It's no trick...
it's a vision system

Vision
Components
The Smart Camera People

Extension Library Manual

Revision 2.4 December 18th, 2009 ME
Document name: Extensionlib.Doc

Vision Components
The Smart Camera People

Vision Components GmbH
Ottostraße 2 • D-76275 Ettlingen
Tel. +49(7243)2167-0
Fax +49(7243)2167-11
sales@vision-components.de

www.vision-components.com

Foreword and Disclaimer

This documentation has been prepared with most possible care. However, Vision Components GmbH does not take any liability for possible errors. In the interest of progress, Vision Components GmbH reserves the right to perform technical changes without further notice.
Please notify **support@vision-components.com** if you become aware of any errors in this manual or if a certain topic requires more detailed documentation.
This manual is intended for information of Vision Component's customers only. Any publication of this document or parts thereof requires written permission by Vision Components GmbH.

Trademarks

Code Composer Studio and TMS320C6000, Windows XP, Total Commander, Tera Term, Motorola are registered Trademarks. All trademarks are the property of their respective owners.

References

Since the VC40XX smart camera family employs a TI processor, the programming environment and functions for the VC20XX cameras can be used for this camera.

Please also consult the following resources for further reference:

"**Knowledge Base / FAQ**" for a searchable data base of SW and HW questions / answers.

Description	Title on Website	Download Area
Quick start Manual for VC camera set up and programming	Getting Started VC Smart Cameras with TI DSP	Getting Started VC SDK Ti
Schnellstart VC – deutsche Version of „Getting Started VC".	Schnellstart VC Smart Kameras	Getting Started VC20XX and VC40XX Cameras
Introduction to VC Smart Camera programming	Programming Tutorial for VC20XX and VC40XX Cameras	Getting Started VC20XX and VC40XX Cameras
Demo programs and sample code used in the Programming Tutorial	Tutorial_Code	Getting Started VC20XX and VC40XX Cameras
VC4XXX Hardware Manual	VC40XX Smart Cameras Hardware Documentation	Hardware Documentation VC Smart Cameras
VCRT Operation System Functions Manual	VCRT 5.0 Software Manual	Software documentation VC Smart Cameras
VCRT Operation System TCP/IP Functions Manual	VCRT 5.0 TCP/IP Manual	Software documentation VC Smart Cameras
VCLIB 2.0 /3.0 Image Processing Library Manual	VCLIB 2.0/ 3.0 Software Manual	Software documentation VC Smart Cameras

- The Light bulb highlights hints and ideas that may be helpful for a development.
- This warning sign alerts of possible pitfalls to avoid. Please pay careful attention to sections marked with this sign.

Author: VC Support, **mailto:support@vision-comp.com**

Table of Contents

1 Affine and non-affine coordinate transformations

rotate90l	rotate image by 90 degrees counter-clockwise
rotate90r	rotate image by 90 degrees clockwise
rotate180	rotate image by 180 degrees
move_image_alpha	move image with 2D interpolation
affine_image_transform	general affine image transform
affine_image_transform2	general affine image transform (floating-point)
calc_rotation_matrix	calculate affine rotation matrix
polar_image_transform	polar to cartesian image transformation
polar_image_transform2	polar to cartesian image transformation
mirror_hor	mirror image horizontally
mirror_ver	mirror image vertically
xshear	horizontal image shear
lens_transform	lens distortion correction
lens_transform2	lens distortion correction, type 2

Affine transformations map one coordinate system into a different one using a **linear** mapping. All affine transformations can be specified by the following formula

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a[0][0] & a[0][1] \\ a[1][0] & a[1][1] \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t[0] \\ t[1] \end{pmatrix}$$

Where

(x, y)	coordinate in the source coordinate system
(x', y')	coordinate in the target system
a[0][0], a[0][1]	coefficients of the transformation matrix
a[1][0], a[1][1]	
(t[0], t[1])	displacement vector

With the above formula **any combination** of the following transformation may be performed:

- displacement (movement) parallel to x and y axis
- rotation with an arbitrary rotation center point
- shear
- size scaling (zoom up / down) in x and y separately
- mirroring

For the function `affine_image_transform()` **the inverse transformation matrix is used.** I.e. (x,y) now is the coordinate in the target image variable and (x',y') in the source image.

rotate90l — **rotate image by 90 degrees counter-clockwise**

synopsis

```
void rotate90l(image *src, image *dst)
```

description

The function `rotate90l()` rotates an image by 90 degrees counter-clockwise. It is not possible to use the function in-place, i.e. source and destination images must not overlap.

The function rotates the whole image if the dimensions of source and destination image fulfil the following conditions:

```
src->dx = dst->dy
src->dy = dst->dx
```

If this is not the case, only a fraction of the source image is rotated with the following dimensions:

```
dx = min(src->dx, dst->dy)
dy = min(src->dy, dst->dx)
```

The function splits the source image into tiles which can be processed very efficiently even with limited cache memory size.

memory

none

see also

`rotate90r()`

rotate90r — **rotate image by 90 degrees clockwise**

synopsis

```
void rotate90r(image *src, image *dst)
```

description

The function `rotate90r()` rotates an image by 90 degrees clockwise. Please refer to the documentation of `rotate90r ()` for the details.

memory

none

see also

`rotate90l()`

rotate180 — **rotate image by 180 degrees**

synopsis

```
void rotate180(image *src, image *dst)
```

description

The function `rotate180()` rotates an image by 180 degrees. It is possible to use the function in-place, i.e. source and destination images may be identical. In this case, the function uses a slightly faster exchange-and-reverse routine.

If source and destination images are not identical but overlap, the result is not defined.

The function rotates the whole image if the dimensions of source and destination image fulfil the following conditions:

```
src->dx = dst->dx
src->dy = dst->dy
```

If this is not the case, only a fraction of the source image is rotated with the following dimensions:

```
dx = min(src->dx, dst->dx)
dy = min(src->dy, dst->dy)
```

memory none

see also **rotate90l**()

move_image_alpha — move image with 2D interpolation

synopsis

```
I32 move_image_alpha(image *src, image *dst,
                        float mx, float my, U8 bgnd)
```

description The function **move_image_alpha**() moves an image by a fractional number of pixels in x and y direction using bilinear interpolation.

The function uses the following parameters:

`src` : pointer to source image
`dst` : pointer to destination image
`mx` : movement vector x-component, mx >0 : movement to right
`my` : movement vector y-component, my >0 : downward movement
`bgnd` : background color

The movement vector (mx, my) is added to the start of source image. The image is then copied using bilinear interpolation. If (mx, my) = (0, 0), the result of the operation is the same as for the `copy()` function.

The resulting image always has the dimensions of the destination image. Missing areas are set to `bgnd`. The function also performs a bilinear interpolation between edges of the source image and the background color where applicable.

It is possible to use the function in-place if my<0, otherwise the result is not defined.

Returned Values : error codes

-1 : one of the image variables is NULL
-2 : memory allocation error

memory dst->dx or less

see also **rotate90l**()

affine_image_transform

general affine image transformation

synopsis

```
I32 affine_image_transform(image *src,
      image *dst, float a[2][2], float t[2], U8 bgnd)
```

description

This function performs a general affine image transformation on the source image and outputs the result to the destination image. The function uses bilinear interpolation.

The function uses the following parameters:

src : source image variable
dst : destination image variable
a[2][2]: inverse 2D transformation matrix
t[2] : inverse 2D translation vector
bgnd : background grey value

The resulting image always has the dimensions of the destination image. Missing areas are set to bgnd.

Please be aware, that the function needs the **inverse transformation matrix and translation vector** as an input.

It is not possible to use the function in-place, i.e. source and destination images must not overlap.

memory none

see also matrix()

affine_image_transform2

general affine image transformation

synopsis

```
I32 affine_image_transform2(image *src,
      image *dst, float a[2][2], float t[2], U8 bgnd)
```

description

Same as **affine_image_transform**() but with (slow) floating-point calculation. It is recommmended to use function **affine_image_transform2**() instead.

calc_rotation_matrix — calculate affine rotation matrix

synopsis

```
void calc_rotation_matrix(float angle,
      float cx, float cy, float a[2][2], float t[2])
```

description

This function calculates an inverse affine transformation matrix and translation vector for a clockwise rotation. The result matrix and vector may be used for the functions **affine_image_transform**() or **affine_image_transform2**().

The function uses the following parameters:

`angle`	: angle for clockwise rotation [degrees]
`cx`	: rotation center x-coordinate
`cy`	: rotation center y-coordinate
`a[2][2]`	: inverse 2D transformation matrix (result)
`t[2]`	: inverse 2D translation vector (result)

memory none

see also **affine_image_transform**(),**calc_rotation**()

calc_rotation — calculate affine rotation matrix using angle and 2 points

synopsis

```
void calc_rotation(float angle, point *p,
      point *target_p, float a[2][2], float t[2])
```

description

This function calculates an inverse affine transformation matrix and translation vector for a clockwise rotation. The result matrix and vector may be used for the functions **affine_image_transform**() or **affine_image_transform2**(). This function can be used if the angle of the rotation is known as well as a point p in the original image and its mapping in the target image.

The function uses the following parameters:

`angle`	: angle for clockwise rotation [degrees]
`p`	: point in original image (x, y)
`target_p`	: target image of point p after rotation
`a[2][2]`	: inverse 2D transformation matrix (result)
`t[2]`	: inverse 2D translation vector (result)

memory none

see also **affine_image_transform**(), **calc_rotation_matrix**()

polar_image_transform

polar to cartesian image transformation

synopsis

```
I32  polar_image_transform(image *src,
                           image *dst, float t[2], U32 r0, U8 bgnd)
```

description

This function calculates a bilinear interpolated transformation from polar coordinates into cartesian coordinates.

The function uses the following parameters:

`src`	: pointer to source image
`dst`	: pointer to destination image
`t[2]`	: inverse 2D translation vector
`r0`	: minimum radius for transform
`bgnd`	: background grey value

applications

This function may be used to unwrap circular barcode or characters written e.g. on a CD-ROM or wafer.

memory

none

see also

`affine_image_transform``()`

polar_image_transform2 — **polar to cartesian image transformation**

synopsis

```
I32 polar_image_transform2(image *src,
                  image *dst, float t[2], U32 r0, U8 bgnd)
```

description

Same as `polar_image_transform()` but with (slow) floating-point calculation. It is recommmended to use function `polar_image_transform()` instead.

mirror_hor — **mirror image horizontally**

synopsis

```
void mirror_hor(image *src, image *dst)
```

description

The function `mirror_hor()` mirrors an image horizontally with respect to its central axis. It is possible to use the function in-place, i.e. source and destination images may be identical. In this case, the function uses a slightly faster exchange-and-reverse routine.

If source and destination images are not identical but overlap, the result is not defined.

The function mirrors the whole image if the dimensions of source and destination image fulfil the following conditions:

```
src->dx = dst->dx
src->dy = dst->dy
```

If this is not the case, only a fraction of the source image is mirrored with the following dimensions:

```
dx = min(src->dx, dst->dx)
dy = min(src->dy, dst->dy)
```

memory

none

see also

mirror_ver()

mirror_ver — **mirror image vertically**

synopsis

```
void mirror_ver(image *src, image *dst)
```

description

The function **mirror_ver**() mirrors an image vertically with respect to its central axis. It is possible to use the function in-place, i.e. source and destination images may be identical. In this case, the function uses a slightly faster exchange-and-reverse routine.

If source and destination images are not identical but overlap, the result is not defined.

The function mirrors the whole image if the dimensions of source and destination image fulfil the following conditions:

```
src->dx = dst->dx
src->dy = dst->dy
```

If this is not the case, only a fraction of the source image is mirrored with the following dimensions:

```
dx = min(src->dx, dst->dx)
dy = min(src->dy, dst->dy)
```

memory none

see also **mirror_hor**()

xshear — horizontal image shear

synopsis

```
void xshear(image *src, float shear,
            image *dst, float offset, U8 bgnd)
```

description

The function **xshear**() performs a horizontal image shear of the source image by a shear factor of `shear`. A horizontal displacement offset `offset` may be added. Pixels for which the corresponding source image is not defined, are set to `bgnd`.

If source and destination images are not identical but overlap, the result is not defined.

memory none

see also **yshear**()

threepoint_calculate — **three-point formula for affine transformations**

synopsis

```
I32 threepoint_calculate(
      point *p0, point *p1, point *p2,
      point *q0, point *q1, point *q2,
                  float **a, float *t)
```

description

The function **threepoint_calculate**() calculates the inverse affine transformation matrix for 2 x 3 coordinate points.

When two patterns must be compared very accurately, e.g. in print inspection, it is useful to match both patterns using affine transformation with subpixel accuracy before comparing them. This function allows to compute the inverse transformation matrix and displacement vector. Three known points are necessary in the original image and their image points in the transformed image (i.e. 6 in total).
These points may be chosen by binary blob analysis or correlation methods as an example. For the affine transformation itself, always the *inverse* transformation matrix and displacement vector is required as computed by the function. If, for some reason, the non-inverse matrix is required, just exchange the points p and q for the original and transformed image.

The function uses the following parameters:

`p0, p1, p2`	point coordinates for first image
`q0, q1, q2`	point coordinates for second image
`a`	inverse 2D transformation matrix 2x2 (result)
`t`	inverse 2D translation vector 2x1 (result)

points are defined by the following struct:

```
typedef struct           /* coordinate point          */
  {
  float x;               /* x coordinate (float)      */
  float y;               /* y coordinate (float)      */
  } point;
```

The function returns the standard error code, including `ERR_SINGULAR`. This will be the case, if the three points are on a line.
Even if the function does not indicate a singular situation, it is not wise to have all three points on a line.

recommendation Do not place the three points on a line

memory none

see also **affine_image_transform**(), **calc_rotation_matrix**()

lens_transform **lens distortion correction**

synopsis

```
I32  lens_transform(image *src, image *dst,
                    point *center, float k3, U8 bgnd)
```

description

The function `lens_transform()` performs a lens correction transformation.

Pincushion- and barrel-type distortions can be corrected. Distortions of this type typically increase with the square of the distance to the optical centerpoint.

The following parameters are used:

`src`	source image
`dst`	transformed destination image
`center.x, center.y`	transformation centerpoint
`k3`	transformation parameter
`bgnd`	background color

The transformation is performed using the following formula:

$$\begin{pmatrix} x' - center.x \\ y' - center.y \end{pmatrix} = \begin{pmatrix} x - center.x \\ y - center.y \end{pmatrix} * (1 + k3/1000 * \begin{pmatrix} x - center.x \\ y - center.y \end{pmatrix}^2)$$

The sign and value of the parameter `k3` determines the type of the distortion:

`k3 = 0`	no distortion
`k3 > 0`	pincushion type distortion
`k3 < 0`	barrel type distortion

Pixels for which the corresponding source image is not defined, are set to `bgnd`.

memory none

see also `lens_transform2()`

lens_transform2 **lens distortion correction, type 2**

synopsis

```
I32  lens_transform2(image *src, image *dst,
            point *center, float f, float mag, U8 bgnd)
```

description

The function `lens_transform2()` performs a lens correction for barrel-type distortions.

The function corrects non-linear circular symmetric distortions based on a universal model of the lens. The model assumes that the lens maps a part of a sphere to the CCD-sensor. All the user needs to know is the focal length of the lens in units of the sensor pixel size.
The model cannot be applied to telecentric lenses and some specially corrected lenses and has to be approved for a specific lens. Also, it is possible that focal length parameter for this routine deviates from the actual f-value.

The function uses the following parameters:

`src`	source image
`dst`	transformed destination image
`center.x, center.y`	transformation centerpoint
`f`	focal length parameter
`mag`	magnification (1.0: no magnification)
`bgnd`	background color

The transformation is performed using the following formula:

$$\begin{pmatrix} x' - center.x \\ y' - center.y \end{pmatrix} = \begin{pmatrix} x - center.x \\ y - center.y \end{pmatrix} * mag * correction(r^2)$$

The key parameter for the correction is `f`, namely the focal length of the lens divided by the pixel size.

The following table lists the pixelsize for the different camera models:

Camera model	Sensor	pixelsize [µm]
VC4018, VC4038	ICX424	7.4 x 7.4
VC4016, VC4066	ICX204	4.65 x 4.65
VC4068	ICX205	4.65 x 4.65
VC4065	ICX415	8.3 x 8.3
VC4472	ICX274	4.4 x 4.4
VC4058	KAI-0340	7.4 x 7.4
SBC4012	MT9P031	2.2 x 2.2

For example, using a lens with a focal length of 6 mm together with a VC4472 camera would be:

```
f = 1000.0 * 6.0/4.4
```

The factor of 1000 is due to the fact that one mm equals 1000 µm.

The image can be magnified (`mag > 1.0`) or demagnified (`mag < 1.0`).
A tilt in the object plane (object plane not perpendicular to optical axis) can be easily adjusted by setting the transformation centerpoint `(center.x, center.y)` to some pixel outside the CCD midpoint, since this effectively rotates the optical axis.

