(x,y) (x,y) (x,y) ...` NULL= memory allocation error or r1=0 and r2=0 (# of generated points = 0)

Return code:

Return code	Description
>=0	Number of points (x,y), stored in **xy_buf**
-1	Memory allocation error

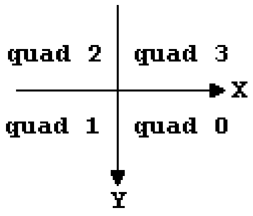
cl_ellip_quad = Generate ellipse quadrant

Prototype:

```
#include “cl_lib.h”
int cl_ellip_quad ( unsigned int r1, unsigned int r2, int **xy_buf )
```

Description:

The function generates one ellipse quadrant (1/4 of ellipse) and stores the x/y coordinates of the ellipse points in the output buffer **`xy_buf`**. The function generates ellipse quadrant 0 (see the picture below). The ellipse center is **`(0,0)`**. The code is based on the Bresenham's algorithm. The output buffer **`xy_buf`** is allocated by this function and must be freed by the calling function. In case of error, this function frees **`xy_buf`** itself. Note: The function stores in the output buffer the coordinates of the two border quadrant points. The ellipse quadrants are:



Arguments:

Argument	Description
`r1`	[in] horizontal ellipse radius
`r2`	[in] vertical ellipse radius
`xy_buf`	[out] buffer with x/y coordinates of ellipse points in format: `(x,y) (x,y) (x,y) ...` NULL= memory allocation error or r1=0 and r2=0 (# of generated points = 0)

Return code:

Return code	Description
`>=0`	Number of points `(x,y)`, stored in **`xy_buf`**
`-1`	Memory allocation error
`-2`	Internal count mismatch error

cl_poly_to_seg = Generate polygon segment definition

Prototype:

```
#include “cl_lib.h”
int cl_poly_to_seg ( int *pt_buf, int pt_cnt, SEG_DEF *seg_def )
```

Description:

The function generates segment definition data for a convex polygon `P1,P2,...,Pk (P1 != Pk)`, defined by its corners. The function stores results into the output segment definition structure **`seg_def`**. The **`seg_buf`** buffer (a member of **`seg_def`**) contains the minimum and the maximum x-coordinates of all pixel line segments, which are covered by the polygon:

```
                         seg_buf[2*i] seg_buf[2*i+1]
                         ------------ -------------
  i = min_y          :      min_x        max_x
  i = min_y + 1      :      min_x        max_x
  . . . . . . . . . . . . . . . . . . . . . . .
  i = min_y + dy - 1 :      min_x        max_x
where:
    min_x = min x-coord. of current segment
    max_x = max x-coord. of current segment
    dy    = # of segments
    seg_buf size = dy*2 integer elements
```

***ATTENTION**. The **`seg_buf`** buffer in the output **`seg_def`** structure is allocated by this function and must be freed by the calling function. In case of error this function frees **`seg_buf`**.*

Arguments:

Argument	Description
`pt_buf`	[in] buffer with x/y coordinates of **`pt_cnt`** sequential convex corners in format: `(x,y) (x,y) ...`
`pt_cnt`	[in] number of convex corner points (x,y pairs)
`seg_def`	[out] segment-definition structure (**`seg_buf`** = NULL: NA)

Return code:

Return code	Description
`0`	Success
`CL_ALLOC_ERROR`	Memory allocation error
`CL_INTERNAL_ERR`	Internal error (data corrupted)

cl_ellip_to_seg = Generate ellipse segment definition

Prototype:

```
#include “cl_lib.h”
int cl_ellip_to_seg ( int x0, int y0, unsigned int r1, unsigned int r2,
                      SEG_DEF *seg_def )
```

Description:

The function generates segment definition data for non-rotated circle or ellipse, defined by **`x0,y0`**, **`r1`** and **`r2`**. The function stores the results into the output segment definition structure **`seg_def`**. The **`seg_buf`** buffer (a member of **`seg_def`**) contains the minimum and the maximum x-coordinates of the ellipse pixel lines:

```
                         seg_buf[2*i] seg_buf[2*i+1]
                         ------------ -------------
  i = min_y          :      min_x        max_x
  i = min_y + 1      :      min_x        max_x
  . . . . . . . . . . . . . . . . . . . . . . .
  i = min_y + dy - 1 :      min_x        max_x
```

```
where:
   min_x = min x-coord. of current segment
   max_x = max x-coord. of current segment
   dy    = # of segments
   seg_buf size = dy*2 integer elements
```

***ATTENTION**. The **`seg_buf`** buffer in the output **`seg_def`** structure is allocated by this function and must be freed by the calling function. In case of error this function frees **`seg_buf`**.*

Arguments:

Argument	Description
`x0`	[in] x-coordinate of ellipse center
`y0`	[in] y-coordinate of ellipse center
`r1`	[in] horizontal ellipse radius
`r2`	[in] vertical ellipse radius
`seg_def`	[out] segment-definition structure (**`seg_buf`** = NULL: NA)

Return code:

Return code	Description
`0`	Success
`CL_ALLOC_ERROR`	Memory allocation error
`CL_INVALID_ARG`	Invalid ellipse parameters
`CL_INTERNAL_ERR`	Internal error (data corrupted)
Other	Returned by called functions

cl_rot_xybuf = Rotate X/Y buffer

Prototype:

```
#include “cl_lib.h”
void cl_rot_xybuf ( int *xybuf, int pt_cnt, float x0, float y0, float ang )
```

Description:

The function rotates the points in the input/output buffer **`xybuf`** around the rotation point **`(x0,y0)`**. The rotation angle **`ang`** is in radians in the range `[0,2*PI]` and increases in counter-clockwise direction. The input/output buffer contains integer coordinates.

Arguments:

Argument	Description
`xybuf`	[i/o] buffer with x/y point coordinates: `(x,y) (x,y) ...`
`pt_cnt`	[in] number of points in **`xybuf`**

x0	[in] x-coordinate of rotation origin
y0	[in] y-coordinate of rotation origin
ang	[in] rotation angle in radians

Return code:

None

cl_rot_ellip = Calculate rotated ellipse

Prototype:

```
#include “cl_lib.h”
int cl_rot_ellip ( float xcen, float ycen, float a, float b, float phi,
                   SEG_DEF *seg_def )
```

Description:

The function generates segment definition data for a rotated ellipse and stores the results into the output segment definition structure **seg_def**. The **seg_buf** buffer (a member of **seg_def**) contains the minimum and the maximum x-coordinates of the ellipse pixel lines:

```
                        seg_buf[2*i] seg_buf[2*i+1]
                        ------------ --------------
  i = min_y          :      min_x        max_x
  i = min_y + 1      :      min_x        max_x
  . . . . . . . . . . . . . . . . . . . . . . .
  i = min_y + dy - 1 :      min_x        max_x
where:
    min_x = min x-coord. of current segment
    max_x = max x-coord. of current segment
    dy    = # of segments
    seg_buf size = dy*2 integer elements
```

__ATTENTION__. The __seg_buf__ buffer in the output __seg_def__ structure is allocated by this function and must be freed by the calling function. In case of error this function frees __seg_buf__.

Algorithm

The function code is based on the QuickEllipse() open-source function, written by James Tough, 7th May 1992. QuickEllipse uses the same method as Ellipse, but uses incremental methods to reduce the amount of work that has to be done inside the main loop. The speed increase is very noticeable.

An Ellipse Generator.

Written by James Tough   7th May 92

The following routines displays a filled ellipse on the screen from the semi-minor axis 'a', semi-major axis 'b' and angle of rotation 'phi'. It works along these principles .....

The standard ellipse equation is:

```
    x*x     y*y
    ---  +  ---  = 0
    a*a      b*b
```

Rotation of a point (x,y) is well known through the use of

```
    x' = x*COS(phi) - y*SIN(phi)
    y' = y*COS(phi) + y*COS(phi)
```

Taking these to together, this gives the equation for a rotated ellipse centered around the origin.

```
    [x*COS(phi) - y*SIN(phi)]^2   [x*SIN(phi) + y*COS(phi)]^2
    --------------------------- + --------------------------- = 0
               a*a                            b*b
```

Some of the above equation can be pre-computed:

```
    i = COS(phi)/a        and    j = SIN(phi)/b
```

y is constant for each line so,

```
    m = -yk*SIN(phi)/a    and     n = yk*COS(phi)/b
```

where yk stands for the k-th line (y subscript k)

Where yk=y, we get a quadratic,

```
    (i*x + m)^2 + (j*x + n)^2 = 1
```

Thus for any particular line, y, there is two corresponding x values. These are the roots of the quadratic. To get the roots, the above equation can be rearranged using the standard method,

```
         -(i*m + j*n) +- sqrt[i^2 + j^2 - (i*n -j*m)^2]
     x = ----------------------------------------------
                          i^2 + j^2
```

NOTE - again much of this equation can be pre-computed.

```
    c1 = i^2 + j^2
    c2 = [COS(phi)*SIN(phi)*(a-b)]
    c3 = [b*b*(COS(phi)^2) + a*a*(SIN(phi)^2)]
    c4 = a*b/c3
    x = c2*y +- c4*sqrt(c3 - y*y),
```

where +- must be evaluated once for plus, and once for minus.

We also need to know how large the ellipse is. This condition arises when the **sqrt** of the above equation evaluates to zero. Thus the height of the ellipse is give by

```
    sqrt[ b*b*(COS(phi)^2) + a*a*(SIN(phi)^2) ]
```

which just happens to be equal to sqrt(c3).

It is now possible to create a routine that will scan convert the ellipse on the screen.

NOTES:

```
     c2 is the gradient of the new ellipse axis.
     c4 is the new semi-minor axis, 'a'.
     sqr(c3) is the new semi-major axis, 'b'.
```

These values could be used in a 4WS or 8WS ellipse generator that does not work on rotation, to give the feel of a rotated ellipse. These ellipses are not very accurate and give visible bumps along the edge of the ellipse. However, these routines are very quick, and give a good approximation to a rotated ellipse.

Arguments:

Argument	Description
xcen	[in] x-coordinate of ellipse center
ycen	[in] y-coordinate of ellipse center
a	[in] horizontal ellipse radius
b	[in] vertical ellipse radius
phi	[in] rotation angle in radians
seg_def	[out] ellipse segment-definition structure (**seg_buf** = NULL: NA)

Return code:

Return code	Description
0	Success
-1	Memory allocation error

cl_isqrt = Fast inverse SQRT

Prototype:

```
#include “cl_lib.h”
float cl_isqrt ( float x )
```

Description:

The function calculates fast inverse square root of IEEE single-precision floating-point number **x** and returns a single-precision result:

```
   Result = 1 / sqrt(x)
```

Use this function on cameras without floating-point ALU to achieve significant improvement of SQRT calculations. If you need normal SQRT, use the formula:

```
   SQRT(x) = x * cl_isqrt (x)
```

Algorithm:

Original:	Created: Ken Turkowski: lookup bits <= 8 (Graphics Gems)
Modified:	Plamen Kozhuharov, 31.01.2006:
	Uses 12-bit lookup table for 1 iteration (2 times faster)

Arguments:

Argument	Description
x	[in] single-precision function argument

Return code:

```
1/sqrt(x)
```

cl_minma = Find local min/max values

Prototype:

```
#include "cl_lib.h"
int cl_minma ( int *in_buf, int in_cnt, int mode, int minmax_type,
               int threshold, int *res_buf, int *res_cnt )
```

Description:

The function finds sequential alternating local maximums or minimums in the input buffer **in_buf** with detection sensitivity **threshold**:

```
   min, max, min, max, ...
```

The **minmax_type** argument specifies the type of the searched minimums/maximums.

The function works on non-circular or circular input point-list as specified by the **mode** argument. Use the circular mode if the input buffer contains circular data, for example circle pixels. The function appends the beginning item at the end of the input buffer to perform circular operation.

Format of result buffer **res_buf**:

```
   (idx, type, val) (idx, type, val) ...
```

where:

`idx`	=	**in_buf** index of detected local min/max
`type`	=	min/max type: 0=minimum, 1=maximum
`val`	=	min/max strength - greater values mean greater magnitudes of the detected peaks and valleys (always positive)

Algorithm

Based on Joerg Beutel's code for local min/max detection, used in the "Point-list edge detection" tool.

The **threshold** (**T**) argument specifies min/max detection sensitivity – greater threshold values search greater peaks and valleys in the input data **P[]**:

P[imax] is a local maximum if there is a next element **P[k] (k>imax)**, such that:

P[imax]–T > P[k].

P[imin] is a local minimum if there is a next element **P[k] (k>imin)**, such that:

P[imin]+T < P[k].

After each found min/max value, the function switches the search direction – from min to max or vice versa.

Arguments:

Argument	Description
in_buf	[in] buffer to scan for local min/max values
in_cnt	[in] number of **in_buf** elements
mode	[in] operation mode: bit 0 = circular mode bit 1 = spare bit 2 = apply 1st derivative on input data before local min/max: `(P[i+1] – P[i])` bit 3 = use absolute values of input data. If both bits 2 and 3 are set, absolute data is calculated after the 1st derivative calculation.
minmax_type	[in] type of searched local minimum/maximums:

	0 = find local minimums 1 = find local maximums 2 = find both
`threshold`	[in] min/max detection threshold
`res_buf`	[out] result buffer in format: `(idx, type, val) (idx, type, val) ...`
`res_cnt`	[i/o] result count: [in] : **`res_buf`** size in items `(idx,type,val)`, each item = 3 ints [out] : actual number of stored **`res_buf`** items

Return code:

Return code	Description
`0`	Success
`CL_ALLOC_ERROR`	Memory allocation error
`CL_RESBUF_OVF`	Result buffer overflow (res_cnt results stored in res_buf)

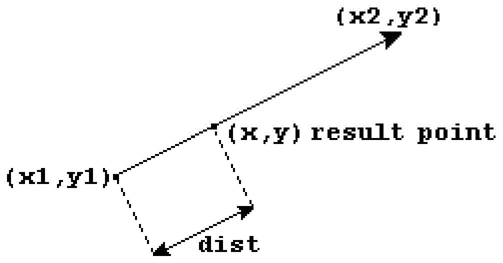
cl_line_pt = Calculate line point

Prototype:

```
#include “cl_lib.h”
int cl_line_pt ( float line[4], float dist, float *x, float *y )
```

Description:

The function finds a points on a line at distance **`dist`** from the beginning line point **`(x1,y1)`**. The input line and the coordinates of the result point are float.



The distance is signed:

```
dist > 0 : find line point (x,y) in the direction (x1,y1) -> (x2,y2)
dist < 0 : find line point (x,y) in the opposite direction
```

Arguments:

Argument	Description
`line`	[in] 4-element line buffer: `(x1,y1) (x2,y2)`
`dist`	[in] distance from `(x1,y1)` to the result point
`x`	[out] x-coordinate of result point
`y`	[out] y-coordinate of result point

Return code:

Return code	Description
`0`	OK
`CL_INVALID_ARG`	Invalid input line (dx=0 && dy=0)

cl_line_segm = Line segmentation (division into equal parts)

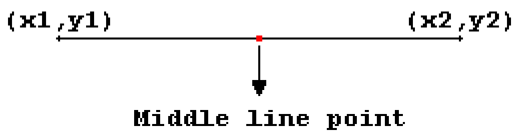
Prototype:

```
#include "cl_lib.h"
#include "ip_lib.h"
int ip_line_segm (float x1, float y1, float x2, float y2, int pt_cnt,
                  int **res_xy, float *line_len, float *seg_len )
```

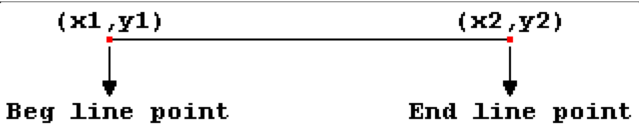
Description:

The function divides the line **`(x1,y1) (x2,y2)`** into N equal segments and returns the x/y coordinates of the segment-division points on the line. The **`pt_cnt`** input argument defines the number of result points:

`pt_cnt = 1:` return the middle line point:

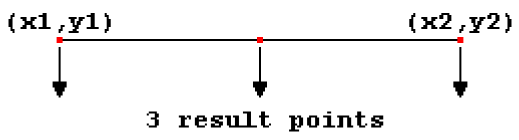


`pt_cnt = 2:` return the beginning and the end line points:

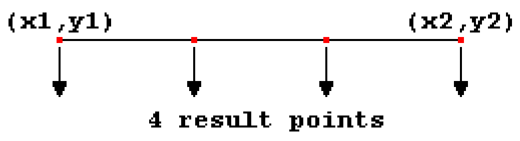


`pt_cnt >= 3:` divide the line by (**`pt_cnt-1`**) equal segments and return the division points, including the beginning and the end line points:

`pt_cnt = 3:`



`pt_cnt = 4:`



. . . . . .

The function calculates the x/y-coordinates of the result points with 1/1000 pixel accuracy. It allocates a result buffer for **`pt_cnt`** points and stores scaled*1000 integer coordinates in the result buffer in format:

`(x,y)  x,y) ... (x,y)    :` **`pt_cnt`** points (**`pt_cnt*2`** integers)

***ATTENTION**. The calling function must free the result buffer **`res_xy`** ! On error return this function frees the allocated result buffer.*

The function also returns (if enabled) the total length of the line and length of the segment, which divides the line into equal parts.

Note:

The **cl_line_segm** is a macro, which defines an alias name of the image processing function **ip_line_segm**. This function is actually a part of the image processing library, but is included in here because it perfectly fits by purpose in the calculation and geometry library.

Arguments:

Argument	Description
`x1`	[in] x-coordinate of beginning line point
`y1`	[in] y-coordinate of beginning line point
`x2`	[in] x-coordinate of end line point
`y2`	[in] y-coordinate of end line point
`pt_cnt`	[in] number of result line segment points
`res_xy`	[out] buffer with pixel coordinates
`line_len`	[out] line length (NULL: don't return)
`seg_len`	[out] segment length (NULL: don't return)

Return code:

Return code	Description
`0`	Success
`IP_ALLOC_ERROR`	Memory allocation error
`IP_INV_ARG`	Invalid function argument(s)
Other	Other errors returned by called functions

cl_sort = Sort general multi-field buffer

Prototype:

```
#include “cl_lib.h”
int cl_sort ( void *buf, int el_cnt, int el_size, int data_offs,
              int data_type, int sort_dir )
```

Description:

The function sorts a buffer, composed of structures or other multi-field elements in respect to a structure member or data field with given type and length (see the **data_type** argument). The supported data are **CL_DATA_TYPES**, defined in **cl_lib.h**.

The function uses the **qsort()** C run-time library function.

Example:

```
#include <stddef.h>     /* offsetof() definition  */
#include “cl_lib.h”

    typedef struct
    {
      int    i;
      float  f;
    } BUF_EL;

    BUF_EL buf[5] = { ... };

/* Sort buf by the 'i' field in ascending order: */
    cl_sort(buf, 5, sizeof(BUF_EL), offsetof(BUF_EL,i), CL_INT, 0);

/* Sort buf by the 'f' field in descending order: */
    cl_sort(buf, 5, sizeof(BUF_EL), offsetof(BUF_EL,f), CL_FLOAT, 1);
```

Arguments:

Argument	Description
buf	[i/o] data buffer to sort
el_cnt	[in] # of **buf** elements (data structures)
el_size	[in] size of buffer element in bytes
data_offs	[in] byte offset of the element data field (structure member) to sort by

data_type	[in] type of data element field:
	CL_CHAR = 8-bit char
	CL_UCHAR = 8-bit unsigned char
	CL_SHORT = 16-bit integer
	CL_USHORT = 16-bit unsigned integer
	CL_INT = 32-bit integer
	CL_UINT = 32-bit unsigned integer
	CL_INT64 = 64-bit integer
	CL_UINT64 = 64-bit unsigned integer
	CL_FLOAT = 32-bit float
	CL_DOUBLE = 64-bit float
	CL_STR = 0-terminated string with arbitrary length

sort_dir	[in] sort direction: 0 = ascending order 1 = descending order

Return code:

Return code	Description
0	Success
CL_INVALID_ARG	Invalid data field offset (data_offs + data length > el_size) or unsupported data type

Fast integer SIN/COS generator

The SIN/COS generator allows fast image rotation with integer calculations by the function **sc_rot_img**. You can use fixed number of rotation angles, uniformly distributed in the range [0,2PI]. The number of angles is set by the **sc_size** argument when you open the SIN/COS generator by the **sc_gen_open** function.

Example:

```
#include “cl_lib.h”

  int   rc = 0;
  SC_HND hnd = NULL;
  image img;                /* source image to rotate                  */
  image img1={0,0,0,0};     /* destination rotated image               */
  int   sc_size = 360;      /* # of rotation angles (step = 1 degree) */
  int   ang_id  = 45;       /* rotation angle id = 45 degrees          */
  int   pix_fill = 128;     /* pixel value to fill                     */
  int   bw_mode = 0;        /* B&W mode: 0=gray image, 1=B&W image     */
  int   bw_clr1 = 0;        /* B&W image color 1, used if bw_mode = 1 */
  int   bw_clr2 = 255;      /* B&W image color 2, used if bw_mode = 1 */
  float ang;                /* float rotation angle in radians         */
  int   rot_dx;             /* width of destination image              */
  int   rot_dy;             /* height of destination image             */
  . . . . . . . . .

/* Open SIN/COS generator */
  hnd = sc_gen_open(sc_size, 0);
  if(!hnd) goto done;     /* open error */

/* Get angle in radians, corresponding to angle index ang_id */
  ang = sc_rot_ang(hnd, ang_id);

/* Get dimensions rot_dx, rot_dy of destination image */
  rc = sc_rot_image_size (img.dx, img.dy, ang, &rot_dx, &rot_dy);
  if(rc) goto done;

/* Allocate destination rotated image */
  rc = ip_alloc_img(rot_dx, rot_dy, &img1);
  if(rc) goto done;

  . . . . . . . . .
/*
* Rotate image by 45 degrees. Set to pix_fill all pixels in the destination
* image, which do not have respective pixels in the source image.
*/
  rc = sc_rot_img(hnd, &img, &img1, ang_id, pix_fill, bw_mode, bw_clr1, bw_clr2);
  if(rc) goto done;       /* error */
  . . . . . . . . .

/* Exit */
done:
  ip_free_img(&img1);  /* free allocated destination image */
  sc_gen_close(&hnd);  /* close SIN/COS generator          */
```

sc_gen_open = Open SIN/COS generator

Prototype:

```
#include “cl_lib.h”
```

```
SC_HND sc_gen_open ( int sc_size, int sc_fract )
```

Description:

The function opens the fast integer SIN/COS generator. It allocates memory tables, calculates and stores integer sin/cos values into the tables and returns SIN/COS generator handle.

ATTENTION*. In case of error the function frees the allocated handle memory and returns NULL.*

The function calculates **`sc_size`** sine and cosine values for angles, uniformly distributed in the range `[0,2*PI]` with step `2*PI/sc_size`. A value of 360 for example means that you can rotate images by angle step of 1 degree.

The **`fract_cnt`** argument specifies number of sin/cos fraction digits, i.e. the accuracy of used sin/cos values. The recommended **`fract_cnt`** value is 4. Greater values increase the rotation accuracy, but may cause integer overflow when rotating big images. Values of **`fract_cnt`** below 3 and above 5 are not recommended.

Arguments:

Argument	Description
`sc_size`	[in] SC generator size = # of sin/cos values uniformly distributed in the range [0,2PI], which are stored in the generator tables `0` = default : 360
`fract_cnt`	[in] number of sin/cos fraction digits [1,6]: `0` = default : 4 fraction digits

Return code:

Return code	Description
`NULL`	Error
`!=NULL`	SIN/COS generator handle

sc_gen_close = Close SIN/COS generator

Prototype:

```
#include “cl_lib.h”
void sc_gen_close (SC_HND *hnd )
```

Description:

The function closes the fast integer SIN/COS generator and frees the buffers the handle memory, allocated by **sc_gen_open**.

Arguments:

Argument	Description
hnd	[i/o] SIN/COS generator handle (set to NULL on exit)

Return code:

None

sc_rot_ang = Get rotation angle in radians

Prototype:

```
#include “cl_lib.h”
float sc_rot_ang (SC_HND hnd, int ang_id )
```

Description:

The function returns float rotation angle in radians, which corresponds to the angle index **ang_id**. The **ang_id** argument is rotation angle index in the range **[0,sc_size-1]**, where **sc_size** was used to open the SC generator. The returned angle is calculated by the formula:

```
   ang = ang_id * 2PI / sc_size
```

The angle increases in counter-clockwise direction.

Arguments:

Argument	Description
hnd	[in] SIN/COS generator handle
ang_id	[in] rotation angle index in the range [0,sc_size-1]

Return code:

Return code	Description
[0,2*PI]	Rotation angle in radians
-1. 0	Error (invalid handle)

sc_rot_image_size = Calculate dimensions of rotated image

Prototype:

```
#include “cl_lib.h”
int sc_rot_image_size ( int dx, int dy, float ang, int *rot_dx,
                        int *rot_dy )
```

Description:

The function calculates the dimensions of the result rotated image, when a source image with dimensions **dx**, **dy** is rotated by angle **ang** in radians. This function is useful to determine the dimensions (and allocation) of destination rotation image before the call to **sc_rot_img**.

Algorithm

The dimensions of the result rotated image **rot_dx**, **rot,dy** are calculated by the following formula:

```
   sin_a = abs(sin(ang))
   cos_a = abs(cos(ang))
   rot_dx = dy*sin_a + dx*cos_a
   rot_dy = dy*cos_a + dx*sin_a
```

Arguments:

Argument	Description
dx	[in] width of source image in pixels
dy	[in] height of source image in pixels
ang	[in] rotation angle in radians
rot_dx	[out] width of rotated image
rot_dy	[out] height of rotated image

Return code:

Return code	Description
0	OK
CL_INV_ARG	Invalid function argument(s)

sc_rot_img = Rotate image

Prototype:

```
#include “cl_lib.h”
int sc_rot_img (SC_HND hnd, image *in_img, image *out_img, int ang_id,
                int pix_fill, int bw_mode, int bw_clr1, int bw_clr2 )
```

Description:

The function rotates the input image **in_img** by an angle **ang_id**, using the sin/cos generator. The function stores the result rotated image into **out_img**. The function needs preliminary setup by the **sc_gen_open** function.

The **ang_id** argument specifies rotation angle index in the range **[0,sc_size-1]**, where **sc_size** was used to open the SC generator. The actual rotation angle in radians is:

```
  ang = ang_id * 2PI / sc_size
```

The angle increases in counter-clockwise direction.

The output image must be allocated by the calling function. The result rotated image is stored (eventually truncated) into the upper left corner of the output image. The rotation can be done in place:

```
  in_img == out_img.
```

The true dimensions of the rotated image can be obtained by **sc_rot_img_size**. Rotated pixels in the output image, which have no respective pixels in the input image, are filled with value **pix_fill**.

In B&W mode (**bw_mode = 1**) the function supports rotation of monochrome images, composed of 2 colors **bw_clr1** and **bw_clr2**. This mode ensures that the destination image will also be composed of the 2 colors.

Tips and tricks

You can use the B&W mode to binarize a gray-level source image on the fly while rotating it. For example:

```
  bw_mode = 1, bw_clr1 = 50, bw_clr2 = 150
```
The function rotates and binarizes the source image with threshold:
```
  (bw_clr1 + bw_clr2)/2.
```
The pixels below the threshold are set to **`bw_clr1`** and the pixels above - to **`bw_clr2`**.

Execution speed example:
Rotation of 640x480 image on 400 MHz TMS320C64xx DSP:
sc_rot_img (integer calculations) : 663 ms
ip_img_rotate (float calculations) : 1835 ms

Arguments:

Argument	Description
`hnd`	[in] SIN/COS generator handle
`in_img`	[in] source image to rotate
`out_img`	[in] destination rotated image
`ang_id`	[in] rotation angle index in the range `[0,sc_size-1]`
`pix_fill`	[in] pixel value to fill
`bw_mode`	[in] B&W image mode: `0` = gray-level input image `1` = 2-color B&W input image with colors **`bw_clr1`** and **`bw_clr2`**
`bw_clr1`	[in] B&W image color 1
`bw_clr2`	[in] B&W image color 2
`out_img`	[out] rotated image

Return code:

Return code	Description
`0`	Success
`CL_INV_ARG`	Invalid argument
`CL_ALLOC_ERROR`	Memory allocation error

5.4. Drawing library

This library contains 2-D drawing functions. All functions draw into a drawing image of type **DR_IMAGE**, currently equivalent to the **image** structure. Most of the drawing functions do not need setup.

Some functions however, for example the text drawing functions, need setup. You must call first **dr_init** (or **vrm_vmlib_dsp_init**, which calls **dr_init**) to install a default font, used by the text drawing functions.

Note: The function **dr_txt** does not need font setup.

Header files:

`dr_lib.h`	Drawing library header
`lib_err.h`	VM_LIB library error codes

The ***basic*** drawing functions are those functions, which actually set image pixels. Here is a list with the basic drawing functions:

```
dr_line        = Draw line
dr_dot_line    = Draw dotted line
dr_mat         = Draw pixel matrix
dr_bitmat      = Draw bit pixel matrix
dr_image       = Draw image
dr_clr_image   = Draw color image
dr_dot         = Draw dot
dr_fill_box_xy = Fill box with color (X/Y)
dr_fill_box    = Fill box with color
dr_fill_img    = Fill image with color
dr_clear_img   = Clear image
dr_copy_img    = Copy image
dr_ellip       = Draw ellipse
dr_rot_ellip   = Draw rotated ellipse
```

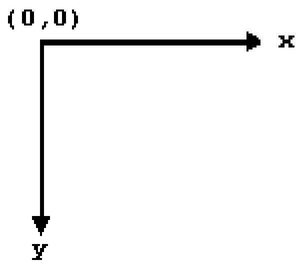
These functions also perform:

- coordinate translation relative to the drawing origin
- clipping inside current clipping box (default clipping box = the drawing image)

The ***general*** (non-basic) drawing functions call the basic functions. They do not perform coordinate clipping, coordinate translation and color conversion.

5.4.1. Coordinate system

The VM_LIB library functions use left-handed coordinate system with an origin (0,0) at the top left screen corner:



5.4.2. VM_LIB colors

The available drawing colors have values from 0 to DR_COLOR_CNT-1. Color 0 is reserved transparent color and can't be changed by **dr_set_palette** function. The next table shows the default drawing colors, set by **dr_init**. The color macros are defined in **dr_lib.h**.

Default colors

Color macro	Color value	Description
`DR_COLOR_NONE`	0	no color (transparent)
`DR_COLOR_BLUE`	1	blue
`DR_COLOR_GREEN`	2	green
`DR_COLOR_CYAN`	3	cyan
`DR_COLOR_RED`	4	red
`DR_COLOR_MAGENTA`	5	magenta
`DR_COLOR_YELLOW`	6	yellow
`DR_COLOR_GRAY`	7	gray
`DR_COLOR_BRIGHT_GRAY`	8	bright gray
`DR_COLOR_BRIGHT_BLUE`	9	bright blue
`DR_COLOR_BRIGHT_GREEN`	10	bright green
`DR_COLOR_BRIGHT_CYAN`	11	bright cyan
`DR_COLOR_BRIGHT_RED`	12	bright red
`DR_COLOR_BRIGHT_MAGENTA`	13	bright magenta
`DR_COLOR_BRIGHT_YELLOW`	14	bright yellow
`DR_COLOR_WHITE`	15	white (high intensity)
`DR_COLOR_BLACK`	16	black (foreground text color)
`DR_COLOR_DARK_GRAY`	17	dark-gray (shadow 3-D corner)
`DR_COLOR_DEF_BGND`	18	middle-gray (windows-like background)
`DR_COLOR_LIGHT_GRAY`	19	light-gray (light 3-D corner)
`DR_COLOR_TIT_BGND`	20	unselected window title background color
`DR_COLOR_TIT_SBGND`	21	selected window (on focus) title background color
`DR_COLOR_TRANSL_BLUE`	64	translucent blue
`DR_COLOR_TRANSL_GREEN`	65	translucent green
`DR_COLOR_TRANSL_RED`	66	translucent red

5.4.3. Fonts

The drawing library supports bitmap fonts with fixed character width and height in the so called **FNT** format. The default built-in font has character size 8x10 and contains full ASCII code table 0-255. A font file may contain full ASCII table or not, but must begin with an ASCII code 0.

