Documentation of the image processing library VCLIB version 3.0

This documentation was created very conscientiously. No liability will be assumed for any errors or misleading descriptions which it may contain. The statements made in this documentation are informative in nature and not a guarantee of features. The right is reserved to make changes in the interest of technical progress.

This documentation describes the programs of the image processing library version 3.0. You can also consult the following documentation:

- Hardware documentation — Hardware
- Documentation VC/RT — Operating system and basic functions
- Documentation FLIB — Fast vector functions

Main changes with respect to VCLIB2.0 release 2 are:

All functions have been completely revised. They have been rewritten to account for the structure and possibilities of the TI C62xx and C64xx architecture. This allows for future improvements in speed and functionality. VCLIB 3.0 is not backward compatible to the (older) ADSP architecture.

CONTENTS

Changes with respect to VCLIB2.0 release 2

Functions working on image variables

Functions working on image variables behave pretty similar to their previous counterparts. Images and regions of interest are specified by the **image** structure as an input to the function.
The internal operational philosophy, however, has changed: The ADSP compatible philosophy required image data (**U8**) to be copied to an integer line buffer. The modification then took place with a basic function with output data in a second integer line buffer. The result had to be transferred from this second buffer back to pixel memory. To copy the data from image data memory to line buffers and back functions like `blrdb()` and `blwrb()` were used. As a result, all functions could only work on images with even start address `st` and even number of horizontal pixels `dx`.
With the new TI philosophy, this restriction does not apply. All regions of interest may start wherever they want and the size may be arbitrary within reasonable bounds, because `blrdb()` and `blwrb()` are no longer used. For the new philosophy, no copying of image data is necessary, which also improves the speed performance of the functions. Basic functions now operate directly on image data.

Functions working on runlength code

This is where the major changes have been done. A pointer to RLC is no longer a **long** variable. Instead it is a **U16 ***, which it is supposed to be. In addition, the RLC address itself is no longer half the value of the corresponding memory address, it is just the memory address and nothing else. As in the case for the image variables, all unnecessary data copying was avoided.
The following example may be helpful:

ADSP version:

```
image a = { 0L, 752, 582, 768};
long rlc;

a.st = (long)getvar(CAPT_START);

rlc = (long)vcmalloc(0x10000);

if(rlcmk(&a, 128, rlc>>1, 0x10000L) != 0L)
  rlcout(&a, rlc>>1, 0, 255);

vcfree((int *)rlc);
```

TI version:

```
image a = { 0L, 640, 480, 768};
U16 *rlc;

a.st = (long)getvar(CAPT_START);

rlc = (U16 *)vcmalloc(0x10000);     /* 0x10000 * 4 bytes of memory      */

if(rlcmk(&a, 128, rlc, 0x40000) != NULL)
  rlcout(&b, rlc, 0, 255);

vcfree((int *)rlc);
```

What can also be seen is that for the RLC size now bytes instead of integers are used.

Basic functions

All basic functions for image variables and RLC had to be changed. They now operate directly on **U8** pixels or **U16** RLC data. Basic functions have been taken off this documentation. They are most promising for future speed improvements and will be included in a different library called FLIB (Fast Library)

Contour functions:

Like for RLC, the pointer for the resulting contour code has been changed from **long** to **U32 ***, which results in **U32 *dst** for the new `contour`() function. Like for the RLC, addreses to contour code are now "real" addresses, not addresses divided by 2 as it was the case for the old version. Contour Code is now stored as byte values (instead of integers) which reduces memory requirement by a factor of 4. Please keep in mind that there always must be 16 additional bytes of memory available for contour length, error code and position of contour start.
The following example may be helpful:

ADSP version:

```
image a  = {0L, 256, 256, 768};
int x0, y0=200;
long dest, cc;

a.st = (long)getvar(CAPT_START);

cc=(long)vcmalloc(1005);      /* allocate space for contour code */
dest=cc/2;

x0=cfind(&a, y0, 128);        /* find contour start              */

if(x0!=0)
  {
  contour8(&a, x0, y0, ~2, 128, 1000, &dest);
  }

cdisp_d(&a, cc/2,  255);
vcfree((int *)cc);
```

TI version:

```
image a  = {0L, 256, 256, 768};
int x0, y0=200;
U32 *dest, *cc;

a.st = (long)getvar(CAPT_START);

cc=(long)vcmalloc(256+16);    /* allocate space for contour code */
dest=cc;                      /* 1000 bytes for CC + 16 bytes    */
                              /* for size, error code and x0, y0 */

x0=cfind(&a, y0, 128);        /* find contour start              */

if(x0!=0)
  {
  contour8(&a, x0, y0, ~2, 128, 1000, &dest);
  }

cdisp_d(&a, cc,  255);

vcfree((int *)cc);
```