Pixels for which the corresponding source image is not defined, are set to `bgnd`.

When called the first time, the function builds up a table for the correction values. The memory for the table is automatically allocated and the table is calculated, which may take some time. The table is kept in memory for further use with the same parameters. If the parameters are changed, the old table will be released and a new table will be set up.

To deallocate the table memory, the function

```
void deinit_lens_transform2()
```

should be called.

memory `4*(dst->dx - p->cx)*(dst->dy - p->cy)` bytes, use function **`deinit_lens_transform2`**`()` to release memory

see also **`lens_transform`**`()`

2 Filter functions

isef	infinite symmetric exponential filter (recursive)
isef_hor	infinite symmetric exponential horizontal filter (recursive)
isef_ver	infinite symmetric exponential vertical filter (recursive)
gauss	recursive gauss filter
gauss_hor	horizontal recursive gauss filter
gauss_ver	vertical recursive gauss filter
gauss_fir	non-recursive gauss filter 3x3, 5x5, etc.
gradient_2x2	vector gradient (robert's cross)
gradient_3x3	vector gradient (sobel)
maxMxN	moving maximum (dilation) filter
minMxN	moving minimum (erosion) filter

isef **infinite symmetric exponential filter (recursive)**

synopsis

```
I32 isef(image *src, image *dst, float b) )
```

description

The function `isef()` calculates the infinite symmetric exponential filter with filter parameter `b` with `0.0 < b < 1.0`. `b` defines the equivalent of the filter kernel size for this recursive filter: the larger the value of `b`, the larger the kernel size. Since this function is designed as a recursive filter, the execution speed does not depend on the size of `b`.

Remark: The function calls **isef_hor()** and **isef_ver()** in succession.

The function returns the standard error code.

memory `(dx*(dy+2)+2)/2` bytes of heap memory

see also **isef_hor()**, **isef_ver()**

isef_hor **horizontal infinite symmetric exponential filter (recursive)**

synopsis

```
I32 isef_hor(image *src, image *dst, float b)
```

description

The function **isef_hor()** calculates the horizontal infinite symmetric exponential filter with filter parameter `b` with `0.0 < b < 1.0`. `b` defines the equivalent of the filter kernel size for this recursive filter: the larger the value of `b`, the larger the kernel size. Since this function is designed as a recursive filter, the execution speed does not depend on the size of `b`.

The function returns the standard error code.

memory `2*(dx+1)` bytes of heap memory

see also **isef()**, **isef_ver()**

isef_ver **vertical infinite symmetric exponential filter (recursive)**

synopsis `I32 isef_ver(image *src, image *dst, float b) )`

description The function **isef_ver**() calculates the vertical infinite symmetric exponential filter with filter parameter `b` with `0.0 < b < 1.0`. `b` defines the equivalent of the filter kernel size for this recursive filter: the larger the value of `b`, the larger the kernel size. Since this function is designed as a recursive filter, the execution speed does not depend on the size of `b`.

The function returns the standard error code.

memory `2*(dx*(dy+2)+2)` bytes of heap memory

see also **isef**(), **isef_hor**()

gauss **recursive gauss filter**

synopsis `I32 gauss(image *src, image *dst, float sigma) )`

description The function **gauss**() calculates the recursive gauss filter with filter parameter `sigma` with `0.0 < sigma < 5.0`. `sigma` defines the equivalent of the filter kernel size (standard deviation) for this recursive filter: the larger the value of `sigma`, the larger the kernel size. Since this function is designed as a recursive filter, the execution speed does not depend on the size of `sigma`.

Remark: The function calls **gauss_hor**() and **gauss_ver**() in succession.

The function returns the standard error code.

memory `2*(dx*(dy+6)+1)` bytes of heap memory

see also **gauss_fir**(), **gauss_hor**(), **gauss_ver**()

point spread funtions for different values of sigma

```
sigma = 0.625

000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
001: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
003: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
005: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
006: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
007: 00 00 00 00 00 00 00 00 00 01 01 01 00 00 00 00 00 00 00 00 00
008: 00 00 00 00 00 00 00 00 02 05 07 05 02 00 00 00 00 00 00 00 00
009: 00 00 00 00 00 00 00 01 05 0e 16 0e 05 01 00 00 00 00 00 00 00
010: 00 00 00 00 00 00 00 01 07 16 23 16 07 01 00 00 00 00 00 00 00
011: 00 00 00 00 00 00 00 01 05 0e 16 0e 05 01 00 00 00 00 00 00 00
012: 00 00 00 00 00 00 00 00 02 05 07 05 02 00 00 00 00 00 00 00 00
013: 00 00 00 00 00 00 00 00 00 01 01 01 00 00 00 00 00 00 00 00 00
014: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
015: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
016: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
017: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
018: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
019: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

sigma = 1.0

000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
001: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
003: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
005: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
006: 00 00 00 00 00 00 00 00 01 01 01 01 01 00 00 00 00 00 00 00 00
007: 00 00 00 00 00 00 00 01 02 03 03 03 02 01 00 00 00 00 00 00 00
008: 00 00 00 00 00 00 01 02 04 05 06 05 04 02 01 00 00 00 00 00 00
009: 00 00 00 00 00 00 01 03 05 08 09 08 05 03 01 00 00 00 00 00 00
010: 00 00 00 00 00 00 01 03 06 09 0b 09 06 03 01 00 00 00 00 00 00
011: 00 00 00 00 00 00 01 03 05 08 09 08 05 03 01 00 00 00 00 00 00
012: 00 00 00 00 00 00 01 02 04 05 06 05 04 02 01 00 00 00 00 00 00
013: 00 00 00 00 00 00 00 01 02 03 03 03 02 01 00 00 00 00 00 00 00
014: 00 00 00 00 00 00 00 00 01 01 01 01 01 00 00 00 00 00 00 00 00
015: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
016: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
017: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
018: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
019: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00


sigma = 1.5

000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
001: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
003: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004: 00 00 00 00 00 00 00 00 00 01 01 01 00 00 00 00 00 00 00 00 00
005: 00 00 00 00 00 00 00 01 01 01 01 01 01 01 00 00 00 00 00 00 00
006: 00 00 00 00 00 01 01 01 01 02 02 02 01 01 01 01 00 00 00 00 00
007: 00 00 00 00 00 01 01 02 02 03 03 03 02 02 01 01 00 00 00 00 00
008: 00 00 00 00 01 01 01 02 03 03 04 03 03 02 01 01 01 00 00 00 00
009: 00 00 00 00 01 01 02 03 03 04 04 04 03 03 02 01 01 00 00 00 00
010: 00 00 00 00 01 01 02 03 04 04 05 04 04 03 02 01 01 00 00 00 00
011: 00 00 00 00 01 01 02 03 03 04 04 04 03 03 02 01 01 00 00 00 00
012: 00 00 00 00 01 01 01 02 03 03 04 03 03 02 01 01 01 00 00 00 00
013: 00 00 00 00 00 01 01 02 02 03 03 03 02 02 01 01 00 00 00 00 00
014: 00 00 00 00 00 00 01 01 02 02 02 02 02 01 01 00 00 00 00 00 00
015: 00 00 00 00 00 00 00 01 01 01 01 01 01 01 00 00 00 00 00 00 00
016: 00 00 00 00 00 00 00 00 01 01 01 01 01 00 00 00 00 00 00 00 00
017: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
018: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
019: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

gauss_hor **horizontal gauss filter (recursive)**

synopsis `I32 gauss_hor(image *src, image *dst, float sigma)`

description The function **gauss_hor**() calculates the horizontal recursive gauss filter with filter parameter `sigma` with `0.0 < sigma < 5.0`. `sigma` defines the equivalent of the filter kernel size (standard deviation) for this recursive filter: the larger the value of `sigma`, the larger the kernel size. Since this function is designed as a recursive filter, the execution speed does not depend on the size of `sigma`.

The function returns the standard error code.

memory `2*(dx+1)` bytes of heap memory

see also **gauss**(), **gauss_ver**()

gauss_ver **vertical gauss filter (recursive)**

synopsis `I32 gauss_ver(image *src, image *dst, float sigma)`

description The function **gauss_ver**() calculates the vertical recursive gauss filter with filter parameter `sigma` with `0.0 < sigma < 5.0`. `sigma` defines the equivalent of the filter kernel size (standard deviation) for this recursive filter: the larger the value of `sigma`, the larger the kernel size. Since this function is designed as a recursive filter, the execution speed does not depend on the size of `sigma`.

The function returns the standard error code.

memory `2*(dx*(dy+6)+1)` bytes of heap memory

see also **gauss**(), **gauss_hor**()

gauss_fir **non-recursive gauss filter**

synopsis

```
I32 gauss_fir(image *src, image *dst, float sigma)
```

description

This is the non-recursive version of the gauss low-pass filter. `sigma` is the standard deviation of the filter.

sigma	Filter size
0.391	3x3
0.625	5x5
0.812	7x7

Values for `sigma` between the values in the table are allowed. The function switches to whatever filter size comes closer to the value of `sigma`.

The function returns the standard error code.

see also

gauss()

gradient_2x2 **vector gradient (robert's cross)**

synopsis

```
I32 gradient_2x2(image *src, image *dst)
```

description

The function **gradient_2x2**() calculates the vector gradient, i.e. a gradient with separate magnitude and direction components for the source image `src`. The destination image `dst` is therefore of type `IMAGE_VECTOR`. A 2x2 robert's cross type filter is used. The directional component has values in the range of [0..256] corresponding to angles between 0 and 360 degrees according to the following table.

```
0xe0 0x00 0x20
0xc0  xx  0x40
0xa0 0x80 0x60
```

Directions of an edge are defined normal to the edge pointing from the dark side to the bright side.

The function returns the standard error code.

The function requires a large table for the calculation which can be initialized using the function

```
I32 init_gradient_2x2(),
```

which allocates memory for the table and initializes it with the proper values. It returns the standard error code.

To deallocate the memory, the function

```
void deinit_gradient_2x2()
```

should be used.

gradient_2x2() also works, if **init_gradient_2x2**() is not called beforehand. It does the memory allocation and initialisation, but this may take some time, the first time the function is called, so the user might like to do the initialisation at the time when the program starts to guarantee equal processing times.

memory 256 KB of heap memory

see also **gradient_3x3**(), **robert**()

gradient_3x3 — vector gradient (sobel)

synopsis

```
I32 gradient_3x3(image *src, image *dst)
```

description

The function **gradient_3x3**() calculates the vector gradient, i.e. a gradient with separate magnitude and direction components for the source image `src`. The destination image `dst` is therefore of type `IMAGE_VECTOR`. A 3x3 sobel type filter is used. The directional component has values in the range of [0..256] corresponding to angles between 0 and 360 degrees according to the following table.

```
0xe0 0x00 0x20
0xc0  xx  0x40
0xa0 0x80 0x60
```

Directions of an edge are defined normal to the edge pointing from the dark side to the bright side.

The function returns the standard error code.

The function requires a large table for the calculation which can be initialized using the function

```
I32 init_gradient_3x3(),
```

which allocates memory for the table and initializes it with the proper values. It returns the standard error code.

To deallocate the memory, the function

```
void deinit_gradient_3x3()
```

should be used.

gradient_3x3() also works, if **init_gradient_3x3**() is not called beforehand. It does the memory allocation and initialisation, but this may take some time, the first time the function is called, so the user might like to do the initialisation at the time when the program starts to guarantee equal processing times.

memory 256 KB of heap memory

see also **gradient_2x2**(), **sobel**()

maxMxN **moving maximum (dilation)) filter**

synopsis `I32 maxMxN(image *src, image *dst, I32 mx, I32 my)`

description The function **maxMxN**() calculates the moving maximum filter (grey value dilation) with a filter kernel of size (mx, my). It is possible to set either mx or my to one, in which case a linear horizontal or vertical structuring element is used.

It is not possible to use this function in-place, i.e src and dst must be different. The exection time is independent of the mask size

The function returns the standard error code.

memory dx bytes of heap memory

see also **minMxN**()

minMxN **moving minimum (erosion)) filter**

synopsis `I32 minMxN(image *src, image *dst, I32 mx, I32 my)`

description The function **minMxN**() calculates the moving minimum filter (grey value erosion) with a filter kernel of size (mx, my). It is possible to set either mx or my to one, in which case a linear horizontal or vertical structuring element is used.

It is not possible to use this function in-place, i.e src and dst must be different. The exection time is independent of the mask size

The function returns the standard error code.

memory dx bytes of heap memory

see also **maxMxN**()

3 Edge detection

In an image, homogeneous regions, i.e. regions with slowly moving grey values are of minor importance for the recognition process. Most of the information is located where grey values change rapidly, i.e. in the edges of an image. Edge detection is a method to locate the relevant pixel changes precisely and robustly in an image.

Edge detection is quite vulnerable to noise. Noise can be reduced using low-pass filters. For this very reason, all edge detection algorithms essentially use some kind of low-pass filter as a preprocessing stage. Some images have much noise, others not. The noise does not even have to stem from the sensor or the camera electronic, e.g. if you imagine a rough or grinded surface on an industrial part, the surface structure might be considered as noise, whereas for a similar part with a polished, shiny surface, a rough structure might be a flaw that must be detected. Edge detection solves this conflict, using low-pass filters with different filter size. So for an object with rough surface a large filter size would be required to average over the surface structure; for the second example a smaller size of the filter kernel would allow to detect even tiny flaws.

The edge detection itself is performed by calculating the first or second derivative of an image and thresholding. For the detection of the edges, clever methods have been developed, to

- be as insensitive to noise as possible
- to precisely locate the edges
- to produce edges lines that are only one pixel wide (if possible)

One of the methods is a maximum search technique, that detects the maximum of the gradient image either directly in the first order or as zero crossings in the second order derivative.

In the literature, quite a lot of edge detection algorithms have been suggested, the Marr-Hildredth, Canny, Shen-Castan, SUSAN etc.

The algorithms mostly differ in their low-pass filter design. Some of them are even “optimum” detectors, i.e. they give the best possible result – according to an edge criterion or an edge model.

In practice, the differences are not so much of importance. Since some of the techniques used require quite a bit of computational effort, it is sometimes worth taking a somewhat sub-optimal approach and saving a lot of computing time. We have therefore provided options that allow the user to tailor the edge detector to the specific application.

There is a variety of low-pass filters to choose from. From a theoretical point of view, a Gauss filter should be preferred. We have fixed-sized 3x3, 5x5 and 7x7 filters, as well as a recursive filter design with variable size. For edge binarization, there are 3 modes available. For `BinMode=0` gradient values below `MinContrast` are set to zero, all other edge grey values are kept without binarization. The second mode uses a locally variable threshold using moving average for binarization, and the third mode uses a global threshold that is automatically calculated, so that a predefined percentage of all pixels are above the threshold.

In addition, a technique called hysteresis thresholding may optionally be used. Here, two different thresholds are used. The high threshold is used to detect the edge. Due to the high threshold, the detected edge might have some holes. To close the holes, the low threshold is used. The edges are then extended to those edges of the low-threshold image that connect to those of the high-threshold image. The whole procedure produces high quality edge images with less noise than a simple threshold.

edge **calculate image edges**

synopsis