FNT file format:

```
"FNT"
font_cnt      2-byte # of icons
font_dx       2-byte icon width in pixels
```

```
font_dy       2-byte icon width in pixels
-------------------------------------------------------+
byte,byte,...,byte  (pixel row = (font_dx+7)/8 bytes ) |
byte,byte,...,byte  (pixel row = (font_dx+7)/8 bytes ) | icon 0 (font_dy rows)
. . . . . . . . .    . . . . .                          |
byte,byte,...,byte  (pixel row = (font_dx+7)/8 bytes ) |
-------------------------------------------------------+
byte,byte,...,byte  (pixel row = (font_dx+7)/8 bytes ) |
byte,byte,...,byte  (pixel row = (font_dx+7)/8 bytes ) | icon 1
. . . . . . . . .    . . . . .                          |
byte,byte,...,byte  (pixel row = (font_dx+7)/8 bytes ) |
-------------------------------------------------------+
. . . . . . . . . . . . . . . .                        .
-------------------------------------------------------+
byte,byte,...,byte  (pixel row = (font_dx+7)/8 bytes ) |
byte,byte,...,byte  (pixel row = (font_dx+7)/8 bytes ) | icon (font_cnt-1)
. . . . . . . . .    . . . . .                          |
byte,byte,...,byte  (pixel row = (font_dx+7)/8 bytes ) |
-------------------------------------------------------+
```

5.4.4. Drawing origin

All drawings are done relative to the drawing origin, i.e. all pixel coordinates `(x,y)` are added to the coordinates of the current origin point. The default drawing origin is (0,0) = the top/left corner of the drawing image.

***ATTENTION**. The drawing origin is ignored in clipping - the coordinates of the current clipping box are always absolute.*

5.4.5. Clipping

The drawing functions clip their drawings inside the current clipping box. The default clipping box is the drawing image. The coordinates of the current clipping box are absolute – the drawing origin is ignored. The clipping can be enabled or disabled by **dr_set_clipmode**. Clipping is always done inside the drawing image.

dr_init = Initialize the drawing library

Prototype:

```
#include "dr_lib.h"
int dr_init ()
```

Description:

The function initializes the drawing library. It installs the default 8x10 font, which is used by the text display functions.

***ATTENTION**. Don't forget that **dr_init** requires closing of drawing library by **dr_close** before program exit.*

Arguments:
None

Return code:

Return code	Description
`DR_OK`	Success
Other	Returned by the called functions

dr_close = Close the drawing library

Prototype:
```
#include "dr_lib.h"
void dr_close ()
```

Description:
The function closes the drawing library and frees all allocated memory buffers (the font buffer).

Arguments:
None

Return code:
None

dr_set_palette = Set color palette

Prototype:
```
#include "dr_lib.h"
void dr_set_palette ( int clr, int r, int g, int b )
```

Description:
The function sets RGB palette for the input color **`clr`**. The color must be in the range of available colors **`[0,DR_COLOR_CNT-1]`**, otherwise the function does nothing.

Notes:
This function is hardware-dependent – it is not supported on all library platforms.

Arguments:

Argument	Description
`clr`	[in] color
`r`	[in] red intensity [0,255]

g	[in] green intensity [0,255]
b	[in] blue intensity [0,255]

Return code:
None

dr_line = Draw line

Prototype:
```
#include "dr_lib.h"
void dr_line ( DR_IMAGE *dr_img, int x1, int y1, int x2, int y2, int clr )
```

Description:
The function draws a line into the drawing image **`dr_img`** with specified color **`clr`**. The line is clipped inside current clipping box (default clipping box = the drawing image). The line is drawn relative to the current drawing origin (default = 0.0). To achieve best speed, the code is divided into partial cases for horizontal, vertical, 45-degree and arbitrary lines.

The function draws different types of lines specified by the current ***line type***. The line type is set by the function **dr_set_linetype**.

Arguments:

Argument	Description
`dr_img`	[in] drawing image
`x1`	[in] x-coordinate of first line point
`y1`	[in] y-coordinate of first line point
`x2`	[in] x-coordinate of second line point
`y2`	[in] y-coordinate of second line point
`clr`	[in] drawing color (value stored in the image pixels)

Return code:
None

dr_dot_line = Draw dotted line

Prototype:
```
#include "dr_lib.h"
void dr_dot_line ( DR_IMAGE *dr_img, int x1, int y1, int x2, int y2,
                   int clr )
```

Description:

The function draws a dotted line in the input drawing image **`dr_img`** with specified color **`clr`**. The line is clipped inside current clipping box (default clipping box = the drawing image). The line is drawn relative to the current drawing origin (default = 0.0). To achieve best speed, the code is divided into partial cases for horizontal, vertical, 45-degree and arbitrary lines.

Arguments:

Argument	Description
`dr_img`	[in] drawing image
`x1`	[in] x-coordinate of first line point
`y1`	[in] y-coordinate of first line point
`x2`	[in] x-coordinate of second line point
`y2`	[in] y-coordinate of second line point
`clr`	[in] drawing color (value stored in the image pixels)

Return code:

None

dr_mat = Draw pixel matrix

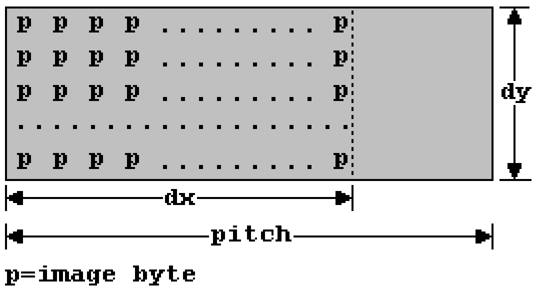
Prototype:

```
#include "dr_lib.h"
void dr_mat ( DR_IMAGE *dr_img, DR_IMAGE *mat, int x, int y, int clr,
              int bgnd, int type )
```

Description:

This is the basic pixel-drawing function. It draws a single-color or a multi-color pixel matrix into the drawing image **`dr_img`**. The function works with input matrix buffer **`mat`** in DR_IMAGE format. Each image byte corresponds to one pixel. The function clips the pixel matrix inside the drawing image and in the current clipping box if clipping is enabled.

Format of the input pixel matrix **`mat`**:



The function interprets the input pixel matrix and works in different modes depending on the **`type`** argument:

Single-color mode (`type == 0`):

Non-zero bytes in the input matrix **`mat`** are drawn with color **`clr`**. Zero bytes are drawn with color **`bgnd`**.

Multi-color mode (`type != 0`):

The input matrix **`mat`** contains color pixel values, which are stored directly into **`dr_img`**. In multi-color mode the **`bgnd`** argument is used to define transparent color:

`bgnd >= 0 :` don't draw color matrix pixels, equal to **bgnd**

`bgnd = -1 :` draw all color matrix pixels

***TIPS & TRICKS**. The function **dr_bitmat** draws a mono-color pixel matrix, where 1 bit defines one pixel.*

Arguments:

Argument	Description
`dr_img`	[in] drawing image
`mat`	[in] pixel matrix image in ***byte*** format (1 byte = 1 pixel)
`x`	[in] x-coordinate of top left matrix corner in **`dr_img`**
`y`	[in] y-coordinate of top left matrix corner in **`dr_img`**
`clr`	[in] foreground color used to draw non-zero **`mat`** pixels (used if **`type = 0`**)
`bgnd`	[in] background color, used to draw zero "**`mat`**" pixels in single-color mode (**`type=0`** or **`2`**): `-1 :` leave respective **`dr_img`** pixels unchanged `>=0 :` set respective **`dr_img`** pixels to **bgnd** color In color mode **(`type=1`** or **`3`**) **bgnd** defines transparent color: `-1 :` draw all color matrix pixels `>=0 :` don't draw color matrix pixels, equal to **bgnd**
`type`	[in] matrix type: `bit 0 = 0:` monochrome (single-color) matrix `      = 1:` multi-color matrix `bit 1 =` spare

Return code:

None

dr_bitmat = Draw bit pixel matrix

Prototype:

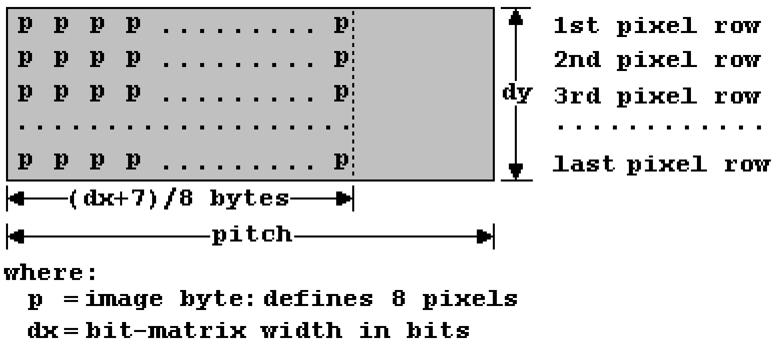
```
#include “dr_lib.h”
int dr_bitmat ( DR_IMAGE *dr_img, DR_IMAGE *mat, int x, int y, int dx,
                int clr, int bgnd )
```

Description:

The function draws a monochrome (single-color) bit matrix **mat** in the drawing image **dr_img**. The input matrix buffer **mat** is a DR_IMAGE image. Each image byte defines 8 pixels. Non-zero **mat** bits are drawn with color **clr**. Zero bits are drawn with color **bgnd**.

The function clips the pixel matrix inside the drawing image **dr_img**. It also clips the pixel matrix in the current clipping box if clipping is enabled.

Bit-matrix format :



Bits 7-0 in each byte are converted from left to right to bytes. Trailing unused bits in the last byte of each pixel row are ignored.

Arguments:

Argument	Description
dr_img	[in] drawing image
mat	[in] pixel matrix image in ***bit*** format (1 byte = 8 pixels)
x	[in] x-coordinate of top left matrix corner in **dr_img**
y	[in] y-coordinate of top left matrix corner in **dr_img**
dx	[in] bit-matrix width in bits (pixels)
clr	[in] foreground color used to draw non-zero **mat** bits
bgnd	[in] background color, used to draw zero "**mat**" bits: -1 : leave respective **dr_img** pixels unchanged >=0 : set respective **dr_img** pixels to **bgnd** color

Return code:

Return code	Description
DR_OK	Success
DR_ALLOC_ERROR	Memory allocation error

dr_image = Draw monochrome image

Prototype:

```
#include "dr_lib.h"
void dr_image ( DR_IMAGE *dr_img, DR_IMAGE *img, int x, int y, int clr,
                int bgnd )
```

Description:

The function draws the image **`img`** at position **`x,y`** (top/left corner) in the drawing image **`dr_img`**. Non-zero **`img`** pixels are drawn with color **`clr`**. Zero **`img`** pixels are not drawn.

Arguments:

Argument	Description
`dr_img`	[in] drawing image
`img`	[in] image to draw (***byte*** format: 1 byte = 1 pixel)
`x`	[in] x-coordinate of top left **`img`** corner in **`dr_img`**
`y`	[in] y-coordinate of top left **`img`** corner in **`dr_img`**
`clr`	[in] foreground color of non-zero **`img`** pixels
`bgnd`	[in] background color, used to draw zero **img** pixels: `-1` : leave respective **dr_img** pixels unchanged `>=0` : set respective **dr_img** pixels to **bgnd** color

Return code:

None

dr_clr_image = Draw color image

Prototype:

```
#include “dr_lib.h”
void dr_clr_image ( DR_IMAGE *dr_img, DR_IMAGE *img, int x, int y )
```

Description:

The function draws a color image **`img`** at position **`x,y`** (top/left corner) in the drawing image **`dr_img`**. Non-zero **`img`** pixels are stored directly into respective **`dr_img`** pixels. Zero **`img`** pixels are not drawn.

Arguments:

Argument	Description
`dr_img`	[in] drawing image
`img`	[in] color image to draw (***byte*** format: 1 byte = 1 pixel)
`x`	[in] x-coordinate of top left **`img`** corner in **`dr_img`**
`y`	[in] y-coordinate of top left **`img`** corner in **`dr_img`**

Return code:

None

dr_dot = Draw dot

Prototype:

```
#include “dr_lib.h”
void dr_dot ( DR_IMAGE *dr_img, int x, int y, int clr )
```

Description:

The function draws a dot (pixel) with integer coordinates in the drawing image **`dr_img`**. The function clips the dot inside the image. The function also clips the dot inside the current clipping box if clipping is enabled. The dot is drawn relative to the current drawing origin.

Arguments:

Argument	Description
`dr_img`	[in] drawing image
`x`	[in] dot x-coordinate
`y`	[in] dot y-coordinate
`clr`	[in] drawing color = pixel value stored in **`dr_img`**

Return code:

None

dr_fill_box_xy = Fill box with color (X/Y)

Prototype:

```
#include “dr_lib.h”
void dr_fill_box_xy ( DR_IMAGE *dr_img, int x1, int y1, int x2, int y2,
                      int clr )
```

Description:

The function fills a box in the input drawing image **`dr_img`** with color **`clr`**. The box is specified by **x/y** coordinates of the top/left and bottom/right corners. The box is clipped inside the current clipping box (default = the drawing image). The box coordinates are relative to the current drawing origin (default = 0,0).

Arguments:

Argument	Description
`dr_img`	[in] drawing image
`x1`	[in] x-coordinate of top/left box corner
`y1`	[in] y-coordinate of top/left box corner
`x2`	[in] x-coordinate of bottom/right box corner
`y2`	[in] y-coordinate of bottom/right box corner
`clr`	[in] drawing color

Return code:
None

dr_fill_box = Fill box with color

Prototype:
```
#include "dr_lib.h"
void dr_fill_box ( DR_IMAGE *dr_img, int x, int y, int dx, int dy,
                   int clr )
```

Description:
The function fills a box in the input drawing image **dr_img** with color **clr**. The box is defined by top/left corner, width and height. The box is clipped inside the current clipping box (default = the drawing image). The box coordinates are relative to the current drawing origin (default = 0,0).

Arguments:

Argument	Description
dr_img	[in] drawing image
x	[in] x-coordinate of top/left box corner
y	[in] y-coordinate of top/left box corner
dx	[in] box width in pixels
dy	[in] box height in pixels
clr	[in] drawing color

Return code:
None

dr_fill_img = Fill image with color

Prototype:
```
#include "dr_lib.h"
void dr_fill_img ( DR_IMAGE *dr_img, int clr )
```

Description:
The function sets all **img** pixels to **clr**.

Arguments:

Argument	Description
dr_img	[out] drawing image

`clr`	[in] drawing color

Return code:
None

dr_clear_img = Clear drawing image

Prototype:
```
#include "dr_lib.h"
void dr_clear_img ( DR_IMAGE *dr_img )
```

Description:
The function sets all **`img`** pixels to zero.

Arguments:

Argument	Description
`dr_img`	[out] drawing image

Return code:
None

dr_copy_img = Copy drawing image

Prototype:
```
#include "dr_lib.h"
void dr_copy_img ( DR_IMAGE *in_img, DR_IMAGE *out_img )
```

Description:
The function copies the pixels of the source image **`in_img`** into the destination image **`out_img`**. In case of image dimensions mismatch, the function copies **`in_img`** into the upper left area of **`out_img`**, eventually truncated. The **`out_img`** dimensions **dx** and **dy** are not changed.

Arguments:

Argument	Description
`in_img`	[in] source image
`out_img`	[in] destination image

Return code:
None

dr_ellip = Draw ellipse

Prototype:

```
#include "dr_lib.h"
int dr_ellip ( DR_IMAGE *dr_img, int x0, int y0, int r1, int r2, int clr )
```

Description:

The function draws a non-rotated ellipse in **dr_img**, specified by a center point **(x,y)**, horizontal radius **r1** and vertical radius **r2**. The ellipse is clipped inside the current clipping box (default = the drawing image). The ellipse coordinates are relative to the current drawing origin (default = 0,0).

Arguments:

Argument	Description
dr_img	[in] drawing image
x0	[in] x-coordinate of ellipse center
y0	[in] y-coordinate of ellipse center
r1	[in] horizontal ellipse radius in pixels
r2	[in] vertical ellipse radius in pixels
clr	[in] drawing color

Return code:

Return code	Description
DR_OK	Success
DR_ALLOC_ERROR	Memory allocation error

dr_rot_ellip = Draw rotated ellipse

Prototype:

```
#include "dr_lib.h"
int dr_rot_ellip ( DR_IMAGE *dr_img, float xc, float yc, float r_max,
                   float r_min, float ang, int clr )
```

Description:

The function draws a rotated ellipse in **dr_img**, specified by a center point **(xc,yc)**, greater radius **r_max**, smaller radius **r_min2** and rotation angle **ang**. The rotated ellipse is clipped inside current clipping box (default = the drawing image). The ellipse coordinates are relative to the current drawing origin (default = 0,0).

The function applies the non-rotated Bresenham's ellipse algorithm for angles = k*PI/2.

Arguments:

Argument	Description

`dr_img`	[in] drawing image
`xc`	[in] x-coordinate of ellipse center
`yc`	[in] y-coordinate of ellipse center
`r_max`	[in] greater (major) ellipse radius in pixels
`r_min`	[in] smaller (minor) ellipse radius in pixels
`ang`	[in] rotation angle in radians [0,2PI] – increases counter clockwise
`clr`	[in] drawing color

Return code:

Return code	Description
`DR_OK`	Success
`DR_ALLOC_ERROR`	Memory allocation error

dr_set_linetype = Set line type

Prototype:

```
#include “dr_lib.h”
void dr_set_linetype ( int  type )
```

Description:

The function sets current line type for line drawing. This type is used by the function **dr_line** to draw different types of lines.

Arguments:

Argument	Description
`type`	[in] line type: `DR_LINE_SOLID  :` solid line `DR_LINE_DOTTED :` dotted line

Return code:

None

dr_get_linetype = Get line type

Prototype:

```
#include “dr_lib.h”
int dr_get_linetype ()
```

Description:

The function returns the current line type used for line drawing by **dr_line** (see **dr_set_linetype**) **.**

Arguments:
None

Return code:
Current line type.

dr_norm_coord = Normalize coordinates

Prototype:
```
#include “dr_lib.h”
void dr_norm_coord ( int *x1, int *x2 )
```

Description:
The function normalizes the input/output coordinates - it assures that **`x1 <= x2`** by swapping the two coordinates if the first coordinate is greater than the second one.

Arguments:

Argument	Description
x1	[i/o] 1st coordinate
x2	[i/o] 2nd coordinate

Return code:
None

dr_set_draworg = Set drawing origin

Prototype:
```
#include “dr_lib.h”
void dr_set_draworg ( int  x, int y )
```

Description:
The function sets the current drawing origin. All drawings, done by the library functions, are relative to the drawing origin, i.e. all pixel coordinates `(x,y)` are added to the coordinates of the current origin point. The default drawing origin is (0,0) = the top/left corner of the drawing image.

ATTENTION*. The drawing origin is ignored in clipping - the coordinates of the current clipping box are always absolute.*

Arguments:

Argument	Description

x	[in] x-coordinate of drawing origin point
y	[in] y-coordinate of drawing origin point

Return code:
None

dr_get_draworg = Get drawing origin

Prototype:
```
#include “dr_lib.h”
void dr_get_draworg ( int  *x, int *y )
```

Description:
The function gets current drawing origin. All drawings, done by the library functions, are relative to the drawing origin, i.e. all pixel coordinates `(x,y)` are added to the coordinates of the current origin point. The default drawing origin is (0,0) = the top/left corner of the drawing image.

ATTENTION*. The drawing origin is ignored in clipping - the coordinates of the current clipping box are always absolute.*

Arguments:

Argument	Description
x	[out] x-coordinate of drawing origin point
y	[out] y-coordinate of drawing origin point

Return code:
None

dr_set_clipbox = Set clipping box

Prototype:
```
#include “dr_lib.h”
void dr_set_clipbox ( DR_IMAGE *dr_img, int x1, int y1, int x2, int y2 )
```

Description:
The function sets the current clipping box, used for clipping by the drawing functions. Pixels, which are outside the clipping box, are ignored and not drawn.

The default clipping box is the drawing image. The coordinates of the current clipping box are absolute – the drawing origin is ignored. The clipping mode (enabled/disabled) is set by the function **dr_set_clipmode**. No matter what is the clipping mode, clipping is always done inside the drawing image.

***TIPS & TRICKS**. Some functions in future library versions may support inverse clipping – pixels outside the clipping box are drawn and inner pixels are not drawn – please refer to the function description.*

Arguments:

Argument	Description
dr_img	[in] drawing image
x1	[in] x-coordinate of top/left box corner
y1	[in] y-coordinate of top/left box corner
x2	[in] x-coordinate of bottom/right box corner
y2	[in] y-coordinate of bottom/right box corner

Return code:

None

dr_get_clipbox = Get clipping box

Prototype:

```
#include "dr_lib.h"
void dr_get_clipbox ( DR_IMAGE *dr_img, int *x1, int *y1, int *x2,
                      int *y2 )
```

Description:

The function returns in **x1**, **y1**, **x2** and **y2** the corners of the current clipping box, used for clipping by the drawing functions. Pixels, which are outside the clipping box, are ignored and not drawn.

The default clipping box is the drawing image. The coordinates of the current clipping box are absolute – the drawing origin is ignored. The clipping mode (enabled/disabled) is set by the function **dr_set_clipmode**. No matter what is the clipping mode, clipping is always done inside the drawing image.

***TIPS & TRICKS**. Some functions in future library versions may support inverse clipping – pixels outside the clipping box are drawn and inner pixels are not drawn – please refer to the function description.*

Arguments:

Argument	Description
dr_img	[in] drawing image
x1	[out] x-coordinate of top/left box corner
y1	[out] y-coordinate of top/left box corner
x2	[out] x-coordinate of bottom/right box corner
y2	[out] y-coordinate of bottom/right box corner

Return code:

None

dr_set_clipmode = Set clipping mode

Prototype:

```
#include “dr_lib.h”
void dr_set_clipmode ( int clip_mode )
```

Description:

The function sets current clipping mode for the drawing functions. The default clipping mode is `DR_CLIP_OFF`, which means drawings are clipped inside the drawing image. This function does not affect the current clipping box.

***TIPS & TRICKS**. Some functions in future library versions may support the inverse clipping option DR_CLIP_INV – please refer to the function description.*

Arguments:

Argument	Description
`clip`	[in] clipping mode: `DR_CLIP_OFF` = clipping off (default) `DR_CLIP_ON` = normal clipping (draw inside the clipping box) `DR_CLIP_INV` = inverse clipping (draw outside the clipping box) **(currently NA)**

Return code:

None

dr_get_clipmode = Get clipping mode

Prototype:

```
#include “dr_lib.h”
int dr_get_clipmode ()
```

Description:

The function returns the current clipping mode, set by **dr_set_clipmode**.

Arguments:

None

Return code:

Current clipping mode - `DR_CLIP_OFF`, `DR_CLIP_ON` or `DR_CLIP_INV`.

dr_clip_point = Clip point inside image

Prototype:

```
#include "dr_lib.h"
void dr_clip_point ( DR_IMAGE *dr_img, int *x, int *y )
```

Description:

The function clips the input/output point **(x,y)** so that the point remains inside the image.

Arguments:

Argument	Description
dr_img	[in] drawing image
x	[i/o] x-coordinate of point
y	[i/o] y-coordinate of point

Return code:

None

dr_box_xy = Draw box (X/Y)

Prototype:

```
#include "dr_lib.h"
void dr_box_xy ( DR_IMAGE *dr_img, int x1, int y1, int x2, int y2,
                 int clr )
```

Description:

The function draws a box (non-rotated rectangle) in **dr_img**, specified by top/left and bottom/right corners. The box is clipped inside the current clipping box (default = the drawing image). The box is drawn relative to the current drawing origin (default = 0,0).

Arguments:

Argument	Description
dr_img	[in] drawing image
x1	[in] x-coordinate of top/left box corner
y1	[in] y-coordinate of top/left box corner
x2	[in] x-coordinate of bottom/right box corner
y2	[in] y-coordinate of bottom/right box corner
clr	[in] drawing color

Return code:

None

dr_box = Draw box

Prototype:

```
#include “dr_lib.h”
void dr_box ( DR_IMAGE *dr_img, int x, int y, int dx, int dy, int clr )
```

Description:

The function draws a box (non-rotated rectangle) in **`dr_img`**, specified by a top/left corner point, width and height. The box is clipped inside the current clipping box (default = the drawing image). The box is drawn relative to the current drawing origin (default = 0,0).

Arguments:

Argument	Description
`dr_img`	[in] drawing image
`x`	[in] x-coordinate of top/left box corner
`y`	[in] y-coordinate of top/left box corner
`dx`	[in] box width in pixels
`dy`	[in] box height in pixels
`clr`	[in] drawing color

Return code:

None

dr_marker = Draw marker

Prototype:

```
#include “dr_lib.h”
void dr_marker ( DR_IMAGE *dr_img, int x, int y, int type, int size,
                 int clr )
```

Description:

The function draws a marker in the drawing image **`dr_img`** with specified color **`clr`** at position **`(x,y)`**=marker center. The marker is clipped inside the current clipping box (default = the drawing image). The marker is drawn relative to the current drawing origin (default = 0,0).

The **`type`** argument specifies different marker shapes. The **`size`** argument defines `'+'/'x'` marker size from the center to the end points (0 = draw single dot). In the case of marker type 2, the function draws a thick square dot with dimensions **`size`** x **`size`**.

Arguments:

Argument	Description
`dr_img`	[in] drawing image

x	[in] x-coordinate of marker center point
y	[in] y-coordinate of marker center point

type	[in] marker type:
	`0` = cross marker `'+'`
	`1` = `'x'` marker
	`2` = draw thick square dot with dimensions **`size`** x **`size`**

size	[in] marker size in pixels (from center to end points): `0` = draw single dot
clr	[in] drawing color

Return code:

None

dr_markers = Draw multiple markers

Prototype:

```
#include “dr_lib.h”
void dr_markers ( DR_IMAGE *dr_img, void *xy_buf, int xy_type, int xy_cnt,
                  int type, int size, int clr )
```

Description:

The function draws multiple markers in the drawing image **`dr_img`** with specified color **`clr`**. The input buffer **`xy_buf`** contains `(x,y)` coordinates of marker center points. When **`xy_buf`** contains fractional point coordinates (**`xy_type`** `= 1` or `2`), the function rounds the coordinates to nearest integers.

The **`type`** argument specifies different marker shapes. The **`size`** argument defines `'+'/'x'` marker size from the center to the end points (0 = draw single dots). In the case of marker type 2, the function draws thick square dots with dimensions **`size`** x **`size`**.

The function clips the markers inside current clipping box (default = the drawing image). The function draws the markers relative to the current drawing origin (default = 0,0).

Arguments:

Argument	Description
dr_img	[in] drawing image
xy_buf	[in] source buffer with x/y marker coordinates: `(x,y) (x,y) (x,y) ...`
xy_type	[in] type of values in **`xy_buf`**: `0` = integer `1` = scaled*1000 integer (1/1000 pixel accuracy) `2` = floating-point
xy_cnt	[in] number of **`xy_buf`** points `(x,y)`
type	[in] marker type: `0` = cross markers `'+'` `1` = `'x'` markers

	2 = thick square dots with dimensions **`size`** x **`size`**
`size`	[in] marker size in pixels (from center to end points): `0` = draw single dots
`clr`	[in] drawing color

Return code:

None

dr_poly = Draw polygon

Prototype:

```
#include "dr_lib.h"
void dr_poly ( DR_IMAGE *dr_img, int *poly, int n, int clr )
```

Description:

The function draws a polygon in the drawing image **`dr_img`** with specified color **`clr`**. The polygon is a figure, composed of N corners and N lines connecting the corners. The polygon is clipped inside the current clipping box (default = the drawing image). The polygon is drawn relative to the current drawing origin (default = 0,0).

Arguments:

Argument	Description
`dr_img`	[in] drawing image
`poly`	[in] buffer with coordinates of **`n`** sequential polygon corner points (size = **`2*n`** integers): `(x,y) (x,y) ...`
`n`	[in] number of polygon corners
`clr`	[in] drawing color

Return code:

None

dr_rot_rect = Draw rotated rectangle

Prototype:

```
#include "dr_lib.h"
void dr_rot_rect ( DR_IMAGE *dr_img, int rect[8], int clr )
```

Description:

The function draws a general rotated rectangle in the drawing image **`dr_img`** with specified color **`clr`**. The rotated rectangle is specified by 4 corner points with integer `(x,y)` coordinates. The rectangle is clipped inside the current clipping box (default = the drawing image). The rectangle is drawn relative to the current drawing origin (default = 0,0).

Arguments:

Argument	Description
`dr_img`	[in] drawing image
`rect`	[in] 8-element buffer with coordinates of 4 sequential rectangle corner points: `(x1,y1) (x2,y2) (x3,y3) (x4,y4)`
`clr`	[in] drawing color

Return code:
None

dr_arrow = Draw arrow

Prototype:
```
#include “dr_lib.h”
void dr_arrow ( DR_IMAGE *dr_img, int x1, int y1, int x2, int y2, int clr )
```

Description:
The function draws an arrow into the drawing image **`dr_img`** with specified color **`clr`**. The arrow starts from **`(x1,y1)`** and ends at **`(x2,y2)`**. The arrow is clipped inside the current clipping box (default = the drawing image). The arrow is drawn relative to the current drawing origin (default = 0,0).

Arguments:

Argument	Description
`dr_img`	[in] drawing image
`x1`	[in] x-coordinate of beginning arrow point
`y1`	[in] y-coordinate of beginning arrow point
`x2`	[in] x-coordinate of end arrow point (the arrow tip)
`y2`	[in] y-coordinate of end arrow point (the arrow tip)
`clr`	[in] drawing color

Return code:
None

dr_set_font = Set (install) font

Prototype:
```
#include “dr_lib.h”
int dr_set_font ( char *font_file )
```

Description:

The function installs a text font from the font file **`font_file`**. The text drawing functions use bitmap fonts with fixed width and height. In case of missing or invalid font file, the function installs the default built-in font with character size 8x10, defined by an internal table. The format of the font file is described in "6.4.3. Fonts"

Arguments:

Argument	Description
`font_file`	[in] font file name (NULL: install default font)

Return code:

Return code	Description
`DR_OK`	Success
`DR_ALLOC_ERROR`	Font table allocation error
`DR_FONT_FILE_ERR`	Font file error

dr_close_font = Close (uninstall) current font

Prototype:

```
#include "dr_lib.h"
void dr_close_font ()
```

Description:

The function uninstalls the current font and frees the font buffer memory.

Arguments:

None

Return code:

None

dr_get_font = Get current font

Prototype:

```
#include "dr_lib.h"
int dr_get_font ( DR_FONT *cur_font )
```

Description:

The function returns in **`cur_font`** the parameters of the current installed font. The **`cur_font`** output argument is a pointer to a **`DR_FONT`** structure, defined in **dr_lib.h**:

```
typedef struct
{
    VL_UINT32 font_dx;  /* font width in pixels                         */
    VL_UINT32 font_dy;  /* font height in pixels                        */
    VL_UINT32 font_cnt; /* number of char. icons in font_tab            */
    char     *font_tab; /* font table with char. definition data in     */
                        /* bitmap format (1 byte = 1 icon pixel)        */
} DR_FONT;
```

Arguments:

Argument	Description
`cur_font`	[out] font definition structure (see the description above)

Return code:

Return code	Description
`DR_OK`	Success
`DR_INVALID_FONT`	Font not installed

dr_get_font_size = Get current font size

Prototype:

```
#include “dr_lib.h”
int dr_get_font_size ( int *font_dx, int *font_dy )
```

Description:

The function returns the width and the height of the current font in **`font_dx`** and **`font_dy`**.

***TIPS & TRICKS**. Getting of font size helps in calculations of text dimensions in pixels.*

Arguments:

Argument	Description
`font_dx`	[out] font width in pixels (NULL: don't return)
`font_dy`	[out] font height in pixels (NULL: don't return)

Return code:

Return code	Description
`DR_OK`	Success
`DR_INVALID_FONT`	Font not installed

dr_text = Draw text

Prototype:

```
#include “dr_lib.h”
int dr_text ( DR_IMAGE *dr_img, char *str, int x, int y, int clr,
              int bgnd )
```

Description:

The function draws a text in the drawing image **`dr_img`** with current font. The text is clipped inside the drawing image and by the current clipping box if clipping is enabled. The text is drawn relative to the current drawing origin (default = 0,0).