Pixellist functions:

Pixellist functions are not part of VCLIB 2.0 but of VCRT. In order to account for the new programming philosophy new pixellist functions have been added to VCLIB 3.0. These are:

```
ad_calc32
rp_list32
wp_list32
wp_set32
wp_xor32
```

The new graphics functions like `frame()` or `line()` rely on the new pixellist functions, but this fact is hidden inside these functions.
If you use the old pixellist functions (of VCRT), it is recommended to change to the new ones in order to make advantage of some possible future routines using pixel lists.

Functions returning long:

Several functions return long values in VCLIB 2.0. Those have been changed to U32
The following functions have been changed:

`histo():` uses U32 array instead of long array for result.
`mean(), focus(), variance(), arx(), arx2()` return their result as U32 instead of long

Summary: Changes necessary to use VCLIB 3.0 for existing programs:

Functions for image variables	No changes
Functions for RLC	Change all RLC pointers from **long** to **U16 ***
	Change all numerical values from half addresses to real addresses
	Change the maximum size for call of `rlcmk()` from integer to byte (factor 4)
Functions for Contour Code	Change CC pointers from **long** to **U32 ***
	Change all numerical values from half addresses to real addresses
	Change the maximum size for `contour8()` from integer to byte (factor 4)
Basic functions	Contact VC
Pixellist functions	It is recommended, but not necessary to change to the new functions with different names
Functions returning long	change definition for result or use cast

General comments on the image processing library

Image processing involves relatively large amounts of data. A video image of the size 512x512 pixels requires 256 KBytes of memory, an image with 740x574 pixels requires 415 KBytes, and a high-resolution image of the size 1024x1024 pixels even requires 1 MByte of memory. This fact naturally affects computing time.
Let's assume some format-filling image operation requires only one microsecond per pixel. Then, a 512x512 image requires 262 msec, a 740x574 image requires 425 msec, and a format of 1024x1024 requires around 1 sec. This is unacceptable in many cases, especially in industrial image processing.
Naturally, one can try to work around this problem by use of faster and faster processors. On the other hand, technical progress which produces faster processors also produces higher-resolution sensors. This comparison illustrates the problem well. If the clock rate of a processor is doubled, it will work twice as fast (assuming a double-speed memory). However, if the format of a sensor changes from 512x512 to 1024x1024, this is **four times** as much.
For this reason, there are some rules for developing fast image processing programs

1. Avoid format-filling image processing
2. Use optimized programs
3. Use processes which are as simple as possible
4. To the extent possible, make calculations beforehand
5. Use run length codes for binary images

Avoid format-filling image processing

In most cases, it is not necessary to evaluate **all** pixels of an image, even though their **existence**, i.e., a high resolution, is often very useful.
Numerous examples will be provided below which illustrate how this can be done.
Knowledge of the problem to be solved is of vital importance. If certain pixels are unimportant for a particular task, then they do not need to be evaluated. With this method, the computing speed can often be increased **several thousand times**.

Use optimized programs

The programs included in the library described here are almost all highly optimized assembly language programs. Thus, in many cases it pays to find a way to create the desired image processing program from library calls, even if the required algorithm cannot be found in the library. In most cases, this is better than writing your own program in C.

Use processes which are as simple as possible

Complicated algorithms tend to require a lot of processor time. If this is not possible, at least try to use a combination of simple steps.

To the extent possible, make calculations beforehand

Many calculations can be made beforehand, and the results can be saved in tables. This includes, for example, trigonometric functions which can be calculated from a table faster than from an algorithm. Also, in many cases image coordinates can be converted to video memory addresses beforehand.

Use run length code for binary images

Many programs in this library work with run length code (described in detail below).
In many cases, the use of run length code (specifically for binary images) can increase the evaluation speed several fold. Only the function which creates run length code from a gray-scale image requires some processing time.

Methods for avoiding format-filling image processing

1. Areas of Interest
2. Forgoing high resolution
3. One dimensional instead of two-dimensional image processing

Areas of Interest

This procedure limits itself to the relevant image sections (windows, areas of interest). E.g., in a relatively large image section first the position of an object could be determined. Depending on this search, much smaller windows are calculated. The presence, for instance, of a bored hole or a bar code could be evaluated with these windows.
This relatively simple procedure often increases speed considerably. Remember that the number of pixels in a square window increases with the square of the length of one side. A window with a side 100 pixels long has an area of only 10000 pixels, while one with a side of 1000 pixels has an area of a million pixels. That is one hundred times more!