```
I32 edge(image *src, image *dst,
      I32 type, float sigma, I32 BinMode,
      I32 MinContrast, float fthresh, I32 binar_value)
```

description

The function **edge**() performs edge detection on `src`. Various operating modes can be set.

The function uses the following parameters:

`src`	: source image of type = `IMAGE_GREY`
`dst`	: destination image of type = `IMAGE_GREY` or `IMAGE_VECTOR`
`type`	: type of low-pass filter
`sigma`	: low-pass filter size
`BinMode`	: binarization mode
`MinContrast`	: minimum contrast for binarization
`fthresh`	: threshold percentage (only used when `BinMode=2`)
`binar_value`	: binary value for edge output

If the destination image is of type `IMAGE_VECTOR`, the full directional information of the edges image is provided.

The following types of low-pass filters are supported:

type	Low pass filter	Gradient routine
0	no filter	**gradient_2x2**()
1	sobel	**gradient_3x3**()
2	moving average	**gradient_2x2**()
3	Gauss	**gradient_2x2**()
4	Gauss FIR	**gradient_2x2**()
5	ISEF	**gradient_2x2**()

`sigma` specifies the size of the filter and therefore the amount of noise reduction. For `type<2`, `sigma` is not used. For `type=2`, we have `kx=ky=sigma` for the moving average filter. See documentation of `avgm` for further information. For `type=3` and `4`, see documentation of the functions **gauss**() and **gauss_fir**() for the description of `sigma`. For `type=5`, we have `b=sigma`. See documentation of the function **isef**() for further information. Please note, that in this case the value of `sigma` must be less than `1.0`.

type	Low pass filter	sigma
2	moving average	`kx=ky=sigma`
3	Gauss	`0.0 < sigma <= 5.0`
4	Gauss FIR	`sigma = 0.391/0.625/0.812`
5	ISEF	`0.0 < b=sigma < 1.0`

edge() will return `ERR_PARAM`, if the above limitations for sigma are violated.

The following binarization modes are supported:

BinMode	mode	hysteresis threshold
0	no binarization	n/a
1	VC style	no
2	Canny style	no
-1	VC style	yes
-2	Canny style	yes

If `BinMode=0`, edge values below `MinContrast` are still set to zero. For `BinMode=1`, the edge image is subtracted from a 3x3 moving average filter. All pixels above `MinContrast` are set to `binar_value`. If hysteresis threshold is selected, the high threshold is `MinContrast`, the low threshold is `MinContrast/4`.

For `BinMode=2`, the internal global threshold `thr` for the edge image is automatically computed so that always a certain percentage of all pixels, `fthresh`, of the gradient image is above the computed threshold. Reasonable values for `fthresh` are between `0.05` (5 %) and `0.20` (20 %). The automatic threshold also never falls below `MinContrast`. If hysteresis threshold is selected, the high threshold is `thr`, the low threshold is `thr/4`.

return values

The function returns the standard error code. For example, `ERR_PARAM` is returned, if the values for `sigma` and `fthresh` are *outside* the following range:

a) `BinMode = 2` AND `0.0 <= fthresh < 1.0`
b) `type = 3` AND `0.0 < sigma <= 5.0`
c) `type = 5` AND `0.0 < sigma < 1.0`

Since **edge**() calls the functions **gradient_2x2**() or **gradient_3x3**() depending on the value of `type`, the necessary tables should be initialized using the functions

I32 init_gradient_2x2() or
I32 init_gradient_3x3()

To deallocate the memory, the functions

void deinit_gradient_2x2() or
void deinit_gradient_3x3()

should be used.

macros

since **edge**() has quite a few options, there are the following macros to simplify the use:

```
edge_canny(src, dst, binar_value)
edge_fast(src, dst, binar_value)
edge_sobel(src, dst, binar_value)
```

memory

256 KB of heap memory

see also

gradient_2x2(), **sobel**()

4 Programs for gray scale correlation

vc_corr2 **small kernel correlation routine / extended search area + extended kernel**

synopsis

```
I32 vc_corr2(image *a, image *b, I32 mcn,
                        I32 mcr, I32 *x0, I32 *y0)
```

description

The function **vc_corr2**() calculates the normalized gray scale correlation function (NCF) of an image variable a with respect to a correlation kernel or sample b.

NCF may be a useful tool to find a given pattern (sample) in an image. The search result depends heavily on the rotation and the size of the pattern. If more than one pattern similar to the sample is present, the one with the closest match is found. **vc_corr2**() is intended for use with small kernels and small images.

Valid kernel sizes must comply to kx*ky <= 1024, e.g. 32x32 or 16x64. The size of the image (dx,dy) is only limited by heap memory (see below). A good idea is to zoom down sample and image to be searched in using (multiple) **pyramid**() operation(s).

mcn is the minimum required contrast. For mcn=0 the function will find the pattern regardless of its contrast. This may result in false pattern detections in almost homogeneous images where no patterns are present. Therefore a certain minimum contrast is recommended. (local contrast is defined as the variance of gray values in an image region with the size of the kernel)

mcr is the minimum required correlation coefficient. Values for mcr are in the range [0..1024] with 0: no correlation and 1024: absolute identity. Negative correlation coefficients (inverse image) are not supported.

vc_corr2() returns the correlation coefficient for the pattern found. If no pattern is found (due to low contrast or low correlation) it will return ERR_CORR (-100).
The function also returns the x0 and y0 coordinates of the closest match.
If the function detects a format error (e.g. kernel > image or kernel > 1024 pixels), it will return ERR_FORMAT.

memory 8*(dx-kx+1) bytes of heap memory

see also **vc_corr0**(), **vc_corr3**()

vc_corr3 — small kernel correlation routine / 32bit image output

synopsis

```
I32 vc_corr3 (image *src, image *smp,
              image *dst32, I32 mcn)
```

description

The function **vc_corr3()** calculates the normalized gray scale correlation function (NCF) of an image variable `src` with respect to a correlation kernel or sample `smp` and stores the resulting *correlation image* in `dst32`.

Image Variable	Image Type
src	IMAGE_GREY
smp	IMAGE_GREY
dst32	IMAGE_GREY32

NCF may be a useful tool to find a given pattern (sample) in an image. The search result depends heavily on the rotation and the size of the pattern. If more than one pattern similar to the sample is present, this will be reflected in the destination image by several high values. The closest match will have the highest value in the correlation image. **vc_corr3**() is intended for use with small kernels and small images.

Valid kernel sizes must comply to `kx*ky <= 1024`, e.g. `32x32` or `16x64`. The size of the image (`dx,dy`) is only limited by heap memory (see below). A good idea is to zoom down sample and image to be searched in using (multiple) **pyramid**() operation(s).

`mcn` is the minimum required contrast. For `mcn=0` the function will find the pattern regardless of its contrast. This may result in false pattern detections in almost homogeneous images where no patterns are present. Therefore a certain minimum contrast is recommended. If the minimum contrast is not found, the resulting correlation value will be set to zero.

vc_corr3() produces correlation coefficients unnormalized by the search pattern. In order to get normalized correlation values in the range of [0..1023], the following formula may be used:

```
nc  = (float) corr_result *  (float) contr3(smp) / 4194304.0
```

where `corr_result` is one element of the destination image `dst32` and **contr3()** is a function that calculates the inverse contrast for the sample image.

The function returns the standard error format, e.g. `ERR_TYPE` for incompatible types of the source and destination images or `ERR_FORMAT` if the sizes of the images are not within the range (e.g. kernel > image or kernel > 1024 pixels).

memory

`8*(dx-kx+1)` bytes of heap memory

see also

vc_corr0(), **vc_corr2**()

corrcheck — calculate correlation coefficient for two small images

synopsis

```
float corrcheck(image *a, image *b)
```

description

The function **corrcheck**() calculates the normalized gray scale correlation coefficient (NCF) for the two images `a` and `b`. Both images must have the same size. The calculation is performed with maximum possible accuracy using integer and floating-point calculations internally wherever appropriate.

The result is in the range of [-1.0, +1.0]. A value of +1.0 indicates a complete correlation, i.e. identity (except for differences in brightness and contrast). A value of -1.0 also indicates a complete correlation but with inverse contrast.

For comparison of the result with correlation coefficients used by the functions **vc_corr0**() and **vc_corr2**(), the following conversion may be helpful:

```
corrf = corrcheck(a, b) * 1024;
corr  = (I32)((corrf > 0.0) ? corrf + 0.5 : 0.0);
```

Valid image sizes must comply to `kx*ky <= 1024`, e.g. `32x32` or `16x64`.

```
nc  = (float) corr_result *  (float) contr3(smp) / 4194304.0
```

where `corr_result` is one element of the destination image `dst32` and **contr3()** is a function that calculates the inverse contrast for the sample image.

The function returns the standard error format, i.e `ERR_FORMAT` if the sizes of the images are not within the range (e.g. kernel != image or kernel > 1024 pixels).

memory

none

see also

vc_corr0(), **vc_corr2**()

5 Programs for processing binary images in (unlabelled) run length code

rlcmkbit	create run length code of bitplane for an image variable
rlc2	logical functions of 2 images in run length code

rlcmkbit **create run length code of bitplane for an image variable**

synopsis

```
U16 *rlcmkbit(image *a, I32 bit, U16 *rlc, I32 size)
```

description

The function **rlcmkbit**() creates run length code for the image variable a and stores it in memory. `bit` indicates the bitplane for which the run length code is created. It should be a power of two.
A pixel with the corresponding bit being set (`pixel & bit != 0`) is interpreted as white, otherwise as black.
`rlc` is the starting address at which the RLC is stored in memory, `size` is the number of words in memory available for the RLC. If there is not enough space here, creation of the RLC is aborted and the function returns NULL.

This function returns a pointer (U16 *) to the next memory address which is not yet written with RLC. The pointer is aligned to the next integer address. In case of error, it returns NULL.

see also `rlcmk()`, `rlcout()`

memory none

rlc2 **logical functions of two images in run length code**

synopsis

```
U16 *rlc2(U16 *rlca, U16 *rlcb, U16 *dest,
                          U16 * (*func)() )
```

description

The function **rlc2**() makes it possible to calculate any functions of two run length coded images.
`rlca` and `rlcb` pass the memory address of both RLCs. The memory address of the resulting RLC is passed with `dest`. `dest` must be different from `rlca` and `rlcb` (no in-place operations allowed !)
The RLCs to be linked must have the identical format (dx, dy). If this is not the case, then the function returns NULL.
On success, the function **rlc2**() returns the next not yet written memory address for the resulting RLC `dest` (integer aligned).
For execution, it does not matter if the RLC is labelled or unlabelled. In both cases, the result is an unlabelled RLC.
A pointer to the basic function to be executed specifies the nature of the function.

The following macros are available:

	Call	Basic function	Operation
	rlcand(a, b, dest)	**rlc_andf**()	AND
	rlcor(a, b, dest)	**rlc_orf**()	OR
	rlcxor(a, b, dest)	**rlc_xorf**()	XOR
new:	**rlcnand**(a, b, dest)	**rlc_nandf**()	NAND
new:	**rlcnor**(a, b, dest)	**rlc_norf**()	NOR
new:	**rlcequiv**(a, b, dest)	**rlc_equivf**()	EQUIV=NXOR

Of course, you can write your own basic functions. Pass their address (function pointer) to **rlc2**().

memory none

6 Programs for processing binary images in labelled run length code

rlc_label	segment run length code (object labelling)
rlc_qin	object inclusion property for labelled RLC
rlc_nhls	number of holes property for labelled RLC
rlc_arf	RLC area filter for small objects
rlc_select	RLC object selection with guide image
rlc_delete	delete RLC objects using a selection list
rlc_moments	calculate moments of order 0, 1, 2 for labelled RLC
mom_calc_cgx	calculate center of gravity from moments (x-coordinate)
mom_calc_cgy	calculate center of gravity from moments (y-coordinate)
mom_calc_angle	calculate angle of inertial axis from moments (result in degrees)
mom_calc_rad	calculate angle of inertial axis from moments (result in radiants)
mom_calc_ecc	calculate object eccentricity from moments
mom_calc_ellipse_a	calculate ellipse half-parameter a
mom_calc_ellipse_b	calculate ellipse half-parameter b
mom_calc_phi1	calculate Hu moment 1
mom_calc_phi2	calculate Hu moment 2

rlc_label **segment run length code (object labelling)**

synopsis `U16 *rlc_label(U16 *rlc, U16 *slc, I32 mode)`

description The function **rlc_label**() segments the run length code stored starting at the memory address `rlc`.
A pointer to the object number information `slc`, which the function will output, must also be passed to the function - enough memory must be available for the memory needs of the SLC. (The SLC needs (size_of_rlc – 4) bytes of memory)

`mode` indicates a certain neighborhood connectedness for the segmentation according to the following table:

mode	connectedness	macro	library
0	4/4 (standard)	**sgmt**(rlc, slc) **label44**(rlc, slc)	standard
1	8/8	**label88**(rlc, slc)	extension
2	8/4	**label84**(rlc, slc)	extension
3	4/8	**label48**(rlc, slc)	extension

A connectedness of 8/4 means that the white pixels must be 8-connected and the black pixels must be 4-connected to form an object. The above macros are available including the standard **sgmt**() function.

note Please note that since functions like **rlc_qin**() might produce nonsensical results when called with RLC data of different connectedness, we recommend using the standard 4/4 connectedness model (macro **sgmt**(rlc, slc)).

The `slc` pointer is stored in the run length code at address rlc and rlc+1. This indicates a labelled RLC.
The number of objects found and the object numbers for the individual RLC segments are stored in the SLC.
The object numbers begin at 0; a total of 32000 object numbers are allowed. An “object number overrun“ occurs if this number is exceeded.

The return value of this function is the next free memory address (integer aligned).
The function returns NULL in case of an error. This might be a licence error or an object number overrun. In the latter case, the number of objects field in the SLC is also set to zero.

memory 256000 bytes of heap memory (= 8 * 32000)

rlc_qin **object inclusion property for labelled RLC**

synopsis

```
I32 rlc_qin(U16 *rlc, I32 qin[], U32 n)
```

description

The function `rlc_qin()` computes the object inclusion property. It is calculated, which object is contained by which other object. This topological relationship is stored in the array qin[].

If an object touches more than one image boundary, this property cannot be computed for this object. In this case, qin is set to -1 for this object.

The function uses the following parameters:

`rlc`	: pointer to labelled RLC
`qin`	: array for QIN values for each object (output) qin is an array of dimension n which must be allocated by the user
`n`	: size of qin[]

The function returns the number of objects on success or the standard error code.

example

`qin[0] = -1`	object 0 touches more than 1 image boundary
`qin[1] =  0`	object 1 is inside object 0
`qin[2] =  1`	object 2 is inside object 1
`qin[3] =  0`	object 3 is inside object 0

memory

`4*nobj*sizeof(int)` bytes heap memory

see also

rlc_nhls()

rlc_nhls **number of holes property for labelled RLC**

synopsis `I32 rlc_nhls(U16 *rlc, U32 holes[], U32 n)`

description The function **`rlc_nhls`**`()` computes the number of holes for all objects in labelled RLC. Please be aware that even very small objects with a few pixels are counted as holes. It is therefore recommended to clean the image with the function **`rlc_arf`**`()` first.

The function uses the following parameters:

`rlc` : pointer to labelled RLC
`holes` : array for number_of_holes value for each object (output)
`holes[]` is an array of dimension n which must be allocated by the user
`n` : size of `holes[]`

The function returns the number of objects on success or the standard error code.

memory (`4*nobj`+ n)`*sizeof(int)` bytes heap memory

see also **`rlc_qin`**`()`

rlc_arf **RLC area filter for small objects (works on labelled RLC)**

synopsis `I32 rlc_arf(U16 *src, U16 *dst, U32 min_area)`

description The function **`rlc_arf`**`()` deletes all objects of the source RLC with an area less than `min_area`. This can be used to drastically reduce the amount of RLC entries and to speed up the following routines operating on RLC. The function is intended to eliminate small objects. If the value for `min_area` is larger than the horizontal size of the image, it may occur that objects touching the left and right image boundaries must be deleted. In this case nonsensical results may be produced.

The function uses the following parameters:

`src`	pointer to **labelled** source RLC
`dst`	pointer to **unlabelled** destination RLC
`min_area`	objects with an area less than `min_area` are deleted

Note, that the output RLC is **unlabelled**. It might therefore be necessary to call `sgmt()` again for object labelling.
The function returns the number of objects on success or the (negative) standard error code on error.

memory `nobj*sizeof(int)` bytes heap memory

see also **`rlc_mf`**`()`

rlc_select — RLC object selection with guide image

synopsis

```
I32 rlc_select(U16 *rlc, U16 *rlc2, I32 select[],
               U32 n, I32 mode)
```

description

The function **rlc_select**() is used for the selection of objects in a binary image given by rlc with a second binary guide image rlc2. The guide image defines with its white regions where object in the first image are selected. Together with the function **rlc_delete**(), this functionality is sometimes called *morphological reconstruction*.

The format of both RLCs (i.e. the size of the image in x and y) must be identical, otherwise the function returns with ERR_RLCFMT.

The function uses the following parameters:

rlc	pointer to **labelled** source RLC
rlc2	pointer to **unlabelled** guide RLC
select[n]	result array with values for each object indicating the selection: 1: object selected 0: object not selected
n	size of select = maximum number of objects that can be processed
mode	selects the operating mode mode = 0: standard mode mode = 1: black objects are ALWAYS marked as selected.

The function returns the number of objects on success or the standard error code.

memory none

see also **rlc_delete**(), **rlc_arf**()

rlc_delete — delete RLC objects using a selection list

synopsis

```
I32 rlc_delete(U16 *src, U16 *dst, I32 select[])
```

description

The function **rlc_delete**() deletes objects from an RLC using a selection list. The source RLC must be labelled, whereas the result RLC is unlabelled.

The function uses the following parameters:

src	pointer to **labelled** source RLC
dst	pointer to **unlabelled** destination RLC
select[n]	selection array with values for each object indicating the selection: != 0 : object selected 0: object not selected

The function returns the number of objects on success or the standard error code.

memory none

see also **rlc_delete**(), **rlc_arf**()

rlc_moments **calculate moments of order 0, 1, 2 for labelled RLC**

synopsis `I32 rlc_moments(U16 *rlc, moment *mom, U32 n)`

description The function `rlc_moments()` calculates the ***centralized moments*** of order 0, 1 and 2 for the labelled runlength code `rlc`. The centralized moments maybe used to calculate useful features present in the RLC. For example the moment $\mu 00$ is equal to the total pixel area of the object. With moments $\mu 10$ and $\mu 01$, the center of gravity for the object can be calculated by dividing these values by $\mu 00$. Higher moments may be used to calculate translation-, rotation- and scaling-invariant object features.

The output of the function is stored in the struct array `mom`. All values of this struct are stored in our proprietary ***multi-precision integer format***.
Since there are additional functions available to calculate all meaningful features, it is not necessary to use these values directly.

The definition of the moment struct:

```
typedef struct    /* centralized moments          */
  {
  BITN mu00;      /* order 0                      */
  BITN mu10;      /* order 1                      */
  BITN mu01;      /* order 1                      */
  BITN mu20;      /* order 2                      */
  BITN mu11;      /* order 2                      */
  BITN mu02;      /* order 2                      */
  BITN mu30;      /* order 3                      */
  BITN mu21;      /* order 3                      */
  BITN mu12;      /* order 3                      */
  BITN mu03;      /* order 3                      */
  } moment;
```

Moments of order 3 in this struct are reserved for future use. The function `rlc_moments()` only calculates moments up to the second order.

Input variables for the function:

`rlc`	pointer to labelled RLC (input to the function)
`mom`	array of struct (moment struct) / function output
`n`	number of array elements in `mom`. The function checks wether this number is sufficiently high for all objects in the RLC. If not, the function returns with an error code.

On success, the function returns the number of objects in the RLC. This value may be used to address the output struct array. The standard error code is returned on error: RLC is unlabelled or it contains more objects than `n`.

memory `nobj*(tbd)` bytes heap memory

see also **`mom_calc_cgx()`**, **`mom_calc_cgy()`**, **`mom_calc_angle()`**, **`mom_calc_ecc()`**, **`mom_calc_phi1()`**, **`mom_calc_phi2()`**

mom_calc_cgx
mom_calc_cgy

calculate center of gravity from moments

synopsis

```
float mom_calc_cgx(moment *mom)
float mom_calc_cgy(moment *mom)
```

description

The functions **mom_calc_cgx**() and **mom_calc_cgx**() compute the x- and y-coordinates of the center of gravity for the object with the centralized moments given by `mom`. The output of the funtion is a float value with subpixel accuracy.

memory

none

see also

rlc_moments()

mom_calc_angle

calculate angle of inertial axis from moments (result in degrees)

synopsis

```
float mom_calc_angle(moment *mom)
```

description

The function **mom_calc_angle**() computes the angle of the inertial axis (minimum moment of inertia) for the object with the centralized moments given by `mom`. This may be used as the main object orientation, e.g. for robot applications.

The output of the function is a float value with subangle accuracy ranging from 0.0 to 179.9 degrees. 0 means, the object is oriented parallel to the (horizontal) x-axis, 90 means it is parallel to the y-axis. The user must be careful: Since the output ***does not cover*** the full range from 0 to 360 degrees, it is not possible to differentiate mirrored positions of the object. The function is also useless for objects with circular symmetry, e.g. for disks, squares and the like. Use the function `mom_calc_ecc()`, to see if the object has some kind of eccentricity which is necessary for a unique inertial axis.

memory

none

see also

rlc_moments(), **mom_calc_rad**()

mom_calc_rad **calculate angle of inertial axis from moments (result in radiants)**

synopsis `float mom_calc_rad(moment *mom)`

description The function **`mom_calc_rad()`** computes the angle of the inertial axis (minimum moment of inertia) for the object with the centralized moments given by `mom`. This may be used as the main object orientation, e.g. for robot applications.

The output of the funtion is a float value with subangle accuracy ranging from 0.0 to π. 0 means, the object is oriented parallel to the (horizontal) x-axis, $\pi/2$ means it is parallel to the y-axis. The user must be careful: Since the output ***does not cover*** the full range from 0 to 2π, it is not possible to differentiate mirrored positions of the object. The function is also useless for objects with circular symmetry, e.g. for disks, squares and the like. Use the function `mom_calc_ecc()`, to see if the object has some kind of eccenticity which is necessary for a unique inertial axis.

memory none

see also `rlc_moments()`

mom_calc_ecc **calculate object eccentricity from moments**

synopsis `float mom_calc_ecc(moment *mom)`

description The function `mom_calc_ecc()` computes the eccentricity for the object with the centralized moments given by `mom`. This may be used as the main object orientation, e.g. for robot applications.

The output of the funtion is a float value with ranging from 0.0 to 1.0. 0.0 means, the object is totally symmetric, like a disk or a square, 1.0 means the object is totally eccentric, like a needle. The eccentricity for an ellipsoid with diameter a and b, the eccentricity would be (a-b)/(a+b).

memory none

see also **`rlc_moments()`**

mom_calc_ellipse_a **calculate ellipse half-parameters a and b**
mom_calc_ellipse_b

synopsis

```
float mom_calc_ellipse_a(moment *mom)
float mom_calc_ellipse_b(moment *mom)
```

description These functions calculate the half-parameters a and b of the equivalent ellipse for the object with the centralized moments given by `mom`.

memory none

see also `rlc_moments()`, `mom_calc_ecc()`

mom_calc_phi1 **calculate Hu moments**
mom_calc_phi2

synopsis

```
float mom_calc_phi1(moment *mom)
float mom_calc_phi2(moment *mom)
```

description The functions `mom_calc_phi1()` and `mom_calc_phi2()` compute the first two Hu moments.

With the definition:

η20 = μ20 / (μ00*μ00)
η11 = μ11 / (μ00*μ00)
η02 = μ02 / (μ00*μ00)

the first two Hu moments are defined as follows:

φ1 = η20 + η20

φ2 = (η20 - η20)* (η20 - η20) + 4*η11*η11

memory none

see also `rlc_moments()`

rl_ftr3 **calculate object features in the labelled RLC (subpixel version)**

synopsis

```
I32 rl_ftr3(U16 *rlc, ftr *f, U32 n)
```

description

The function `rl_ftr3()` calculates object features of all objects in the labelled RLC.

The following features are calculated:

area:	object area
x_center:	x-coordinate of the center of gravity
y_center:	y-coordinate of the center of gravity
x_cf:	x-coordinate of the center of gravity (subpixel)
y_cf:	y-coordinate of the center of gravity (subpixel)
x_min:	smallest x-coordinate
x_max:	largest x-coordinate
y_min:	smallest y-coordinate
y_max:	largest y-coordinate
x_lst:	last x-coordinate in the last line
color:	object color (0 = black, -1 = white)

The maximum and minimum values of x and y define the bounding box around the chosen object.
The coordinates (x_lst,y_max) specify a point which can serve as the initial value for contour following. The object pixels are guaranteed to be contiguous. In contrast to function `rl_ftr2()` which calculates the center of gravity only with pixel resolution, the function `rl_ftr2()` calculates it also in subpixel resolution and places the result in two additional variables in the data structure which are not used otherwise.

`rlc` is the start address of the labelled run length code in memory.

`f` is a pointer to the feature list (here: a *struct* array), `n` is the maximum number of objects, i.e. usually the dimension of the *struct* array.

The *struct* used has the following structure:

```
typedef struct
  {
  U32 area;        /* object area                */
  U32 x_center;    /* x_center - normalized      */
  U32 y_center;    /* y_center - normalized      */
  I32 x_min;       /* x_min                      */
  I32 x_max;       /* x_max                      */
  I32 y_min;       /* y_min                      */
  I32 y_max;       /* y_max                      */
  I32 x_lst;       /* last x                     */
  I32 color;       /* object color 0 = black     */
  float x_cf;      /* x_center, normalized, float*/
  float y_cf;      /* y_center, normalized, float*/
  } ftr;
```

A pointer to the *struct* array is passed to this function. The pointer need **not** be initialized before you call this function.
The *struct* array is provided with the correct features of all objects after the function is called.

The function returns the standard error code or, if no error occurred , the number of objects in the labelled RLC.

see also `rl_area2(), rlc_feature(), rl_ftr2()`

memory no heap space required

example

```
U16 *rlc, *next;
ftr f[100];

next = rlcmk(&a, 128, rlc, 0x40000);
next=sgmt(rlc,next);
nobj=rl_ftr3(rlc, f, 100);
```

7 Miscellaneous Image Functions

get_component	get image component
equalize	equalize image
set_ovl_false_color	set translucent overlay LUT to false color palette
set_translucent_to_value	set translucent overlay LUT to fixed value
display_directions	display a directional image using overlay
mask_frame	mask a frame with programmable frame width

get_component **get image component**

synopsis `I32 get_component(image *src, image *dst, I32 comp)`

description With this function it is possible to copy just one component out of a multi-component image, e.g. a color or vector image. `src` is an image variable of different types, like `IMAGE_VECTOR` or `IMAGE_RGB`, whereas `dst` must be of `type = IMAGE_GREY`.

The function returns the standard error code

parameters:

`src`	source image variable, any type
`dst`	destination image variable, grey image type
`comp`	component to be copied = 0, 1, 2

memory none

see also `copy()`

equalize **equalize image**

synopsis `I32 equalize(image *src, image *dst)`

description In some cases, a grey image does not cover the complete range of grey values from 0 to 255. With this function the range can be expanded to ease the human interpretation of the image on a computer screen.
First, maximum and minimum grey levels are calculated in the search image. The range between maximum and minimum is then expanded to values between 0 and 255 using a lookup table.

The function returns the standard error code

memory none

see also `look()`

set_ovl_false_color **set translucent overlay LUT to false color palette**

synopsis `I32 set_ovl_false_color(I32 table)`

description This function sets one of the translucent overlay tables to a color palette with equal intensity and saturation and colors covering the complete spectrum

`table` is the number of the translucent table (1, 2, 3); table 1 corresponds to bit 0 in the overlay, table 2 to bit 2 and table 3 to bit 2.

The function returns 0 on success and -1 on error.

set_translucent_to_value **set translucent overlay LUT to fixed value**

synopsis `I32 set_translucent_to_value(I32 t, I32 r, I32 g, I32 b)`

description This function sets one of the translucent overlay tables to a fixed color defined by `r` (red), `g` (green) and `b` (blue).

`t` is the number of the translucent table (1, 2, 3); table 1 corresponds to bit 0 in the overlay, table 2 to bit 2 and table 3 to bit 2.

The function returns 0 on success and -1 on error.

display_directions **display a directional image using overlay**

synopsis

```
I32 display_directions(image *src, I32 thresh,
                       I32  startx, I32 starty)
```

description

For the display of a directional image (e.g. from a vector gradient) a color palette is quite useful. The input image variable must be of type `IMAGE_VECTOR`.This function displays the directions using a false color palette in the translucent overlay. In addition, a threshold may be selected for the magnitude, i.e. all image pixels with a magnitude larger than the threshold are displayed in false colors, all other pixels are black.

The function copies the directions of the source image to a display area in main display memory with identical size and coordinates given by `(startx, starty)`. In the overlay memory, the translucent bit planes 0 and 1 are used for this feature. All other overlay planes are set to zero by this function.

parameters:

`src`	source image variable, `type = IMAGE_VECTOR`
`thresh`	threshold for magnitude binarisation
`startx, starty`	left upper corner coordinate of display

note

The logical pages for display and overlay must be set correctly for this function.

memory

none

mask_frame **mask a frame with programmable frame width**

synopsis

```
void mask_frame(image *src, I32 sx0, I32 sx1,
                I32 sy0, I32 sy1, I32 value)
```

description

This function sets a frame inside the image `src` to the value `value`.

parameters:

`src`	source image variable, type not checked
`sx0`	frame size on left side
`sx1`	frame size on right side
`sy0`	frame size / top
`sy1`	frame size / bottom
`value`	value that is written to frame

note

The logical pages for display and overlay must be set correctly for this function.

memory

none

8 Programs for processing pixel lists

bestline	calculate chi-square bestline for a pixellist
bestcircle	calculate chi-square bestcircle for a pixellist
draw_line	draw a line in normalized floatingpoint form
draw_circle	draw a circle with window-clipping
clip	perform window-clipping for coordinates in pixellist
translate	perform translation of coordinates in pixellist
IntersectionPoints	intersection of a line with image borders
PL_line_stats	line statistics for pixel list
PL_line_ending	line ending for pixel list

bestline — calculate chi-square bestline for a pixellist

synopsis

```
I32 bestline(I32 *xy, I32 N,
             float *cx, float *cy, float *b)
```

description

The function **bestline**() computes a line minimizing the sum of all quadratic distances of the points in the pixellist to the line. Unlike other methods, this function uses the shortest distance to the line (i.e. perpendicular to the line).

The resulting line is defined as:

cx * x + cy *y – b = 0

The function uses the following parameters:

`xy`	pointer to the pixellist, i.e. alternating x- and y-coordinates
`N`	number of points in pixellist
`cx`	pointer to result-parameter cx (float)
`cy`	pointer to result-parameter cy (float)
`b`	pointer to result-parameter b (float)

The pixels in the pixellist may have the full I32 coordinate range, the minimum number of pixels in the list is 2. There is no upper limit. If `N` is less than 2, the function returns `ERR_PARAM`, otherwise `ERR_NONE` (0).

memory none

see also **bestcircle**(), **draw_line**()

bestcircle **calculate chi-square bestcircle for a pixellist**

synopsis

```
I32 bestcircle(I32 *xy, I32 N, float *px,
                            float *py, float *rad)
```

description

The function `bestcircle()` computes a circle minimizing the sum of all quadratic distances of the points in the pixellist to the circle. This function uses the shortest distance to the circle (i.e. perpendicular to the circle).

The resulting circle is defined as:

x = floor (rad * sin(phi) + px + 0.5);
y = floor (rad * cos(phi) + py + 0.5);

The function uses the following parameters:

`xy`	pointer to the pixellist, i.e. alternating x- and y-coordinates
`N`	number of points in pixellist
`px`	pointer to result-parameter px (float)
`py`	pointer to result-parameter py (float)
`rad`	pointer to result-parameter radius (float)

(`px, py`) defines the centerpoint and `rad` the radius of the circle found. Please make sure that the camera you use has a quadratic pixel architecture or use affine transformation to get virtually quadratic pixels. Otherwise circles in reality will be ellipses in image memory and the function will not be able to make a proper fit.

The pixels in the pixellist may have a coordinate range of [-16384, +16383],the minimum number of pixels in the list is 3, the maximum number is 65535. If this range is exceeded, the function returns `ERR_PARAM`, otherwise `ERR_NONE` (0). It is possible, that a singularity occurs within the calculation. This might be the case, if the pixels in the pixellist do not define a circle but a line instead. In this case, the function returns `ERR_SINGULAR`.

memory none

see also `bestline()`

draw_line **draw a line in normalized floatingpoint form**

synopsis

```
I32 draw_line(image *a, float cx, float cy, float b,
                              I32 col, void (*func)())
```

description

The function `draw_line()` draws a line in normalized floatingpoint form in an image variable. The clipping area for the drawing is defined by the size of the image variable. The drawing method is given by the rendering function `func()`.

The line is defined as:

cx * x + cy *y – b = 0

The function uses the following parameters:

`a`	image variable used for drawing the line
`cx`	line-parameter cx (float)
`cy`	line-parameter cy (float)
`b`	line-parameter b = distance from origin (float)

It may turn out that the specified line does not cross the image field at all, in this case, the function returns `ERR_PARAM`, otherwise `ERR_NONE` (0).

2 macros are available:

`draw_lined(a, cx, cy, b, c)`	set pixel to value = `c` (`wp_set32`)
`draw_linex(a, cx, cy, b, c)`	XOR pixel with `c` (`wp_xor32`)

memory none

see also **bestline**()

draw_circle **draw a circle with window-clipping**

synopsis

```
I32 draw_circle(image *a, I32 px, I32 py,
                 I32 rad, I32 col, void (*func)())
```

description

The function `draw_circle()` draws a circle with radius `rad` and center point `(px, py)` in an image variable. The clipping area for the drawing is defined by the size of the image variable. The drawing method is given by the rendering function `func()`.

The function uses the following parameters:

`a`	image variable used for drawing the circle
`px`	center point x-coordinate
`py`	center point y-coordinate
`rad`	circle radius

The function returns `ERR_NONE` (0) on success, `ERR_MEMORY`, when there is a memory allocation error.

2 macros are available:

`draw_circled(a, px, py, r, c)`	set pixel to value = `c` (`wp_set32`)
`draw_circlex(a, px, py, r, c)`	XOR pixel with `c` (`wp_xor32`)

memory (`48 * rad`) bytes of heap memory

see also **`bestcircle`**`()`, **`draw_line`**`()`

clip **perform window-clipping for coordinates in pixellist**

synopsis

```
I32 clip(I32 N, I32 *xy_src, I32 *xy_dst,
      I32 x_min, I32 x_max, I32 y_min, I32 y_max)
```

description

The function `clip()` performs window-clipping for the (x, y) coordinates in the pixellist. The coordinates are copied from `xy_src` to `xy_dst`, if they are in a rectangle defined by `x_min`, `x_max`, `y_min` and `y_max`, i.e.

`x_min <= x < x_max` and
`y_min <= y < y_max`

`N` is the number of coordinates in `xy_src`. The function returns the number of coordinates in the result list `xy_dst`. It is allowed to use the function in-place, i.e. `xy_dst = xy_src`.

memory none

translate **perform translation of coordinates in pixellist**

synopsis