***ATTENTION**. This function requires installed font by a previous call to **dr_init** for example.*

Arguments:

Argument	Description
`dr_img`	[in] drawing image
`str`	[in] 0-terminated string with text to draw
`x`	[in] x-coordinate of top left text corner
`y`	[in] y-coordinate of top left text corner
`clr`	[in] foreground text color
`bgnd`	[in] background text color: `-1` : leave respective **`dr_img`** pixels unchanged `>=0` : set respective **`dr_img`** pixels to **bgnd** color

Return code:

Return code	Description
`DR_OK`	Success
`DR_ALLOC_ERROR`	Memory allocation error
Other	Returned by called functions

dr_txt = Draw text (no font setup needed)

Prototype:

```
#include “dr_lib.h”
int dr_txt ( image *dr_img, char *str, int x, int y, int size, int clr,
             int bgnd )
```

Description:

The function draws text in **`dr_img`** with color **`clr`** starting from **`(x,y)`** = top/left text corner. The function supports different text dimensions – by doubling, tripling, etc. the character dimensions. The function uses the built-in DR_LIB 8x10 font table in ***bit*** format (1bit = 1 pixel). Since the font table format needs unpacking from bits to bytes, this function is slower than **dr_text**.

The text is clipped inside the drawing image and by the current clipping box if clipping is enabled. The text is drawn relative to the current drawing origin (default = 0,0).

ATTENTION*. This function does not require DR_LIB setup by call to* ***dr_init()****.*

Arguments:

Argument	Description
`dr_img`	[in] drawing image
`str`	[in] 0-terminated string with text to draw
`x`	[in] x-coordinate of top left text corner
`y`	[in] y-coordinate of top left text corner
`size`	[in] text size: 1=8x10, 2=16x20, 3=24x30, ...
`clr`	[in] foreground text color
`bgnd`	[in] background text color (0 = transparent, leave respective **`dr_img`** pixels unchanged)

Return code:

Return code	Description
`DR_OK`	Success
`DR_INV_IMAGE`	Invalid drawing image
`DR_ALLOC_ERROR`	Memory allocation error

dr_xybuf = Draw XY buffer

Prototype:

```
#include "dr_lib.h"
void dr_xybuf ( DR_IMAGE *dr_img, int *xy_buf, int n, int clr )
```

Description:

The function draws a set of pixels into **`dr_img`** with color **`clr`**. The pixels are specified by x/y coordinates contained in the input **`xy_buf`** buffer. The pixels are clipped inside the drawing image.

Arguments:

Argument	Description
`dr_img`	[in] drawing image
`xy_buf`	[in] buffer with pixel x/y coordinates in format: `(x,y) (x,y) ...` Size of **`xy_buf`** = **`2*n`** int elements

`n`	[in] number of points (x/y pairs) in **`xy_buf`**
`clr`	[in] drawing color

Return code:

None

dr_fill_poly = Fill convex polygon with color

Prototype:

```
#include “dr_lib.h”
int dr_fill_poly ( DR_IMAGE *dr_img, int *pt_buf, int pt_cnt, int clr )
```

Description:

The function sets to **`clr`** all **`dr_img`** pixels inside the convex polygon `P1,P2,...,Pk (P1 != Pk)`, defined by the input buffer **`pt_buf`**. Pixels with coordinates outside the image are ignored.

Arguments:

Argument	Description
`dr_img`	[in] drawing image
`pt_buf`	[in] buffer with x/y coordinates of **`pt_cnt`** sequential convex polygon corner points in format: `(x,y) (x,y) ...`
`pt_cnt`	[in] number of polygon corners points (x,y pairs)
`clr`	[in] foreground text color

Return code:

Return code	Description
`DR_OK`	Success
Other	Returned by called functions

dr_bitmat_to_bytemat = Convert bit matrix to byte matrix

Prototype:

```
#include “dr_lib.h”
void dr_bitmat_to_bytemat ( char *bit_mat, int dx, int dy, char *byte_mat )
```

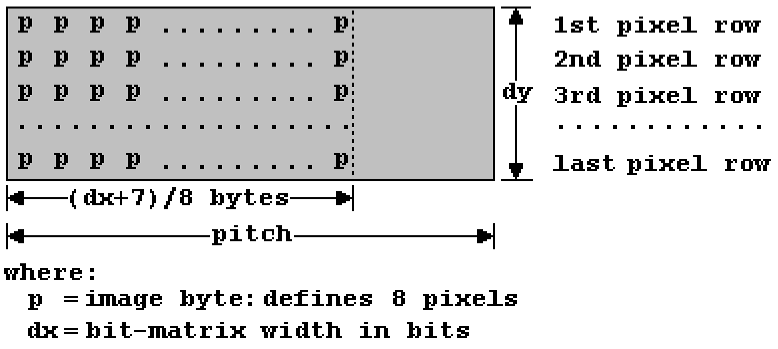
Description:

The function converts the input bit matrix buffer **`bit_mat`** into a byte matrix buffer **`byte_mat`**. Each bit in **`bit_mat`** is converted to 1 result byte in **`byte_mat`**. The first (leftmost) **`dx`** bits in each **`bit_mat`** row are converted; any trailing bits in the last byte are ignored.

Bit to byte conversion:

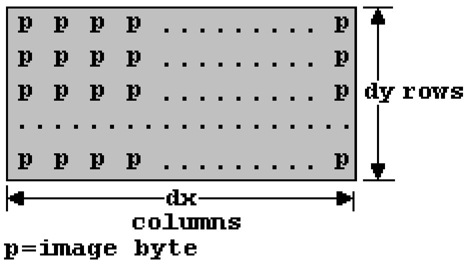
```
* <br> bit=0 => byte=0x00
* <br> bit=1 => byte=0x01
```

Format of input bit-matrix:



Each matrix row is composed of **dx** bits **((dx+7)/8 bytes)** and defines one pixel row in a bitmap picture. Bits 7-0 in each byte are converted from left to right into bytes. Excess bits above **dx** in the last byte of each matrix row are ignored.

Format of output byte-matrix:



Arguments:

Argument	Description
bit_mat	[in] bit matrix buffer with size **(dx+7)/8 * dy** bytes
dx	[in] bit-matrix width in bits (pixels) = **(dx+7)/8** bytes
dy	[in] bit-matrix height in bits = number of pixel rows
byte_mat	[out] byte matrix buffer with size = **dx * dy** bytes (not checked for overflow)

Return code:

None

5.5. Edge detection library

This chapter describes the edge-detection library - a group of image-processing functions, intended for reading of image pixels and detection of edges.

5.5.1. Introduction

The edge-detection library is a set of C functions for pixel sampling and searching of edges in sets of pixel values or other data. The library works on PC or intelligent programmable cameras with C compiler.

All library functions are written in plain C code and can easily be ported on new hardware platforms and operating systems.

5.5.1.1. Basic philosophy of the edge detection library

The library functions are divided into relatively independent groups according to their purpose:

- **Pixel sampling functions.** These functions read single pixels or whole pixel lines in a gray-level image. Pass the output buffers of these functions to the edge-detection functions. See "6.5.2.2. Pixel sampling functions".
- **Edge detection functions.** These functions find edges in 32-bit integer data buffers. They may receive pixel values [0,255], produced by the pixel-sampling functions, or other data where transitions of values from light to dark (high to low) or dark to light (low to high) must be found. See "6.5.2.3. Edge detection functions".
- **Calculation of edge coordinates.** The edge detection functions return edge postions in the input data buffers. Use the functions from this group to find the exact x/y coordinates of the detected edge points. See "6.5.2.4. Calculation of edge coordinates".
- **Drawing of edge points.** Useful to show the detected edges. See "6.5.2.5. Drawing of edge points".
- **High-level edge-detection tool.** The **ip_edge** function incorporates into one universal tool all pixel sampling and edge-detection methods, including the drawing of the detected edges. See "6.5.2.6. High-level edge-detection tool".

The library is created with the aim to be universal and easily upgraded. You can use for example a new data sampling function with an existing edge-detection function, or an existing pixel-sampling function with a new edge-detection function.

5.5.1.2. Simple example

The **ED_SDEMO.C** file contains a simple program, which demonstrates how to use the library functions. It reads the pixels on one image line, finds edges, calculates edge coordinates and draws the detected edge points. The program can be used as template code for development of more complex custom applications.

```
/* Simple demo program for the edge-detection library */

#include "vm_lib.h"

void main()
{
    int     rc = 0;
    image   vd_img;         /* gray-level video image                   */
    image   dr_img;         /* drawing image                            */

    int     line[4];        /* line: (x1,y1) (x2,y2)                    */
    int     line_type = 0;  /* line type: 0=int                         */
    float   pix_step = 1.;  /* pixel step, used if sub_pix=1            */
    int     sub_pix  = 0;   /* 0=Bresenham, 1=sub-pixel mode            */
    int     pix_type = 0;   /* type of result pixel coordinates: 0=int  */
    int     pix_val[640];   /* result pixel value buffer                */
    int     pix_xy[2*640];  /* result pixel coordinate buffer           */
    int     pix_cnt;        /* result pixel count                       */

    int     edge_dir = 2;   /* edge direction: 0-2=L->D,D->L,both       */
    int     edge_type = 3;  /* edge types: 0-3=1st,last.strongest,all   */
    int     edge_th = 127;  /* edge detection threshold                 */
    int     pos_type = 0;   /* type of result edge positions: 0=int     */
    int     edge_pos[640];  /* result edge position buffer              */
    int     edge_val[640];  /* result edge strength buffer              */
    int     edge_cnt;       /* result edge count                        */

    int     edge_xy[2*640]; /* result edge coordinate buffer            */
    int     xy_type = 0;    /* type of edge coordinates: 0=int          */

    printf("Edge-detection library - simple demo program\n");

/*............ Fill gray-level video image -> vd_img */
/* . . . . . . . . . . */

/*............ Read pixel line */
/*
*  Pixel values            -> pix_val[]
*  Pixel (x,y) coordinates -> pix_xy[]
*  Number of stored pixels -> pix_cnt
*/
    line[0] = 100; line[1] = 100;   /* 1st line point (x1,y1)   */
    line[2] = 200; line[3] = 200;   /* 2nd line point (x2,y2)   */
    rc = ip_read_pixel_line(&vd_img, line, line_type, pix_step, sub_pix,
                            pix_type, 640, pix_xy, pix_val, &pix_cnt);
    if(rc) goto done;

/*............ Find edges in the pixel value buffer 'pix_val' */
/*
* Use threshold-based edge detection:
*   Edge positions       -> edge_pos[]
*   Edge strength values -> edge_val[]  (>0: D->L, <0:L->D)
*/
    rc = ip_edge_det_th(pix_val, pix_cnt, 0, edge_dir, edge_type, edge_th,
                        pos_type, 640, edge_pos, edge_val, &edge_cnt);
    if(rc) goto done;

/*............ Calculate edge coordinates */
/*
* Edge x,y coordinates -> edge_xy[edge_cnt]
* Type of coordinates = xy_type: 0=int, 1=scaled*1000 int, 2=float
*/
    rc = ip_calc_edge_coord(pix_xy, pix_type, pix_cnt,
                            edge_pos, pos_type, edge_cnt,
                            edge_xy, xy_type);
    if(rc) goto done;

/*............ Draw small 'x' markers on detected edge points */
    dr_markers(&dr_img, edge_xy, xy_type, edge_cnt, 1, 1, 10);

/*............ Exit */
done:
    return;
}
```

5.5.2. Usage

The chapter describes how to use the edge-detection functions. You need a prerequisite experience in C programming.

The library functions are divided into several groups according to their purpose - pixel-sampling functions, edge-detections functions, drawing functions, etc. The pixel-sampling functions work on a gray-level image of type **`VD_IMAGE`**. The drawing functions draw into a **`DR_IMAGE`** image. The **`VD_IMAGE`** and **`DR_IMAGE`** structures are defined in **cdef.h**. Currently these image structures are equivalent to the **`image`** structure.

5.5.2.1. Header files

Include the basic library header **vm_lib.h** in your code. The file includes other header files, described below:

Header file	Description
cdef.h	Common platform-dependent definitions
ip_lib.h	Pixel-sampling and edge detection functions
dr_lib.h	Drawing functions
lib_err.h	Error codes

***ATTENTION**. Do not use macros, type definitions and function prototypes from the header file(s), which are not documented in this manual. They are intended for internal use only and may be changed without a notice.*

5.5.2.2. Pixel sampling functions

The pixel sampling functions read pixel values from a gray-level video image of type **`VD_IMAGE`**:

`ip_get_subpixel`	=	Get sub-pixel value (use float pixel coordinates)
`ip_read_pixel`	=	Read pixel value (use integer pixel coordinates)
`ip_read_subpixel`	=	Read sub-pixel value (use scaled*1000 integer pixel coordinates)
`ip_read_pixel_line`	=	Read pixel line
`ip_read_pixel_stripe`	=	Read pixel stripe – a thick line (band) with mean pixel values
`ip_read_rect_lines`	=	Read rectangle pixel lines (basic function)
`ip_read_pixel_rect`	=	Read pixel rectangle

The functions read single pixels/sub-pixels, pixel lines and whole pixel rectangles, composed of multiple parallel pixel lines or stripes (bands of pixel lines with pixel averaging). The functions return optional buffers with pixel coordinates, which may be necessary to calculate exact edge positions after the edge-detection stage (see **ip_calc_edge_coord**).

The pixel sampling functions use two methods for calculation of line pixel coordinates:

- The Bresenham's algorithm – the fastest method when you don't need sub-pixel accuracy.
- Calculation of fractional sub-pixel coordinates by ***scaled*1000 integer*** arithmetic. This method achieves the highest execution speed when you need sub-pixels.

5.5.2.3. Edge detection functions

The buffers with pixel values, read by the first group of functions, must be passed as input buffers to the edge detection functions:

`ip_edge_det_th`	=	Edge detection by threshold
`ip_edge_det_grad`	=	Edge detection by gradient

These functions find edges (essesntial transitions from light to dark and/or dark to light) and return edge positions - integer or fractional indexes in the input data buffers.

The edge-detection functions use 32-bit integer buffers with data values. These buffers are not restricted to hold pixel values in the range [0,255], so you can find edges on other types of data.

5.5.2.4. Calculation of edge coordinates

As already mentioned, the edge-detection functions return edge positions as indexes in the source data buffers. If you need exact edge coordinates with rounded integer or sub-pixel accuracy, you can use the following function:

`ip_calc_edge_coord` = Calculate x/y edge coordinates

This function works on input edge positions (indexes) and coordinates of the pixels, the values of which were used to find the edge positions. The function applies linear interpolation to find the exact coordinates of each edge point, which lies between to pixels.

5.5.2.5. Drawing of edge points

Use the following functions to draw markers on the detected edge points into a drawing image of type **`DR_IMAGE`**:

`dr_markers` = Draw markers on detected edge points - universal function with arbitrary marker size and type

Drawing of edge points could be very useful for debugging and test purposes.

5.5.2.6. High-level edge-detection tool

The **ip_edge** function combines all methods for pixel sampling and all edge-detection algorithms into one high-level tool:

- Pixel sampling on a single scan line or multiple scan lines, defined by a rotated rectangle.
- Working with stripes (bands) instead of single scan lines.
- Edge detection by the threshold and the gradient methods.
- Calculation of edge coordinates.
- Drawing of markers on detected edge points.

5.5.2.7. Scaled integers

The edge-detection library uses the so called ***scaled*N integers*** to present fractional quantities by integer numbers. See section “6.3.3. Scaled integers” for more details.

5.5.3. Edge-detection examples

This chapter presents complete demo programs, which show how to use the edge-detection library. The distribution file contains the following demo source files:

File	Description
ED_SDEMO.C	Template code, shown in the introduction section (not a functional program)
ED_DEMO1.C	Edge detection on single scan line
ED_DEMO2.C	Edge detection on multiple scan lines in a rotated rectangle
ED_DEMO3.C	Edge detection on multiple scan lines in rotated rectangle, using the high-level **ip_edge** function

The demo programs ED_DEMO1, ED_DEMO2 and ED_DEMO3 are complete functional programs, which can be compiled and run on PC. The demo programs generate test gray-level image with expected edges:

- L->D edges on approximately horizontal scan lines at x-coordinates 100 and 300.
- D->L edges on approximately horizontal scan lines at x-coordinates 200 and 400.

Usage:

```
ED_DEMOi [calc_mode [method]]
```

where:

`calc_mode` =	Calculation mode: 0=integer(default), 1=sub-pixels
`method` =	Edge-detection method: 0=threshold(default), 1=gradient

The demo programs work in two calculation modes, specified by the command line parameter **`calc_mode`**:

- 0 : use Bresenham's algorithm to calculate integer line pixel coordinates, find integer edge positions and edge coordinates
- 1 : use scaled*1000 integer calculations with 1/1000 sub-pixel accuracy for line pixel coordinates, edge positions and edge coordinates

The demo programs implement two edge-detection algorithms, specified by the command line parameter **`method`**:

- 0 : use threshold-based edge-detection
- 1 : use gradient-based edge-detection

ip_get_subpixel = Get sub-pixel value (float coordinates)

Prototype:

```
#include "ip_lib.h"
int ip_get_subpixel ( image *img, float x, float y )
```

Description:

The function calculates the value of a sub-pixel with floating-point coordinates **`(x,y)`**. The sub-pixel value is calculated by bilinear interpolation. This is the slowest pixel-reading function; please use **ip_read_pixel** or **ip_read_subpixel** when possible.

Arguments:

Argument	Description
`img`	[in] source gray-level image
`x`	[in] floating-point x-coordinate of pixel
`y`	[in] floating-point y-coordinate of pixel

Return code:

Return code	Description
`0-255`	Sub-pixel value
`-1`	Pixel outside image

ip_read_pixel = Read pixel value (integer coordinates)

Prototype:

```
#include "ip_lib.h"
```

```
int ip_read_pixel ( image *img, int x, int y )
```

Description:

The function reads the value of image pixel, defined by integer **`(x,y)`** coordinates. The function checks that the pixel is inside the image, otherwise it returns -1.

Arguments:

Argument	Description
`img`	[in] source image
`x`	[in] integer x-coordinate of pixel
`y`	[in] integer y-coordinate of pixel

Return code:

Return code	Description
`0-255`	Pixel value
`-1`	Pixel outside image or other error

ip_read_subpixel = Read sub-pixel value (scaled*1000 integer coordinates)

Prototype:

```
#include "ip_lib.h"
int ip_read_subpixel ( image *img, int x, int y )
```

Description:

The function calculates the value of a sub-pixel with ***scaled*1000 integer*** **`(x,y)`** coordinates by a fast bilinear interpolation method with integer calculations.

TIPS & TRICKS. *This is the fastest sub-pixel read function.*

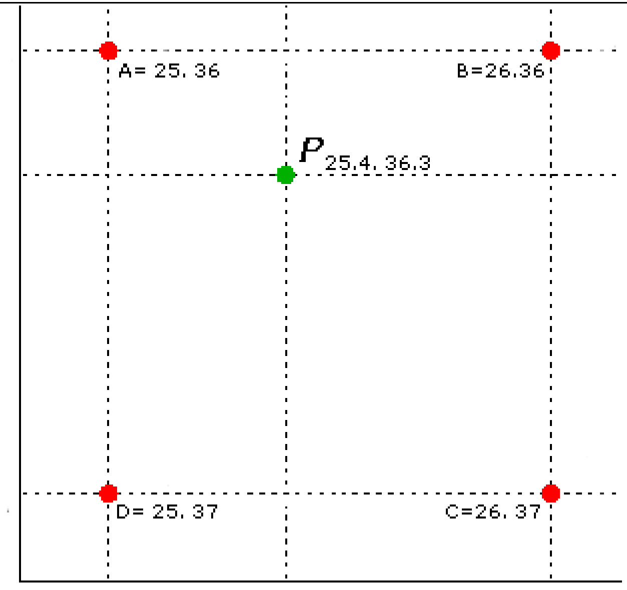
Example of scaled*1000 integer coordinate (see section "6.3.3. Scaled integers"):

```
  x = 12650
  Actual x value = 12.650
```

Bilinear Interpolation algorithm:

The bilinear interpolation is a technique used when you want to get a pixel value at coordinates, which are not whole numbers. The bilinear interpolation algorithm solves this by taking a weighted average of the 4 pixels around it.

Let us calculate the value of the pixel *P=(25.4, 36.3)*. P lies between four pixels with coordinates A=(25,36), B= (26,36), C= (26,37) and D= (25,37) and respective pixel values **a**, **b**, **c** and **d**.



The value of P will be a weighted combination of the values of **a**, **b**, **c** and **d**. We need the .4 and .3 parts of P alone; we can find these by rounding P down and then subtracting:

```
  dx = Px - round(Px) = .4
  dy = Py - round(Py) = .3
```

Now, we can multiple **a**, **b**, **c** and **d** by (1 - the area of the rectangle near them), where 1 = the total rectangle area, and sum these values together to receive the interpolated sub-pixel value:

```
  Sub-pixel value = a * (1-dx)*(1-dy) +  /* upper left  */
                    b * dx*(1-dy)     +  /* upper right */
                    c * dx*dy         +  /* lower right */
                    d * (1-dx)*dy;       /* lower left  */
```

Arguments:

Argument	Description
`img`	[in] source gray-level image
x	[in] ***scaled*1000 integer*** x-coordinate of pixel
y	[in] ***scaled*1000 integer*** y-coordinate of pixel

Return code:

Return code	Description
`0-255`	Sub-pixel value
`-1`	Pixel outside image

ip_read_pixel_line = Read pixel line

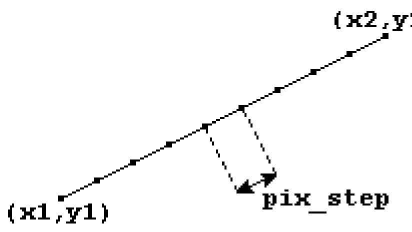
Prototype:

```
#include “ip_lib.h”
int ip_read_pixel_line ( VD_IMAGE *img, void *line, int arg_type,
                         float pix_step, int sub_pix, int res_type,
                         int res_size, void *res_xy, int *res_val,
                         int *res_cnt )
```

Description:

The function reads a pixel line from the input image **`img`**, defined by the 4-element input **`line`** buffer in format `(x1,y1) (x2,y2)`. The function stores the coordinates and the values of the line pixels into the output buffers **`res_xy`** and **`res_val`** respectively. The line pixels are read in the direction from `(x1,y1)` to `(x2,y2)`. The reading of pixel coordinates or values can be disabled by passing of NULL pointers to the respective arguments.

The **`pix_step`** argument defines Euclidean distance between successive line pixel samples. The pixels are uniformly distributed on the line starting from `(x1,y1)` to `(x2,y2)`:



The function calculates line pixel coordinates in two modes, specified by **`sub_pix`**:

- **Integer mode**. The function uses the Bresenham's algorithm to generate integer coordinates of line pixels.
- **Sub-pixel mode**. The function generates pixel coordinates with sub-pixel accuracy, defined by **`pix_step`**, and reads sub-pixel values. The function uses internally scaled*1000 integer calculations to achieve best execution speed.

Example of ***scaled*1000 integer*** coordinate (see section "6.3.3. Scaled integers"):

```
x = 12650
Actual x value = 12.650
```

Arguments:

Argument	Description
`img`	[in] source gray-level image (NULL : don't return pixel values in **`res_val`**)
`line`	[in] 4-element line buffer `(x1,y1) (x2,y2)` with coordinate type **`arg_type`**
`arg_type`	[in] type of x/y coordinates in the **`line`** buffer: `0` = integer `1` = scaled*1000 integer (1/1000 pixel accuracy) `2` = floating-point
`pix_step`	[in] float pixel step used to read line pixels in sub-pixel mode: `>= 1.0` : use fractional pixel step > 1 `<  1.0 [0.5-0.001]` : use fractional pixel step < 1 Ignored when **`sub_pix=0`** (Bresenham).
`sub_pix`	[in] sub-pixel calculation mode (type of calculated pixel coordinates): `0` = Integer mode - use integer pixel coordinates to read line pixels, generated by the Bresenham's algorithm (ignores **`pix_step`**). `1` = Sub-pixel mode - use scaled*1000 integer sub-pixel coordinates to read line pixels
`res_type`	[in] type of result pixel coordinates, stored in **`res_xy`**: `0` = integer `1` = scaled*1000 integer (1/1000 pixel accuracy) `2` = floating-point

`res_size`	[in] size of output buffers **`res_val`** and **`res_xy`** in elements:
	`res_val` `:` # of integers, size in bytes `= res_size*sizeof(int)`
	`res_xy` `:` # of points `(x,y)`, size in bytes `= res_size*2"sizeof(int)`

`res_xy`	[out] buffer with pixel coordinates in format (NULL : don't return): `(x,y) (x,y) ... (x,y) :` **`res_cnt`** pixel coordinates The coordinates are integer, scaled*1000 integer or floating-point numbers depending on **`res_type`**.
`res_val`	[out] buffer with pixel values in format (NULL : don't return): `val val ... val :` **`res_cnt`** pixel values `val = 0-255 :` valid pixel value `= -1 :` pixel outside image
`res_cnt`	[out] actual number of elements (values and points), stored into the output buffers (**`<= res_size`**)

Return code:

Return code	Description
`0`	Success
`IP_INV_IMAGE`	Invalid image
`IP_INV_ARG`	Invalid input argument
`IP_ALLOC_ERROR`	Memory allocation error
`IP_RESBUF_OVF`	Result buffer overflow
`IP_INV_PLATFORM`	Invalid PC/camera platform : `sizeof(int) != sizeof(float)`

ip_read_pixel_stripe = Read pixel stripe

Prototype:

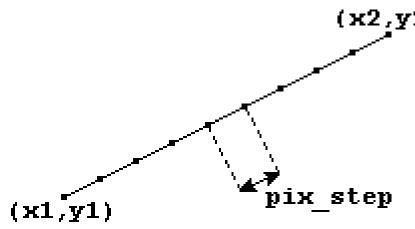
```
#include “ip_lib.h”
int ip_read_pixel_stripe ( VD_IMAGE *vd_img, DR_IMAGE *dr_img, void *line,
                           int arg_type, int stripe_wid, int stripe_lcnt,
                           float pix_step, int sub_pix, int res_type,
                           int res_size, void *res_xy, int *res_val,
                           int *res_cnt, int dr_mode, int *dr_clrs )
```

Description:

The function reads a pixel stripe (band) from the input image **`vd_img`**, defined by the scan line **`line`** in format `(x1,y1) (x2,y2)`, and stores the pixel values into **`res_val`**. Optionally the function can save the pixel coordinates of the middle scan line in **`res_xy`**.

The stripe can be considered as a thick scan line with thickness **`stripe_wid`** pixels. The stripe position is determined by the input scan line **`line`**, which lies in the middle of the stripe. The stripe is composed of **`stripe_lcnt`** pixel lines, uniformly distributed in the stripe width and parallel to the scan line. The respective pixels in the pixel lines are summed, divided by **`stripe_lcnt`** and the results are stored as pixel values in **`res_val`**. If **`stripe_wid <= 1`** or **`stripe_lcnt <= 1`**, then the function reads one single scan line.

The **pix_step** argument defines Euclidean distance between successive line pixels. The pixels are uniformly distributed on each stripe line starting from (x1,y1) to (x2,y2):



The function calculates line pixel coordinates in two modes, specified by **sub_pix**:

- **Integer mode**. The function uses the Bresenham's algorithm to generate integer coordinates of line pixels.
- **Sub-pixel mode**. The function generates pixel coordinates with sub-pixel accuracy, defined by **pix_step**, and reads sub-pixel values. The function uses internally scaled*1000 integer calculations to achieve best execution speed.

Example of ***scaled*1000 integer*** coordinate (see section "6.3.3. Scaled integers"):

```
x = 12650
Actual x value = 12.650
```

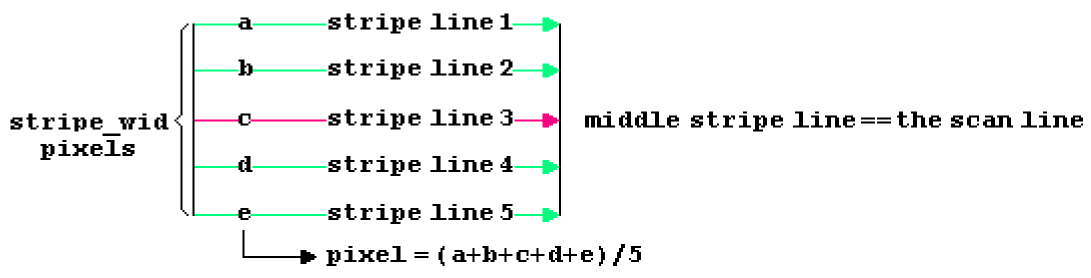
The function draws the stripe into the drawing image **dr_img**. The **dr_mode** argument specifies what elements of the stripe should be drawn: stripe enclosing rectangle, middle scan line, stripe pixels lines, etc.

ATTENTION*. The* ***stripe_lcnt*** *parameter is set internally by the function to* ***odd*** *value. Using of very small* ***pix_step*** *values may decrease the execution speed. Sub-pixel accuracy in the range [0.5, 0.1] is enough in most practical cases.*

TIPS & TRICKS*. Reading of stripes is useful for example in edge-detection. Using of mean stripe pixel values is tolerant to noise and yields stable detection of edges.*

Stripe example (stripe_lcnt=5):

5 pixel lines parallel to the scan line, are uniformly distributed in the stripe width. The red scan line, which defines the stripe position, is in the center of the stripe. Here the middle scan line is horizontal, but it can be arbitrary non-horizontal, non-vertical line.



Arguments:

Argument	Description
vd_img	[in] source gray-level image to read stripe from
dr_img	[in] drawing image (NULL: skip drawing)
line	[in] 4-element buffer (x1,y1) (x2,y2) with coordinates of the middle scan line.

	The line coordinates have type **`arg_type`**.
`arg_type`	[in] type of x/y coordinates in the **`line`** buffer: `0` = integer `1` = scaled*1000 integer (1/1000 pixel accuracy) `2` = floating-point
`stripe_wid`	[in] stripe (band) width in pixels (the scan line is in the middle of the stripe): `<=1 :` no stripe, single scan line `>1 :` width of stripe in pixels
`stripe_lcnt`	[in] stripe line count - number of pixel lines, which compose the stripe (**odd** value): `<=1 :` no stripes, single scan line `>1 :` number of pixel lines uniformly distributed in the stripe width on both sides of the scan line (see the example)
`pix_step`	[in] float pixel step used to read line pixels in sub-pixel mode: `>= 1.0 :` use fractional pixel step > 1 `< 1.0 [0.5-0.001] :` use fractional pixel step < 1 Ignored when **`sub_pix=0`** (Bresenham).
`sub_pix`	[in] sub-pixel calculation mode (type of calculated pixel coordinates): `0` = Integer mode - use integer pixel coordinates to read line pixels, generated by the Bresenham's algorithm (ignores **`pix_step`**). `1` = Sub-pixel mode - use scaled*1000 integer sub-pixel coordinates to read line pixels.
`res_type`	[in] type of result pixel coordinates, stored in **`res_xy`**: `0` = integer `1` = scaled*1000 integer (1/1000 pixel accuracy) `2` = floating-point

`res_size`	[in] size of output buffers **`res_val`** and **`res_xy`** in elements:
	`res_val` : # of integers, size in bytes = `res_size*sizeof(int)`
	`res_xy` : # of points `(x,y)`, size in bytes = `res_size*2"sizeof(int)` or `res_size*2"sizeof(int)`

`res_xy`	[out] buffer with pixel coordinates of the middle scan line in format (NULL : don't return): `(x,y) (x,y) ... (x,y) :` **`res_cnt`** pixel coordinates The coordinates are integer, scaled*1000 integer or floating-point numbers depending on **`res_type`**.
`res_val`	[out] buffer with mean pixel values from all stripe lines in format (NULL : don't return): `val val ... val :` **`res_cnt`** pixel values `val = 0-255 :` valid pixel value `= -1 :` pixel outside image
`res_cnt`	[out] actual number of elements (values and points), stored into the output buffers (**`<= res_size`**)
`dr_mode`	[in] drawing mode (OR of bits): `0` = don't draw `bit 1` = draw stripe rectangle `bit 2` = draw middle scan line `bit 3` = draw all stripe pixel lines `bit 4` = draw top and bottom dotted rectangle lines with scan line color
`dr_clrs`	[in] 3-element buffer with drawing colors (NULL = use default colors): `[0]` = stripe rectangle color (default = bright green) `[1]` = color of middle scan line and arrow (default = bright green) `[2]` = color of stripe lines/arrows (default = yellow)

Return code:

Return code	Description

`0`	Success
`IP_INV_IMAGE`	Invalid image
`IP_INV_ARG`	Invalid input argument
`IP_ALLOC_ERROR`	Memory allocation error
`IP_RESBUF_OVF`	Result buffer overflow
`IP_INV_PLATFORM`	Invalid PC/camera platform : `sizeof(int) != sizeof(float)`

ip_read_rect_lines = Read rectangle lines (basic function)

Prototype:

```
#include "ip_lib.h"
int ip_read_rect_lines ( VD_IMAGE *vd_img, DR_IMAGE *dr_img,
                         float rect[8], int line_cnt,
                         int stripe_wid, int stripe_lcnt,
                         float pix_step, int sub_pix, int scan_dir,
                         int res_type, int res_size, void *res_xy,
                         int *res_val, int *res_cnt_buf, int *res_cnt,
                         int dr_mode, int *dr_clrs )
```

Description:

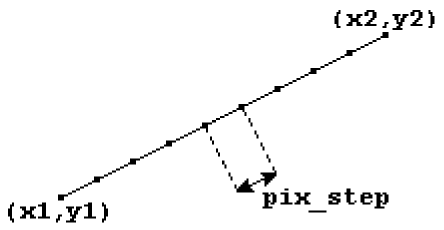
The function reads **`line_cnt`** pixel lines in a rotated rectangle area in the input gray-level image **`vd_img`**. The rectangle is defined by 4 corner points with float x/y coordinates:

```
A(x,y) = rect[0,1]
B(x,y) = rect[2,3]
C(x,y) = rect[4,5]
D(x,y) = rect[6,7]
```

The line AB is the ***top*** rectangle line and the line DC is the ***bottom*** rectangle line in non-rotated rectangle state.