Forgoing high resolution

Some operations do not need the full resolution of the image. As an example, if you want to look for an object which a certain known minimum size, then it suffices to include every other, every fourth, or more generally every nth pixel in the search. This effect can be used horizontally as well as vertically, so the acceleration is n^2.

One dimensional instead of two-dimensional image processing

One-dimensional image processing includes the following procedures:

- Edge sampling - a sudden change of brightness is located along on a line (one-dimensional).
- Contour following - the contour of an object is a one-dimensional structure, even if it is very jagged due to poor image quality.

For edge sampling, the maximum number of pixels to be examined is the number of pixels in the image diagonal (and it is only this number under difficult circumstances). As a rule, a few hundred pixels are evaluated in such cases.

For contour following, experience shows a few thousand pixels are evaluated (in seldom cases, up to ten thousand pixels).

In both cases, the number of pixels to be evaluated is much less than for a full frame, even though the algorithms used here are often somewhat more complex.

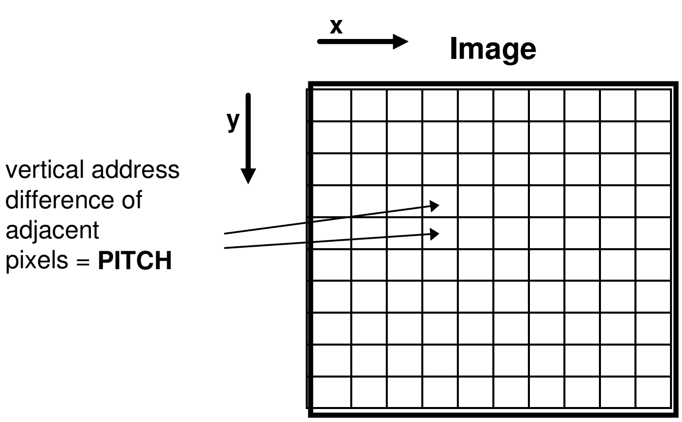
Important image processing data structures

Gray-scale images/image windows

Gray-scale images are usually saved as two-dimensional arrays (unsigned char).
Since computer memories always have a linear structure, the video data is saved in sequence, pixel for pixel, line for line. It is possible for a gap of exactly identical length to occur between the individual lines (e.g. when taking and showing an image). The address of a pixel can then be calculated with the following formula:

```
long addr, startad;
addr = startad + (long) y * PITCH + (long) x;
```

Here, `startad` is the start address of the video memory area, x and y are the coordinates of the pixel (in image processing, the origin is in the upper left corner of the image, the x-axis corresponds to the usual mathematical convention, while the y-axis is pointed down in contrast to the convention).
The constant `PITCH` is the difference of the address of two vertically adjacent pixels.
Access functions are used to access the pixels of the image array. These functions are described in detail in the VC/RT documentation.



Images and image windows are described by means of so-called image variables, which are described in detail below.

Color images

Color image processing and the corresponding data structures are described in the documentation of the color library.

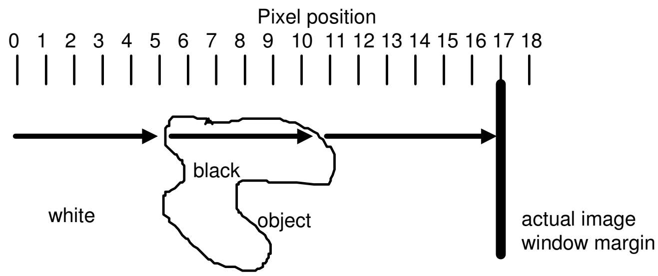
Run length code (RLC)

Probably the best known use of the run length code (RLC) is for telefax. In contrast, RLC is used in image processing not to reduce the amount of information but rather due to the execution speed of the RLC-based programs. For this reason, the run length code used in image process has a slightly different structure than for telefax.

As a matter of principle, RLC is especially suited for binary images, or - in modified form - for images with few quantization steps (a maximum of 16). If there are too many quantization steps, there is the potential hazard that encoding in RLC will not reduce the amount of information of the original image but quite the opposite might actually increase the amount of information. The reduction in the amount of information is the reason why RLC-based programs run faster.