```
void translate(I32 N, I32 *xy_src,
               I32 *xy_dst, I32 mx, I32 my)
```

description

The function **translate**() performs a translation operation for the (x, y) coordinatesin the pixellist `xy_src`. The vector (`mx, my`) is added to all coordinates, the result is written to `xy_dst`.

`N` is the number of coordinates in `xy_src` and `xy_dst`. It is allowed to use the function in-place, i.e. `xy_dst = xy_src`.

memory none

IntersectionPoints **intersection of a line with image borders**

synopsis

```
I32 IntersectionPoints(image *a, float cx, float cy,
               float b, I32 *x0, I32 *y0, I32 *x1, I32 *y1)
```

description

This function calculates the intersection points of a line with the borders of an image (`image *a`). The function uses the following parameters:

`a`	: image variable
`cx, cy, b`	: parameters for line in normalized vector format cx, cy are components of the unit vector normal to the line, b is the distance to the origin
`x0, y0`	: coordinatesof the first intersection point (output)
`x1, y1`	: coordinatesof the second intersection point (output)

The intersection points are calculated in the following order:

left, right, top, bottom

The function returns `ERR_NONE`, if 2 intersection points could be calculated, otherwise it returns `ERR_PARAM`.

memory none

see also **clip**(), **draw_line**()

MarkCross **draw cross into image**

synopsis

```
void MarkCross(image *img, I32 ix,
               I32 iy, I32 color, I32 size)
```

description

The function **`MarkCross`**`()` draws a cross with `2*size` pixels and the grey value `color`.

memory

none

PL_line_stats **line statistics for pixel list**

synopsis

```
I32 PL_line_stats(I32 *xy, I32 nr, vcline *line,
                  float *dmin, float *dmax, float *sigma)
```

description

The function **`PL_line_stats`**`()` calculates some statistics for a line that comes in two representations:

(1) as **`vcline *`**`line` in normalized vector representation
(2) as a pixel-list (**`I32 *`**`xy`)

The normalized vector representation could be the output of the function `bestline()`, the line between the first and the last point of the list or any other preferred line.
The pixel-list `xy` could be the result of the contour algorithm, a Hough-line, extracted with the function `GetHoughPixels()`, or any other source.

The function parses through the points in the list and calculates the closest distance to the theoretical line (**`vcline *`**`line`). It outputs the minimum (`dmin`, should be a negative value or 0, representing a deviation on one side of the line) and the maximum (`dmax`, should be a positive value or 0, representing a deviation on the other side of the line) and the standard deviation `sigma`, which is the square root of the sum of the squared differences divided by the number `nr` of pixels in the list.

return value

`ERR_NONE`

PL_line_ending **line ending for pixel list**

synopsis

```
I32 PL_line_ending(I32 *xy, I32 nr, vcline *line,
                   I32 *imin, I32 *imax)
```

description

The function `PL_line_ending()` calculates the first and the last pixel of the pixel-list in the direction of `vcline *line` and outputs the index of the corresponding coordinate. When the pixel-list is the result of a contour-following or the output of the function `GetHoughPixels()`, the pixels in the list are not ordered in general as a line. The function can also use arbitrary pixel-list and output the minimum and maximum coordinate in a given direction specified by `vcline *line`.

The coordinate of the minimum and maximum pixels are then retrieved by

```
I32 imin, imax, *xy;
I32 minx, miny, maxx, maxy;

PL_line_ending(xy, nr, line, &imin, &imax);

minx = xy[2*imin];
miny = xy[2*imin+1];
maxx = xy[2*imax];
maxy = xy[2*imax+1];
```

return value `ERR_NONE`

9 Geometric tools

The basic data structures for geometric processing are:

```
typedef struct          /* coordinate point                     */
  {
  float x;              /* x coordinate (float)                 */
  float y;              /* y coordinate (float)                 */
  } point;

typedef struct          /* line                                 */
  {
  float cx;             /* cx, cy, b-parameters for             */
  float cy;             /* line in normalized                   */
  float b;              /* vector form:                         */
  } vcline;             /* (cx * x) + (cy * y) - b    = 0       */
```

For lines the normal vector (cx, cy) should be normalized to 1, although the vcline-struct still defines a line if this is not the case. b cannot be used as the distance from the origin to the line and some trigonometric functions could have some problems. Please be aware, that even in the case of a normalized vector, the representation is not unique, since all values could be replaced by their negative, describing the very same line. All functions in this chapter use floating-point values and floating-point calculations.

LineIntersection **calculate intersection point of two lines**

synopsis `I32 LineIntersection(vcline *a, vcline *b, point *r)`

description This function finds the intersection point of 2 lines given in the standard normal form

`cx * x + cy * y - b = 0`

In case of parallel lines, the function returns an error (ERR_SINGULAR).

PointDistance **calculate Euclidean distance between two points**

synopsis `float PointDistance(point *a, point *b)`

description The function calculates the Euclidean distance between two points and returns the result.

PointLineDistance **calculate distance beween a point and a line**

synopsis `float PointLineDistance(point *p, vcline *l)`

description This function calculates the distance beween a point and a line and returns the result.

LinePerpendicular **calculates a line perpendicualar to a given one through a point**

synopsis `void LinePerpendicular(point *p, vcline *l, vcline *r)`

description This function calculates a line perpendicualar to a given one through a point. The result is stored as `vcline *r`.

LineParallel **calculates a line parallel to a given one through a point**

synopsis `void LineParallel(point *p, vcline *l, vcline *r)`

description This function calculates a line parallel to a given one through a point. The result is stored as `vcline *r`.

Norm **calculates the norm (length) of a vector (point)**

synopsis `float Norm(point *a)`

description This function calculates the norm or length of a vector in point representation and returns the result as a float value.

AngleP **calculates the angle of a vector (point)**

synopsis `float AngleP(point *a)`

description This function calculates the angle between the horizontal x-axis and the vector `a` and returns the result in radiants. A value of 0 is output for a horizontal vector pointing right. The result increases for a clockwise rotation up to a value of 2π (360deg) and is always positive.

Angle **calculates the angle of a line**

synopsis `float Angle(vcline *a)`

description This function calculates the angle between the horizontal x-axis and line `a` and returns the result in radiants. A value of 0 is output for a horizontal line. The result increases for a clockwise rotation up to a value of π (180deg) and is always positive.

LineAngle **calculates the angle between two lines**

synopsis

```
float LineAngle(vcline *a, vcline *b)
```

description

This function calculates the angle between two lines `a` and `b` and returns the result in radiants. A value of 0 is output if both lines are parallel. In all other cases, the result is the angle that line a must be rotated clockwise to be parallel to b.
The result increases for a clockwise rotation up to a value of π (180deg) and is always positive.

LineParameters **calculates the line parameters using two points**

synopsis

```
void LineParameters(point *p1, point *p2, vcline *line)
```

description

This function calculates the line parameters for a line running through two points. If the two points are too close together, (distance < 5.0e-7), all line parameters will be set to 0.0.

10 Hough Transform

The Hough Transform is a tool to identify a certain class of shapes in a given image. It is mostly used for finding lines, but more complex shapes like circles and ellipses may also be searched with special versions of the Hough Transform. The general idea is to use an accumulator space and a voting procedure.
The Hough Transform for lines uses the polar (or normal) representation of a line. The accumulator space is 2-dimensional and has the following features

parameter	description	dimension	size *)	origin
ρ	distance from origin	horizontal	`sqrt(dx*dx+dy*dy)+1`	centered
φ	line angle	vertical	`128`	top

*) the size for ρ is always rounded up to the next even number

In each column for ρ, there are 128 bins for φ, covering the range from 0° to 180°, which results in an angular resolution of 1.4°.

The Hough Transform is typically applied to edge images. Although it is possible to use images with grey levels as an input, it is recommended to set unused image pixels to zero, since this provides a significant speed improvement. This results in a particularly good performance for binary images, where the edge pixels are set to some arbitrary value in the range of 1..255 and all other pixels are zero.

The Hough Transform can be applied to an image using the function **`HoughTransform`**`()`. For each non-zero pixel of the input image with coordinates(x, y), the Hough Transform calulates the function

$$\rho = x * \sin(\varphi) + y * \cos(\varphi)$$

for all 128 values of the parameter φ. The corresponding bins in the accumulator space are incremented by the grey value of the pixel (x, y).

If there are linear structures in the image, there will be corresponding peaks in the accumulator space with a height (peak strength) proportional to the number of pixels of the line. It does not matter if the line is solid, dotted or randomly distributed along its way, only the number of pixel counts for its representation in accumulator space (Hough space).

The second main task is therefore to identify peaks in Hough Space. This is done by the function **`FindHoughLine`**`()`.

The following shows the control struct which selects the features of the Hough Transform and the line finding algorithm:

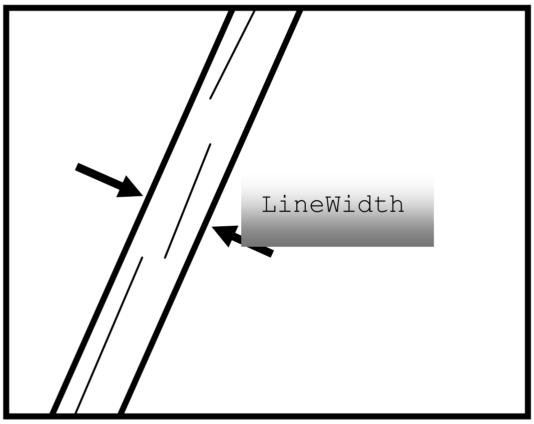
```
typedef struct
{
  I32 HoughMode;              /* Hough operating mode = 0-3       */
  I32 LineNumber;             /* number of lines for output       */
  I32 delta_phi;              /* phi tolerance                    */
  I32 MinPix;                 /* threshold for number of pixels   */
  I32 LineWidth;              /* width of lines to be searched    */
  I32 bestfit;                /* bestfit line flag 1=bestfit      */
  struct hlstruct *maxptr;    /* pointer to active lines list     */
  struct hlstruct *empty;     /* pointer to empty line list       */
  }
HoughControl;
```

Not all of the parameters are used in both **HoughTransform**() and **FindHoughLine**().

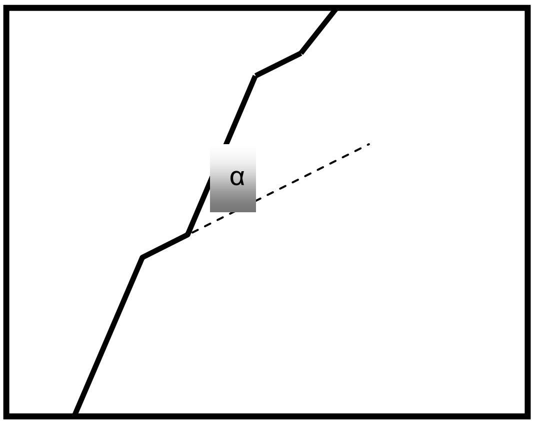
parameter	description	used in function:	remark
HoughMode	Hough operating mode	both	user input
LineNumber	Number of lines requested	FindHoughLine()	user input
delta_phi	see text	both	user input
MinPix	see text	FindHoughLine()	user input
LineWidth	see text	FindHoughLine()	user input
bestfit	perform bestfit (1=yes, 0=no)	FindHoughLine()	user input
maxptr	pointer to first element of output list	FindHoughLine()	set by HoughInit2()
empty	reserved for internal use	FindHoughLine()	set by HoughInit2()

An important parameter for the function **FindHoughLine**() is MinPix. This is basically a threshold for the Hough space, corresponding to the minimum number of pixels for a line. Unfortunately, due to rounding errors and possibly to due to the different grey values in the original image, this value is far from exact. There may be up to a factor of 2 difference to the correct number of pixels. So this value must be set low enough, in order not to miss a line. On the other hand, if MinPix is set too low, the routine must search through a large number of tiny peaks, wasting computing time.

Most real lines are not straight lines. To account for this and to robustly detect somewhat noisy, wavy or disturbed lines, 2 parameter have been added to tune the search characteristics.



All pixels in the stripe with width of LineWidth count for a single line



If α ≤ delta_phi, all pixel count for a single line, otherwise multiple lines are output. Note that the condition on the left side must also be fulfilled.

`LineWidth` defines a channel of a certain width, where the pixels in the original image vote for this line in Hough space.

`delta_phi` is the amount of angular tolerance for the line detection.

The accuracy for the detection of lines with reasonably good quality in the original image is about 0.2 degrees. If this is not enough, setting `bestfit` to 1 will force **FindHoughLine**`()` to calculate a least squares fit for all lines. For this fit, only pixels in the area specified by `LineWidth` and with a direction specified by `delta_phi` will be taken into account. As a result, outliers will not play a role for the bestfit procedure. For the bestfit, pixels with a value ≠ 0 are used, pixels with value = 0 are ignored. Therefore, it does not make much sense using grey images as an input, if `bestfit = 1`, unless values below a certain threshold are set to zero. We recommend using binary images, when the bestfit feature is selected.

It must be remarked, that the accuracy of the combined Hough- and bestline algorithms (sometimes even without the bestline) is so high, that the geometric distortion of the lens used for taking the picture might play an important role. This is certainly the case for wide angle lenses. If wide angle lenses are required for the application, the use of a correcting geometric image transform before performing the Hough Transform might be considered.

Since a number of functions work on the same data, unexpected behavior might result, if the operating modes are changed between calls of those functions. It is therefore a good idea, to set the operating modes in `control` at the beginning of the program and never change parameters afterwards. For setting default values, you can use the function **HoughDefaults**`()`.

In the following we present an example program:

```
#define BINAR_VALUE    (1)

// allocate memory for the 32bit Hough image
ImageAllocate(&hough32, IMAGE_GREY32, HoughCalcDx(&src), 128);

// set defaults
HoughDefaults(&control);

// threshold for number of pixels in line
control.MinPix = 20 * BINAR_VALUE;

// allocate memory and initialize
HoughInit();
HoughInit2(&control);

// binarize image with 0 and 1
binarize(&src, &src, thr, 0, BINAR_VALUE);

// do the Hough Transform
HoughTransform(&src, &hough32, &control);

// search Hough space for lines
FindHoughLine(&hough32, &src, &control);

// get the number of elements in the list
nr = HoughRank(control.maxptr, HL_VALUE, 0);

// step through the list of lines and draw them in the original image
p = control.maxptr;

while(p != NULL)
  {
  draw_lined(&src, p->cx, p->cy, p->b, 255);
  p = p->next;
  }


// deallocate memory
HoughDeinit();
HoughDeinit2(&Control);
ImageFree(&Hough32);
```

If the program must work in a loop, we would have started the loop with the function **binarize**() ending with the while-loop that draws the lines.

FindHoughLine() outputs its results as follows: After execution, the `control->maxptr` points to the chained list of line descriptors, which have the following format:

```
typedef struct hlstruct
{
  struct hlstruct *next;            /* pointer to next element in list   */
  I32    state;                     /* internal state                    */
  I32    value;                     /* line detection value              */
  I32    phi;                       /* line angle                        */
  I32    rho;                       /* distance from center of image     */
  I32    strength;                  /* line detection strength           */
  float cx;                         /* cx, cy, bparameters for           */
  float cy;                         /* line in normalized                */
  float b;                          /* vector form                       */
}
HLine;
```

`next` points to the next element in the list or to `NULL` for the last element in the list. `value` is the peak value in Hough space for the line (with a fixpoint integer format of 28.4). `phi` and `rho` are the line angle and distance from the origin, i.e. the variables φ and ρ of the Hough transform. The latter two values are scaled by 256, i.e. they have the fixpoint integer format of 24.8 .

`strength`, like `value` is the number of pixels for the line multiplied with their grey-value, but it takes into account all pixels in a stripe with width = `LineWidth` in the direction `phi` +/- `delta_phi`. It is therefore a better characterization of the line. `cx`, `cy` and `b` are the parameters for the line in the normalized vector form defined by

$$cx * x + cy * y - b = 0$$

The origin of the coordinate system is located at the upper left corner of the image variable `src`, like usual for most functions, whereas the origin for `phi` and `rho` is right in the middle of `src`.

The line-list is sorted by the parameter `value`, highest value first.

We recommend using only the output parameters `strength`, `cx`, `cy` and `b`. The parameters `value`, `phi` and `rho` are more for internal use inside the function. If the user selects the chi-square bestfit feature, this also influences only `cx`, `cy` and `b`.

If the user prefers the list to be sorted according to their `strength`, this can be accomplished using **HoughSortLine**().

HoughTransform — Hough Transform for lines

synopsis