The pixel lines (called ***scan*** lines) are distributed uniformly in the rectangle from the top to the bottom rectangle line, including the top/bottom lines with one exception – a single scan line (**`line_cnt=1`**) is in the middle of the rectangle, see the examples below. In non-rotated rectangle state the scan lines are horizontal and the pixels are read from left to right if **`scan_dir=0`** or from right to left if **`scan_dir=1`** - see the arrows in the examples below.

The **`pix_step`** argument defines Euclidean distance between successive line pixels. The pixels are uniformly distributed on each scan/stripe line when the scan direction is set from `(x1,y1)` to `(x2,y2)`:



The function calculates line pixel coordinates in two modes, specified by **`sub_pix`**:

- **Integer mode**. The function uses the Bresenham's algorithm to generate integer coordinates of line pixels.

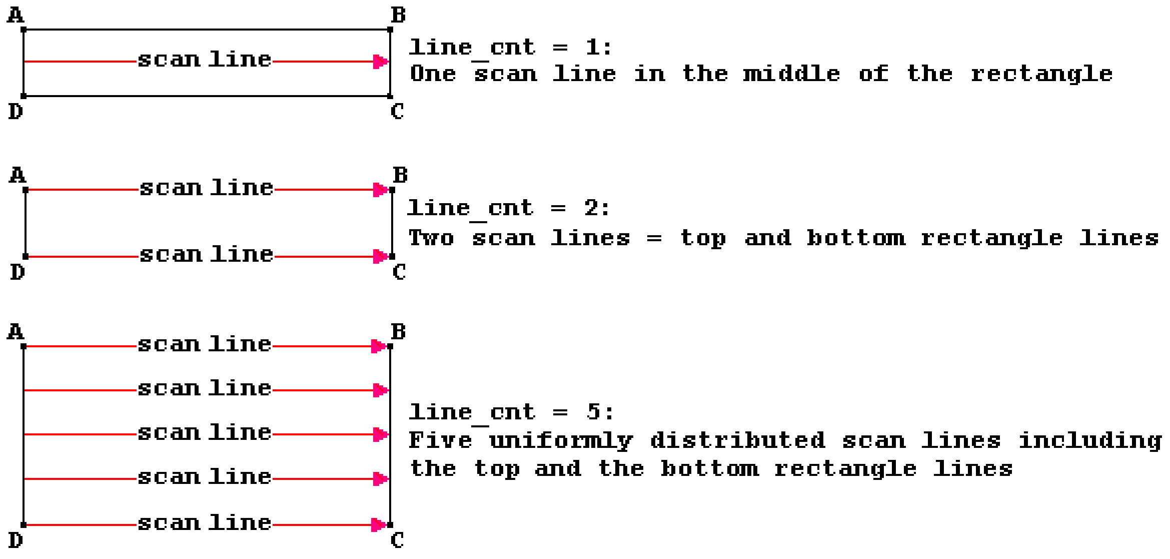
- **Sub-pixel mode**. The function generates pixel coordinates with sub-pixel accuracy, defined by **`pix_step`**, and reads sub-pixel values. The function uses internally scaled*1000 integer calculations to achieve best execution speed.

Example of ***scaled*1000 integer*** coordinate (see section “6.3.3. Scaled integers”):

```
x = 12650
Actual x value = 12.650
```

***ATTENTION**. Using of very small **`pix_step`** values may decrease the execution speed. Sub-pixel accuracy in the range [0.5, 0.1] is sufficient in many practical cases.*

Examples with different number of scan lines for non-rotated rectangle. The arrows show the scan direction:

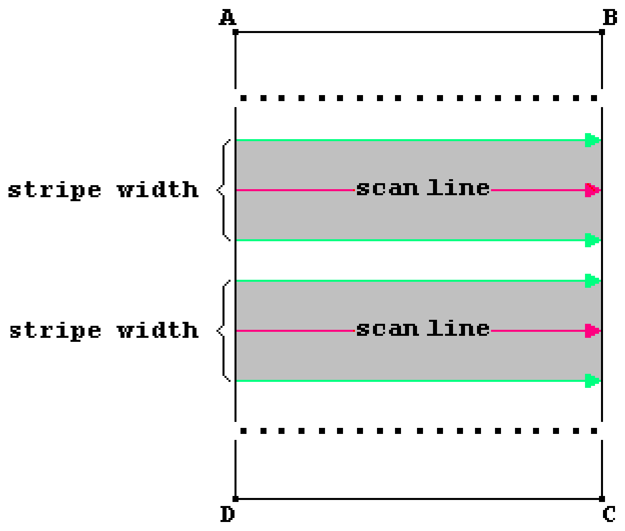

A
B
scan line
line_cnt = 1:
One scan line in the middle of the rectangle
D
C
A
B
scan line
scan line
line_cnt = 2:
Two scan lines = top and bottom rectangle lines
D
C
A
B
scan line
scan line
scan line
scan line
scan line
line_cnt = 5:
Five uniformly distributed scan lines including the top and the bottom rectangle lines
D
C

Working with stripes:

Working with single scan lines is sometimes sensitive to noise, for example in edge detection. In such cases we recommend using of mean pixel values on whole stripes (bands), which method is tolerant to noise and yields stable detection of edges. The stripe mode is enabled by the following stripe control parameters:

`stripe_wid`	=	stripe (band) width in pixels
`stripe_lcnt`	=	stripe line count - number of pixel lines, which compose each stripe (**odd value**)

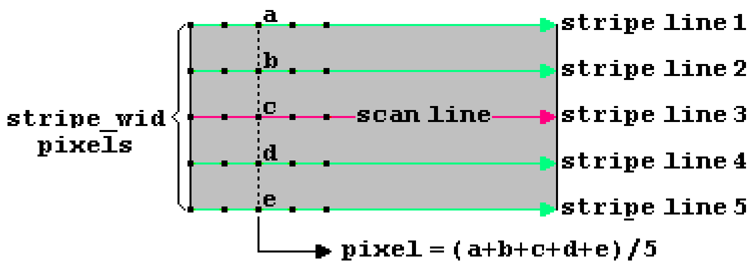
The stripes can be considered as thicker scan lines with thickness defined by **`stripe_wid`**. The stripe positions are determined by the scan lines, which lie in the middle of the stripes:



Each stripe is composed of **`stripe_lcnt`** pixel lines (called ***stripe lines***), uniformly distributed in the stripe width from both sides of the scan line. The respective pixels in the stripe lines are summed and divided by **`stripe_lcnt`** and the results are stored as pixel values in **`res_val`**.

Stripe example (`stripe_lcnt=5`):

5 pixel lines parallel to the scan line, are uniformly distributed in the stripe width. The red scan line, which defines the stripe position, is in the center of the stripe. In this example with non-rotated rectangle the middle scan line and all stripe lines are horizontal, but in the general case they can be arbitrary non-horizontal, non-vertical lines.



When the strip width is set to 1 (**`stripe_wid <= 1`** or **`stripe_lcnt <= 1`**) the stripes are disabled and the function uses single scan lines.

The function draws the stripe into the drawing image **`dr_img`**. The **`dr_mode`** argument specifies what elements of the stripe should be drawn: stripe enclosing rectangle, middle scan line, stripe pixels lines, etc.

TIPS & TRICKS*. Difference between **ip_read_pixel_lines** and **ip_read_pixel_rect**:*

* **ip_read_pixel_lines** : reads arbitrary rectangle, defined by 4 corners*

* **ip_read_pixel_rect** : reads rotated rectangle, specified by point, dx, dy and angle.*

Arguments:

Argument	Description
`vd_img`	[in] source gray-level image
`dr_img`	[in] drawing image (NULL: skip drawing)
`rect`	[in] float 8-element x/y buffer with rectangle corners: `(x,y) (x,y) (x,y) (x,y) :` coordinates of A, B, C, D corners
`line_cnt`	[in] number of scan lines in the region-of-interest (ROI) rectangle
`stripe_wid`	[in] stripe (band) width in pixels (the scan lines are in the middle of the stripes):

	`<=1` : no stripes, single scan lines `>1` : stripe width in pixels
`stripe_lcnt`	[in] stripe line count - number of pixel lines, which compose each stripe (**odd** value): `<=1` : no stripes, single scan lines `>1` : number of pixel lines uniformly distributed in the stripe width from both sides of the scan line (see the example)
`pix_step`	[in] float pixel step used to read line pixels in sub-pixel mode: `>= 1.0` : use fractional pixel step >= 1 `< 1.0 [0.5-0.001]` : use fractional pixel step < 1 Ignored when **`sub_pix=0`** (Bresenham).
`sub_pix`	[in] sub-pixel calculation mode (type of calculated pixel coordinates): `0` = Integer mode - use integer pixel coordinates to read line pixels, generated by the Bresenham's algorithm (ignores **`pix_step`**). `1` = Sub-pixel mode - use scaled*1000 integer sub-pixel coordinates to read line pixels.
`scan_dir`	[in] scan direction on each pixel line: `0` = normal : left to right (`L->R`) `1` = reverse: right to left (`R->L`)
`res_type`	[in] type of result pixel coordinates, stored in **`res_xy`**: `0` = integer `1` = scaled*1000 integer (1/1000 pixel accuracy) `2` = floating-point
`res_size`	[in] max size of result buffers in elements (values/points). The sizes of the result buffers in bytes must be: **`res_xy`** `: 2*res_size*sizeof(int): if` **`res_type=0,1`** (integer) `: 2*res_size*sizeof(float): if` **`res_type=2`** (float) **`res_val`** `: res_size*sizeof(int)`
`res_xy`	[out] buffer with x/y coordinates of line pixels, allocated by the calling function (NULL: don't return): `(x,y) (x,y) ... (x,y) : res_cnt_buf[0]` pixels of 1st line (top) `(x,y) (x,y) ... (x,y) : res_cnt_buf[1]` pixels of 2nd line `...................................................` `(x,y) (x,y) ... (x,y) : res_cnt_buf[line_cnt-1]` pixels of the last line (bottom) The coordinates are integer, scaled*1000 integer or floating-point numbers depending on **`res_type`**.
`res_val`	[out] buffer with line pixel values, allocated by the calling function (NULL : don't return): `p p ... p : res_cnt_buf[0]` pixel values of 1st line (top) `p p ... p : res_cnt_buf[1]` pixel values of 2nd line `............................................` `p p ... p : res_cnt_buf[line_cnt-1]` pixel values of last bottom line where: `p = 0-255` : valid pixel value `= -1` : pixel outside image
`res_cnt_buf`	[out] buffer with **`line_cnt`** elements with # of points on each scan line from top to bottom rectangle line. These values show the numbers of the pixel values or pixel coordinates stored in the result buffers for each scan line (NULL : don't return): `N0 N1 ... Nk (k =` **`line_cnt-1`**`)`
`res_cnt`	[out] total number of results `(N0 + N1 + ... + Nk)`, stored in the output buffers (**<= `res_size`**)
`dr_mode`	[in] drawing mode (OR of bits): `0` = don't draw `bit 1` = draw input rectangle

	bit 2 = draw middle scan line with arrow, showing the scan direction bit 3 = draw all scan lines with arrows
dr_clrs	[in] 3-element buffer with drawing colors (NULL = use default colors): [0] = rectangle color (default = bright green) [1] = color of middle scan line and arrow (default = bright green) [2] = color of all scan lines and arrows (default = yellow) [3] = error color (default = bright red)

Return code:

Return code	Description
0	Success
IP_ALLOC_ERROR	Memory allocation error
IP_INV_IMAGE	Invalid image
IP_INV_ARG	Invalid input argument(s)
IP_RESBUF_OVF	Result buffer overflow
IP_INTERNAL_ERR	Internal error
Other	Other errors returned by called functions

ip_read_pixel_rect = Read pixel rectangle

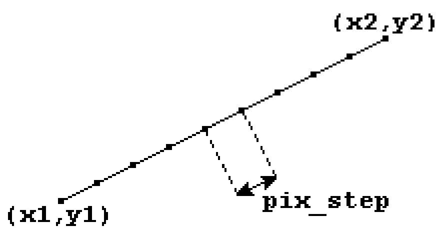
Prototype:

```
#include "ip_lib.h"
int ip_read_pixel_rect ( VD_IMAGE *vd_img, DR_IMAGE *dr_img,
                         int x, int y, int dx, int dy, int pt_pos,
                         float ang, int line_cnt,
                         int stripe_wid, int stripe_lcnt,
                         float pix_step, int sub_pix, int scan_dir,
                         int res_type, int res_size, void *res_xy,
                         int *res_val, int *res_cnt_buf, int *res_cnt,
                         int dr_mode, int *dr_clrs )
```

Description:

The function reads **line_cnt** pixel lines in a rotated rectangle area in the input gray-level image **vd_img**. The region-of-interest (ROI) rectangle is defined by a rectangle-origin point **(x,y)**, dimensions **dx**, **dy**, rotation angle **ang** around the origin point, and point position **pt_pos**. The pixel lines (called ***scan*** lines) are distributed uniformly in the rectangle from the top to the bottom rectangle line including the top/bottom lines with one exception – a single scan line (**line_cnt=1**) is in the middle of the rectangle, see the examples below. In non-rotated rectangle state (**ang=0**) the scan lines are horizontal and the pixels are read from left to right if **scan_dir=0** or from right to left if **scan_dir=1** - see the arrows in the examples below.

The **pix_step** argument defines Euclidean distance between successive line pixels. The pixels are uniformly distributed on each scan/stripe line when the scan direction is set from (x1,y1) to (x2,y2):



The function calculates line pixel coordinates in two modes, specified by **sub_pix**:

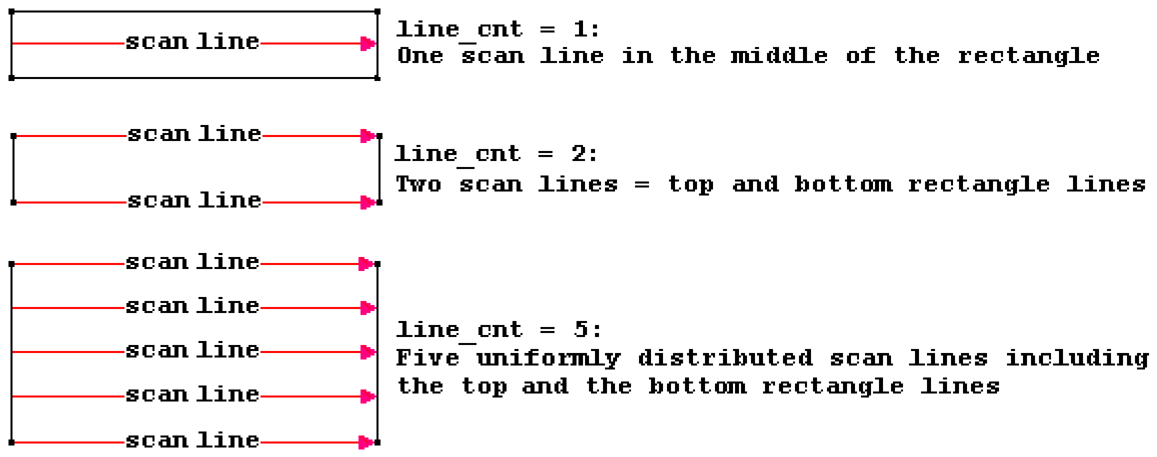
- **Integer mode**. The function uses the Bresenham's algorithm to generate integer coordinates of line pixels.
- **Sub-pixel mode**. The function generates pixel coordinates with sub-pixel accuracy, defined by **pix_step**, and reads sub-pixel values. The function uses internally scaled*1000 integer calculations to achieve best execution speed.

Example of ***scaled*1000 integer*** coordinate (see section "6.3.3. Scaled integers"):

```
x = 12650
Actual x value = 12.650
```

ATTENTION. *Using of very small* ***pix_step*** *values may decrease the execution speed. Sub-pixel accuracy in the range [0.5, 0.1] is sufficient in many practical cases.*

Examples with different number of scan lines for non-rotated rectangle. The arrows show the scan direction:

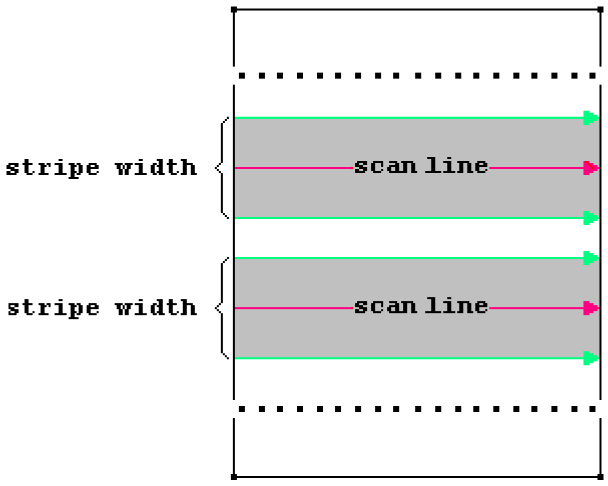

scan line
line_cnt = 1:
One scan line in the middle of the rectangle
scan line
scan line
line_cnt = 2:
Two scan lines = top and bottom rectangle lines
scan line
scan line
scan line
scan line
scan line
line_cnt = 5:
Five uniformly distributed scan lines including the top and the bottom rectangle lines

Working with stripes:

Working with single scan lines is sometimes sensitive to noise, for example in edge detection. In such cases we recommend using of mean pixel values on whole stripes (bands), which method is tolerant to noise and yields stable detection of edges. The stripe mode is enabled by the following stripe control parameters:

stripe_wid	=	stripe (band) width in pixels
stripe_lcnt	=	stripe line count - number of pixel lines, which compose each stripe (**odd value**)

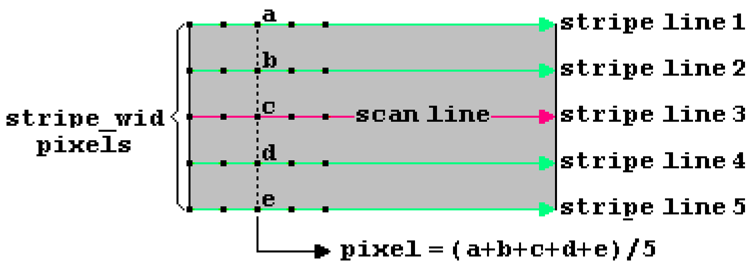
The stripes can be considered as thicker scan lines with thickness defined by **stripe_wid**. The stripe positions are determined by the scan lines, which lie in the middle of the stripes:



Each stripe is composed of **`stripe_lcnt`** pixel lines (called ***stripe lines***), uniformly distributed in the stripe width from both sides of the scan line. The respective pixels in the stripe lines are summed and divided by **`stripe_lcnt`** and the results are stored as pixel values in **`res_val`**.

Stripe example (`stripe_lcnt=5`):

5 pixel lines parallel to the scan line, are uniformly distributed in the stripe width. The red scan line, which defines the stripe position, is in the center of the stripe. In this example with non-rotated rectangle the middle scan line and all stripe lines are horizontal, but in the general case they can be arbitrary non-horizontal, non-vertical lines.



The function draws the stripe into the drawing image **`dr_img`**. The **`dr_mode`** argument specifies what elements of the stripe should be drawn: stripe enclosing rectangle, middle scan line, stripe pixels lines, etc.

TIPS & TRICKS. *Difference between **ip_read_pixel_lines** and **ip_read_pixel_rect**:*
*   **ip_read_pixel_lines** : reads arbitrary rectangle, defined by 4 corners*
*   **ip_read_pixel_rect** : reads rotated rectangle, specified by point, dx, dy and angle.*

Arguments:

Argument	Description
`vd_img`	[in] source gray-level image
`dr_img`	[in] drawing image (NULL: skip drawing)
`x`	[in] x-coordinate of rectangle-origin point
`y`	[in] y-coordinate of rectangle-origin point
`dx`	[in] rectangle width in pixels
`dy`	[in] rectangle height in pixels
`pt_pos`	[in] position of rectangle-origin point **`(x,y)`**: `0` = at rectangle center

	`1 =` at top/left rectangle corner
`ang`	[in] rectangle rotation angle in radians [0,2PI] around the rectangle-origin point. The angle increases in counter-clockwise direction. Angle 0 corresponds to a non-rotated rectangle with horizontal scan arrows from left to right (**`scan_dir = 0`**) or right to left (**`scan_dir = 1`**).
`line_cnt`	[in] number of scan lines in the region-of-interest (ROI) rectangle
`stripe_wid`	[in] stripe (band) width in pixels (the scan lines are in the middle of the stripes): `<=1 :` no stripes, single scan lines `>1 :` stripe width in pixels
`stripe_lcnt`	[in] stripe line count - number of pixel lines, which compose each stripe (**odd** value): `<=1 :` no stripes, single scan lines `>1 :` number of pixel lines uniformly distributed in the stripe width from both sides of the scan line (see the example)
`pix_step`	[in] float pixel step used to read line pixels in sub-pixel mode: `>= 1.0 :` use fractional pixel step >= 1 `<  1.0 [0.5-0.001] :` use fractional pixel step < 1 Ignored when **`sub_pix=0`** (Bresenham).
`sub_pix`	[in] sub-pixel calculation mode (type of calculated pixel coordinates): `0 =` **Integer mode** - use integer pixel coordinates to read line pixels, generated by the Bresenham's algorithm (ignores **`pix_step`**). `1 =` **Sub-pixel mode** - use scaled*1000 integer sub-pixel coordinates to read line pixels.
`scan_dir`	[in] scan direction on each pixel line: `0 =` normal : left to right (`L->R`) `1 =` reverse: right to left (`R->L`)
`res_type`	[in] type of result pixel coordinates, stored in **`res_xy`**: `0 =` integer `1 =` scaled*1000 integer (1/1000 pixel accuracy) `2 =` floating-point
`res_size`	[in] max size of result buffers in elements (values/points). The sizes of the result buffers in bytes must be: **`res_xy`** `: 2*res_size*sizeof(int): if` **`res_type=0,1`** (integer) `: 2*res_size*sizeof(float): if` **`res_type=2`** (float) **`res_val`** `: res_size*sizeof(int)`
`res_xy`	[out] buffer with x/y coordinates of line pixels, allocated by the calling function (NULL: don't return): `(x,y) (x,y) ... (x,y) : res_cnt_buf[0]` pixels of 1st line (top) `(x,y) (x,y) ... (x,y) : res_cnt_buf[1]` pixels of 2nd line `..................................................` `(x,y) (x,y) ... (x,y) : res_cnt_buf[line_cnt-1]` pixels of the last line (bottom) The coordinates are integer, scaled*1000 integer or floating-point numbers depending on **`res_type`**.
`res_val`	[out] buffer with line pixel values, allocated by the calling function (NULL : don't return): `p p ... p : res_cnt_buf[0]` pixel values of 1st line (top) `p p ... p : res_cnt_buf[1]` pixel values of 2nd line `............................................` `p p ... p : res_cnt_buf[line_cnt-1]` pixel values of last bottom line where: `p = 0-255 :` valid pixel value `= -1 :` pixel outside image
`res_cnt_buf`	[out] buffer with **`line_cnt`** elements with # of points on each scan line from top to bottom rectangle line. These values show the number of the pixel values or

	coordinates stored in the result buffers for each scan line (NULL : don't return): `N0 N1 ... Nk  (k = `**`line_cnt-1`**`)`
`res_cnt`	[out] total number of results `(N0 + N1 + ... + Nk)`, stored in the output buffers (**`<= res_size`**)
`dr_mode`	[in] drawing mode (OR of bits): `0      =` don't draw `bit 1 =` draw input rectangle `bit 2 =` draw middle scan line with arrow, showing the scan direction `bit 3 =` draw all scan lines with arrows
`dr_clrs`	[in] 3-element buffer with drawing colors (NULL = use default colors): `[0] =` rectangle color (default = bright green) `[1] =` color of middle scan line and arrow (default = bright green) `[2] =` color of all scan lines and arrows (default = yellow) `[3] =` error color (default = bright red)

Return code:

Return code	Description
`0`	Success
`IP_ALLOC_ERROR`	Memory allocation error
`IP_INV_IMAGE`	Invalid image
`IP_INV_ARG`	Invalid input argument(s)
`IP_RESBUF_OVF`	Result buffer overflow
`IP_INTERNAL_ERR`	Internal error
Other	Other errors returned by called functions

ip_edge_det_th = Edge detection by threshold

Prototype:

```
#include "ip_lib.h"
int ip_edge_det_th ( int *data_buf, int data_cnt, int mode, int edge_dir,
                     int edge_type, int edge_th, int res_type,
                     int res_size, void *res_pos, int *res_val,
                     int *res_cnt )
```

Description:

The function finds edges in the input data buffer **`data_buf`** by a simple threshold method. The input data buffer usually contains pixel values in the range [0,255].

An edge between two successive data values **`p1`**, **`p2`** occurs when the two values cross the edge-detection threshold **`edge_th`**:

```
p1 >= edge_th && p2 <  edge_th : falling edge L->D (light to dark)
p1 <  edge_th && p2 >= edge_th : rising edge  D->L (dark to light)
```

The **`edge_th`** argument specifies edge-detection threshold, usually pixel brightness in the range [0,255]. A good practice is to calculate this value dynamically for each input image - for example by the percent-threshold tool **ip_perc_threshold**. Put the percent-threshold working rectangle on a place where both dark and light pixels occur. Typical threshold value = 127 for gray-level image with good contrast.

Results:

The **`res_type`** argument specifies the type of the edge positions values, stored into the output buffer **`res_pos`**:

- **Integer -** the function stores rounded edge positions. Each edge position is equal to the index of the neighboring data point, which is nearer to the detected edge.
- **Scaled*1000 integer** - the function stores fractional edge indexes as scaled*1000 integer numbers with accuracy 1/1000.
- **Float** - the function stores fractional edge indexes as floating point numbers.

You must pass respective pointer **`(int *)`**, **`(int *)`** or **`(float *)`** in **`res_pos`**.

The **`res_val`** output buffer contains signed edge strength values, which show also the edge directions. Greater magnitudes (absolute strength values) mean stronger edges:

```
res_val[i] < 0 : L->D edge
res_val[i] > 0 : D->L edge
```

You can disable returning of edge positions and/or edge strength values in **`res_pos`** and **`res_val`** by passing NULL pointers to the respective arguments. When the function finds more edges than **`res_size`**, it stores the first detected **`res_size`** edges into the output buffers and returns error `IP_RESBUF_OVF`.

The function may return one edge (first/last/strongest) or all edges with direction, defined by **`edge_dir`** (`L->D`, `D->L` or both).

Circular mode:

The function can work in circular mode (**`mode`** bit 0 == 1), when the first data element is appended at the end of the data buffer. Thus you can find an edge between the first and the last data elements in a circular data buffer.

Using negative and/or positive data values:

The function works on positive data values only (**`mode`** bit 1 == 0), or both on positive and negative data values in **`data_buf`** (**`mode`** bit 1 == 1). In the first case the function ignores negative data values when searching edges.

Arguments:

Argument	Description

`data_buf`	[in] source data buffer (pixel values):
	`p p p ...`
	where:
	`p >= 0 :` valid pixel value
	`p  < 0 :` pixel out of image if `mode(bit 1)==0`

`data_cnt`	[in] number of **`data_buf`** elements
`mode`	[in] operation mode: `bit 0 =` circular mode `bit 1 =` use negative data values
`edge_dir`	[in] direction for searched edge(s): `0 =` light to dark: `L->D` (falling edge) `1 =` dark to light: `D->L` (rising edge)

	2 = both directions
edge_type	[in] type of searched edge(s): 0 = first 1 = last 2 = strongest 3 = all
edge_th	[in] edge-detection threshold - usually pixel brightness in the range [0,255]
res_type	[in] type of edge position values, stored in **res_pos**: 0 = integer (rounded to the index of the nearest data value) 1 = scaled*1000 integer (1/1000 accuracy) 2 = floating-point
res_size	[in] number of elements in the result buffers **res_pos** and **res_val** (max number of results to store)
res_pos	[out] destination buffer with edge position results (**NULL**: don't return). This buffer contains indexes in **data_buf**, where edges are detected. The indexes are integer or float values depending on **res_type**.
res_val	[out] destination buffer with signed edge strength values (**NULL**: don't return). These values show also the edge directions: <0 : L->D edge >0 : D->L edge
res_cnt	[out] number of results, stored in the output buffers **res_pos** and **res_val** (**res_cnt <= res_size**)

Return code:

Return code	Description
0	Success
IP_INV_ARG	Invalid function argument(s)
IP_ALLOC_ERROR	Memory allocation error
IP_RESBUF_OVF	Result buffer overflow

ip_edge_det_grad = Edge detection by gradient

Prototype:

```
#include “ip_lib.h”
int ip_edge_det_grad ( int *data_buf, int data_cnt, int mode,
                       int edge_dir, int edge_type, int edge_th,
                       int grad_size, int grad_dist,
                       int res_type, int res_size,
                       void *res_pos, int *res_val, int *res_cnt )
```

Description:

The function finds edges in the input data buffer **data_buf** by a slope-based gradient algorithm. The input data buffer usually contains pixel values in the range [0,255].

Algorithm:

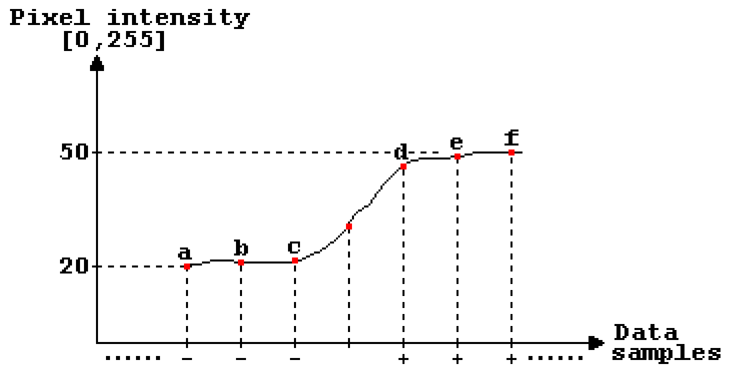
The edge detection is based on the calculation of gradient values by accumulating two groups of values with given size **`grad_size`** and distance **`grad_dist`**, and then subtracting one from the other. The difference is normalized by dividing the sum by **`grad_size`** to keep the threshold independent from **`grad_size`**. Edges are reported on places, where the gradient magnitude exceeds the threshold **`edge_th`**:

	grad_dist = 0	grad_dist = 1	grad_dist = 2
grad_size = 1	-+	- +	- +
grad_size = 2	--++	-- ++	-- ++
grad_size = 3	---+++	--- +++	--- +++

When a sequence of adjacent edge points with equal edge directions is found, only the edge with the greatest gradient is reported.