```
I32 HoughTransform(image *src, image *hough32,
                              HoughControl *control)
```

description

This function calculates the Hough Transform for the source image `src`. The 32bit image `hough32` is the output of the function. Operating modes are selected with `control`.

The function uses the following parameters:

`src`	: image variable of type `IMAGE_GREY` or `IMAGE_VECTOR`
`hough32`	: Hough accumulator image of type `IMAGE_GREY32`
`control`	: control struct; the following parameters used for this function:
`HoughMode`	: operating mode, must be 0 or 3
`delta_phi`	: +/- angular detection tolerance in units of 1.4 degrees

The size of `hough32` depends on the size of the input image `src`:

```
hough32->dx    = sqrt(src->dx * src->dx + src->dy * src->dy) + 1
hough32->dy    = 128
```

`hough32->dx` is then *rounded* to the next higher even number. For convenience, we provide the function

```
I32 HoughCalcDx(image *src)
```

which returns the horizontal hough image size for a given source image `src`.

It is recommended to set `HoughMode` to 0 and to use an image of type `IMAGE_VECTOR`. This allows you to make use of the parameter `delta_phi` and also results in a considerable speed advantage. Setting `HoughMode` to 3 or using an image of type `IMAGE_GREY` as input, automatically sets `delta_phi` to 64 (= +/- 64), essentially switching off the angular tolerance feature.

The function requires some tables for the calculation which can be allocated and initialized using the function

```
I32 HoughInit()
```

This function returns the standard error code. To deallocate the memory, the function

```
void HoughDeinit()
```

should be used.

`HoughTransform()` also works, if **`HoughInit()`** is not called beforehand. It does the memory allocation and initialisation, but this may take some time, the first time the function is called, so the user might like to do the initialisation at the time when the program starts to guarantee equal processing times.

HoughTransform() returns the standard error code. An error is detected when:

a) the image variables do not have the proper type (ERR_TYPE)
b) hough32->dx < **HoughCalcDx**(src) (ERR_FORMAT)
c) hough32->dy ≠ 128 (ERR_FORMAT)
d) the function is out of memory (ERR_MEMORY)

memory (640 + 4*src->dx) bytes including **HoughInit**()

see also **FindHoughLine**()

FindHoughLine — find lines in Hough Transform

synopsis

```
I32 FindHoughLine(image *hough32, image *src,
                                  HoughControl *control)
```

description

This function searches the 32bit image hough32 for peaks representing lines in the original image src. hough32 should be the result of the function **HoughTransform**(). Operating modes are selected with control.

The function uses the following parameters:

hough32	: Hough accumulator image of type IMAGE_GREY32
src	: image variable of type IMAGE_GREY or IMAGE_VECTOR
control	: control struct; the following parameters used for this function:
HoughMode	: operating mode, must be 0 or 3
delta_phi	: +/- angular detection tolerance in units of 1.4 degrees
LineNumber	: maximum number of output lines
MinPix	: minimum number of pixels for line times grey value *)
LineWidth	: width detection tolerance for lines
bestfit	: perform chi-square bestfit (1=yes, 0=no)

*) MinPix is an approximate value for the minimum number of pixels in a line multiplied with their pixel value. It is used as a threshold for speeding-up the algorithm.

The function requires some heap memory for the storage of the line descriptors and for some tables. The memory allocation and initialization should be done using the functions

```
I32 HoughInit()
I32 HoughInit2(HoughControl *control)
```

Be sure to set the control struct control before calling **HoughInit2**(), since the amount of memory allocated, depends on the parameter LineNumber. This function returns the standard error code.

To deallocate the memory, the functions

```
void HoughDeinit()
void HoughDeinit2(HoughControl *control)
```

should be used.

FindHoughLine() also works, if **HoughInit**() and **HoughInit2**() is not called beforehand. It does the memory allocation and initialization, but this may take some time when the function is called for the first time, so the user might like to do the initialization at the time when the program starts to guarantee equal processing times.

The function outputs a chained list of line descriptors with the pointer `(Hline *)control->maxptr` pointing to the start of the list. The list is sorted according to the line parameter `value`.

FindHoughLine() returns the standard error code. An error is detected when:

a) the image variables do not have the proper type (`ERR_TYPE`)
b) `hough32->dx < ` **HoughCalcDx**`(src)` (`ERR_FORMAT`)
c) `hough32->dx` is odd (`ERR_FORMAT`)
d) `hough32->dx < 96` (`ERR_FORMAT`)
e) `hough32->dy ≠ 128` (`ERR_FORMAT`)
f) the function is out of memory (`ERR_MEMORY`)
g) parameters are out of range (`ERR_PARAM`)

The function has two internal error states which it may return on occasion:

h) internal out-of-memory state (`ERR_HOUGH0`). In this case, it might help to increase `LineNumber`.
i) maximum iteration error (`ERR_HOUGH1`). This error can only occur, if the `control`-struct is inconsistent for the functions **HoughTransform**() and **FindHoughLine**() or otherwise some tables have been damaged.

remark **FindHoughLine**() changes the contents of `hough32`.

memory

`640`	bytes for **HoughInit**() tables
`2*LineNumber*sizeof(HLine)`	bytes for **HoughInit2**()
`< 128`	kBytes for **FindHoughLine**() routine

see also **HoughTransform**()

HoughDefaults **set defaults for the Hough Transformation**

synopsis

```
void HoughDefaults(HoughControl *control)
```

description

This function sets default values for the Hough transformation. The parameters are set as follows:

```
control->HoughMode   = 0;
control->LineNumber  = 100;
control->delta_phi   = 5;
control->MinPix      = 5;
control->LineWidth   = 5;
control->bestfit     = 0;
```

memory none

see also **HoughTransform**(), **FindHoughLine**()

HoughSortLine **sort line list according to different sorting criteria**

synopsis

```
void HoughSortLine(HLine **listptr, I32 offset)
```

description

With this function the user can sort the line list according to different sorting criteria. `listptr` is a handle for the line list and `offset` is the element number in the struct. `vclib.h` provides a number of predefined values for `offset` that you can choose from:

```
#define HL_VALUE      (2)
#define HL_PHI        (3)
#define HL_RHO        (4)
#define HL_STRENGTH   (5)
```

as an example the line list is sorted according to line strength:

example

```
HoughSortLine(&(control->maxptr), HL_STRENGTH);
```

Please note, that **HoughSortLine**() might change the value of `control->maxptr` if it needs to point to a different first element in the list

memory none

see also **HoughRank**()

HoughRank **get number of lines above a certain value**

synopsis

```
I32 HoughRank(HLine *listptr, I32 offset, I32 value)
```

description

With this function it is possible to get the number of lines with a `value` above a certain threshold, e.g. the number of lines above a certain `strength`.
Please note, that the lines *have to be sorted first* with the function **`HoughSortLine()`** according to the selected criterion.

`listptr` is a pointer to the line list and `offset` is the element number in the struct. `vclib.h` provides a number of predefined values for `offset` that you can choose from:

```
#define HL_VALUE      (2)
#define HL_PHI        (3)
#define HL_RHO        (4)
#define HL_STRENGTH   (5)
```

For the struct values that are always positive, namely `line->value` and `line->strength`, calling **`HoughRank( , , 0)`** will return the total number of lines in the list.

as an example the rank is computed according to line strength:

example

```
HoughSortLine(&(control->maxptr), HL_STRENGTH);
count = HoughRank(control->maxptr, HL_STRENGTH, 200);
```

memory none

see also **`HoughSortLine()`**

GetHoughPixels **extract xy-list from Hough source image**

synopsis

```
I32 GetHoughPixels(image *Src, HLine *line,
                        HoughControl *Control, I32 *xy)
```

description

The Hough Transform can detect lines in an image which can consist of an arbitrary set of pixels on a line. This function returns an exact list of the pixels which contribute to the Hough line.

parameters

`Src`	: source image (`IMAGE_VECTOR` or `IMAGE_GREY`)
`line`	: line descriptor with values for `phi,cx,cy,b`
`Control`	: Hough Control struct (always use same values for all functions)
`xy`	: xy co-ordinate list (result)

return values number of pixels found (positive value) or standard error return (negative)

memory

`4*(dx + 2*dy + 4*max(dx, dy) + 2*width*(dx + dy) + 6)`
bytes of heap memory

see also **`FindHoughLine()`**

11 Hough Transform for Circles

Similar to the Hough Transform for lines, it is possible to define a Hough Transform for circles. The parameter space would be three-dimensional, since a circle is defined by three parameters: (x0, y0) and r. This adds another order of magnitude for the memory space and computation time requirements. In order to reduce memory space and computation time, we calculate the center (x0, y0) of the circle first. The radius is calculated internally as a second step. We also find only one circle at a time not a multiple, so it is up to the user to call the function as long as necessary to locate all the relevant circles. This ensures rapid execution times.

Like the Hough Transform for lines, we use a 2-dimensional accumulator space and vote for the most likely centerpoints of circles. In most cases the accumulator has the same dimensions as the original image. The Hough-space may however be larger, which is important, if the user wants to locate circles with centerpoints outside the active field.

Like the Hough Transform for lines, it is not important that the structures in the images are connected full circles. They could be any part of a circle, like a half- or quatercircle, fully or partly connected. However, unlike the transform for lines, it is not possible to detect circles consisting only of single isolated pixels. Please also keep in mind, that if you only have a small segment of a circle smaller than a quatercircle, the centerpoint must be extrapolated from the data, so centerpoint and radius will have poor accuracy.

The Hough Transform for circles uses the following twodimensional accumulator space:

parameter	**description**	**dimension**	**size *)**	**origin**
x0	centerpoint x-position	horizontal	`dx` (default)	left
y0	centerpoint y-position	vertical	`dy` (default)	top

*) Hough space may be smaller or larger than the source image to extend or restrict the search area

The Hough Transform is typically applied to edge images. Although it is possible to use images with grey levels as an input, it is recommended to set unused image pixels to zero, since this provides a significant speed improvement. This results in a particularly good performance for binary images, where the edge pixels are set to some arbitrary value in the range of 1..255 and all other pixels are zero.

The Hough Transform for circles can be applied to an image using the function **`HoughCircleTransform`**`()`.

The function increments the bins in Hough space for the possible centerpoints of potential circles by the grey value of the pixel (x, y).

If there are circular structures in the image, there will be corresponding peaks in the accumulator space with a height (peak strength) proportional to the number of pixels on the circle. It does not matter if the circle is solid, or randomly distributed along its way, only the number of pixel counts for its representation in accumulator space (Hough space). Isolated pixels, however, cannot contribute to the accumulator space.

The second main task is to identify peaks in Hough Space. This is done by the function **FindHoughCircle**().

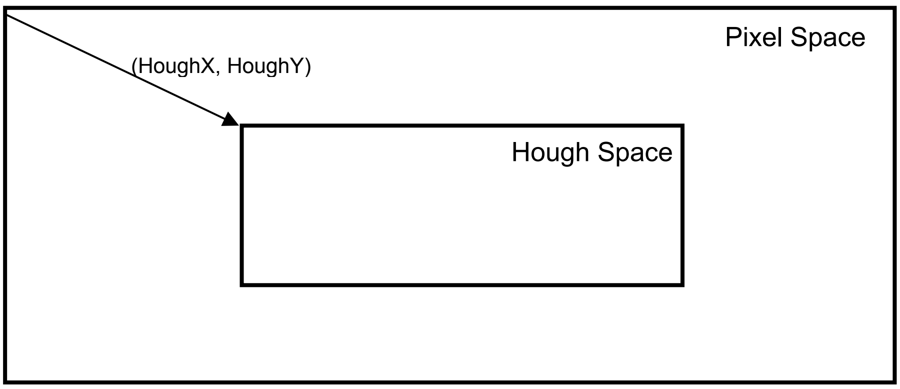
The following shows the control struct which selects the features of the Hough Transform and the circle finding algorithm:

```
typedef struct
{
  I32   Thresh;       /* threshold for binarization of source image      */
  I32   MinRad;       /* minimum radius for circle detection             */
  I32   MaxRad;       /* maximum radius for circle detection             */
  I32   Error;        /* error code                                      */

  /* hough parameters (with relative hough position to the image)        */
  I32   HoughX;       /* relative starting point x-coordinate            */
  I32   HoughY;       /* relative starting point y-coordinate            */

  I32   bestfit;      /* 0: no bestcircle, 1: bestcircle approximation   */

  + reserved additional parameters for internal use

}
HCirclePar;
```

The parameters of this struct should be kept constant throughout the detection cycle including the initialization. Since additional internal parameters are used, the function **DefaultParHoughCircle**() must be called first to initialize the struct to the default parameters. The above parameters may then be changed to taylor the transform to specific requirements.
The function **InitHoughCircle**() must be called after the struct has been set to the correct values. **InitHoughCircle**() initializes some tables depending on the values of `MinRad` and `MaxRad`, the minimum and maximum radius defining the range of circles to be searched for.

`(HoughX, HoughY)` is the relative position of the Hough space in respect to the pixel space. The default value is (0, 0). Together with same dimensions for pixel- and Hough space, this results in the full coordinate range for the centerpoints of the circle. However, the centerpoints of the circle may easily lie outside the original pixel range. In this case, the Hough space may be made larger and the vector `(HoughX, HoughY)` should have negative values for `HoughX` and/or `HoughY` .

On the other hand, if it is clear from the application, that the centerpoints are restricted to a certain range, the Hough space may be made smaller with the positive vector `(HoughX, HoughY)` pointing to the left upper corner of the detection range.



This results in faster processing times since a smaller Hough Space must be searched.

HoughCDefaults — **set Hough Circle parameters to default**

synopsis

```
I32 HoughCDefaults(HCirclePar *HCP)
```

description

This function sets all the internal and external parameters of the control structure HCP. The function must be executed before any other function for the Hough Circle Transform may be called. The following values for the external parameters are set:

```
HCP->Thresh  = 128;     /* threshold for binarization  */
HCP->MinRad  = 2;       /* minimum radius for circle   */
HCP->MaxRad  = 256;     /* maximum radius for circle   */
HCP->Error   = 0;       /* error code                  */
HCP->HoughX  = 0;       /* relative starting point x   */
HCP->HoughY  = 0;       /* relative starting point y   */
HCP->bestfit = 0;       /* 0: no bestcircle appr.      */
```

The external parameters may be changed to some appropriate value afterwards. Please be sure to keep the parameters fixed for a complete cycle of the Hough Transform including the functions

```
InitHoughCircle()
HoughCircleTransform()
FindHoughCircle()
```

return values — standard error return

memory — none

InitHoughCircle — **initialize Hough Circle Transform**

synopsis

```
I32 InitHoughCircle(HCirclePar *HCP)
```

description

This function basically allocates and sets some tables necessary for the Hough Circle Transform. The function may take several hundred milliseconds depending on the value of MaxRad. Therefore it is recommended to call it only once at the start of the user program. Please be aware, that depending on the value of MaxRad the function may require some memory space. For the same reason, the function must be called whenever the value of MaxRad changes.

return values — standard error return

memory — `4*(MaxRad+1)*(MaxRad+1) + 640` bytes of heap memory

see also — **DeinitHoughCircle**()

DeinitHoughCircle **deinitialize Hough Circle Transform**

synopsis

```
I32 DeinitHoughCircle(HCirclePar *HCP)
```

description

This function releases memory space used for the tables and other structures previously allocated by the functions

InitHoughCircle()
HoughCircleTransform()

return values

standard error return

memory

none

see also

InitHoughCircle(), **HoughCircleTransform**()

HoughCircleTransform **Hough Transform for circles**

synopsis

```
I32 HoughCircleTransform(image *ImgVec,
                         image *Hough32, HCirclePar *HCP)
```

description

This function calculates the Hough Circle Transform for the source image `ImgVec`. The 32bit image `Hough32` is the output of the function. Operating modes are selected with `HCP`.

parameters

`ImgVec`	: image variable of type `IMAGE_VECTOR`
`Hough32`	: Hough accumulator image of type `IMAGE_GREY32`
`HCP`	: control struct; must be set with **HoughCDefaults**()

`ImgVec` must be a vector image, typically as the result of function **edge**() .

The size of `Hough32` is typically the same as for the source image `ImgVec`.

The function requires some tables for the calculation which can be allocated and initialized using the function

InitHoughCircle()

This function returns the standard error code. To deallocate the memory, the function

DeinitHoughCircle()

should be used.

return values

HoughCircleTransform() returns the standard error code.
An error is detected when:

a) the image variables do not have the proper type (`ERR_TYPE`)
b) the function is out of memory (`ERR_MEMORY`)

memory

see also **FindHoughCircle**()

FindHoughCircle **find circles in Hough Circle Transform**

synopsis