Example:

Suppose `edge_th=10`, `grad_size=3` and `grad_dist=1`. A data section of 7 elements is checked for edges by sliding the section on all possible positions inside the input data buffer **`data_buf`**.



In each position the gradient value is calculated as:

```
grad_val = ((d+e+f – (a+b+c))/3
```

An edge is detected if:

`grad_val >= 10  :` if searching `D->L` : rising edge direction
`grad_val <= -10 :` if searching `L->D` : falling edge direction

Choosing threshold and gradient values

The edge-detection threshold and the gradient sizes should be chosen depending on the contents of processed image. The threshold **`edge_th`** depends on the image contrast and how steep are the slopes between dark and light pixels, where the function finds edges.

In the example above the flat pixel levels between the edge occupy at least 3 pixels and the distance between them is 1 pixel, so it is adequate to chose `grad_size=3` and `grad_dist=1`. Other threshold and gradient values could be adjusted by experiments.

Remember that greater threshold values ignore noise and find distinct edges. Smaller threshold values find more edges, but are sensitive to noise.

Suggested default values:

```
edge_th   = 10
grad_size = 1
grad_dist = 0
```

Results:

The **`res_type`** argument specifies the type of the edge positions values, stored in the output buffer **`res_pos`**:

- **Integer** - the function stores rounded edge positions. Each edge position is equal to the index of the neighboring data point, which is nearer to the detected edge.
- **Scaled*1000 integer** - the function stores fractional edge indexes as scaled*1000 integer numbers with accuracy 1/1000.
- **Float** - the function stores fractional edge indexes as floating-point numbers.

You must pass respective pointer **`(int *)`**, **`(int *)`** or **`(float *)`** in **`res_pos`**.

The **`res_val`** output buffer contains signed edge strength values, which show also the edge directions. Greater magnitudes (absolute strength values) mean stronger edges:

```
  res_val[i] < 0 : L->D edge
  res_val[i] > 0 : D->L edge
```

You can disable returning of edge positions and/or edge strength values in **`res_pos`** and **`res_val`** by passing NULL pointers to the respective arguments. When the function finds more edges than **`res_size`**, it stores the first detected **`res_size`** edges into the output buffers and returns error `IP_RESBUF_OVF`.

The function may return one edge (first/last/strongest) or all edges with direction, defined by **`edge_dir`** (`L->D`, `D->L` or both).

Circular mode:

The function can work in circular mode (**`mode`** bit 0 == 1), when the first `(grad_len-1)` input elements are appended at the end of the data buffer. The `grad_len` value is equal to the length of the data section being checked for edges:

```
   grad_len = grad_size*2 + grad_dist
```

Thus you can find an edge in circular data between the first and the last data elements.

Using negative and/or positive data values:

The function works on positive data values only (**`mode`** bit 1 == 0), or both on positive and negative data values in **`data_buf`** (**`mode`** bit 1 == 1). In the first case the function ignores negative data values when searching edges.

Arguments:

Argument	Description

`data_buf`	[in] source data buffer (pixel values):
	`p p p ...`
	where:
	`p >= 0 :` valid pixel value
	`p  < 0 :` pixel out of image if `mode(bit 1)==0`

`data_cnt`	[in] number of **`data_buf`** elements
`mode`	[in] operation mode: `bit 0 =` circular mode `bit 1 =` use negative data values
`edge_dir`	[in] direction for searched edge(s): `0 =` light to dark: `L->D` (falling edge) `1 =` dark to light: `D->L` (rising edge) `2 =` both directions
`edge_type`	[in] type of searched edge(s): `0 =` first `1 =` last

	`2` = strongest `3` = all
`edge_th`	[in] gradient edge-detection threshold (suggested default value = 10)
`grad_size`	[in] gradient size (>= 1, see the description above), default=1
`grad_dist`	[in] gradient distance (>=0, see the description above), default=0
`res_type`	[in] type of edge position values, stored in **`res_pos`**: `0` = integer (rounded to the index of the nearest data value) `1` = scaled*1000 integer (1/1000 accuracy) `2` = floating-point
`res_size`	[in] number of elements in the result buffers **`res_pos`** and **`res_val`** (max number of results to store)
`res_pos`	[out] destination buffer with edge position results (**NULL**: don't return). This buffer contains indexes in **`data_buf`**, where edges are detected. The indexes are integer or float values depending on **`res_type`**.
`res_val`	[out] destination buffer with signed edge strength values (**NULL**: don't return). These values show also the edge directions: `<0 : L->D` edge `>0 : D->L` edge
`res_cnt`	[out] number of results, stored in the output buffers **`res_pos`** and **`res_val`** (**`res_cnt <= res_size`**)

Return code:

Return code	Description
`0`	Success
`IP_INV_ARG`	Invalid function argument(s)
`IP_ALLOC_ERROR`	Memory allocation error
`IP_RESBUF_OVF`	Result buffer overflow

ip_calc_edge_coord = Calculate x/y edge coordinates

Prototype:

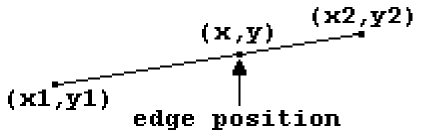
```
#include “ip_lib.h”
int ip_calc_edge_coord ( void *pix_xy, int xy_type, int pix_cnt,
                         void *edge_pos, int pos_type, int edge_cnt,
                         void *res_xy, int res_type )
```

Description:

The function calculates x/y coordinates of edges, detected previously by one of the edge-detection functions. The function receives input buffers with edge positions and pixel coordinates. The **`pix_xy`** buffer contains the coordinates of the pixels on one scan line, and the **`edge_pos`** buffer contains the edge positions detected on this line by the edge-detection functions **ip_edge_det_th** or **ip_edge_det_grad**.

Algorithm:

The function applies linear interpolation method to get the **`(x,y)`** coordinates of an edge, which lies between two successive pixels **`(x1,y1)`** and **`(x2,y2)`**:



The linear interpolation is done by integer calculations with 1/100 pixel accuracy.

Arguments:

Argument	Description
`pix_xy`	[in] source buffer with x/y pixel coordinates: `(x,y) (x,y) (x,y) ...`
`xy_type`	[in] type of values in **`pix_xy`**: `0 =` integer `1 =` scaled*1000 integer (1/1000 accuracy) `2 =` floating-point
`pix_cnt`	[in] number of **`pix_xy`** elements (x,y pairs)

`edge_pos`	[in] source buffer with edge positions:
	`pos pos pos  ...`
	where:
	`pos =` edge index in the pixel buffer **`pix_xy`** in the range `[0, pix_cnt-1]`

`pos_type`	[in] type of values in **`edge_pos`**: `0 =` integer `1 =` scaled*1000 integer (1/1000 accuracy) `2 =` floating-point
`edge_cnt`	[in] number of **`edge_pos`** elements

`res_xy`	[out] destination buffer with **`edge_cnt`** (x,y) edge coordinates in format:
	`(x,y) (x,y) (x,y) ...`
	The minimum buffer size in bytes must be:
	`res_type = 0,1 : 2 * edge_cnt * sizeof(int)`
	`res_type = 2   : 2 * edge_cnt * sizeof(float)`

`res_type`	[in] type of edge coordinates, stored in **`res_xy`**: `0 =` integer (rounded to nearest integer coordinates) `1 =` scaled*1000 integer (1/1000 accuracy) `2 =` floating-point

Return code:

Return code	Description
`0`	Success
`IP_ALLOC_ERROR`	Memory allocation error
`IP_INV_ARG`	Invalid function argument(s)
`IP_INV_BUFFER`	Mismatching edge position and pixel coordinate buffers: `edge_pos[i] >= pix_cnt`

ip_edge = Edge-detection tool

Prototype:

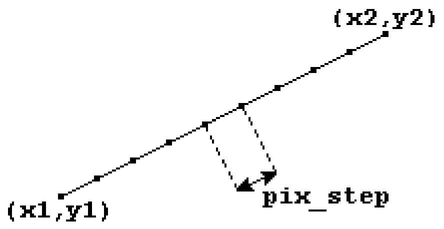
```
#include “ip_lib.h”
int ip_edge ( VD_IMAGE *vd_img, DR_IMAGE *dr_img,
              int x, int y, int dx, int dy, int pt_pos, float ang,
              int line_cnt, int stripe_wid, int stripe_lcnt,
              float pix_step, int sub_pix, int scan_dir,
              int edge_dir, int edge_type, int edge_th,
              int grad_size, int grad_dist, int method,
              int res_mode, int res_type, int res_size, void *res_xy,
              int *res_val, int *res_cnt_buf, int *res_cnt,
              int dr_mode, int dr_clr, int err_clr )
```

Description:

The function is the top-level edge-detection tool, which implements all methods for data sampling and all edge-detection algorithms, currently present in the edge-detection library.

The function finds edges on **`line_cnt`** pixel lines in a rotated rectangle area in the input gray-level image **`vd_img`**. The region-of-interest (ROI) rectangle is defined by a rectangle-origin point **`(x,y)`**, dimensions **`dx`**, **`dy`**, rotation angle **`ang`** around the origin point, and point position **`pt_pos`**. The pixel lines (called ***scan*** lines) are distributed uniformly in the rectangle from the top to the bottom rectangle line including the top/bottom lines with one exception – a single scan line (**`line_cnt=1`**) is in the middle of the rectangle, see the examples below. In non-rotated rectangle state (**`ang=0`**) the scan lines are horizontal and the pixels are read from left to right if **`scan_dir=0`** or from right to left if **`scan_dir=1`** - see the arrows in the examples below.

The **`pix_step`** argument defines Euclidean distance between successive line pixels. The pixels are uniformly distributed on each scan/stripe line when the scan direction is set from `(x1,y1)` to `(x2,y2)`:



The function calculates line pixel coordinates in two modes, specified by **`sub_pix`**:

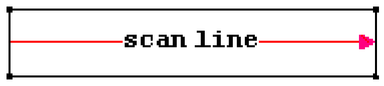
- **Integer mode**. The function uses the Bresenham's algorithm to generate integer coordinates of line pixels (fastest method, **`pix_step`** is ignored).
- **Sub-pixel mode**. The function generates pixel coordinates with sub-pixel accuracy, defined by **`pix_step`**, and reads sub-pixel values. The function uses scaled*1000 integer calculations to achieve high execution speed.

Example of ***scaled*1000 integer*** coordinate (see section “6.3.3. Scaled integers”):

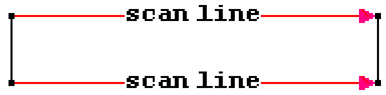
```
x = 12650
Actual x value = 12.650
```

***ATTENTION**. Using of very small **`pix_step`** values may decrease the execution speed. Sub-pixel accuracy in the range [0.5, 0.1] is sufficient in many practical cases.*

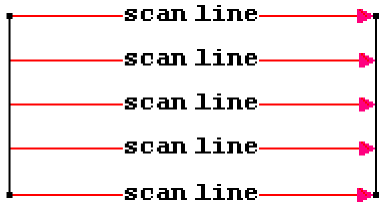
Examples with different number of scan lines for non-rotated rectangles. The arrows show the scan direction:



line_cnt = 1:
One scan line in the middle of the rectangle



line_cnt = 2:
Two scan lines = top and bottom rectangle lines



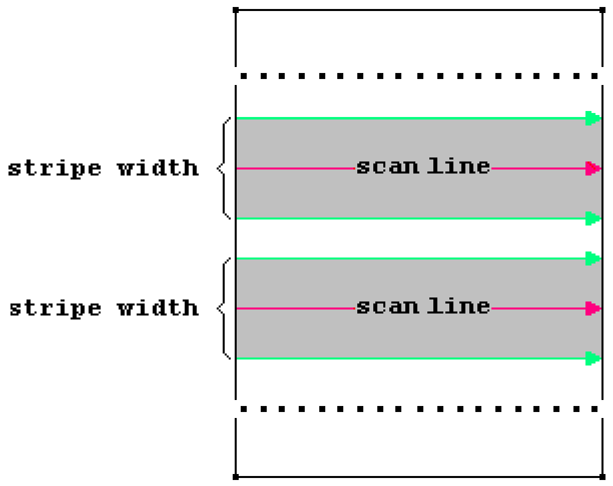
line_cnt = 5:
Five uniformly distributed scan lines including the top and the bottom rectangle lines

Working with stripes:

Working with single scan lines to find edges is sometimes sensitive to noise. In such cases we recommend using of mean pixel values on whole stripes (bands of pixel lines), which are tolerant to noise and yield more stable detection of edges. The stripe mode is enabled by the following stripe control parameters:

`stripe_wid`	=	stripe (band) width in pixels
`stripe_lcnt`	=	stripe line count - number of pixel lines, which compose each stripe (**odd value**)

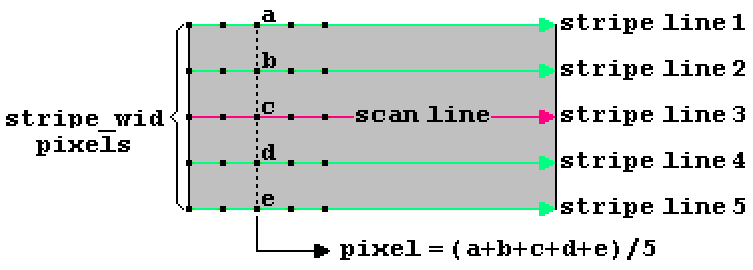
The stripes can be considered as thicker scan lines with thickness **`stripe_wid`** pixels. The stripe positions are determined by the scan lines, which lie in the middle of the stripes:



Each stripe is composed of **`stripe_lcnt`** pixel lines (called ***stripe lines***), uniformly distributed in the stripe width from both sides of the scan line. The respective pixels in the stripe lines are summed and divided by **`stripe_lcnt`** and the tool searches edges on these mean pixel values.

Stripe example (`stripe_lcnt=5`):

5 pixel lines parallel to the scan line, are uniformly distributed in the stripe width. The red scan line, which defines the stripe position, is in the center of the stripe. This example shows non-rotated rectangle with horizontal scan and stripe lines, but in the general case they can be arbitrary non-horizontal, non-vertical lines.



The function uses one of the following edge-detection methods:

Threshold-based edge-detection:

An edge between two successive pixels **p1**, **p2** is detected when the two values cross the edge-detection threshold **`edge_th`**:

```
p1 >= edge_th && p2 <  edge_th : falling edge L->D (light to dark)
p1 <  edge_th && p2 >= edge_th : rising edge  D->L (dark to light)
```

The **`edge_th`** argument specifies pixel brightness in the range [0,255]. A good practice is to calculate this value dynamically for each input image - for example by the percent-threshold tool **ip_perc_threshold**. Put the percent-threshold working rectangle on a place where both dark and light pixels occur. Typical threshold value = 127 for gray-level image with good contrast.

Gradient (slope)-based edge detection:

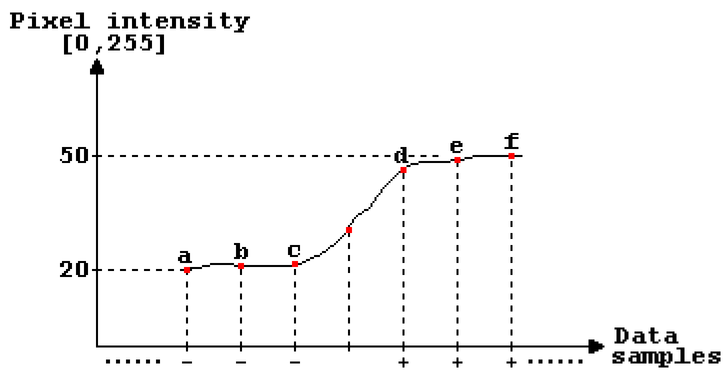
The edge detection is based on the calculation of gradient values by accumulating two groups of values with given size **`grad_size`** and distance **`grad_dist`**, and then subtracting one from the other. The difference is normalized by dividing the sum by **`grad_size`** to keep the threshold independent from **`grad_size`**. Edges are reported on places, where the gradient magnitude exceeds the threshold **`edge_th`**:

	grad_dist = 0	grad_dist = 1	grad_dist = 2
grad_size = 1	-+	- +	- +
grad_size = 2	--++	-- ++	-- ++
grad_size = 3	---+++	--- +++	--- +++

When a sequence of adjacent edge points with equal edge directions is found, only the edge with the greatest gradient is reported.

Example:

Suppose `edge_th=10`, `grad_size=3` and `grad_dist=1`. A data section of 7 elements is checked for edges by sliding the section on all possible positions on each scan line:



In each position the gradient value is calculated as:

```
grad_val = ((d+e+f – (a+b+c))/3
```

An edge is detected if:

```
  grad_val >= 10  : if searching D->L : rising edge direction
  grad_val <= -10 : if searching L->D : falling edge direction
```

Choosing gradient threshold and gradient sizes

The edge-detection threshold and the gradient sizes should be chosen depending on the contents of processed image. The threshold **`edge_th`** depends on the image contrast and how steep are the slopes between dark and light pixels, where the function finds edges.

In the example above the flat pixel levels between the edge occupy at least 3 pixels and the distance between them is 1 pixel, so it is adequate to choose `grad_size=3` and `grad_dist=1`. Other threshold and gradient values could be adjusted by experiments.

Remember that greater threshold values ignore noise and find distinct edges. Smaller threshold values find more edges, but are sensitive to noise.

Suggested default values:

```
  edge_th   = 10
  grad_size = 1
  grad_dist = 0
```

Results:

The **`res_type`** argument specifies the type of the edge coordinate values, stored into the output buffer **`res_xy`**:

- **Integer** - the function stores rounded x/y edge coordinates. Each edge coordinates are equal to the coordinates of the neighboring pixel, which is nearer to the detected edge.
- **Scaled*1000 integer** - the function stores fractional edge coordinates as scaled*1000 integer numbers with accuracy 1/1000.
- **Float** - the function stores fractional edge coordinates as floating point numbers.

You must pass respective pointer **`(int *)`**, **`(int *)`** or **`(float *)`** in **`res_xy`**.

The **`res_val`** output buffer contains signed edge strength values, which show also the edge directions. Greater magnitudes (absolute strength values) mean stronger edges:

```
  res_val[i] < 0 : L->D edge
  res_val[i] > 0 : D->L edge
```

You can disable returning of edge coordinates and/or edge strength values in **`res_xy`** and **`res_val`** by passing NULL pointers to the respective arguments. When the function finds more edges than **`res_size`**, it stores the first detected **`res_size`** edges into the output buffers and returns error `VM_RESBUF_OVF`.

On each scan line the function may return one edge (first, last orstrongest) or all edges with direction, defined by **`edge_dir`** (`L->D`, `D->L` or both).

Normal result mode:

In normal result mode (**`res_mode`** = 0), the tool stores the parameters of the detected edges into the result buffers. Edge coordinates and edge strength values are stored sequentially into **`res_xy`** and **`res_val`** from top to bottom scan line. In order to see which edges belong to a given scan line (and if any), you should read sequentially the elements of **`res_xy`** and **`res_val`**, taking into account the numbers of detected edges on each scan line, which are stored into **`res_cnt_buf`**.

Example:

```
                                          Edge strength     Edge (x,y)
                                         --------------- ----------------
  line 0: 1 edge  : res_cnt_buf[0] = 1 :   res_val[0]       res_xy[0,1]
  line 1: 0 edges : res_cnt_buf[1] = 0 :      -                  -
  line 2: 3 edges : res_cnt_buf[2] = 3 :   res_val[1-3]     res_xy[2-7]
  line 3: 2 edges : res_cnt_buf[3] = 2 :   res_val[4-5]     res_xy[8-11]
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
```

Dummy edges:

The dummy edges (enabled by **`res_mode`** = 1) are introduced to facilitate the processing of edges on multiple scan lines. In many cases you will request searching of single edges on the scan lines - 1st, last or strongest (**`edge_type`** = 0-2). Now you may disable the **`res_cnt_buf`** result buffer by passing a NULL pointer. The tool will store exactly **`line_cnt`** results into **`res_xy`** and/or **`res_val`**, and the edge values and coordinates will show whether an edge is found or not on the respective scan line **`'i'`**:

`res_val[i] = v =` edge strength value:

  `v != 0 :` valid edge: `<0:L->D, >0:D->L`

  `v == 0 :` dummy edge (edge not detected)

`res_xy[2*i,2*i+1] = (x,y) =` edge coordinates:

  `x,y >= 0 :` valid edge

  `x,y  < 0 :` dummy edge (edge not detected)

Although not needed, if you have enabled **`res_cnt_buf`** in ***dummy edge*** mode, all **`res_cnt_buf`** elements will be set to 1 - a dummy edge or a valid edge is stored into **`res_xy`** and/or **`res_val`**. Check the edge strength values or the edge coordinates, as described above.

Example:

```
                                         Edge strength   Edge type
                                         -------------  -----------
line 0: 1 edge : res_cnt_buf[0] = 1 : res_val[0]=20   D->L edge
line 1: 1 edge : res_cnt_buf[1] = 1 : res_val[1]=0        -
line 2: 1 edge : res_cnt_buf[2] = 1 : res_val[2]=-30  L->D edge
line 3: 1 edge : res_cnt_buf[3] = 1 : res_val[3]=15   D->L edge
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
```

***ATTENTION**.*

*The **dummy edge** mode is disabled internally (**`res_mode`**=0) when searching of all edges is specified by **edge_type=3**.*

*Remember that **`res_cnt`** contains the actual total number of detected valid edges. In **res_mode**=0 this number is equal to the total number of stored results. In **res_mode**=1 this number can be less than the total number of stored results due to stored dummy edges (the actual number of stored edge results is equal to **`line_cnt`**).*

*The described above result modes are important if you want to know the distribution of edges on the scan lines. If you don't bother about which edge on which scan line is located, you may work in normal mode **res_mode=**0 with **res_cnt_buf=NULL**. The tool will return in **res_cnt** the number of results, stored into **res_xy** and **res_val**.*

Drawing:

The tool draws in **`dr_img`** the ROI rectangle and a center scan line with arrow, showing the scan direction. In stripe mode the tool draws two additional dashed lines, parallel to the middle scan lines, which show the stripe width. The tool draws also small **`'x'`** markers on all detected edge points.

NOTE: Edge markers are drawn only when **`res_xy`** `!= NULL`.

***ATTENTION**. Using stripes with the gradient edge-detection method need scan direction of the working rectangle, which is approximately perpendicular to the searched edges. Since pixels are summed on the whole stripe width, reliable results are obtained when perpendicular edges occur on the whole stripe width.*

***TIPS & TRICKS**. To achieve reliable and stable results, adjust the rotation angle so that the scan lines are approximately perpendicular to the object's contour.*

Arguments:

Argument	Description
`vd_img`	[in] source gray-level image
`dr_img`	[in] drawing image (NULL: skip drawing)
`x`	[in] x-coordinate of rectangle-origin point
`y`	[in] y-coordinate of rectangle-origin point
`dx`	[in] rectangle width in pixels
`dy`	[in] rectangle height in pixels
`pt_pos`	[in] position of rectangle-origin point **`(x,y)`**: `0` = at rectangle center `1` = at top/left rectangle corner
`ang`	[in] rectangle rotation angle in radians [0,2PI] around the rectangle-origin point. The angle increases in counter-clockwise direction. Angle 0 corresponds to a non-rotated rectangle with horizontal scan arrows from left to right (**`scan_dir = 0`**) or right to left (**`scan_dir = 1`**).
`line_cnt`	[in] number of scan lines in the rectangle
`stripe_wid`	[in] stripe (band) width in pixels (the scan lines are in the middle of the stripes): `<=1` : no stripes, single scan lines `>1` : stripe width in pixels
`stripe_lcnt`	[in] stripe line count - number of pixel lines, which compose each stripe (**odd** value): `<=1` : no stripes, single scan lines `>1` : number of pixel lines uniformly distributed in the stripe width from both sides of the scan line (see the example)
`pix_step`	[in] pixel step on each scan/stripe line, used to read line pixels in sub-pixel mode: `>= 1.0` : use fractional pixel step >= 1 `<  1.0 [0.5-0.001]` : use fractional pixel step < 1 Ignored when **`sub_pix=0`** (Bresenham).
`sub_pix`	[in] sub-pixel calculation mode (type of calculated pixel coordinates): `0` = **Integer mode** – the function uses integer pixel coordinates to read line pixels, generated by the Bresenham's algorithm (ignores **`pix_step`**). `1` = **Sub-pixel mode** – the function uses scaled*1000 integer sub-pixel coordinates to read line pixels.
`scan_dir`	[in] scan direction on each pixel line: `0` = normal : left to right (`L->R`) `1` = reverse: right to left (`R->L`)
`edge_dir`	[in] direction for searched edge(s): `0` = light to dark: `L->D` (falling edge) `1` = dark to light: `D->L` (rising edge) `2` = both directions
`edge_type`	[in] type of searched edge(s) on each scan line: `0` = first `1` = last `2` = strongest `3` = all
`edge_th`	[in] edge-detection threshold. Choose threshold values depending on the edge-

	detection method. Suggested default values: `127` – for threshold-based edge-detection `10` – for gradient-based edge-detection
`grad_size`	[in] gradient size, used for method 1: [1,2,3,…], default=1
`grad_dist`	[in] gradient distance, used for method 1: [0,1,2,…], default=0
`method`	[in] edge-detection method: `0 =` threshold-based edge detection `1 =` gradient(slope)-based edge-detection `2 =` ... (spare)
`res_mode`	[in] result mode: `0 =` ***normal*** result mode - store only detected edges into result buffers `1 =` ***dummy edge*** result mode - store detected and dummy edges to facilitate processing of edge results when one edge per scan line is searched (no need of **`res_cnt_buf`**). Ignored if **`edge_type`** = 3
`res_type`	[in] type of result edge coordinates, stored in **`res_xy`**: `0 =` integer `1 =` scaled*1000 integer (1/1000 pixel accuracy) `2 =` floating-point
`res_size`	[in] max size of result buffers in elements (values/points) = max number of edge points to detect. The minimum sizes of the result buffers in bytes must be: **`res_xy`** `: 2*res_size*sizeof(int): if` **`res_type=0,1`** (integer) `: 2*res_size*sizeof(float): if` **`res_type=2`** (float) **`res_val`** `: res_size*sizeof(int)`
`res_xy`	[out] buffer with x/y coordinates of detected edge points, allocated by the calling function (NULL: don’t return): `(x,y) (x,y) ... (x,y) : res_cnt_buf[0]` edges on 1st line (top) `(x,y) (x,y) ... (x,y) : res_cnt_buf[1]` edges on 2nd line `...................................................` `(x,y) (x,y) ... (x,y) : res_cnt_buf[line_cnt-1]` edges on the last line (bottom) where: `x,y >= 0 :` valid edge `x,y < 0 :` dummy edge (if **`res_mode`** = 1) The coordinates are integer, scaled*1000 integer or floating-point numbers depending on **`res_type`**.
`res_val`	[out] buffer with signed edge strength values, allocated by the calling function (NULL : don't return): `v v ... v : res_cnt_buf[0]` values on 1st line (top) `v v ... v : res_cnt_buf[1]` values on 2nd line `.........................................` `v v ... v : res_cnt_buf[line_cnt-1]` values on last line (bottom) where: `v < 0 : L->D` edge `v > 0 : D->L` edge `v = 0 :` dummy edge (if **`res_mode`** = 1)
`res_cnt_buf`	[out] **`line_cnt`**-element result buffer with numbers of stored results for each scan line from top to bottom, allocated by the calling function (NULL: don't return): `N0 N1 ... Nk` where: `Ni =` Number of stored results for scan line **`‘i’`**. Minimum **`res_cnt_buf`** size in bytes = `line_cnt*sizeof(int)` **NOTE:** When **`res_mode`** = 1 a dummy edge result is stored for each scan line if and

	edge is not found on this line (`Ni = 1`).
`res_cnt`	[out] total number of detected edges, excluding the dummy edges if **`res_mode`**=1 (<= **`res_size`**).
`dr_mode`	[in] drawing mode: `0 :` draw on (always) `1 :` draw off (never) `2 :` on success only `3 :` on error only
`dr_clr`	[in] drawing color on success (0=default: bright green)
`err_clr`	[in] drawing color on error (0=default: bright red)

Return code:

Return code	Description
`0`	Success
`VM_INV_ARG`	Invalid input argument(s)
`VM_INV_IMAGE`	Invalid input video image
`VM_ALLOC_ERROR`	Memory allocation error
`VM_RESBUF_OVF`	Result buffer overflow
Other	Other errors returned by called functions

5.6. Image processing library

The library contains image processing functions. All functions use gray-level video image of type **VD_IMAGE** (***image*** structure). Unless otherwise stated the functions do not need any setup, see the function reference.

Header files:

`ip_lib.h`	Image-processing library header
`lib_err.h`	VM_LIB library error codes

ip_alloc_img = Allocate image

Prototype:

```
#include "ip_lib.h"
int ip_alloc_img (int dx, int dy, image *img )
```

Description:

The function (re)allocates an image with dimensions, specified by the input parameters and setups the output image **`img`**. The pitch of the image currently is equal to **`dx`**. If the image is already allocated, the function reallocates it (frees and allocates).

Arguments:

Argument	Description
`dx`	[in] image width in pixels
`dy`	[in] image height in pixels
`img`	[out] allocated image

Return code:

Return code	Description
`0`	Success
`IP_INV_ARG`	Invalid argument(s)
`IP_ALLOC_ERROR`	Memory allocation error

ip_free_img = Free image

Prototype:

```
#include "ip_lib.h"
void ip_free_img (image *img )
```

Description:
The function frees the memory of the input/output image **`img`** and marks the image as unallocated by setting the image pointer **`img->st`** to zero.

Arguments:

Argument	Description
`img`	[i/o] image

Return code:
None

ip_img_copy = Image copy

Prototype:
```
#include “ip_lib.h”
void ip_img_copy (VD_IMAGE *in_img, VD_IMAGE *out_img )
```

Description:
The function copies the pixels of the source image **`in_img`** into the destination image **`out_img`**. In case of image dimensions mismatch, the function copies **`in_img`** into the upper left area of **`out_img`**, eventually truncated. The **`out_img`** dimensions **`dx`** and **`dy`** are not changed.

Arguments:

Argument	Description
`in_img`	[in] source image
`out_img`	[out] destination image

Return code:
None

ip_copy_imgbox = Copy image box (universal function)

Prototype:
```
#include “ip_lib.h”
void ip_copy_imgbox (image *in_img, image *out_img, VM_BOX *box, int mode )
```

Description:
The function copies an image box from **`in_img`** to **`out_img`**. The **`mode`** argument specifies box copy mode:

`mode = 0 :` copy **`in_img`** rectangle area, defined by **`box`**, into **`out_img`**
`mode = 1 :` copy **`in_img`** into **`out_img`** rectangle area, defined by **`box`**

In case of image/box dimensions mismatch, the function copies the source area into the upper left corner of the destination area eventually truncated.

Arguments:

Argument	Description
`in_img`	[in] source image
`out_img`	[out] destination image
`box`	[in] source/destination image box (NULL: copy whole image **`in_img->out_img`**)

`mode`	in] copy mode:
	`0 =` copy **`in_img`** `box` **`-> out_img`**
	`1 =` copy **`in_img`** `->` **`out_img`** `box`

Return code:

None

ip_imgbox_to_img = Copy image box to image

Prototype:

```
#include “ip_lib.h”
void ip_imgbox_to_img (image *in_img, image *out_img, VM_BOX *box )
```

Description:

The function copies **`in_img`** rectangle area, defined by **`box`**, into **`out_img`**. In case of image/box dimensions mismatch, the function copies the source area into the upper left corner of the destination area eventually truncated.

Arguments:

Argument	Description
`in_img`	[in] source image
`out_img`	[out] destination image
`box`	[in] source image box (NULL: copy whole image **`in_img->out_img`**)

Return code:

None

ip_img_to_imgbox = Copy image to image box

Prototype:

```
#include “ip_lib.h”
void ip_img_to_imgbox (image *in_img, image *out_img, VM_BOX *box )
```

Description:

The function copies **in_img** into **out_img** rectangle area, defined by **box**. In case of image/box dimensions mismatch, the function copies the source area into the upper left corner of the destination area eventually truncated.

Arguments:

Argument	Description
in_img	[in] source image
out_img	[out] destination image
box	[in] destination image box (NULL: copy whole image **in_img->out_img**)

Return code:

None

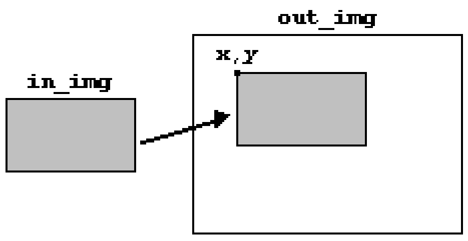
ip_copy_img = Copy image to given destination position

Prototype:

```
#include "ip_lib.h"
void ip_copy_img (image *in_img, image *out_img, int x, int x )
```

Description:

The function copies **in_img** at position **(x,y)** in the destination image **out_img**, where **(x,y)** defines the top/left image corner. If the source image goes beyond the destination image, it is truncated.



The **out_img** dimensions **dx** and **dy** are not changed.

Arguments:

Argument	Description
in_img	[in] source image
out_img	[out] destination image
x	x-coordinate of the top/left image corner in the destination image
y	y-coordinate of the top/left image corner in the destination image

Return code:

None

ip_img_pyramid = Image pyramid

Prototype:

```
#include "ip_lib.h"
void ip_img_pyramid ( VD_IMAGE *a, VD_IMAGE *b )
```

Description:

The function applies a pyramid filter on the input image **a** and stores the result image into **b**. The result image is 4-times smaller than the input one - it is shrunk 2 times horizontally and 2 times vertically. If the result image is too big to fit in **b**, it is truncated from right and bottom.

The operation can be done in place (**`a == b`**).

Arguments:

Argument	Description
a	[in] source image
b	[out] destination image

Return code:

None

ip_img_subsample = Image sub-sample

Prototype:

```
#include "ip_lib.h"
void ip_img_subsample ( VD_IMAGE *in_img, VD_IMAGE *out_img, int hor_step,
                        int ver_step )
```

Description:

The function sub-samples the input image **`in_img`** and stores the result image into **`out_img`**. The function copies pixels from each **`hor_step`**-th column and each **`ver_step`**-th row from the input image into the output image. Thus the result image is shrunk **`hor_step`** times horizontally and **`ver_step`** times vertically. If the result image is too big to fit in **`out_img`**, it is truncated from right and bottom.

The operation can be done in place (**`in_img == out_img`**).

Arguments:

Argument	Description
`in_img`	[in] source image
`out_img`	[out] destination image
`hor_step`	[in] horizontal sub-sampling step (>=1)
`ver_step`	[in] vertical sub-sampling step (>=1)

Return code:

None

ip_img_combine = Image combine

Prototype:

```
#include “ip_lib.h”
void ip_img_combine ( VD_IMAGE *img_a, VD_IMAGE *img_b, VD_IMAGE *img_c,
                      void (*func)(), int q )
```

Description:

The function combines the input images **img_a** and **img_b** and stores the result image into **img_c**. The nature of the "combine" operation is specified by the basic function **func()**. The **q** parameter is passed to the basic function (shift value for **ip_add2f**). In case of image dimensions mismatch, the function uses the image with the minimum dimensions. The result image is stored in the top/left area of **img_c**, eventually truncated. The **img_c** dimensions **dx** and **dy** are not changed.

The header file **IP_LIB.H** contains macros with different basic functions:

`ip_add2(a, b, c, q)`	= Sum images with shifting by **q** bits (**q>0:**shift right, **q<0:** shift left)
`ip_sub2(a, b, c)`	= Subtract images
`ip_max2(a, b, c)`	= Image max
`ip_min2(a, b, c)`	= Image min
`ip_and2(a, b, c)`	= Image logical AND
`ip_or2(a, b, c)`	= Image logical OR
`ip_xor2(a, b, c)`	= Image logical XOR
`ip_not2(a, b, c)`	= Image logical NOT

The operation can be done in place: **img_c** can be identical with one or both input images.

Arguments:

Argument	Description
img_a	[in] source image A
img_b	[in] source image B
img_c	[out] destination image C
func	[in] basic combination function
q	[in] **func** parameter (shift value)

Return code:

None

ip_set_img = Set image with constant

Prototype:

```
#include “ip_lib.h”
void ip_set_img ( VD_IMAGE *img, int val )
```

Description:
The function sets all pixels of **`img`** to **`val`**.

Arguments:

Argument	Description
`img`	[out] destination image
`val`	[in] pixel value to set

Return code:
None

ip_img_set = Set image with mask

Prototype:
```
#include “ip_lib.h”
void ip_img_set ( VD_IMAGE *img, VD_IMAGE *mask_img, int val )
```

Description:
The function sets some pixels in **`img`** as specified by the mask image **`mask_img`**:

- **`img`** pixels corresponding to non-zero pixels in **`mask_img`** are set to **`val`**
- the remaining **`img`** pixels are left unchanged

If the two images have different dimensions, then the function uses mask for the overlapping image pixels.

Arguments:

Argument	Description
`img`	[out] destination image
`mask_img`	[in] mask image (NULL: no mask, set all pixels)
`val`	[in] pixel value to set

Return code:
None

ip_clear_img = Clear image

Prototype:
```
#include “ip_lib.h”
void ip_clear_img ( VD_IMAGE *img )
```

Description:
The function sets all pixels of **`img`** to zero.

Arguments:

Argument	Description
`img`	[out] destination image

Return code:

None

ip_truncate_rect = Truncate image rectangle

Prototype:

```
#include “ip_lib.h”
int ip_truncate_rect (VD_IMAGE *img, int *x, int *y, int *dx, int *dy )
```

Description:

The function truncates the rectangle, defined by **`x`**, **`y`**, **`dx`** and **`dy`** inside the input image **`img`** and returns the parameters of the truncated rectangle in the same arguments.

Arguments:

Argument	Description
`img`	[in] image
`x`	[i/o] x-coordinate of top/left rectangle corner
`y`	[i/o] y-coordinate of top/left rectangle corner
`dx`	[i/o] rectangle width in pixels
`dy`	[i/o] rectangle height in pixels

Return code:

Return code	Description
`0`	Success
`1`	Error: Invalid rectangle parameters or the rectangle is entirely outside the image

ip_rect_binar = Binarization of image rectangle

Prototype:

```
#include “ip_lib.h”
int ip_rect_binar ( VD_IMAGE *img, int x, int y, int dx, int dy,
                    int lo_th, int up_th, int bgnd_clr, int obj_clr )
```

Description:

The function binarizes a rectangle area in the input/output image **img**, specified by **x**, **y**, **dx** and **dy**. The function truncates the input rectangle inside the image.

Algorithm:

The function changes pixel values according the following formula:

```
  if(lo_th <= pix <= up_th) then pix = obj_clr;  /* object area     */
                            else pix = bgnd_clr; /* background area */
```

When `up_th < lo_th,` the binarization formula is:

```
  if(pix <= up_th || pix >= lo_th) then pix = obj_clr;
                                   else pix = bgnd_clr;
```

The object area in the input/output image is filled with **obj_clr**. The background area in the input/output image is filled with color **bgnd_clr**. The two values are in the range [0,255] and should be different.

Arguments:

Argument	Description
img	[i/o] source and destination video image
x	[i/o] x-coordinate of top/left rectangle corner
y	[i/o] y-coordinate of top/left rectangle corner
dx	[i/o] rectangle width in pixels
dy	[i/o] rectangle height in pixels
lo_th	[in] lower binarization threshold
up_th	[in] upper binarization threshold
bgnd_clr	[in] color value, written into background pixels
obj_clr	[in] color value, written into object pixels (**bgnd_clr != obj_clr**)

Return code:

Return code	Description
0	Success
IP_AREA_OUTSIDE	Rectangle outside image
Other	Returned by called functions

ip_img_binar = Image binarization

Prototype:

```
#include “ip_lib.h”
int ip_img_binar ( VD_IMAGE *img, int lo_th, int up_th, int bgnd_clr,
                   int obj_clr )
```

Description:

The function binarizes in place the input/output image **`img`**, with thresholds **`lo_th`** and **`up_th`**. Pixels in the range **`[lo_th, up_th]`** represent the object area, pixels outside this range are background.

Algorithm:

The function changes pixel values according the following formula:

```
  if(lo_th <= pix <= up_th) then pix = obj_clr;  /* object area     */
                                 else pix = bgnd_clr; /* background area */
```

When `up_th < lo_th,` the binarization formula is:

```
  if(pix <= up_th || pix >= lo_th) then pix = obj_clr;
                                    else pix = bgnd_clr;
```

The object area in the input/output image is filled with **`obj_clr`**. The background area in the input/output image is filled with color **`bgnd_clr`**. The two values are in the range [0,255] and should be different.

Arguments:

Argument	Description
`img`	[i/o] source/destination video image
`lo_th`	[in] lower binarization threshold
`up_th`	[in] upper binarization threshold
`bgnd_clr`	[in] color value, written into background pixels
`obj_clr`	[in] color value, written into object pixels (**`bgnd_clr != obj_clr`**)

Return code:

Return code	Description
`0`	Success
Other	Returned by called functions

ip_binv = Figure binarization (basic function)

Prototype:

```
#include "ip_lib.h"
int ip_binv (VD_IMAGE *vd_img, SEG_DEF *seg_def, int nbr_dx, int nbr_dy,
             int x_step, int y_step, unsigned char dark_clr,
             unsigned char light_clr )
```

Description:

The function binarizes a segment-defined figure area in the input/output image **`img`** with variable dynamically calculated threshold (see "6.3.1. Segment definitions of figures").

Algorithm:

For each pixel `(x,y)` of the working area, the function finds the minimum and maximum pixel values `minv` and `maxv` in a rectangle with neighboring pixels, defined by:

```
   x,y    = rectangle center
   nbr_dx = rectangle width
   nbr_dy = rectangle height
```

The function binarizes current pixel with threshold `th`, equal to the middle pixel value:

```
   th = (minv + maxv) / 2
   if(pix <= th) then:  pix = dark_clr;
                 else:  pix = light_clr;
```

Choose **`nbr_dx`** and **`nbr_dy`** depending on the input picture so that at each position the rectangle contains both dark and light pixels, otherwise the function will give incorrect results. The **`x_step`** and **`y_step`** parameters are used to skip pixels when calculating the binarization threshold for each pixel, but the whole image figure is binarized without gaps.

Arguments:

Argument	Description
`vd_img`	[i/o] source/destination video image
`seg_def`	[in] figure segment-definition structure (seg_buf = NULL: NA)
`nbr_dx`	[in] width of neighboring pixel rectangle
`nbr_dy`	[in] height of neighboring pixel rectangle
`x_step`	[in] horizontal step to skip pixels in threshold calculation (1=all)
`y_step`	[in] vertical step to skip pixels in threshold calculation (1=all)
`dark_clr`	[in] dark binarization color
`light_clr`	[in] light binarization color

Return code:

Return code	Description
`0`	Success
`IP_INV_ARG`	Invalid function argument(s)
Other	Returned by called functions

ip_ellip_binv = Ellipse binarization with variable threshold

Prototype:

```
#include “ip_lib.h”
int ip_ellip_binv ( VD_IMAGE *vd_img, int x0, int y0,
                    unsigned int r1, unsigned int r2,
                    int nbr_dx, int nbr_dy, int x_step, int y_step,
                    unsigned char dark_clr, unsigned char light_clr )
```

Description:

The function binarizes an ellipse area in the input/output image **`img`** with variable dynamically calculated threshold.

Algorithm:

For each pixel `(x,y)` of the working area, the function finds the minimum and maximum pixel values `minv` and `maxv` in a rectangle with neighboring pixels, defined by:

```
x,y    = rectangle center
nbr_dx = rectangle width
nbr_dy = rectangle height
```

The function binarizes current pixel with threshold `th`, equal to the middle pixel value:

```
th = (minv + maxv) / 2
if(pix <= th) then:  pix = dark_clr;
              else:  pix = light_clr;
```

Choose **`nbr_dx`** and **`nbr_dy`** depending on the input picture so that at each position the rectangle contains both dark and light pixels, otherwise the function will give incorrect results. The **`x_step`** and **`y_step`** parameters are used to skip pixels when calculating the binarization threshold for each pixel, but the whole ellipse is binarized without gaps.

Arguments:

Argument	Description
`vd_img`	[i/o] source/destination video image
`x0`	[in] x-coordinate of ellipse center
`y0`	[in] y-coordinate of ellipse center
`r1`	[in] horizontal ellipse radius
`r2`	[in] vertical ellipse radius
`nbr_dx`	[in] width of neighboring pixel rectangle
`nbr_dy`	[in] height of neighboring pixel rectangle
`x_step`	[in] horizontal step to skip pixels in threshold calculation (1=all)
`y_step`	[in] vertical step to skip pixels in threshold calculation (1=all)
`dark_clr`	[in] dark binarization color
`light_clr`	[in] light binarization color

Return code:

Return code	Description
`0`	Success
Other	Returned by called functions

ip_rect_binv = Rectangle binarization with variable threshold

Prototype:

```
#include “ip_lib.h”
int ip_rect_binv ( VD_IMAGE *vd_img, int x, int y, int dx, int dy,
                   int nbr_dx, int nbr_dy, int x_step, int y_step,
                   unsigned char dark_clr, unsigned char light_clr )
```

Description:

The function binarizes non-rotated rectangle area in the input/output image **img** with variable dynamically calculated threshold.

Algorithm:

For each pixel `(x,y)` of the working area, the function finds the minimum and maximum pixel values `minv` and `maxv` in a rectangle with neighboring pixels, defined by:

```
x,y     = center of neighboring rectangle
nbr_dx  = width of neighboring rectangle
nbr_dy  = height of neighboring rectangle
```

The function binarizes current pixel with threshold `th`, equal to the middle pixel value:

```
th = (minv + maxv) / 2
if(pix <= th) then:  pix = dark_clr;
              else:  pix = light_clr;
```

Choose **nbr_dx** and **nbr_dy** depending on the input picture so that at each position the neighboring rectangle contains both dark and light pixels, otherwise the function will give incorrect results. The **x_step** and **y_step** parameters are used to skip pixels when calculating the binarization threshold for each pixel, but the whole rectangle area is binarized without gaps.

Arguments:

Argument	Description
vd_img	[i/o] source/destination video image
x	[in] x-coordinate of top/left rectangle corner
y	[in] y-coordinate of top/left rectangle corner
dx	[in] rectangle width in pixels
dy	[in] rectangle height in pixels
nbr_dx	[in] width of neighboring pixel rectangle
nbr_dy	[in] height of neighboring pixel rectangle
x_step	[in] horizontal step to skip pixels in threshold calculation (1=all)
y_step	[in] vertical step to skip pixels in threshold calculation (1=all)
dark_clr	[in] dark binarization color
light_clr	[in] light binarization color

Return code:

Return code	Description
0	Success
Other	Returned by called functions

ip_contrast = Image contrast

Prototype:

```
#include "ip_lib.h"
int ip_contrast ( image *img )
```

Description:

The function calculates the contrast value of the gray-level image **`img`**. The result value is in the range [0,128]. You can convert this value to percents in the range [0,100] by:

```
    contrast = (contrast * 100)/ 128;
```

Algorithm:

```
  mean     = SUM(pix) / n      = image mean value
  var      = SUM(pix*pix) / n  = image variance value
  contrast = SQRT(var - mean*mean);
where:
  n  = number of image pixels
```

Arguments:

Argument	Description
`img`	[in] source gray-level image

Return code:

Return code	Description
`>=0`	Contrast value in the range [0,128]
`-1`	Invalid input image

ip_img_mean = Image mean

Prototype:

```
#include “ip_lib.h”
int ip_img_mean ( image *img )
```

Description:

The function calculates the mean value of the gray-level image **`img`**.

Algorithm:

```
  mean = SUM(pixels) / n = image mean value
where:
  n  = number of image pixels
```

Arguments:

Argument	Description
`img`	[in] source gray-level image

Return code:

Return code	Description
`>=0`	Mean value of image pixels
`-1`	Invalid input image

ip_sphere_to_plain = Convert sphere to plain

Prototype:

```
#include “ip_lib.h”
int ip_sphere_to_plain ( image *in_img, int x0, int y0, int rad, int blank,
                         image *out_img, int *min_val, int *max_val )
```

Description:

The function converts a sphere image, received when the camera takes a picture with a ball at the screen center, to a plain image. Half of the sphere (as the camera sees a ball) has center **(x0,y0)** and radius **rad**, which must be calculated by the calling function. The sphere is unwrapped to circle in the plain with radius **PI/2*rad** and the result picture is stored in **out_img**.

The output image must be allocated by the calling function. Since the result circle has radius **PI/2*rad**, the generated plain picture has dimensions **dx=dy=PI*rad**. If **out_img** has smaller size, the result picture is copied truncated to the top/left **out_img** corner.

The transformation can be done in place, i.e. **in_img** == **out_img**.

Arguments:

Argument	Description
in_img	[in] source video image with sphere picture
x0	[in] x-coordinate of sphere center in **in_img**
y0	[in] y-coordinate of sphere center in **in_img**
rad	[in] sphere radius
blank	[in] blank pixel value, filled in the result image outside the plain circle
out_img	[out] destination image with converted to plain sphere picture
min_val	[out] minimum sphere pixel value (NULL: don't return)
max_val	[out] maximum sphere pixel value (NULL: don't return)

Return code:

Return code	Description
0	Success
IP_INV_ARG	Invalid argument(s) - sphere goes outside **in_img**
Other	Returned by called functions

ip_minma = Image min/max and histogram

Prototype:

```
#include “ip_lib.h”
int ip_minma (VD_IMAGE *vd_img, int x, int y, int dx, int dy, int x_step,
              int y_step, int *min_val, int *max_val, int *hist_buf )
```

Description:

The function finds the minimum and maximum pixel values in an image rectangle, specified by **(x,y)**, **dx** and **dy**. Optionally the function calculates the histogram of the rectangle pixels (**hist_buf != NULL**).

Arguments:

Argument	Description
vd_img	[in] source video image
x	[in] x-coordinate of top/left rectangle point
y	[in] y-coordinate of top/left rectangle point
dx	[in] rectangle width in pixels
dy	[in] rectangle height in pixels
x_step	[in] horizontal step to skip pixels (1=all)
y_step	[in] vertical step to skip pixels (1=all)
min_val	[out] min pixel brightness (NULL: don't return)
max_val	[out] max pixel brightness (NULL: don't return)
hist_buf	[out] histogram buffer with size 256 integers, allocated by the calling function (NULL: don't generate)

Return code:

Return code	Description
`0`	Success
`IP_INV_ARG`	Invalid function argument(s)
`IP_AREA_OUTSIDE`	Rectangle outside image

ip_img_scale = Scale image

Prototype:

```
#include “ip_lib.h”
int ip_img_scale (VD_IMAGE *in_img, VD_IMAGE *out_img, int dx, int dy,
                  int bw_mode, int bw_clr1, int bw_clr2, int calc_mode )
```

Description:

The function scales the input image **in_img** and stores the result image with dimensions **dx** and **dy** into **out_img**. Greater **dx**, **dy** dimensions, compared to the **in_img** dimensions, expand the input image. Smaller **dx**, **dy** dimensions shrink the input image. The function uses sub-pixel coordinate calculations to read the pixels of the input image.

The output image must be allocated by the calling function. If the output image has smaller dimensions than **dx*dy**, the result scaled picture is truncated in the top/left **out_img** corner.

The function scales the image by two methods:

- Using fast integer arithmetic with scaled*1000 integer pixel coordinates. This method gives 1/1000 pixel accuracy, which is quite sufficient in most practical cases.
- Using high-precision floating-point calculations (slower).

The scale can be done in place, i.e. **`in_img == out_img`**.

Arguments:

Argument	Description
`in_img`	[in] source image to scale
`out_img`	[out] destination scaled image
`dx`	[in] width of output scaled image
`dy`	[in] height of output scaled image
`bw_mode`	[in] black and white (B&W) image scale mode: `0` = scale 256-color gray-level input image `1` = scale 2-color B&W input image with colors **`bw_clr1`** and **`bw_clr2`**
`bw_clr1`	[in] B&W image color 1
`bw_clr2`	[in] B&W image color 2
`calc_mode`	[in] calculation mode: `0` = use fast integer calculations with 1/1000 pixel accuracy. `1` = use high-precision floating-point calculations (slower)

Return code:

Return code	Description
`0`	Success
`IP_ALLOC_ERROR`	Memory allocation error
`IP_INTERNAL_ERR`	Internal error

ip_img_rotate = Rotate image

Prototype:

```
#include “ip_lib.h”
int ip_img_rotate ( VD_IMAGE *in_img, VD_IMAGE *out_img,
                    float ang, int pix_fill,
                    int bw_mode, int bw_clr1, int bw_clr2 )
```

Description:

The function rotates the pixels in the input image **`in_img`** by angle **`ang`**, and stores the result rotated picture into **`out_img`**. The rotation angle is in radians in the range `[0,2*PI]` and increases in counter-clockwise direction. The function reads the pixels of the input image with sub-pixel accuracy. Pixels in the output image, which have no respective rotation pixels in the input image, are filled with value **`pix_fill`**.

The output image must be allocated by the calling function. The result rotated picture is stored (eventually truncated) to the upper left corner of the output image. You can use the function **ip_rot_image_size** to calculate the dimensions of the destination rotated image.

The rotation can be done in place, i.e. **`in_img == out_img`**.

***TIPS & TRICKS**. There is a faster image rotation function **sc_rot_img**, which uses integer rotation calculations.*

Arguments:

Argument	Description
in_img	[in] source image to rotate
out_img	[out] destination rotated image
ang	[in] rotation angle in radians `[0,2*PI]`
pix_fill	[in] pixel value to fill
bw_mode	[in] black and white (B&W) image rotation mode: `0` = rotate 256-color gray-level input image `1` = rotate 2-color B&W input image with colors **bw_clr1** and **bw_clr2**
bw_clr1	[in] B&W image color 1
bw_clr2	[in] B&W image color 2

Return code:

Return code	Description
0	Success
IP_ALLOC_ERROR	Memory allocation error
IP_INV_IMAGE	Invalid image

ip_rot_image_size = Get rotated image size

Prototype:

```
#include “ip_lib.h”
int ip_rot_image_size ( int dx, int dy, float ang, int *rot_dx,
                        int *rot_dy )
```

Description:

The function calculates the dimensions of the result rotated image, when a source image with dimensions **dx**, **dy** is rotated by angle **ang** in radians.

Algorithm

The result rotated image has dimensions **rot_dx**, **rot,dy**, calculated by the following formula:

```
  rot_dx = dy * abs(cos(PI/2 - ang)) + dx * abs(cos(ang));
  rot_dy = dy * abs(cos(ang))  + dx * abs(cos(PI/2 - ang));
```

Arguments:

Argument	Description
dx	[in] width of source image

dy	[in] height of source image
ang	[in] rotation angle in radians
rot_dx	[out] width of rotated image
rot_dy	[out] height of rotated image

Return code:

Return code	Description
`0`	Success
`IP_INV_ARG`	Invalid arguments

ip_img_bwcorr = Black and white (B&W) image correlation

Prototype:

```
#include “ip_lib.h”
int ip_img_bwcorr (VD_IMAGE *img1, VD_IMAGE *img2, VD_IMAGE *mask_img,
                   unsigned int *cor, unsigned int *area )
```

Description:

The function correlates the two input black and white (B&W) images **img1** and **img2** and returns correlation value and image area (max correlation value).

The function works in two modes:

- Non-mask mode (**mask_img** == NULL) : all pixels in the two images are correlated unconditionally
- Mask mode (**mask_img** != NULL) : two pixels are correlated only if the respective pixel in the mask image has non-zero value.

Image format:

Bit 0 of each image pixel specifies object/background area:

`0` = background pixel
`1` = object pixel

Algorithm:

Two respective pixels (with equivalent coordinates) in the two images are considered equivalent if bits 0 of the pixels are equivalent:

```
img1 pixel(bit 0)   img2 pixel(bit 0)
-----------------   -----------------
        0          ==         0
        1          ==         1
```

In ***mask*** mode the functions checks an additional condition for equivalent pixels – if the respective mask pixel is non-zero:

```
img1 pixel(bit 0)   img2 pixel(bit 0)    mask_img pixel
-----------------   -----------------    --------------
        0          ==         0         &&      != 0
        1          ==         1                 != 0
```

The function returns correlation value, equal to the number of equivalent pixels. If the two images have different dimensions, then the function returns the correlation value of the common overlapping image area. The function returns in area the total number of compared pixels.

***TIPS & TRICKS**. Calculate image matching rate in percents by the formula:*

```
rate (%) = cor*100/area;
```

Arguments:

Argument	Description
`img1`	[in] source B&W image
`img2`	[in] source B&W image
`mask_img`	[in] mask image (NULL: no mask)
`cor`	[out] calculated correlation value
`area`	[out] calculated image area (max correlation value = total # of correlated pixels)

Return code:

Return code	Description
`0`	Success
`IP_INV_ARG`	Invalid argument

ip_line_segm = Line segmentation (division into equal parts)

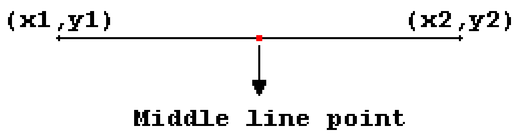
Prototype:

```
#include "ip_lib.h"
int ip_line_segm (float x1, float y1, float x2, float y2, int pt_cnt,
                  int **res_xy, float *line_len, float *seg_len )
```

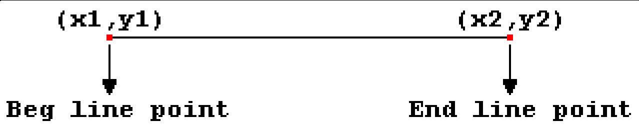
Description:

The function divides the line **`(x1,y1) (x2,y2)`** to N equal segments and returns the x/y coordinates of the segment-division points on the line. The **`pt_cnt`** input argument defines the number of result points:

`pt_cnt = 1:` return the middle line point:

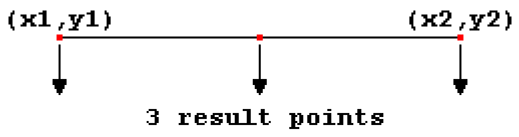


`pt_cnt = 2:` return the beginning and the end line points:

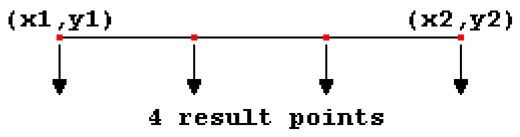


pt_cnt >= 3: divide the line by (**pt_cnt-1**) equal segments and return the division points, including the beginning and the end line points:

pt_cnt = 3:



pt_cnt = 4:



. . . . . .

The function calculates the x/y-coordinates of the result points with 1/1000 pixel accuracy. It allocates a result buffer for **pt_cnt** points and stores scaled*1000 integer coordinates in the result buffer in format:

`(x,y)  x,y) ... (x,y)   :` **pt_cnt** points (**pt_cnt*2** integers)

***ATTENTION**. The calling function must free the result buffer **res_xy** ! The function frees the allocated result buffer on error.*

The function also returns (if enabled) the total length of the line and length of the segment, which divides the line into equal parts.

Note:

This function fits better in the calculation and geometry library. Since the function calls the essential image-processing function **ip_read_pixel_line**, we decided to put this function into the image processing library to keep relatively independent VM_LIB sub-libraries. The calculation library however contains an alias name of this function, defined by the macro **cl_line_segm**, so you can use line segmentation in the calculation library as well.

Arguments:

Argument	Description
x1	[in] x-coordinate of beginning line point
y1	[in] y-coordinate of beginning line point
x2	[in] x-coordinate of end line point
y2	[in] y-coordinate of end line point
pt_cnt	[in] number of result line segment points
res_xy	[out] buffer with pixel coordinates
line_len	[out] line length (NULL: don't return)
seg_len	[out] segment length (NULL: don't return)

Return code:

Return code	Description

`0`	Success
`IP_ALLOC_ERROR`	Memory allocation error
`IP_INV_ARG`	Invalid function argument(s)
Other	Other errors returned by called functions

ip_rgb_to_hsv_convert = Convert RGB to HSV color scheme

Prototype:

```
#include “ip_lib.h”
int ip_rgb_to_hsv_convert ( image *img, int x, int y, int dx, int dy,
                            int prec, unsigned int hor_step,
                            unsigned int ver_step, int mode, int *res )
```

Description:

The function converts image in RGB format (Bayer matrix) to HSV format and vice versa. The conversion is done in place by overwriting the input image. The function works in a rectangle image area, specified by **x**, **y**, **dx** and **dy** (all should be even).

Currently the function works in non-interleaved Bayer matrix mode with horizontal and vertical color pixel increment >=1 (normal pixel increment >=2), which means each color pixel occupies two image rows and two columns at even addresses only.

RGB image format (GRBG):

```
                 +---+-------------- even column
                 | +-|-+------------ odd column
                 | | | |
                 0 1 2 3 ...
               +---------------------
even row    0  | G R G R G R G R . . .
odd  row    1  | B G B G B G B G . . .
even row    2  | G R G R G R G R . . .
odd  row    3  | B G B G B G B G . . .
          ...  | . . . . . . . . . . .
```

HSV image format:

```
                 +---+-------------- even column
                 | +-|-+------------ odd column
                 | | | |
                 0 1 2 3 ...
               +---------------------
even row    0  | V H V H V H V H . . .
odd  row    1  | S x S x S x S x . . .
even row    2  | V H V H V H V H . . .
odd  row    3  | S x S x S x S x . . .
          ...  | . . . . . . . . . . .

'x' byte format:
      7 6 5 4 3 2 1 0
  x = H H H H V V S S
      where: H H H H = 4 least-significant bits of the H byte
             V V     = 2 least-significant bits of the V byte
             S S     = 2 least-significant bits of the S byte
```

The **mode** argument (OR of bits) specifies Bayer matrix format of the input image, destructive mode and type of conversion:

```
bits 1,0 : Bayer matrix format: 0=GRBG, 1=RGGB, 2=GBRG, 3=BGGR
bit 2    : 0=destructive mode, 1=non-destructive mode
bit 3    : 0=RGB->HSV, 1=HSV->RGB
```

These 4 bits give total 16 operating modes:

mode	Operation mode
0	RGB (GRBG Bayer-matrix) to HSV, destructive mode
1	RGB (RGGB Bayer-matrix) to HSV, destructive mode
2	RGB (GBRG Bayer-matrix) to HSV, destructive mode
3	RGB (BGGR Bayer-matrix) to HSV, destructive mode
4	RGB (GRBG Bayer-matrix) to HSV, non-destructive mode
5	RGB (RGGB Bayer-matrix) to HSV, non-destructive mode
6	RGB (GBRG Bayer-matrix) to HSV, non-destructive mode
7	RGB (BGGR Bayer-matrix) to HSV, non-destructive mode
8	HSV to RGB (GRBG Bayer-matrix), destructive mode
9	HSV to RGB (RGGB Bayer-matrix), destructive mode
10	HSV to RGB (GBRG Bayer-matrix), destructive mode
11	HSV to RGB (BGGR Bayer-matrix), destructive mode
12	HSV to RGB (GRBG Bayer-matrix), non-destructive mode
13	HSV to RGB (RGGB Bayer-matrix), non-destructive mode
14	HSV to RGB (GBRG Bayer-matrix), non-destructive mode
15	HSV to RGB (BGGR Bayer-matrix), non-destructive mode

The function calculates and returns in the output **res** buffer min, max and mean pixel values for each color in the input and in the result image:

```
res[0]  = R_Min  = Minimum R value
res[1]  = R_Max  = Maximum R value
res[2]  = R_Mean = Mean (average) R value
res[3]  = G_Min  = Minimum G value
res[4]  = G_Max  = Maximum G value
res[5]  = G_Mean = Mean (average) G value
res[6]  = B_Min  = Minimum B value
res[7]  = B_Max  = Maximum B value
res[8]  = B_Mean = Mean (average) B value

res[9]  = H_Min  = Minimum H value
res[10] = H_Max  = Maximum H value
res[11] = H_Mean = Mean (average) H value
res[12] = S_Min  = Minimum S value
res[13] = S_Max  = Maximum S value
res[14] = S_Mean = Mean (average) S value
res[15] = V_Min  = Minimum V value
res[16] = V_Max  = Maximum V value
res[17] = V_Mean = Mean (average) V value
```

***INFORMATION**. The Hue/Saturation/Value (HSV) color scheme was created by A. R. Smith in 1978. It is based on such intuitive color characteristics as tint, shade and tone (or family, purity and intensity). The coordinate system is cylindrical, and the colors are defined inside a hex cone. The hue value H runs from 0 to 360º. The saturation S is the degree of strength or purity and is from 0 to 1. Purity is how much white is added to the color, so S=1 makes the purest color (no white). Brightness V also ranges from 0 to 1, where 0 is the black.*

Arguments:

Argument	Description
`img`	[i/o] source and destination image (modified in destructive mode)
`x`	[in] x-coordinate of top/left rectangle corner
`y`	[in] y-coordinate of top/left rectangle corner
`dx`	[in] rectangle width in pixels
`dy`	[in] rectangle height in pixels
`prec`	[in] calculation precision: `0` = set high 8-bits of H, S and V components in the result image **x** byte = 0: faster operation `1` = calculate **x** byte with low-order bits of H, S and V components in the result matrix – slower operation
`hor_step`	[in] horizontal step in color pixels (=>1): `1` : horizontal pixel increment = 2 (sequential color columns) `2` : horizontal pixel increment = 4 (skip 1 color column) `3` : horizontal pixel increment = 6 (skip 2 color columns) `...` : . . . .
`ver_step`	[in] vertical step in color pixels (=>1): `1` : vertical pixel increment = 2 (sequential color rows) `2` : vertical pixel increment = 4 (skip 1 color row) `3` : vertical pixel increment = 6 (skip 2 color rows) `...` : . . . .