```
I32 FindHoughCircle(image *Hough32,
                    HCirclePar *HCP, HCircle *Circle)
```

description

This function searches the 32bit image `Hough32` for peaks representing circles in the original image `ImgVec`. `Hough32` should be the result of the function **HoughCircleTransform**(). Operating modes are selected with `HCP`. The result is placed in the struct `Circle`. Unlike the function **FindHoughLines**(), this function returns the result one by one, i.e. it must be called several times to find all circles in an image.

The function uses the following parameters:

`Hough32`	: Hough accumulator image of type `IMAGE_GREY32`
`ImgVec`	: image variable of type `IMAGE_VECTOR`
`HCP`	: control struct
`Circle`	: result struct

The result struct has the following definition:

```
typedef struct hcstruct
{
  I32   x0;           /* centerpoint x-value             */
  I32   y0;           /* centerpoint y-value             */
  I32   r;            /* circle radius                   */
  float x0f;          /* centerpoint x-value float       */
  float y0f;          /* centerpoint y-value float       */
  float rf;           /* circle radius float             */
  I32   strength;
}
HCircle;
```

The circle parameters are available both in integer (**I32**) and floating-point format. The floating-point format is quite helpful when using the bestcircle approximation (bestfit=1). The parameter `strength` gives an indication of the amount of pixels in the image belonging to the circle found. Like for the linear Hough Transform, it is more a rough estimate, a proportional value changing with the number of pixels contributing to the circle than an absolute pixel-counter.

The sequence of circles calculated by this function when called multiple times is only partly dependent on the `strength` parameter. First, the function searches for the point in Hough space with the largest value, representing the centerpoint most frequently used. At this point it is, however, not clear if the large value comes from a large circle with many pixels or several smaller concentric circles with fewer pixels each. After the function has located the centerpoint, it then calculates the radius of the circle with the best pixel coverage, i.e. the best ration of pixels per radius.

The function requires some heap memory for the storage of the line descriptors and for some tables. The memory allocation and initialization should be done using the function

InitHoughCircle()

To deallocate the memory, the function

DeinitHoughCircle()

should be used.

return values **FindHoughCircle**() returns the standard error code. An error is detected when:

a) the image variables do not have the proper type (ERR_TYPE)
b) the function is out of memory (ERR_MEMORY)
c) MinRad is out of range (ERR_PARAM)

remark **FindHoughCircle**() changes the contents of Hough32.

memory

see also **HoughCircleTransform**()

FindMaxImage32 locate maximum value of 32bit image

synopsis `I32 FindMaxImage32(image *Src32, I32 *ix, I32 *iy)`

description This function locates the maximum value in a 32bit image. This may be helpful for finding lines or circles in Hough space. It also gives an indication for the range of values in Hough space. The Vision Components Hough Transform does not rely on this function for the localization, but uses more sophisticated algorithms for this purpose.

The function returns the maximum value and stores its position in (ix,iy).

memory none

ImgConvert_I32_U8 convert image from I32 to U8 representation

synopsis `I32 ImgConvert_I32_U8(image *Src32, image *Dst, I32 Pow)`

description This function converts an image of type IMAGE_GREY32 to an image of type IMAGE_GREY, i.e. to an U8 representation. The image is scaled to its maximum if Pow=0. For other values of Pow the function multiplies the destination by Pow and clips the output to the U8 range.

return values The function returns the standard error code.

memory none

12 Fast Fourier Transform

The Fast Fourier Transform Algorithm (FFT) is an efficient algorithm to compute the discrete Fourier transform and its inverse. In the following we have FFT functions for 2D image data as well as 1D vectors.

vc_fft **perform the 2D FFT (16 bit)**

synopsis

```
I32 vc_fft(image *src, image *dst, I32 *mean, I32 *scale)
```

description

This function calculates the 2D FFT of an image.

Image `src` may be one of the following types:

```
IMAGE_GREY
IMAGE_GREY16
IMAGE_CMPLX16
```

Image `dst` must be of type `IMAGE_CMPLX16`. This image type stores two 16-bit values for real and imaginary part in two consecutive memory position aligned on a 32bit boundary. Real and imaginary parts for the first pixel are stored as follows:

```
I16 real, imag, *p;
p = (U16 *)dst->st;
real = p[0]; imag = p[1];
```

It is possible to use the function in-place, i.e. source and destination images can be identical.

The size of the images must be a power of 2 in x and y dimension or otherwise the FFT will be performed in a smaller subwindow.
Reasonable values for `dx` and `dy` are between 16 and 2048.

Since the function only uses a 16bit FFT for both directions, a method for best accuracy has been implemented. This includes handling the mean or average value `mean` separately. The function also calculates a scale factor (`scale`) depending on the number of bits used during the procedure.
`mean` and `scale` are results calculated by the function.

The function automatically allocates space for some internal tables. Each time the function is called with a new value for `dx` or `dy`, a new table is allocated which will be held in memory until the tables are released calling the function

```
void FFTDeinitTwiddle()
```

Since the function also computes the tables, the first call of **vc_fft**() with new values for dx or dy may take some processing time. If the user requires constant execution times, it is possible to call the function

```
I32 FFTInitTwiddle(I32 size)
```

on program start for all sizes dx and dy (powers of 2) of the images that need to be processed. This function allocates and computes all tables necessary for the forward and inverse FFT and will output the standard error code if the system is out of memory.

return values **vc_fft**() returns the standard error code.

memory 4*dx*dy bytes + several tables allocated by **FFTInitTwiddle**(dx)

see also **vc_ifft**()

vc_ifft — perform the inverse 2D FFT (16 bit)

synopsis

```
I32 vc_ifft(image *src, image *dst,
                        I32 *mean, I32 *scale)
```

description This function calculates the inverse 2D FFT of an image.

Images src and dst must be of type IMAGE_CMPLX16.

See the documentation of **vc_fft**() for further details, since both functions are almost identical.

GenCplxImg — copy image and change type to IMAGE_CMPLX16

synopsis

```
I32 GenCplxImg(image *src, image *dst, I32 *mean)
```

description Image src may be one of the following types:

IMAGE_GREY
IMAGE_GREY16
IMAGE_CMPLX16

Image dst must be of types IMAGE_CMPLX16. If mean=NULL, the function just copies image src to image dst. If src has type IMAGE_CMPLX16, this will be a 100% copy. In all other cases the real part of dst will consist of the copied values from src, the imaginary part will be set to 0.

If mean!=NULL, the mean or average value of the source image will be calculated

FindMaxCplx **find maximum in complex image**

synopsis `I32 FindMaxCplx(image *src, I32 *ix, I32 *iy)`

description This function finds the maximum of the complex absolute value in image `src`. The complex absolute value is defined as the square root of the sum of the squares of real and imaginary part. The maximum square root is the return value of the function as well as the position of its maximum `(ix, iy)`.

return values square root of maximum or (negative) standard error code.

memory none

DisplayFFT **convert complex FFT image to U8**

synopsis `I32 DisplayFFT(image *src, image *dst, I32 log)`

description This function converts the image `src` of type `IMAGE_CMPLX16` into an image of type `IMAGE_GREY` as the destination image `dst`. The function is mainly used for the display of FFT images.

`log=0` will produce a linear, `log=1` a logarithmic output. In both cases, the square root of the sum of the squared real and imaginary parts will be used.

The function also re-arranges the 4 quarters of the source image to produce the conventional FFT image with the frequency 0 in the middle.

The output is also scaled to the maximum value.

return values standard error code.

memory none

DisplayInvFFT **convert complex IFFT image to U8**

synopsis

```
void DisplayInvFFT(image *src, image *dst,
                              I32 mean, I32 scale)
```

description

This function converts the image `src` of type `IMAGE_CMPLX16` into an image of type `IMAGE_GREY` as the destination image `dst`. The function is mainly used for the display of IFFT images (IFFT: inverse FFT).

The function only takes the real part of image `src` , `mean` is added to each pixel. The result is scaled using the value of `scale`. The image is not re-ordered like in **DisplayFFT**().

return values standard error code.

memory none

FL_fft2 **1D FFT (radix 2)**

synopsis

```
void FL_fft2(I32 n, I16 *xy, const I16 *w)
```

description

This function calculates the 1D FFT with 16bit fixpoint arithmetic. The following paramters are used:

`n`	size of the FFT, must be a power of 2
`xy`	complex I16 array for in-place operation of FFT
`w`	twiddle factors

The output of the function overwrites the source in the complex array `xy` and must be bit-reversed using the function

```
void FL_bitrev(I32 *xy, I16 *index, I32 n)
```

The twiddle factors and bit-reverse tables are calculated by the function

```
I32 FFTInitTwiddle(I32 n)
```

and accessed via the global array `FFT_Tab[14]`. Use function

```
void FFTDeinitTwiddle()
```

to deallocate the memory.

example

```
I16 *w, *index;

FFTInitTwiddle(n);

w     = FFT_Tab[cnbits(n)];
index = FFT_Tab[cnbits(n)] + 2 * n;

FL_fft2(n, xy, w);
FL_bitrev(xy, index, n);
```

memory none

CalcPolarCoordinates calculate polar coordinate table

synopsis

```
I32 CalcPolarCoordinates(image *pol, I32 deg)
```

description

This function calculates a table with polar coordinates necessary for some functions operating in the frequency domain. One quadrant of polar coordinates is stored in `pol`, an image of type `IMAGE_CMPLX16`. So, if the original FFT is of size (n x m), the size of `pol` must be (n/2 x m/2).

`deg` is the number of steps per 180 degrees. A typical value for `deg` is 180, i.e. one step per degree. `deg` should be a multiple of 2. The maximum value for `deg` is 65536.

Since the execution time of the function can be considerable depending on the image size, sometimes in the range of several seconds, it is recommended to execute it only once at the beginning of the program and keep the table data as long as necessary.

return values standard error code

memory none

DeleteFreq delete frequency in FFT-space

synopsis

```
I32 DeleteFreq(image *src, image *pol,
               I32 minrad, I32 maxrad)
```

description

`DeleteFreq` allows the deletion of frequencies `f` in the FFT-domain with

```
minrad <= f < maxrad
```

Since the function works in-place, `src` is source and destination. `pol` is the quadrant image of polar coordinates.
See function **`CalcPolarCoordinates`**`()` for further information.
The frequencies are deleted without angular preferences.

The sizes of `src` and `pol` must fit, i.e. either

```
pol->dx == src->dx/2      AND      src is full FFT
pol->dy == src->dy/2
```

OR

```
pol->dx == src->dx/2      AND      src is half FFT
pol->dy == src->dy
```

If both conditions do not hold, the function returns `ERR_PARAM`.

return values standard error code

memory none

CalcAngleHisto **calculate angular FFT histogram**

synopsis

```
I32 CalcAngleHisto(image *src, image *pol, I32 mode,
      I32 minfreq, I32 maxfreq, U32 *AngleHisto, I32 nr)
```

description

This function calculates the angular histogram for the FFT given by `src`.

The sizes of `src` and `pol` must fit, i.e. either

`pol->dx == src->dx/2` `pol->dy == src->dy/2`	AND	`src` is full FFT

OR

`pol->dx == src->dx/2` `pol->dy == src->dy`	AND	`src` is half FFT

If both conditions do not hold, the function returns `ERR_PARAM`.

return values

standard error code

memory

none

13 Routines for Linescan Camera

shading_correct **perform shading correction for an image (linescan type)**

synopsis

```
I32 shading_correct(image *src, image *dst,
                                        I8 *offs, U16 *shade)
```

description

This function calculates the shading correction for image `src` and outputs the result in image `dst`.
The shading correction algorithm consist in a subtraction of the offset values stored in the line offset buffer `offs` and a multiplication with the line shading buffer `shade`. The offset values may be positive (offset will be subtracted) and negative (offset wil be added). The values in the shading table must be 256 for the identity operation. Larger values will result in an amplification, smaller values will result in a de-amplification.

The size of both images must be identical. The size of the buffers must be dx = src->dx

return values standard error code

memory **none**

line_calibrate **calibrate the shading correction for an image (linescan type)**

synopsis

```
I32 line_calibrate(image *src1, image *src2,
                float t1, float t2, I8 *offs, U16 *shade)
```

description

This function performs the line calibration necessary for the shading correction. Images `src1` and `src2` must be images of e.g. a plain white surface taken at different shutter speeds `t1` (for `src1`) and `t2` (for `src2`). The function the calculates the offset and shading tables.

Care must be taken that the source images are not overexposed and have grey values in a reasonable range. If the camera features positive pixel offset, it is possible to have one image taken at a very short exposure time with very low grey values. In this case this image would mainly be responsible for the calculation of the offset table, while the other image would mainly be responsible for the shading gain table `shade`. It is clear that there must be some difference in shutter time (hence in grey values), otherwise the algorithm will not produce stable results.

The routine performs a vertical projection on both of the input images. the images are then compared and the offset and shading tables are calculated so that for the maximum contrast value the multiplication (shade) will be 256 (which is aquivalent to a multiplication with 1.0) parameters:

src1 : source image 1 (grey image)
src2 : source image 2 (grey image)
t1 : shutter value for image 1
t2 : shutter value for image 2
offs : line offset table (output)
shade : shading table (output)

The size of both images must be identical. The size of the buffers must be dx = src->dx

return values standard error code

memory **none**

line_IIR **perform line IIR regulation of background intesity (linescan type)**

synopsis

```
I32 line_IIR(image *src, image *dst,
      I32 tol, I32 int_const, I32 p_const, I32 target)
```

description This function performs a recursive filter used in linescan applications.

The function uses the following parameters:

`src`	source image
`dst`	destination image
`tol`	grey value tolerance for averaging
`int_const`	integration parameter
`p_const`	proportional parameter
`target`	target grey value

The algorithm may be used for surface inspection, where a homogeneous surface is inspected. It is mainly a regulation (tracking) of the grey values in a line.
Assume, that the grey values in a line are in the same range within a tolerance (after the shading correction). Pixels not within this tolerance (`tol`) are not considered. All the pixels within the tolerance are subtracted from the tracking average and the difference is integrated. The difference itself (proportional regulation) and the integrated difference (integration regulation) are multiplied with the corresponding constant `int_const/1024` and `p_const/1024` and used to produce a new tracking average. This average is then subtracted for the complete line, the target value `target` is added to produce a destination image with an average value of `target`.

The size of both images must be identical. The size of the buffers must be dx = src->dx

return values standard error code

14 Numerical algorithms from linear algebra

The functions of this chapter cover a set of numerical algorithms from linear algebra. Since vectors and matrices of floating-point type are used, care must be taken to avoid numerical instabilities. The algorithms themselves were chosen to provide maximum stability. This, however, cannot be guaranteed under all circumstances. Also, additional user calculations for the inputs or outputs of the functions, may introduces significant sources of numerical instability.

allocate a float matrix with subscript range[0..nr)[0..nc)

```
float **matrix(I32 nr, I32 nc)
```

free a float matrix allocated by matrix()

```
void free_matrix(float **m)
```

allocate a float vector with subscript range[0..nh)

```
float *vector(I32 nh)
```

free a float vector allocated by vector()

```
void free_vector(float *v)
```

print a two-dimensional float matrix

```
void matrix_print(I32 n, I32 m, float **a)
```

print a float vector

```
void vect_print(I32 n, float x[])
```

calculate vector norm

```
float vect_norm(I32 n, float x[])
```

multiplication of matrix with vector

```
void matrix_vect_mult(I32 n, I32 m, float **a, float x[], float y[])
```

multiplication of two matrices

```
void matrix_mult(I32 n, I32 m, float **a, float **x, float **y)
```

copy a matrix

```
void matrix_copy(I32 n, I32 m, float **src, float **dst)
```

calculate determinant of n x n matrix using LU decomposition

```
float lu_det(float **a, I32 n)
```

calculate inverse of n x n matrix using LU decomposition

```
I32 lu_inverse(float **a, float **y, I32 n, float *det)
```

15 Solar Wafer Library

MeasureRectangle **find and measure a rectangle**

synopsis

```
I32 MeasureRectangle (MR_Par *Par, image *SearchArea)
```

description

This high-level function finds a rectangle (solar wafer) in the image given by image `image *SearchArea` of type `IMAGE_GREY` and calculates its geometric properties.

The selection of a range of parameters and the output of the data is done by the following parameter struct:

```
typedef struct
{
  //////////////////////////////////////////////////
  // input parameters
  //////////////////////////////////////////////////

  I32    FindZoom;   /* number of demagnification stages           */

  //////////////////////////////////////////////////
  // edge parameters
  //////////////////////////////////////////////////

  I32    Type;         /* see function edge for description         */
  I32    Sigma;        /* 10 x sigma                                */
  I32    BinMode;
  I32    MinContr;
  I32    Thresh;

  //////////////////////////////////////////////////
  // hough parameters
  //////////////////////////////////////////////////

  HoughControl Control;       /* see Hough transform for descr.    */

  //////////////////////////////////////////////////
  // line selection filters
  //////////////////////////////////////////////////

  I32    DeviceCol;  /* 0=Black 1=White                            */
  I32    MinLen;     /* min length in pixel                        */
  I32    MaxLen;     /* max length in pixel                        */
  I32    PropOpp;    /* proportion of opposite lines in percent    */
  I32    PropNext;   /* proportion of adjacent lines in percent    */
  I32    DeltaAngle; /* max delta angle for 90 degree corners      */