`mode`	[in] mode of operation:
	`bits 1,0` : Bayer matrix format: `0=GRBG`, `1=RGGB`, `2=GBRG`, `3=BGGR`
	`bit 2` : `0`=destructive mode, `1`=non-destructive mode
	`bit 3` : `0=RGB->HSV`, `1=HSV->RGB`

`res`	[out] 18-element buffer with function results

Return code:

Return code	Description
`0`	Success
Other	Error

ip_perc_threshold = Percent threshold tool

Prototype:

```
#include “ip_lib.h”
```

```
int ip_perc_threshold ( VD_IMAGE *vd_img, DR_IMAGE *dr_img, int x, int y,
                        int dx, int dy, int pt_pos, int x_step, int y_step,
                        int perc, int mode,
                        int dr_mode, int dr_clr, int err_clr,
                        float *intensity, float *average,
                        int *min_val, int *max_val )
```

Description:

The tool finds the minimum and maximum pixel intensities (brightness) (`Imin, Imax`) within the working rectangle and calculates the following results:

intensity	=	Intermediate intensity (`I`) based on a given percentage (`P`)
average	=	Average brightness of the rectangle pixels (`A`)
min_val	=	Minimum pixel brightness
max_val	=	Maximum pixel brightness

The tool works in 3 operation modes as defined by **mode**:

`0 = MIN-MAX -` normal mode (use min and max intensities from the image)
`1 = 000-MAX -` assume `Imin = 0`
`2 = MIN-255 -` assume `Imax = 255`

Algorithm:

```
I = Imin + (Imax - Imin) * P / 100
```

where:

`I =` intensity result
`P =` percentage argument **perc**

The tool performs optional drawing of the working rectangle and the calculated results in the drawing image **dr_img**. The argument **dr_mode** specifies how to draw – always, never, on success or on error only. The arguments **dr_clr** and **err_clr** define the respective drawing colors. Pass NULL **dr_img** pointer or use **dr_mode=1** to bypass drawing.

Arguments:

Argument	Description
vd_img	[i/o] gray-level video image
dr_img	[i/o] drawing image (NULL: skip drawing)
x	[i/o] x-coordinate of working rectangle definition point
y	[i/o] y-coordinate of working rectangle definition point
dx	[i/o] working rectangle width in pixels
dy	[i/o] working rectangle height in pixels
pt_pos	[in] position of rectangle definition point **(x,y)**: `0 =` at rectangle center `1 =` at top left corner
x_step	[in] horizontal step to skip pixels (1=all)
y_step	[in] vertical step to skip pixels (1=all)
perc	[in] percentage value [0,100]. Use 50% as default value for edge-detection threshold (D->L or L->D).
mode	[in] tool operation mode: `0 = MIN-MAX -` normal (use min and max intensities from the image)

	`1 = 000-MAX - assume Imin = 0`
	`2 = MIN-255 - assume Imax = 255`

`dr_mode`	[in] drawing mode: `0 :` draw on (always) `1 :` draw off (never) `2 :` on success only `3 :` on error only
`dr_clr`	[in] drawing color on success (0=default: bright green)
`err_clr`	[in] drawing color on error (0=default: bright red)
`intensity`	[out] calculated intensity `(I)` (NULL: don't return)
`average`	[out] average (mean) pixel brightness `(A)` (NULL: don't return)
`min_val`	[out] min pixel brightness (NULL: don't return)
`max_val`	[out] max pixel brightness (NULL: don't return)

Return code:

Return code	Description
`VM_OK`	Success
`VM_INV_IMAGE`	Invalid drawing image
`VM_AREA_OUTSIDE`	Input rectangle is outside the image
Other	Returned by the called functions

ip_eros = Image erosion/dilation

Prototype:

```
#include "vm_lib.h"
int ip_eros ( VD_IMAGE *img, int x, int y, int dx, int dy,
              unsigned char *sbuf, int ssize, int oper )
```

Description:

The function performs one erosion or dilation pass on a rectangle area, defined by x, y, dx and dy, in a previously binarized image **`img`**. The binary image must have the following format:

```
  Object pixels     = 1
  Background pixels = 0
```

Currently the function implements erosion and dilation with a structuring element 3x3 only (**`ssize`** = 3). The **`sbuf`** buffer contains **`ssize`** x **`ssize`** matrix, the top row first (see the table below). The function destroys the rectangle area in **`img`**.

Algorithm:

The structural element is applied on all pixels in the working image area by placing the origin of the element (the center pixel) on each image pixel.

***Erosion*.**

An image pixel is set to OFF (background) if the 1's in the structural element do not completely overlap the ON image pixels in the 3x3 or 5x5 neighborhood (ON pixels = object pixels with color **`obj_clr`**).

***Dilation*.**

An OFF (background) image pixel is set to ON (object pixel) if any of the 1's in the structural element overlaps an ON image pixel in the 3x3 or 5x5 neighborhood.

Typical structuring elements:

`selm`	Structural element	Description

0	**3x3 diamond:**
	`0 1 0`
	`1 1 1`
	`0 1 0`
		Shrinks or expands 4-connected pixels

1	**3x3 box:** `1 1 1` `1 1 1` `1 1 1`	Shrinks or expands 8-connected pixels (default)

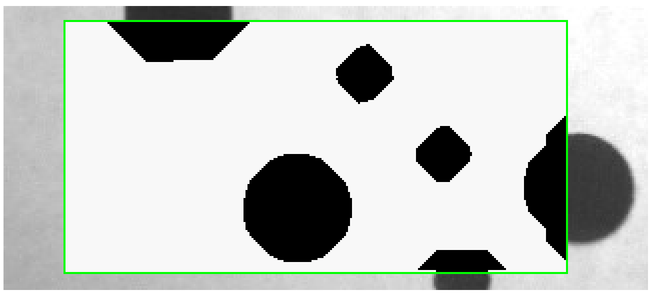
2	**3x3 horizontal:**
	`0 0 0`
	`1 0 1`
	`0 0 0`
		Shrinks or expands objects in horizontal direction

3	**3x3 vertical:**
	`0 1 0`
	`0 0 0`
	`0 1 0`
		Shrinks or expands objects in vertical direction

The function supports the following morphological operations on binarized image:

- **Erosion.** The erosion operation shrinks objects by 1 pixel. This operation removes object boundary pixels and enlarges inner object holes. Objects smaller than the structuring element disappear. Use this operation to remove "salt and pepper" noise outside meaningful objects.
- **Dilation.** The dilation operation expands objects by 1 pixel. This operation adds pixels at object boundaries and fills inner object holes. Use this operation to delete "salt and pepper" noise inside meaningful objects (for example to fill small object holes).
- **Opening.** The opening operation performs **`steps`** erosions followed by **`steps`** dilations. This operation separates objects - deletes noise and thin lines, which connect bigger objects, and preserves object sizes.
- **Closing.** The closing operation performs **`steps`** dilations followed by **`steps`** erosions. This operation fills small holes in objects and preserves object sizes.

Dilation (object expand) example:



Arguments:

Argument	Description
`img`	[i/o] binary image in format: 0=bgnd pixels, 1=object pixels
`x`	[in] x-coordinate of top/left rectangle corner
`y`	[in] y-coordinate of top/left rectangle corner
`dx`	[in] working rectangle width in pixels

`dy`	[in] working rectangle height in pixels
`sbuf`	[in] buffer with structuring element (usually 3x3 or 5x5)
`ssize`	[in] size of structuring element: 3 or 5 (currently ignored, size 3 used only)
`oper`	[in] operation mode: `0 :` erosion `1 :` dilation

Return code:

Return code	Description
`0`	Success
`IP_ALLOC_ERROR`	Memory allocation error
Other	Returned by the called functions

5.7. Normalized correlation (pattern matching)

This section describes the normalized correlation library functions.

ncor = Normalized correlation

Prototype:

```
#include "ncor.h"
int ncor ( image *pat_img, image *srh_img, short mode, short exact,
           short th, short *rate, short *res_x, short *res_y )
```

Description:

This is the root normalized-correlation function, which searches a pattern image in a search image. The function finds one instance of the pattern with the best matching rate. The function returns position and rate of the detected pattern with recognition rate above the threshold **`th`**. The returned position **`(res_x, res_y)`** is relative to the upper left corner of the search image. Result **`rate = -1`** : pattern not found.

The **`mode`** argument specifies 3 search modes – fast, normal and fine. The best execution speed is reached in fast mode. The **`exact`** argument specifies exact result searching. When **`exact`** `= 0` the function returns always even result coordinates (faster, old mode before library version 3.10). When **`exact`** `= 1` the function returns exact odd/even result coordinates. Remember that stable exact results are received in fine mode only.

Arguments:

Argument	Description
`pat_img`	Input pattern image
`srh_img`	Input search image
`mode`	Input search mode: -1=fast, 0=normal, 1-fine
`exact`	Input exact searching mode: `0 :` searching of results with even coordinates only (old faster mode) `1 :` exact searching of results with odd/even coordinates Applies to all search modes (fast/normal/fine), but stable exact results are received in fine mode only.
`th`	Input rate threshold value in the range [512,1023]: search pattern with matching rate above **`th`**.
`rate`	Output matching rate in the range [512,1024] (-1 : pattern not found)
`res_x`	Output x-coordinate of top/left corner of found pattern, relative to `srh_img`
`res_y`	Output y-coordinate of top/left corner of found pattern, relative to `srh_img`

Return code:

Return code	Description
`0`	Success
Other	Error

nc_get_time = Get system time

Prototype:

```
#include "ncor.h"
unsigned long nc_get_time ()
```

Description:

The function returns the system time in milliseconds. The time grows from 0 upwards.

Arguments:

None

Return code:

The system time in ms.

5.8. Object recognition library

This section describes the object recognition library functions. The purpose of the library is 2-D object (pattern) recognition. The pattern must be learned from a pattern image. Once you have learned a pattern, you can perform multiple recognitions in different search images. The searching results are sorted in descending order in respect to the recognition rate. Each detected object is described by the following parameters:

- Result number = number of detected object (lower result number : higher recognition rate)
- Coordinates of the rotation and scale object center, relative to the search image. Coordinates of the 4 corners of the rotated rectangle, enclosing the found object.
- Recognition rate (above a user-defined threshold)
- Rotation angle (in allowed range)
- Scale factor (in allowed range)

Algorithm

The object recognition algorithm contains the following principal processing steps:

edge detection
contour tracking
extraction of basic pattern points
pattern recognition

5.8.1. Simple example

The next example demonstrates how to use the OR library. Although the next chapters in the manual describe the library in details, this example could be useful to give a basic idea about it. The example opens the library by **`or_init`**, learns a pattern by **`or_learn`**, searches the pattern in the search image by **`or_search`** and finally close the library by **`or_exit`**.

```
/* Simple demo program for the object recognition library */
#include "vm_lib.h"

void main()
{
    int     rc = 0;
    OR_HND  or_hnd;         /* OR library handle            */
    image   ovl_img;        /* overlay image                */
    image   pat_img;        /* pattern image                */
    image   srh_img;        /* search image                 */
    OR_RES  res_buf[10];    /* result buffer                */
    int     nres;           /* number of detected results   */

    LEARN_PAR  lpar = {50,12,8,0,0,1};                          /* learn parms  */
    SEARCH_PAR spar = {50,12,8,0,1,552,5,1024,1024,-5,360,0,1}; /* search parms */

    printf("OR library - simple demo program\n");

/* Initialize images: ovl_img = overlay page; pat_img,srh_img = video page */
/* . . . . . . . . . . */

/* Init library handle */
    or_init(&or_hnd);

/* Learn pattern at x,y=100,100, dx,dy=80,80*/
    rc = or_learn(&or_hnd, &pat_img, &ovl_img, NULL, 100, 100, 80, 80, &lpar);
    if(rc) goto done;

/* Search pattern in search area at x,y=10,10, dx,dy=500,400 */
    rc = or_search(&or_hnd, &srh_img, &ovl_img, 10, 10, 500, 400, &spar,
                   res_buf, &nres);
    if(rc) goto done;

/* Close library and exit */
done:
    or_exit(&or_hnd);
    printf("RC=%d\n",rc);

    return;
}
```

5.8.2. Program structure

An object recognition program must have the following structure (see also "*6.8.1. Simple example*"):

- Call **`or_init`** to reset the library handle (1 handle for each pattern).
- Learn stage - setup a ***pattern*** image and teach a pattern by the **`or_learn`** function. Call this function once for a given pattern. The pattern features are stored into the handle.
- Search stage – setup a ***search*** image and search the learned pattern in this image by the **`or_search`** function. You can do multiple recognitions of the learned pattern by multiple calls of this function with the same handle and with different input search images.
- Close the handle by **`or_exit`**. Don't forget to call this function on program exit, because it frees all allocated handle memory.

5.8.3. Input parameters

The "learn" and "search" input parameters (edge-detection thresholds, recognition rate, rotation and scale ranges, etc) are passed to the respective functions by the **`LEARN_PAR`** and the **`SEARCH_PAR`** structures, defined in the header file **`or_lib.h`**. Read this section to learn more about these parameters and how to control the learn/search execution stages.

LEARN_PAR member	Description
`min_edge_h`	Minimum edge height: [7,127], recommended value = 50
`min_edge_g`	Min edge gradient: [6, **min_edge_h**], recommended value = 12
`min_chain`	Min edge contour length: [4,24] pixels
`save_edge_image`	Show contours in the pattern image, stop further processing (diagnostic mode).
`lrn_mode`	Learn mode: `0:` normal learn mode (the edge image **`edg_img`** is ignored) `1:` generate output edge image in **`edg_img`** `2:` learn pattern and use the input edge image **`edg_img`** to mask (discard) the unwanted edges in the pattern image
`dr_clr`	Drawing color [0,255] – value written into the overlay image

SEARCH_PAR member	Description
`min_edge_h`	Minimum edge height: [7,127], recommended value = 50
`min_edge_g`	Min edge gradient: [6, **min_edge_h**], recommended value = 12
`min_chain`	Min edge contour length: [4,24] pixels
`restr_search`	Set restricted search mode

`max_pm_items`	Max number of searched objects (patterns)
`min_rate`	Min recognition rate: 160,..,1023, 1024==100%
`contour_width`	Min edge contour width: 3,5,7,9 pixels
`min_scale`	Min object scale: 512-1024(=100%), keep <= `max_scale`
`max_scale`	Max object scale: 1024(=100%)-1536, keep <200%
`rot_sect_start`	Start rotation angle: -180 to 179 degrees
`rot_sect_width`	Width of rotation sector (range of allowed rotations): 0 to 360 deg
`save_edge_image`	Show contours in the search image, stop further processing (diagnostic mode).
`dr_clr`	Drawing color [0,255] – value written into the overlay image

To achieve good recognition results you need to perform fine-tuning of the “learn” and the “search” parameters. The edge-detection parameters **min_edge_h** and **min_edge_g**, along with appropriate lighting, should be adjusted to achieve clear and stable detection of meaningful contours (edges), as well as to remove noise and clutter. The **min_chain** parameter should be adjusted to remove redundant short contours. Best recognition results are received when the edge contours are longer than 24 pixels.

The number of searched objects is limited by the argument **max_pm_items**. This parameter also limits the searching phase in restricted searching mode, which allows faster recognition times. When this mode is enabled by **restr_search**=1, the search function **`or_search`** stops processing when the first **max_pm_items** results with recognition rates above **min_rate** are detected. Turn off the restricted search to find the best **max_pm_items** results. The restricted searching could be used to improve the recognition times after the achieving of stable recognition results in **restr_search**=0 mode.

The angle parameters **rot_sector_start** and **rot_sector_width** are presented in degrees. The “percent” parameters **min_rate**, **min_scale** and **max_scale** are in units of 1/1024th (e.g. rate 1024 == 100%, scale 1126 == 110%).

In case of recognition fail, decreasing of **min_rate** and/or increasing of **contour_width** may help. Internally the library works with odd values of the contour width. In general, **contour_width** should have small values in case of small/fine patterns or when you try to achieve more precise recognition rates.

The **`or_nsearch()`** function accepts normalized parameters with the following meaning:

`min_rate`	=	min matching rate in percents = [15,100] %
`min_scale`	=	min scale in percents = [50,100] %
`max_scale`	=	max scale in percents = [100,150] %
`rot_sect_start`	=	start rotation angle = [0,360] degrees

5.8.4. Checking detected contours

Finding of good object contours (edges) is of vital importance for the successful object recognition – both in the “learn” and in the “search” stage. You can enter a diagnostic mode of operation and see how edges are detected by setting the parameter **save_edge_image** to 1.

This mode is intended to help you in finding better image edges. Run respective learn/search function with different edge-detection parameters, lights and focus until you observe the best object contours without too much noise (redundant contours). The “learn” and “search” functions **`or_learn`** and **`or_search`** stop further processing after the edge-detection stage and fill the video image with contour data.

Remember that this mode destroys the source image. For example you can't search objects when this option is enabled in the learn stage.

5.8.5. Edge masking in the learn stage

Better recognition results and faster execution times can be achieved by the so called edge-masking in the learn stage. The basic idea is to discard unwanted edge points in the learn image and then search the pattern with meaningful contours only. This can be done by the edge-mask image **`edg_img`** in learn modes 1 and 2. Perform the following steps to generate and apply edge-mask image on the learn pattern:

1. Generate edge-mask image by calling the **`or_learn()`** function in learn mode 1:

```
LEARN_PAR lpar;
image pat_img;
image edg_img;
. . . . . . . . .
lpar.lrn_mode = 1;      /* generate edge-mask image in edg_img */
or_learn(&or_hnd, &pat_img, &ovl_img, &edg_img, ..., &lpar);
```

2. Edit the edge-mask image **`edg_img`** and set unwanted edge pixels to zero. This editing is beyond the scope of this library and therefore an editing program is not supplied. It could be done for example by a Windows program, which displays the edge image **`edg_img`** and allows interactive erasing of unwanted edge points and contours lines by the mouse.
3. Learn the same pattern image **`pat_img`**, which was used to create **`edg_img`** in step 1. Call **`or_learn()`** in learn mode 2. The masked edge points will not be learned and used later in the search stage:

```
lpar.lrn_mode = 2;      /* mask unwanted edge points in pat_img */
or_learn(&or_hnd, &pat_img, &ovl_img, &edg_img, ..., &lpar);
```

5.8.6. Improving performance

To achieve better execution times the scale range [**min_scale**, **max_scale**] and **rot_sect_width** (size of rotation angle range) should be kept as narrow as possible. The **rot_sect_width** parameter should be used together with **rot_sect_start**, which specifies beginning angle of rotation range (sector) in clockwise direction (positive values only).

In case of good recognition results but bad execution speed, you may reduce the execution time by setting "**restr_search**"=1. In this mode both the detection ability and the searching time are proportional to **max_pm_items** and depend heavily on **min_rate** and **contour_width**. Set **min_rate** about 10% below the estimated rate. Set **max_pm_items** parameter to be 2-10 times greater than the number of real objects in the search image. After achieving stable recognition results with relatively short searching time you may check system behavior by slightly varying **contour_width** (and possibly **min_rate**).

Remember that more detected edges in the search image increase the recognition time.

5.8.7. Searching multiple patterns

If you want to perform recognition of several learned in advance patterns, you should declare and initialize several **`OR_HND`** handles by **`or_init`**. Learn the patterns once by passing to **`or_learn`** different pattern images and handles. Call **`or_search`** with different handles to search different patterns. Finally call **`or_exit`** for each one of the initialized handles.

5.8.8. Limitations

The minimum size of the pattern is 12x12 pixels. The minimum size of the search area is 32x32 pixels. The maximum pattern/search area sizes are limited by the respective images (the size of the camera's video page).

or_init = Init OR handle

Prototype:

```
#include "or_lib.h"
void or_init ( OR_HND *or_hnd )
```

Description:

The function initializes (resets) the library handle **or_hnd**. It should be called once for each used handle (1 handle for each searched pattern).

Arguments:

Argument	Description
or_hnd	Output OR library handle

Return code:

None

or_exit = Close OR handle

Prototype:

```
#include "or_lib.h"
void or_exit ( OR_HND *or_hnd )
```

Description:

The function closes OR library handle and frees all allocated memory for this handle.

Arguments:

Argument	Description
or_hnd	Input/output OR library handle

Return code:

None

or_learn = Learn pattern

Prototype:

```
#include "or_lib.h"
int or_learn (OR_HND *or_hnd, image *pat_img, image *ovl_img,
              image *edg_img, int x, int y, int dx, int dy,
            LEARN_PAR *lpar )
```

Description:

The function executes an object recognition learn stage. It learns the pattern from the input learn image, defined by **x**, **y**, **dx** and **dy** and stores description (features) of the learned pattern into the handle **or_hnd**. The pattern rectangle **x**, **y**, **dx**, **dy** is truncated inside the pattern image. The overlay drawing is disabled by `ovl_img == NULL`.

ATTENTION. *The pattern image **pat_img**, used by **or_learn()** and the search image **srh_img**, used by **or_search()** and **or_nsearch()**, must have equivalent dimensions **dx**, **dy** and **pitch**.*

The function generates and/or applies a mask image **edg_img**, which discards unwanted contour points in the pattern image (see "2.4.1. Edge masking in the learn stage").

Arguments:

Argument	Description
or_hnd	Input/output OR library handle
pat_img	Input pattern image (video page)
ovl_img	Input drawing image (overlay page), NULL: disable result drawing
edg_img	Input/output edge-mask image (NULL=ignore edge image, same as lrn_mode=0): `lpar->lrn_mode = 0:` learn pattern in normal mode without edge mask `lpar->lrn_mode = 1:` generate output edge image in `edg_img` and learn pattern `lpar->lrn_mode = 2:` learn pattern and use the input edge image in `edg_img` to mask (discard) the unwanted edge points in the pattern image **ATTENTION: The edge image must have the dimensions dx, dy of the pattern rectangle.** Format of `edg_img`: `pixel = 0 :` mask (discard) respective pattern edge point `pixel > 0 :` use respective pattern edge point in the search stage
x	Input x-coordinate of top/left pattern corner, relative to `pat_img` (must be **even**)
y	Input y-coordinate of top/left pattern corner, relative to `pat_img`
dx	Input pattern width in pixels: from 32 to image `dx` (must be **even**)
dy	Input pattern height in pixels: from 32 to image `dy`
lpar	Input learn parameters (see "2.3. Input parameters" and OR_LIB.H)

Return code:

Return code	Description
`0`	Success
`Other`	Error

or_search = Search pattern

Prototype:

```
#include "or_lib.h"
int or_search (OR_HND *or_hnd, image *srh_img, image *ovl_img,
               int x, int y, int dx, int dy, SEARCH_PAR *spar,
               OR_RES *res_buf, int *nres )
```

Description:

The function searches a learned pattern in a rectangle area of the input search image, defined by **x**, **y**, **dx** and **dy**. The input handle **or_hnd** must be initialized with pattern features by previous call to **or_learn**. The search rectangle **x**, **y**, **dx**, **dy** is truncated inside the search image. The overlay drawing is disabled by `ovl_img == NULL`. The search and the overlay images must have equal pitch..

***ATTENTION**. The pattern image **pat_img**, used by **or_learn()** and the search image **srh_img**, used by **or_search()** and **or_nsearch()**, must have equivalent dimensions **dx**, **dy** and **pitch**.*

The function stores the parameters of detected patterns into the result buffer **res_buf** : an array of **OR_RES** structures with the following members:

x	=	X-coordinate of contour rotation center of detected object, relative to the search image
y	=	Y-coordinate of contour rotation center of detected object, relative to the search image
rate	=	Recognition rate: 1024==100%
rot	=	Detected rotation angle: -180..179 deg
scale	=	Detected scale factor, 1024==100%
corners	=	8-element buffer with X/Y-coordinates of the 4 corner points of the rotated rectangle, which encloses the detected object: `(x0,y0) (x1,y1) (x2,y2) (x3,y3)` The first point `(x0,y0)` corresponds to the top/left corner of the non-rotated pattern rectangle. The points in `corners[]` are ordered in clockwise direction. The coordinates are relative to the search image.

Arguments:

Argument	Description
or_hnd	Input/output OR library handle
srh_img	Input search image (video page)
ovl_img	Input drawing image (overlay page), NULL: disable result drawing
x	Input x-coordinate of top/left search area corner, relative to `srh_img` (**even**)
y	Input y-coordinate of top/left search area corner, relative to `srh_img`
dx	Input search area width in pixels: from 32 to image `dx` (must be **even**)
dy	Input search area height in pixels: from 32 to image `dy`
spar	Input search parameters (see "2.3. Input parameters" and OR_LIB.H)
res_buf	Output result buffer (array of OR_RES structures), allocated by the calling function with minimum size: **spar->max_pm_items**
nres	Output number of detected results (# of elements stored in **res_buf**)

Return code:

Return code	Description
`0`	Success
`Other`	Error

or_nsearch = Normalized search

Prototype:

```
#include “or_lib.h”
int or_nsearch (OR_HND *or_hnd, image *srh_img, image *ovl_img,
                int x, int y, int dx, int dy, SEARCH_PAR *spar,
                OR_RES *res_buf, int *nres )
```

Description:

The function performs normalized search of the learned pattern in the search image and stores the parameters of the detected patterns into `res_buf`. The difference between this function and `or_search()` is that `or_nsearch()` accepts normalized input parameters and returns normalized results:

***ATTENTION**. The pattern image **`pat_img`**, used by **`or_learn()`** and the search image **`srh_img`**, used by **`or_search()`** and **`or_nsearch()`**, must have equivalent dimensions **dx**, **dy** and **pitch**.*

The following input SEARCH_PAR parameters have different meaning:

`min_rate`	=	min matching rate in percents = [15,100] %
`min_scale`	=	min scale in percents = [50,100] %
`max_scale`	=	max scale in percents = [100,150] %
`rot_sect_start`	=	start rotation angle = [0,360] degrees

The following OR_RES result parameters have different meaning:

`rate`	=	recognition rate in percents = [15,100] %
`rot`	=	rotation angle = [0,360] degrees (clockwise)
`scale`	=	scale factor in percents = [50,150] %

The detected angle increases in clockwise direction

The two images **`srh_img`** and **`ovl_img`** must have equal pitch.

Arguments:

Argument	Description
`or_hnd`	Input/output OR library handle
`srh_img`	Input search image (video page)
`ovl_img`	Input drawing image (overlay page), NULL: disable result drawing
`x`	Input x-coordinate of top/left search area corner, relative to `srh_img` (**even**)
`y`	Input y-coordinate of top/left search area corner, relative to `srh_img`

`dx`	Input search area width in pixels: from 32 to image `dx` (must be **even**)
`dy`	Input search area height in pixels: from 32 to image `dy`
`spar`	Input search parameters (see “2.3. Input parameters” and OR_LIB.H)
`res_buf`	Output result buffer (array of OR_RES structures), allocated by the calling function with minimum size: **`spar->max_pm_items`**
`nres`	Output number of detected results (# of elements stored in **`res_buf`**). Use value: **`nres <= spar->max_pm_items`**

Return code:

Return code	Description
`0`	Success
`Other`	Error

or_get_time = Get system time

Prototype:

```
#include “or_lib.h”
unsigned long or_get_time ()
```

Description:

The function returns the system time in milliseconds. The time grows from 0 upwards.

Arguments:

None

Return code:

The system time in ms.

5.9. Utility library

This library contains miscellaneous functions for string processing, parsing of script text files and other general-purpose utilities.

Header files:

`ut_lib.h`	Utility library header

ut_binstr_to_int = Convert binary string to integers

Prototype:

```
#include “ut_lib.h”
void ut_binstr_to_int (char *str, int dig_cnt, int *ibuf )
```

Description:

The function converts the input binary string **`str`** containing '0' and '1' chars into bits:

```
     '0' -> bit 0
   !='0' -> bit 1
```

and stores the results into the output 32-bit integer buffer **`ibuf`**. The first 32 **`str`** chars 0-31 are converted from left to right and stored into bits 0-31 (LSB to MSB) of the first **`ibuf`** element. Next 32 **`str`** chars 32-63 are converted from left to right and stored correspondingly into bits 0-31 of the second output element and so on, until all chars of **`str`** are processed. Last MSB bits of the last **`ibuf`** element, which have no corresponding **`str`** chars to convert, are set to zero.

The input **`str`** buffer contains maximum **`dig_cnt`** chars or less if the 0-terminating char comes first. The output integer buffer **`ibuf`** should have minimum length of **`(dig_cnt+31)/32`** elements.

Example:

```
dig_cnt = 67:
             |0000      33|3333      66|666|
char pos   = |0123......01|2345......23|456|
             +------------+------------+---+
[in] str   = "abcd......ef|ghij......kl|mno"
             +-----32-----+-----32-----+-3-+

             |33      0000|33      0000|33      0000|
bit pos    = |10......3210|10......3210|10......3210|
             +------------+------------+------------+
[out] ibuf = |fe......dcba|lk......jihg|00......0onm|
             +------------+------------+------------+
                                        |       |
                                        |<----->|
                                       set to zero
```

Arguments:

Argument	Description
`str`	[in] string with **`dig_cnt`** binary digits (may be 0-terminated or not)
`dig_cnt`	[in] max number of binary digits to convert
`ibuf`	[out] 32-bit integer buffer

Return code:

None

ut_int_to_binstr = Convert integers to binary string

Prototype:

```
#include “ut_lib.h”
void ut_int_to_binstr (char *str, int dig_cnt, int *ibuf )
```

Description:

The function converts the first **dig_cnt** bits in the input integer buffer **ibuf** into binary string with '0's and '1's. Bits 0-31 (LSB to MSB) of the first **ibuf** element are converted into chars 0-31 of the output string **str**. Bits 0-31 (LSB to MSB) of the 2nd **ibuf** element are converted into chars 32-63 of the output string and so on until **dig_cnt** bits from **ibuf** are converted to chars and stored in **str**.

The function stores exactly **dig_cnt** chars into the output **str** buffer without a 0-term char. The function reads **(dig_cnt+31)/32** 32-bit elements from the input integer buffer **ibuf**.

Example:

```
dig_cnt = 67:
             |0000      33|3333      66|666|
char pos   = |0123......01|2345......23|456|
             +------------+------------+---+
[out] str  = "abcd......ef|ghij......kl|mno"
             +-----32-----+-----32-----+-3-+

             |33      0000|33      0000|33      0000|
bit pos    = |10......3210|10......3210|10......3210|
             +------------+------------+------------+
[in] ibuf  = |fe......dcba|lk......jihg|.........onm|
             +------------+------------+------------+
                                       |       |
                                       |<----->|
                                         unused
```

Arguments:

Argument	Description
str	[out] string with binary digits ‘0’s and ‘1’s
dig_cnt	[in] number of binary digits to convert and store in **str**
ibuf	[in] 32-bit integer buffer

Return code:

None

ut_strtok = Get string token

Prototype:

```
#include “ut_lib.h”
char *ut_strtok (char *str, const char *brks, char *b )
```

Description:

The function extracts next token (sub-string) from **`str`**, delimited by a break character from the **`brks`** 0-terminated string. This is a modified **strtok** function, which returns also the detected break char in **b**.