  //////////////////////////////////////////////////
  // subpixel parameters
  //////////////////////////////////////////////////

  I32    EdgeFilterP; /* must be 4                                 */
  I32    EdgeFilterV; /* must be 2                                 */

  //////////////////////////////////////////////////
  // defect parameters
  //////////////////////////////////////////////////

  I32    DefectWidth; /* maximum defect width in pixels            */
```

```
    ////////////////////////////////////////////////////
    // output parameters
    ////////////////////////////////////////////////////

    ////////////////////////////////////////////////////
    //
    //                  line 0
    // Point 0 ------------------- Point 1
    //         |                  |
    //         |                  |
    //         |                  |
    //  line 3 |                  | line 1
    //         |                  |
    //         |                  |
    //         |                  |
    // Point 3 ------------------- Point 2
    //                   line 2
    //
    ////////////////////////////////////////////////////

    vcline Line     [4];   /* BestLine parameters                    */

    I32    PointsXY[4];   /* number of line points in LineXY        */
    float *LineXY   [4];   /* accurate X and Y positions of the line */

    float  EndX0    [4];   /* start of line, x-coordinate            */
    float  EndY0    [4];   /* start of line, y-coordinate            */
    float  EndX1    [4];   /* end of line, x-coordinate              */
    float  EndY1    [4];   /* end of line, y-coordinate              */

    float  Dmin     [4];   /* min line deviation (negative)          */
    float  Dmax     [4];   /* max line deviation (positive)          */
    float  DSigma   [4];   /* standard deviation from line           */

    // result
    I32   Error;
}

MR_Par;
```

The function performs the following tasks:

(1) Pyramid Demagnification. If selected, the image is zoomed down first to save computation time. No zoom down is selected as default. The function keeps track of the magnification stages and adjusts the output correspondingly
(2) Edge detection using the function **edge** `()`
(3) Hough transformation for lines
(4) Sophisticated line selection with respect to the user paramters
(5) Ordering of lines, 0=top, 1=right, 2=bottom, 3=left
(6) Subpixel measurement of lines. Lines need to have contrast for this stage. Subpixel resolution: 1/100 of a pixel
(7) chi-square bestline for the 4 lines
(8) calculate deviation from bestline including mousebite and sharkteeth

The line selection (4) is mainly a filter that makes sure that the selected lines fulfil a certain geometric relationship. With `MinLen` and `MaxLen` the minimum and maximum length of the lines can be specified. The parameters `PropOpp` and `PropNext` define the proportion of opposite and adjacent lines, e.g. a value of 20 means that one line needs to have at least 20% of the pixels of the other (longer) line. `DeltaAngle` is the tolerance of the 90 degree corners, i.e a value of 10 means that for the corners all angles between 80 and 100 degrees are allowed.

The rectangle does not need to be closed in order to be detected, i.e. it is possible that the lines do not have an intersection. On the other hand, it is also allowable that the lines are longer than necessary to produce a rectangle. With the values `EndX0, EndY0, EndX1, EndY1`, the function outputs the start and end points of each line.
The pixel lists `LineXY[4]`, that are generated by the function also include all the defects in the range of `DefectWidth` around the measured lines.

The function allocates memory for the edge detection tables, the Hough Transform and for the 4 output xy-lists. This memory space is kept after the execution of the function. To release the memory, the function

```
void DeinitMR(MR_Par *Par)
```

should be called, when the function is no longer needed.

return values standard error code

memory `3*dx*dy + sqrt(dx^2 + dy^2)*256` + memory for result
+ tables for edge detection and Hough Transform

Appendix A: Utilities

`void print_image(image *src)` prints image in HEX code.

Appendix B: Corr2 - normalized grey scale correlation

sample size = 32x32 pixels

corr2 is an example for the usage of the correlation functions. On program start the following message appears:

```
place sample in center frame
press any key when ready
```

You may then position an arbitrary pattern in the center frame (128x128 pixels). As soon as you press a key, the sample will be stored and the following message will appear.

```
sample stored
```

The program enters tracking mode, where it shows where the pattern is found in the image. Move the sample around to get an impression of the performance.
The right bar shows the quality of the detection. The higher the marking, the better the comparison.

Appendix C: List of library functions

Affine and non-affine coordinate transformations

Name	Type	Description
`void rotate90l(image *src, image *dst)`	C	rotate image by 90 degrees counter-clockwise
`void rotate90r(image *src, image *dst)`	C	rotate image by 90 degrees clockwise
`void rotate180(image *src, image *dst)`	C	rotate image by 180 degrees
`I32 move_image_alpha(image *src, image *dst, float mx, float my, U8 bgnd)`	C	move image with 2D interpolation
`I32 affine_image_transform(image *src, image *dst, float a[2][2], float t[2], U8 bgnd)`	C	general affine image transformation fast version
`I32 affine_image_transform2(image *src, image *dst, float a[2][2], float t[2], U8 bgnd)`	C	general affine image transformation slow floating-point version
`void calc_rotation_matrix(float angle, float cx, float cy, float a[2][2], float t[2])`	C	calculate affine transformation matrix for a rotation
`I32 polar_image_transform(image *src, image *dst, float t[2], U32 r0, U8 bgnd)`	C	polar to cartesian image transformation fast version
`I32 polar_image_transform2(image *src, image *dst, float t[2], U32 r0, U8 bgnd)`	C	polar to cartesian image transformation slow floating-point version
`void mirror_hor(image *src, image *dst)`	C	mirror image horizontally
`void mirror_ver(image *src, image *dst)`	C	mirror image vertically
`void xshear(image *src, float shear, image *dst, float offset, U8 bgnd)`	C	horizontal image shear
`I32 threepoint_calculate(point *p0, point *p1, point *p2, point *q0, point *q1, point *q2, float **a, float *t)`	C	three-point formula for affine transformations
`I32 lens_transform(image *src, image *dst, point *center, float k3, U8 bgnd)`	C	lens distortion correction
`I32 lens_transform2(image *src, image *dst, point *center, float f, float mag, U8 bgnd)`	C	lens distortion correction, type 2

Filter Functions

Name	Type	Description
`I32 isef(image *src, image *dst, float b)`	C	infinite symmetric exponential filter
`I32 isef_hor(image *src, image *dst, float b)`	C	horizontal infinite symmetric exponential filter (recursive)
`I32 isef_ver(image *src, image *dst, float b)`	C	vertical infinite symmetric exponential filter (recursive)
`I32 gauss(image *src, image *dst, float sigma)`	C	recursive gauss filter
`I32 gauss_hor(image *src, image *dst, float sigma)`	C	horizontal gauss filter (recursive)
`I32 gauss_ver(image *src, image *dst, float sigma)`	C	vertical gauss filter (recursive)
`I32 gauss_fir(image *src, image *dst, float sigma)`	C	non-recursive gauss filter
`I32 gradient_2x2(image *src, image *dst)`	C	vector gradient (robert's cross)
`I32 gradient_3x3(image *src, image *dst)`	C	vector gradient (sobel)
`I32 maxMxN(image *src, image *dst, I32 mx, I32 my)`	C	moving maximum (dilation) filter
`I32 minMxN(image *src, image *dst, I32 mx, I32 my)`	C	moving minimum (dilation) filter

Programs for edge detection

Name	Type	Description
`I32 edge(image *src, image *dst, I32 type, float sigma, I32 BinMode, I32 MinContrast, float fthresh, I32 binar_value)`	C	calculate image edges
`I32 edge_canny(src, dst, binar_value)`	M	edges, canny style
`I32 edge_fast(src, dst, binar_value)`	M	edges, fast routine
`I32 edge_sobel(src, dst, binar_value)`	M	edges, sobel style

Programs for gray scale correlation

Name	Type	Description
`I32 vc_corr2(image *a, image *b, I32 mcn, I32 mcr, I32 *x0, I32 *y0)`	C	small kernel correlation routine extended 32x32 kernel
`I32 vc_corr3(image *src, image *smp, image *dst32, I32 mcn)`	C	small kernel correlation routine 32bit image output
`float corrcheck(image *a, image *b)`	C	calculate correlation coefficient

Programs for processing binary images in (unlabelled) run length code

Name	Type	Description
`rlcnand (U16 *a, U16 *b, U16 *dest)`	M	NAND RLCs
`rlcnor(U16 *a, U16 *b, U16 *dest)`	M	NOR RLCs
`rlcequiv(U16 *a, U16 *b, U16 *dest)`	M	EQUIV=NXOR

Programs for processing binary images in labelled run length code

Name	Type	Description
`U16 *rlc_label(U16 *rlc, U16 *slc, I32 mode)`	C	object labelling
`U16 *sgmt(U16 *rlc, U16 *slc)`	M	object labelling 4/4
`U16 *label44(U16 *rlc, U16 *slc)`	M	object labelling 4/4
`U16 *label88(U16 *rlc, U16 *slc)`	M	object labelling 8/8
`U16 *label84(U16 *rlc, U16 *slc)`	M	object labelling 8/4
`U16 *label48(U16 *rlc, U16 *slc)`	M	object labelling 4/8
`I32 rlc_qin(U16 *rlc, I32 qin[], U32 n)`	C	object inclusion property
`I32 rlc_nhls(U16 *rlc, U32 holes[], U32 n)`	C	number of holes property
`I32 rlc_arf(U16 *src, U16 *dst, U32 min_area)`	C	RLC area filter for small objects
`I32 rlc_select(U16 *rlc, U16 *rlc2, I32 select[], U32 n)`	C	RLC object selection with guide image
`I32 rlc_delete(U16 *src, U16 *dst, I32 select[])`	C	delete RLC objects using a selection list
`I32 rlc_moments(U16 *rlc, moment *mom, U32 n)`	C	calculate moments of order 0, 1, 2
`float mom_calc_cgx(moment *mom)`	C	calculate center of gravity x
`float mom_calc_cgy(moment *mom)`	C	calculate center of gravity y
`float mom_calc_angle(moment *mom)`	C	calculate angle of inertial axis result in degrees
`float mom_calc_rad(moment *mom)`	C	calculate angle of inertial axis result in radiants
`float mom_calc_ecc(moment *mom)`	C	calculate object eccentricity
`float mom_calc_ellipse_a(moment *mom)`	C	calculate ellipse half-parameter a
`float mom_calc_ellipse_b(moment *mom)`	C	calculate ellipse half-parameter a
`float mom_calc_phi1(moment *mom)`	C	calculate Hu moment #1
`float mom_calc_phi2(moment *mom)`	C	calculate Hu moment #2

Miscellaneous Image Functions

Name	Type	Description
`I32 get_component(image *src, image *dst, I32 comp)`	C	get image component
`I32 equalize(image *src, image *dst)`	C	equalize image
`I32 set_ovl_false_color(I32 table)`	C	set translucent overlay LUT to false color palette
`I32 set_translucent_to_value(I32 t, I32 r, I32 g, I32 b)`	C	set translucent overlay LUT to fixed value
`I32 display_directions(image *src, I32 thresh, I32 startx, I32 starty)`	C	display a directional image using overlay
`void mask_frame(image *src, I32 sx0, I32 sx1, I32 sy0, I32 sy1, I32 value)`	C	mask a frame with programmable frame width

Pixellist functions

Name	Type	Description
`I32 bestline(I32 *xy, I32 N, float *cx, float *cy, float *b`	C	calculate chi-square bestline
`I32 bestcircle(I32 *xy, I32 N, float *px, float *py, float *rad)`	C	calculate chi-square bestcircle
`I32 clip(I32 N, I32 *xy_src, I32 *xy_dst, I32 x_min, I32 x_max, I32 y_min, I32 y_max)`	C	perform window-clipping
`void translate(I32 N, I32 *xy_src, I32 *xy_dst, I32 mx, I32 my)`	C	perform translation of coordinates in pixellist
`I32 PL_line_stats(I32 *xy, I32 nr, vcline *line, float *dmin, float *dmax, float *sigma)`	C	line statistics for pixel list
`I32 PL_line_ending(I32 *xy, I32 nr, vcline *line, I32 *imin, I32 *imax)`	C	line ending for pixel list

Geometric tools

Name	Type	Description
`I32 LineIntersection(vcline *a, vcline *b, point *r)`	C	calculate intersection point of two lines
`float PointDistance(point *a, point *b)`	C	calculate Euclidean distance between two points
`float PointLineDistance(point *p, vcline *l)`	C	calculate distance beween a point and a line
`void LinePerpendicular(point *p, vcline *l, vcline *r)`	C	calculates a line perpendicualar to a given one through a point
`void LineParallel(point *p, vcline *l, vcline *r)`	C	calculates a line parallel to a given one through a point
`float Angle(vcline *a)`	C	calculates angle of a line
`float LineAngle(vcline *a, vcline *b)`	C	calculates angle between two lines
`void LineParameters(point *p1, point *p2, vcline *line)`	C	calculates the line parameters using two points

Hough Transform

Name	Type	Description
`I32 HoughInit()`	C	initialize tables for HT
`I32 HoughInit2(HoughControl *control)`	C	initialize tables for **findline()**
`void HoughDeinit()`	C	deallocate tables for HT
`void HoughDeinit2(HoughControl *control)`	C	deallocate tables for **findline()**
`I32 HoughTransform(image *src, image *hough32, HoughControl *control)`	C	Hough Transform for lines
`I32 HoughCalcDx(image *src)`	C	calculate Hough horizontal size
`I32 FindHoughLine(image *hough32, image *src, HoughControl *control)`	C	find lines in accumulator space
`void HoughDefaults(HoughControl *control)`	C	set defaults for Hough Transform
`void HoughSortLine(HLine **listptr, I32 offset)`	C	sort line list according to different sorting criteria
`I32 HoughRank(HLine *listptr, I32 offset, I32 value)`	C	get number of lines above a certain value
`I32 GetHoughPixels(image *Src, HLine *line, HoughControl *ctrl, I32 *xy)`	C	extract xy-list from Hough source image

Graphics functions

Name	Type	Description
`I32 draw_line(image *a, float cx, float cy, float b, I32 col, void (*func)())`	C	draw a line in normalized floatingpoint form
`I32 draw_circle(image *a, I32 px, I32 py, I32 rad, I32 col, void (*func)())`	C	draw a circle with window-clipping
`I32 IntersectionPoints(image *a, float cx, float cy, float b, I32 *x0, I32 *y0, I32 *x1, I32 *y1)`	C	intersection of a line with image borders

Numerical algorithms from linear algebra

Name	Type	Description
`float **matrix(I32 nr, I32 nc)`	C	allocate a float matrix [0..nr)[0..nc)
`void free_matrix(float **m)`	C	free a float matrix allocated by matrix()
`float *vector(I32 nh)`	C	allocate a float vector [0..nh)
`void free_vector(float *v)`	C	free a float vector allocated by vector()
`void matrix_print(I32 n, I32 m, float **a)`	C	print a two-dimensional float matrix
`void vect_print(I32 n, float x[])`	C	print a float vector
`float vect_norm(I32 n, float x[])`	C	calculate vector norm
`void matrix_vect_mult(I32 n, I32 m, float **a, float x[], float y[])`	C	multiplication of matrix with vector
`void matrix_mult(I32 n, I32 m, float **a, float **x, float **y)`	C	multiplication of two matrices
`void matrix_copy(I32 n, I32 m, float **src, float **dst)`	C	copy a matrix
`float lu_det(float **a, I32 n)`	C	calculate determinant of n x n matrix
`I32 lu_inverse(float **a, float **y, I32 n, float *det)`	C	calculate inverse of n x n matrix

Legend: **A: Assembly function** **C: C function** **M: Macro**

Index

Visit the Vision Components site **www.vision-components.com** for further information and documentation and software downloads:

Web Site Menu Links	Content
Home	Latest News from VC
Our Company	VC Company Information
Contact Us	Distributor list / Enquiry forms
News	More News form VC
Products **VC Hardware** **VCXX Camera Series** **VC20XX, VC4XXX Smart Cameras** **VCSBC Board Cameras** **VCM Camera Sensors** **VC Software** **VCRT Operating System** **VCLIB Image Processing Library** **Vision Components' Special Libraries** **3rd Party Software**	Product Overviews: including accessories listings with corresponding order numbers M200 Data Matrix Code Reader VCOCR Text Recognition Color Lib Overview of 3rd Party software available for VC Smart Cameras
Support: **Support News**	Overview about latest features, manuals and SW updates
Knowledge Base / FAQ	Searchable HW and SW information database
Download Area **Public Download Area (free access)** **Registered User Dl Area (registration required)** **Customer Download Area (user- and software registration required)**	Download of all: - Product brochures - Camera Manuals - Programming Manuals - Software updates - Demo Codes - Programming Tutorials

Vision Components GmbH
Ottostraße 2 • D-76275 Ettlingen
Tel. +49(7243)2167-0
Fax +49(7243)2167-11
sales@vision-components.de