Attention: This function destroys the input string.

Usage (identical with **strtok**):

```
   char *tok;
   char *brks=" ,.";  /* break chars = space, comma and dot */
   char  b;
   . . . . . . . . .
   tok = ut_strtok(str, brks, &b);     // get 1st token
   tok = ut_strtok(NULL, brks, &b);    // get 2nd token
   tok = ut_strtok(NULL, brks, &b);    // get 3rd token
   . . . . . . . . . . . . . . . .
```

Example:

Input arguments:

```
  str = “  AAA = .  =bb= c123= xxx.  =  “
  brks = “ =.”
```

Results:

```
  tok=AAA, b=' '
  tok=bb, b='='
  tok=c123, b='='
  tok=xxx, b='.'
  tok=NULL: end of string
```

TIPS & TRICKS. *This function is an alternative of the **ut_get_token** function, which works with more complex interface. The difference between the two functions is:*

ut_get_token	-	*returns empty tokens with valid break characters (if any)*
ut_strtok	-	*returns only non-blank tokens, skips successive break characters without valid tokens between and destroys the input string*

Arguments:

Argument	Description
`str`	[in] 0-terminated string which should be parsed
`brks`	[in] 0-terminated string with break characters
b	[out] the break character, which terminates the extracted token (not included in token)

Return code:

Return code	Description
`!=NULL`	Pointer to next extracted token
`NULL`	Token not found, end of input string

ut_del_comments = Delete comments in text line

Prototype:

```
#include “ut_lib.h”
void ut_del_comments (char *str, char cchar )
```

Description:

The function deletes comments (from a comment character to the end of the text line) in the input/output string **`str`**. Comments are not deleted if the comment character is inside a text, enclosed by double quotation marks `"........",` which defines literal constant.

Arguments:

Argument	Description
`str`	[i/o] text line
`cchar`	[in] comment character

Return code:

None

ut_del_blanks = Delete text blanks

Prototype:

```
#include “ut_lib.h”
void ut_del_blanks ( char *str, char *brks, int mode )
```

Description:

The function deletes redundant blanks in the input/output string **`str`**. A blank is deleted from **`str`**, when the following combination of two neighboring characters occurs:

```
blank      blank
break_char blank
blank      break_char
```

Blanks are not deleted inside a text, enclosed by double quotation marks `"........",` which defines literal constant. The output string optionally can be converted to uppercase letters.

The purpose of this function is to put the delimiter break characters just behind the valid tokens. Thus parsing of the text lines and the detection of lexical errors becomes much easier.

Arguments:

Argument	Description
`str`	[i/o] source/destination string where redundant blanks are deleted
`brks`	[in] 0-terminated string with break characters
`mode`	[in] mode of operation: `0 :` do not up-case the result string `1 :` up-case the result string

Return code:
None

ut_get_token = Get string token

Prototype:

```
#include “ut_lib.h”
int ut_get_token (char *str, int *ipos, char *brks, char *token,
                  char *brk )
```

Description:

The function extracts a substring from **`str`** starting from position **`ipos`**, stores extracted substring into **`token`** and updates **`ipos`** for next search. There is no check for **`token`** buffer overflow; it should be large enough to hold any extracted substring. The token substrings are separated by one or more blanks or other break characters, defined in **`brks`**. A blank should not be included in the **`brks`** string.

Note that output token string may be empty string "" when two adjacent non-blank break characters occur in **`str`**.

Usage:

```
    char tok[128];
    int ipos;
    char *brks=".,=";  /* break chars */
    char  brk;
    . . . . . . . . .
    ipos = 0;
    ut_get_token(str, &ipos, brks, tok, &brk);   // get 1st token
    ut_get_token(str, &ipos, brks, tok, &brk);   // get 2nd token
    ut_get_token(str, &ipos, brks, tok, &brk);   // get 3rd token
    . . . . . . . . . . . . . . . . . . . . .
```

***TIPS & TRICKS**. This function is an alternative of the **ut_strrok** function, which works with simpler interface. The difference between the two functions is:*

- ***ut_get_token*** - *returns empty tokens with valid break characters (if any)*
- ***ut_strtok*** - *returns only non-blank tokens, skips successive break characters without valid tokens between and destroys the input string*

Arguments:

Argument	Description
`str`	[in] 0-terminated string to get token from
`ipos`	[i/o] start string position to search token (>=0)
`brks`	[in] 0-terminated string with break characters
`token`	[out] string with extracted token
`brk`	[out] the break character, which terminates the extracted token (not included in token)

Return code:

Return code	Description
`0`	No more tokens in **`str`**
`>0`	Length of extracted token

ut_search_str = Search string

Prototype:

```
#include “ut_lib.h”
int ut_search_str ( char *str, char **sbuf, int n )
```

Description:

The function searches **`str`** in the string buffer **`sbuf`** and returns the index of the **`sbuf`** element, which is equal to **str** (-1: string not found)

Arguments:

Argument	Description
`str`	[in] 0-terminated string
`sbuf`	[in] string buffer
`n`	[in] number of strings in **`sbuf`**

Return code:

Return code	Description
`>=0`	Index of found string in **`sbuf`**
`-1`	String not found

5.10. Vector function library

This library contains relatively elementary vector functions, optimized for best speed on TI TMS320C6xxx DSP by the ***restrict*** keyword. On other platforms (PC, …) these functions are compiled without any special optimizations, but we recommend using them due to the well optimized C code.

Note: The **`RESTRICT`** macro is defined as **`restrict`** on TMS320C6xxx DSP and empty on all other platforms.

Function notations

The prefixes in the function names show the type of the processed vectors (the ***vector*** is a term for one dimensional array):

Prefix	Vector type
`i8_`	Signed 8-bit vector (char)
`u8_`	Unsigned 8-bit vector (unsigned char)
`i32_`	Signed 32-bit vector (int)
`u32_`	Unsigned 32-bit vector (unsigned int)
`i64_`	Signed 64-bit integer vector
`u64_`	Unsigned 64-bit integer vector

Header files:

`vf_lib.h`	Vector function library header file
`vc_blib.h`	Basic vector function library header

i32_copy = Copy I32 (32-bit integer) vector

Prototype:

```
#include “vf_lib.h”
void i32_copy (int *RESTRICT buf, int *RESTRICT obuf, int n )
```

Description:

The function copies the source 32-bit integer vector **`buf`** into the destination vector **`obuf`**.

Arguments:

Argument	Description
`buf`	[in] source 32-bit integer buffer
`obuf`	[out] destination 32-bit integer buffer
n	[in] number of 32-bit integers to copy

Return code:

None

i64_copy = Copy I64 (64-bit integer) vector

Prototype:

```
#include “vf_lib.h”
void i64_copy (long *RESTRICT buf, long *RESTRICT obuf, int n )
```

Description:

The function copies the source 64-bit integer vector **buf** into the destination vector **obuf**.

Arguments:

Argument	Description
buf	[in] source 64-bit integer buffer
obuf	[out] destination 64-bit integer buffer
n	[in] number of 64-bit integers to copy

Return code:

None

i8_bits_to_bytes = Convert I8 bits to bytes

Prototype:

```
#include “vf_lib.h”
void i8_bits_to_bytes (char *RESTRICT bit_buf, char *RESTRICT byte_buf,
                       int bit_cnt )
```

Description:

The function converts the bits in the input vector **bit_buf** into bytes and stores the results in the output vector **byte_buf**. Each byte in **bit_buf** is unpacked into 8 bytes in **byte_buf** from MSB to LSB. Zero bits are converted to byte 0x00. Non-zero bits are converted to byte 0x01.

Arguments:

Argument	Description
bit_buf	[in] source vector with bits (size = **(bit_cnt+7)/8** bytes)
byte_buf	[out] destination vector with byte values (size = **bit_cnt** bytes)
bit_cnt	[in] number of bits in **bit_buf** and bytes in **byte_buf**

Return code:

None

i8_copy = Copy I8 (char) vector

Prototype:

```
#include “vf_lib.h”
void i8_copy ( char *RESTRICT in_buf, int in_inc, char *RESTRICT out_buf,
               int out_inc, int n )
```

Description:

The function copies the source char vector **`in_buf`** into the destination char vector **`out_buf`** with address increment.

Formula:

```
   out_buf[i*out_inc] = in_buf[i*in_inc], i=0,n-1
```

Arguments:

Argument	Description
`in_buf`	[in] source char vector
`in_inc`	[in] address increment of vector **`in_buf`**
`out_buf`	[out] destination char vector
`out_inc`	[in] address increment of vector **`out_buf`**
`n`	[in] number of vector elements to copy

Return code:

None

i8_copy_clr = Copy I8 (char) color vector

Prototype:

```
#include “vf_lib.h”
void i8_copy_clr ( char *RESTRICT in_buf, char *RESTRICT out_buf,
                   int trans_clr, int ti_enc, int n )
```

Description:

The function copies the source color vector **`in_buf`** into the destination color vector **`out_buf`**. The difference between ordinary vector copying and color vector copying is:

- This function optionally skips copying of **`in_buf`** elements, equal to the transparent color **`trans_clr`**.

Arguments:

Argument	Description
`in_buf`	[in] source vector with normal color values
`out_buf`	[out] destination vector with converted color values
`trans_clr`	[in] transparent color:

	`-1` : copy all **`in_buf`** colors to **`out_buf`**
	`>=0` : skip copying **`in_buf`** colors, equal to **`trans_clr`**

`ti_enc`	[in] color encoding flag (currently not supported on VRmagic DSP platform): `0` : don't convert colors `!=1` : convert **`in_buf`** colors before copying to **`out_buf`**
`n`	[in] number of vector elements to copy

Return code:
None

i8_fill = Fill I8 (char) vector with constant

Prototype:

```
#include “vf_lib.h”
void i8_fill (char *RESTRICT buf, int inc, int c, int n )
```

Description:
The function sets all **`buf`** elements to the constant **`c`**.

Formula:

```
   buf[i*inc] = c, i=0,n-1
```

Arguments:

Argument	Description
`buf`	[out] destination vector
`inc`	[in] vector address increment in bytes
`c`	[in] constant (value to fill)
`n`	[in] number of vector elements

Return code:
None

i8_set = Set I8 (char) vector elements with mask

Prototype:

```
#include “vf_lib.h”
void i8_set ( char *RESTRICT a, int ainc, char *RESTRICT m, int minc,
              int c, int n )
```

Description:
The function sets vector **`A`** elements as specified by the mask vector **`M`**:

- The **A** elements, which correspond to non-zero **M** elements, are set to constant **c**.
- The remaining **A** elements are left unchanged.

Formula:
```
  a[i*ainc] = c if m[i*minc] != 0, i=0,n-1
```

Arguments:

Argument	Description
a	[i/o] source/destination vector **A**
ainc	[in] vector **A** address increment in bytes
m	[in] mask vector **M** (NULL: don't use, set all **A** elements)
minc	[in] vector **M** address increment in bytes
c	[in] constant (value to fill)
n	[in] number of vector elements

Return code:
None

i8_fill_2c = Fill I8 vector with 2 colors

Prototype:
```
#include “vf_lib.h”
void i8_fill_2c ( char *RESTRICT in_buf, char *RESTRICT out_buf,
                  int fgnd, int bgnd, int n )
```

Description:
The function fills the destination vector **out_buf** with two colors **fgnd** and **bgnd**, depending on the values of the respective elements of the input mask vector **in_buf** – see the formula.

Formula:
```
  if(in_buf[i] != 0) then out_buf[i] = fgnd
                     else out_buf[i] = bgnd if bgnd >=0
                                     = leave unchanged if bgnd < 0
  for i=0,n-1
```

Arguments:

Argument	Description
in_buf	[in] mask vector, which defines how to set colors in **out_buf**: `!=0` : set respective **out_buf** byte to **fgnd** `0` : set respective **out_buf** byte to **bgnd** if **bgnd** `>= 0` leave respective **out_buf** byte unchanged if **bgnd** `< 0`
out_buf	[out] destination vector with color values
fgnd	[in] foreground color, used for non-zero **in_buf** elements

bgnd	[in] background color, used for zero **in_buf** elements: `-1` : leave respective **out_buf** elements unchanged `>=0` : set respective **out_buf** elements to **bgnd**
n	[in] number of vector elements in **in_buf** and **out_buf**

Return code:
None

u8_lookup = Convert U8 vector by a look-up table

Prototype:
```
#include “vf_lib.h”
void u8_lookup ( unsigned char *RESTRICT buf, unsigned char *RESTRICT ltab,
                 int n )
```

Description:
The function converts the input/output unsigned char vector **buf** by the look-up table **ltab**. The function can be used for fast binarization of pixel rows using a preliminary generated lookup table.

Formula:
```
  buf[i] = ltab[buf[i]], i=0,n-1
```

Arguments:

Argument	Description
buf	[i/o] source/destination char buffer
ltab	[in] look-up table with size = 256 bytes
n	[in] number of vector elements in **buf**

Return code:
None

u8_bin = Binarize U8 (unsigned char) vector

Prototype:
```
#include “vf_lib.h”
void u8_bin ( unsigned char *RESTRICT buf, int lo_th, int up_th, int
fgnd_clr,
              int bgnd_clr, int n )
```

Description:
The function binarizes the input/output vector **buf** with gray-level pixel values in the range [0,255]:

- The **buf** elements, which are in the range [**lo_th,up_th**] are set to **fgnd_clr**.
- The remaining **buf** elements are set to **bgnd_clr**.

Formula:

```
  if(lo_th <= buf[i] && buf[i] <= up_th) then buf[i] = fgnd_clr;
                                         else buf[i] = bgnd_clr;
  for i=0,n-1
```

When **up_th < lo_th**, the binarization formula is:

```
  if(lo_th <= buf[i] || buf[i] <= up_th) then buf[i] = obj_clr;
                                         else buf[i] = bgnd_clr;
```

Arguments:

Argument	Description
buf	[i/o] source/destination vector
lo_th	[in] lower binarization threshold
up_th	[in] upper binarization threshold
fgnd_clr	[in] foreground color
bgnd_clr	[in] background color
n	[in] number of vector elements in **buf**

Return code:
None

u8_pyrf = Basic pyramid function

Prototype:

```
#include “vf_lib.h”
void u8_pyrf ( unsigned char *RESTRICT in_row1,
               unsigned char *RESTRICT in_row2,
               unsigned char *RESTRICT out_row, int n )
```

Description:
The function calculates one row of the result pyramid image out from two sequential pixel rows from the source image. The size of the two input row buffers **in_row1** and **in_row2** is **n*2** elements.

Formula:

```
  out_row[i] = (in_row1[2*i] + in_row1[2*i+1] +
                in_row2[2*i] + in_row2[2*i+1]) / 4
  for i=0,n-1
```

Arguments:

Argument	Description

`in_row1`	[in] image row 1 buffer
`in_row2`	[in] image row 2 buffer
`out_row`	[out] destination image row buffer
`n`	[in] number of **`out_row`** elements

Return code:
None

u8_dotpr = Dot product (unsigned char)

Prototype:
```
#include "vf_lib.h"
unsigned long u8_dotpr ( unsigned char *RESTRICT a, int ainc,
                         unsigned char *RESTRICT b, int binc, int n )
```

Description:
The function calculates the dot product of the two input vectors **A** and **B** (vector scalar multiplication).

Formula:
```
  result = SUM (a[i*ainc] * b[i*binc]), i=0,n-1
```

Arguments:

Argument	Description
`a`	[in] source vector **A**
`ainc`	[in] vector **A** address increment in elements (bytes)
`b`	[in] source vector **B**
`binc`	[in] vector **B** address increment in elements (bytes)
`n`	[in] number of elements in vectors **A** and **B**

Return code:
The dot product of vectors **A** and **B**.

u8_dotpr_xor = XOR dot product (unsigned char)

Prototype:
```
#include "vf_lib.h"
unsigned long u8_dotpr_xor ( unsigned char *RESTRICT a, int ainc,
                             unsigned char *RESTRICT b, int binc, int n )
```

Description:

The function calculates a XOR dot product of the two input vectors **A** and **B**. The purpose of this function is to correlate two input vectors and to find the number of equivalent elements. Two vector elements are considered equivalent if they have equivalent LSB bits:

```
A element (bit 0)   B element (bit 0)
-----------------   -----------------
       0          ==        0
       1          ==        1
```

Formula:

```
result = n - SUM((a[i*ainc] ^ b[i*binc]) & 1), i=0,n-1
```

Arguments:

Argument	Description
a	[in] source vector **A**
ainc	[in] vector **A** address increment in elements (bytes)
b	[in] source vector **B**
binc	[in] vector **B** address increment in elements (bytes)
n	[in] number of elements in vectors **A** and **B**

Return code:

The number of equivalent elements in vectors **A** and **B**

u8_dotpr_xorm = XOR dot product with mask

Prototype:

```
#include “vf_lib.h”
unsigned long u8_dotpr_xorm ( unsigned char *RESTRICT a, int ainc,
                              unsigned char *RESTRICT b, int binc,
                              unsigned char *RESTRICT m, int minc, int n )
```

Description:

The function calculates a XOR dot product of the two input vectors **A** and **B** with mask **M**. The purpose of this function is to correlate the two input vectors and to find masked number of equivalent elements. Two vector elements are considered equivalent if they have equivalent LSB bits and the respective mask bit is equal to 1.

```
A element (bit 0)   B element (bit 0)   M element (bit 0)
-----------------   -----------------   -----------------
       0          ==        0          &&        1
       1          ==        1                    1
```

Formula:

```
result = SUM(!((a[i*ainc] ^ b[i*binc]) & 1) ^ m[i*minc]), i=0,n-1
```

Arguments:

Argument	Description

`a`	[in] source vector **`A`**
`ainc`	[in] vector **`A`** address increment in elements (bytes)
`b`	[in] source vector **`B`**
`binc`	[in] vector **`B`** address increment in elements (bytes)
`m`	[in] mask vector **`M`**
`minc`	[in] vector **`M`** address increment in elements (bytes)
`n`	[in] number of elements in vectors **`A`**, **`B`** and **`M`**

Return code:

The number of equivalent elements in vectors **`A`** and **`B`**

u8_vsum = Vector sum (unsigned char)

Prototype:

```
#include “vf_lib.h”
unsigned long u8_vsum ( unsigned char *RESTRICT a, int ainc, int n )
```

Description:

The function sums the elements of the input vector **`A`**.

Formula:

```
  result = SUM (a[i*ainc]), i=0,n-1
```

Arguments:

Argument	Description
`a`	[in] source vector **`A`**
`ainc`	[in] vector **`A`** address increment in elements (bytes)
`n`	[in] number of elements in vector **`A`**

Return code:

Sum of the elements of vector **`A`**

u8_histo = U8 (unsigned char) vector histogram

Prototype:

```
#include “vf_lib.h”
void u8_histo ( unsigned char *RESTRICT buf, int inc, int n,
                unsigned int *RESTRICT hist )
```

Description:

The function calculates the histogram of the input U8 buffer **`buf`** and stores the result histogram in the output **`hist`** buffer.

Formula:

```
  hist[buf[i*inc]]++ for i=0,n-1
```

Arguments:

Argument	Description
`buf`	[in] U8 (unsigned char) buffer with pixel values
`inc`	[in] **`buf`** address increment in elements (bytes), 1=successive elements
`n`	[in] number of elements (bytes) in **`buf`**
`hist`	[out] histogram buffer with size = 256 integers

Return code:

None

u8_minmax = U8 (unsigned char) vector min/max

Prototype:

```
#include “vf_lib.h”
void u8_minmax ( unsigned char *RESTRICT a, int ainc, int n,
                 int *minv, int *maxv )
```

Description:

The function finds the minimum and maximum values in the input vector **A**.

Formula:

```
  minv = MIN(a[i*ainc]), i=0,n-1
  maxv = MAX(a[i*ainc]), i=0,n-1
```

Arguments:

Argument	Description
`a`	[in] vector **A**
`ainc`	[in] vector **A** address increment in elements (bytes)
`n`	[in] number of elements in vector **A**
`minv`	[out] min element value (NULL: don't calculate)
`maxv`	[out] max element value (NULL: don't calculate)

Return code:

None

i32_minmax = I32 vector min/max

Prototype:
```
#include “vf_lib.h”
void i32_minmax ( int *RESTRICT a, int ainc, int n, int *minv, int *maxv )
```

Description:
The function finds the minimum and maximum values in the input 32-bit integer vector **A**.

Formula:
```
  minv = MIN(a[i*ainc]), i=0,n-1
  maxv = MAX(a[i*ainc]), i=0,n-1
```

Arguments:

Argument	Description
a	[in] 32-bit integer vector **A**
ainc	[in] vector **A** address increment in elements (integers)
n	[in] number of elements in vector **A**
minv	[out] min element value (NULL: don't calculate)
maxv	[out] max element value (NULL: don't calculate)

Return code:
None

u8_sobel = Basic Sobel function

Prototype:
```
#include “vf_lib.h”
void u8_sobel ( unsigned char *RESTRICT p1, unsigned char *RESTRICT p2,
                unsigned char *RESTRICT p3, unsigned char *RESTRICT pr,
                int count )
```

Description:
The function calculates one row of the result sobel image out from three consecutive pixel rows of the source image

Algorithm:

The Sobel filter is calculated with two masks – vertical and horizontal:
```
   | 1   0  -1 |       | 1   2   1 |
   | 2   0  -2 |       | 0   0   0 |
   | 1   0  -1 |       |-1  -2  -1 |
```

The function takes the 3x3 neighborhood of each input pixel and calculates the convolutions with the two masks. The two magnitudes (abs convolution values) are added and divided by 4 to produce a result pixel without overflow in the range [0,255].

Arguments:

Argument	Description
`p1`	[in] image row 1 buffer
`p2`	[in] image row 2 buffer
`p3`	[in] image row 3 buffer
`pr`	[out] result image row buffer
`count`	[in] number of i/o buffer elements

Return code:

None

5.11. VM_LIB system functions

The VM_LIB library contains a set of general functions, called ***system functions***, which do not belong to a particular sub-library. Here are the functions, which open and close the library.

vm_lib_open = Open VM_LIB library

Prototype:

```
#include “vm_lib.h”
int vm_lib_open ( int vm_key )
```

Description:

The function initializes the VM_LIB library and checks the license. You must call this function before any other VM_LIB function. This function initializes the drawing sub-library by call to **dr_init()**.

Arguments:

Argument	Description
vm_key	[in] VM_LIB license key

Return code:

Return code	Description
0	Success
Other	Returned by called functions

vm_lib_close = Close VM_LIB library

Prototype:

```
#include “vm_lib.h”
void vm_lib_close ()
```

Description:

The function closes the VM_LIB library and frees allocated memory and other system resources. It also closes the drawing sub-library by call to **dr_close()**.

Arguments:

None

Return code:

None

vm_lib_vers = Get library version

Prototype:

```
#include “vm_lib.h”
void vm_lib_vers ( int* version, int* subversion, int* release,
                   int* bugfix )
```

Description:

The function returns 4 library version parameters.

Arguments:

Argument	Description
`version`	[out] basic library version
`subversion`	[out] library sub-version number
`release`	[out] library release number
`bugfix`	[out] library bug-fix number

Return code:

None

vm_lib_vers_str = Get library version string

Prototype:

```
#include “vm_lib.h”
char *vm_lib_vers_str ()
```

Description:

The function returns the library version as string.

Arguments:

None

Return code:

Pointer to library version string.

sys_time = Get system time

Prototype:

```
#include “vm_lib.h”
unsigned long sys_time ()
```

Description:
The function returns the system time in milliseconds. The time grows from 0 upwards.

Arguments:
None

Return code:
The system time in ms.

sys_wait = System wait

Prototype:

```
#include “vm_lib.h”
void sys_wait (int wait_time )
```

Description:
The function delays the execution of the program by **`wait_time`** milliseconds and then continues execution.

On many multitasking systems, including PC (Windows) and Linux machines, this function can be used to pass control to other concurrently working processes (tasks, threads), usually by calling `sys_wait(0)`.

Arguments:

Argument	Description
`wait_time`	[in] wait time in milliseconds (ms)

Return code:
None

6. Error codes

This chapter documents the error codes returned by the VM_LIB library functions and libraries, which are not part of the VM_LIB library

6.1. Object recognition library errors (100 - 149)

The object recognition library error codes are defined in **or_lib.h**:

Code	Macro	Description
100	`OR_INV_PAT_RECT`	Invalid pattern rectangle
101	`OR_INV_EDG_IMG`	Invalid edge image
102	`OR_INV_SEARCH_RECT`	Invalid search rectangle
103	`OR_COPY_PROT_ERR`	Copy protection error

6.2. Basic VM_LIB library errors (150 - 500)

Most of the VM_LIB library functions return the following error codes, defined by macros in **lib_err.h**.

System function errors

Code	Macro	Description
0	`SYS_OK`	Success
150	`SYS_ALLOC_ERROR`	Memory allocation error
151	`SYS_EXIT_EVENT`	System exit event
152	`SYS_CAM_INIT_ERR`	Camera initialization error
153	`SYS_SCREEN_ERR`	Screen size/pitch error (BF camera not initialized)
154	`SYS_TPICT_ERR`	Take picture error
155	`SYS_TPICT_TIMEOUT`	BF take picture timeout error
156	`SYS_IMGOVF_ERR`	Image overflow error

VM_LIB tool errors

Code	Macro	Description
0	`VM_OK`	Success
200	`VM_ALLOC_ERROR`	Memory allocation error
201	`VM_INV_IMAGE`	Invalid input image
202	`VM_AREA_OUTSIDE`	Rectangle area outside image
203	`VM_BLOB_INV_HANDLE`	BLOB handle error
204	`VM_BLOB_INV_ODB_BUFFER`	BLOB invalid ODB buffer
205	`VM_MFIND_ARG_ERROR`	Mark finder argument error

206	VM_GUI_RECT_INV_HANDLE	GUI rectangle handle error
207	VM_GUI_RECT_INV_EXEC_ID	Invalid GUI rectangle exec id.
208	VM_GUI_RESIZE_ERROR	GUI rectangle resize error
209	VM_WIN_INV_HANDLE	Window handle error
210	VM_WIN_INV_EXEC_ID	Invalid window exec id.
211	VM_WIN_SIZE_ERR	Window size error (too small)
212	VM_BUT_INV_HANDLE	Button handle error
213	VM_BUT_INV_EXEC_ID	Invalid button exec id.
214	VM_SPIN_INV_HANDLE	Spin handle error
215	VM_SPIN_INV_EXEC_ID	Invalid spin exec id.
216	VM_SPIN_INV_ARG	Invalid spin argument(s)
217	VM_SPIN_TOGGLE_ERR	Spin toggle error
218	VM_EDIT_INV_HANDLE	Edit handle error
219	VM_EDIT_INV_EXEC_ID	Invalid edit exec id.
220	VM_EDIT_INV_STATE	Invalid edit state
221	VM_EDIT_INV_EDIT_LEN	Invalid edit box length in char #
222	VM_EDIT_INV_ARG	Invalid edit argument(s)
223	VM_RADB_INV_HANDLE	Radio-button handle error
224	VM_RADB_INV_EXEC_ID	Invalid radio-button exec id.
225	VM_RADB_INV_ARG	Invalid radio-button argument(s)
226	VM_CBOX_INV_HANDLE	Check-box handle error
227	VM_CBOX_INV_EXEC_ID	Invalid check-box exec id.
228	VM_CBOX_INV_ARG	Invalid check-box argument(s)
229	VM_DRAW_INV_ARG	Invalid drawing argument(s)
230	VM_INV_ARG	Invalid argument
231	VM_INTERNAL_ERR	Internal error
232	VM_RESBUF_OVF	Result buffer overflow

Drawing library errors

Code	Macro	Description
0	DR_OK	Success
300	DR_ALLOC_ERROR	Memory allocation error
301	DR_INV_ARG	Invalid function argument(s)
302	DR_INV_IMAGE	Invalid image
303	DR_INIT_ERROR	Draw library init error
304	DR_FONT_FILE_ERR	Font file error
305	DR_INVALID_FONT	Invalid font or not installed

Image processing library errors

Code	Macro	Description
350	`IP_ALLOC_ERROR`	Memory allocation error
351	`IP_INV_ARG`	Invalid function argument(s)
352	`IP_INV_IMAGE`	Invalid image
353	`IP_RESBUF_OVF`	Result buffer overflow
354	`IP_AREA_OUTSIDE`	Rectangle outside image
355	`IP_INTERNAL_ERR`	Internal error
356	`IP_INV_PLATFORM`	Invalid PC/camera platform
357	`IP_INV_BUFFER`	Invalid data buffer
358	`IP_NO_EDGES`	No detected edges
359	`IP_INV_RLC`	Invalid RLC buffer
360	`IP_RLC_OVF`	RLC buffer overflow
361	`IP_RLC_MISMATCH`	Mismatching RLC dimensions
362	`IP_RLC_CORR_ERR`	RLC correlation error
363	`IP_INV_FONT`	Invalid IP font buffer
364	`IP_INV_FONT_CODE`	Invalid font code
365	`IP_INV_FONT_ICON`	Invalid font char icon
366	`IP_FONTBUF_OVF`	Font data buffer overflow
367	`IP_CHARBUF_OVF`	Font char buffer overflow
368	`IP_FONT_FILE_ERR`	Font file error

Calculation library errors

Code	Macro	Description
400	`CL_ALLOC_ERROR`	Memory allocation error
401	`CL_INVALID_ARG`	Invalid function argument(s)
402	`CL_RESBUF_OVF`	Result buffer overflow
403	`CL_INTERNAL_ERR`	Internal calculation error

Some libraries have separate error codes, defined in the respective header files…

6.3. Normalized correlation library errors (9001 - 9009)

Code	Macro	Description
0	`RC_OK`	Success
9001	`RC_SYS_NOMEMORY`	DMEM allocation error
9002	`RC_SYS_PATWID_ERR`	Invalid pattern width
9003	`RC_SYS_PATHEI_ERR`	Invalid pattern height
9004	`RC_SYS_SEARCHWID_ERR`	Invalid search area width

9005	RC_SYS_SEARCHHEI_ERR	Invalid search area height
9006	RC_SYS_CORR_ERR	Correlation error
9007	RC_SYS_FIND_FAIL	Recognition failed
9008	RC_SYS_SRHBUF_OVF	Search buffer overflow
9009	RC_SYS_PATBUF_OVF	Unable to allocate pattern buffer

6.4. BLOB library errors (9100 - 9120)

The BLOB library error codes are defined in **bl_lib.h**:

Code	Macro	Description
9100	BLOB_R_NO_MEMORY	Memory allocation error
9101	BLOB_R_IMG_OVF	Image buffer overflow
9102	BLOB_R_RLCBUF_OVF	RLC buffer overflow. Increase the size of the RLC buffer by the bl_set_max_rlc_size() function.
9103	BLOB_R_BINIMG_ERR	Invalid format of binarized image
9104	BLOB_R_MBUF_OVF	Object merge buffer overflow
9105	BLOB_R_OBJLABEL_ERR	Object labeling error
9106	BLOB_R_ODB_OVF	Object database (ODB) buffer overflow
9107	BLOB_R_INTERNAL_ERR	Internal program error
9108	BLOB_R_INV_OBJ_IDX	Invalid object index
9109	BLOB_R_TRACE_ERR	Contour tracing error
9110	BLOB_R_CONTBUF_OVF	Contour buffer overflow
9111	BLOB_R_INV_FEATURE	Invalid object feature
9112	BLOB_R_DELETED_OBJ	Deleted object
9113	BLOB_R_FEATURE_NOCALC	Prerequisite feature not calculated
9114	BLOB_R_INVALID_ARG	Invalid input argument
9115	BLOB_R_DRAW_ERR	Feature drawing error
9116	BLOB_R_NO_ORIENT	Insufficient orientation data
9117	BLOB_R_COPY_PROT	Copy protection error
9118	BLOB_R_INVALID_ODB	Invalid ODB buffer
9119	BLOB_R_CONTOUR_LIMIT	Contour limit exceeded
9120	BLOB_R_NO_RLC_MEMORY	RLC buffer allocation error. Use the bl_set_max_rlc_size() function to increase the size of the work RLC buffer.