The following will assume RLC for a pure binary image. The RLC is created by proceeding from left to right, line by line. Each change from dark to bright or from bright to dark produces an entry in the run length code. The pixel position of the change is stored. If a line begins with white, then an entry is created even for pixel 0. If the line begins with black, then the earliest change can be at pixel 1. An end-of-line mark is entered at the end of the line, independent of the number of changes in the line.
The end-of-line mark is always the last possible pixel position in the line plus 1. Or, stated another way, it is the line width. (Note: the first pixel is numbered 0, the last one is (line width - 1))

The end-of-line mark can vary for different run length codes, so to make sure, it is entered before the actual run length code, as is the number of lines.



Entry no.	**RLC**	**Remark**
1	0	SLC address LSW (0 for unlabelled RLC)
2	0	SLC address MSW (0 for unlabelled RLC
3	18	dx = end-of-line mark
4	25	dy = number of image/image window lines
5	-1	line begins white (0 if line starts black)
6	5	first change from white to black at position 5
7	11	change from black to white at position 11
8	18	end-of-line mark, because image window margin reached
9	...	RLC entries for next line

The SLC address mentioned at the beginning of this example is described below. It is always 0L for unlabelled RLC.

Labelled run length code (SLC)

The labelled run length code contains the segment label code (SLC) in addition to the pure image information of the RLC. The SLC stores information on how the image areas relate to one another.

For example, the example used in the last chapter (figure) consists of two common image areas, the (black) object and the (white) background.

The SLC does not differentiate between objects and background. Thus, in both cases we speak of *objects*.

The SLC is the result of o*bject labeling*.

The SLC requires the same amount of memory as the RLC it is based on. The base address of the SLC is arbitrary, as it is entered in the RLC. However, it is recommended to save the SLC *directly behind* the RLC.

The first entry of the SLC is the number of contained objects, followed by an object number for each RLC entry which always begins with 0.

The SLC for the example of the last chapter is as follows:

Entry no.	**SLC**	**Remark**
1	2	total number of objects (including the background)
2	0	background = object 0
3	1	black object = object 1
4	0	background = object 0
5	---	dummy
6	...	SLC entries for next line

example: labelled RLC (2 lines)

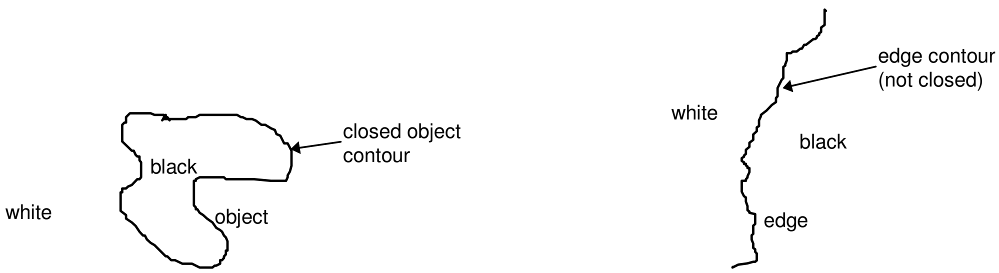
address	code (U16)	comment
0xA0000000	0x0018	SLC address (LSW)
0xA0000002	0xA000	SLC address (MSW)
0xA0000004	18	dx
0xA0000006	25	dy
0xA0000008	-1	color
0xA000000A	5	
0xA000000C	11	
0xA000000E	18	end_of_line
0xA0000010	-1	color
0xA0000012	6	
0xA0000014	12	
0xA0000016	18	end_of_line
0xA0000018	2	number of objects
0xA000001A	0	object 0 (white)
0xA000001C	1	object 1 (black)
0xA000001E	0	
0xA0000020	0 ----	dummy
0xA0000022	0	object 0 (white)
0xA0000024	1	object 1 (black)
0xA0000026	...	

Address lists (pixel lists)

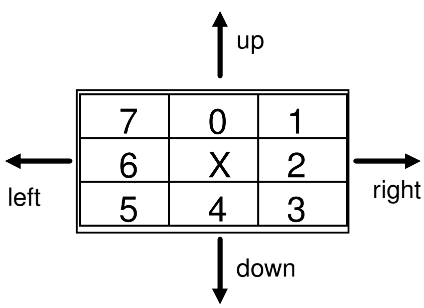
For one-dimensional image structures, it is often recommendable to use so-called pixel lists. Such a list contains both the (x,y) coordinates of the pixels and the video memory addresses of the pixels. The latter can serve to save processor time. The (x,y) coordinates or addresses can also be stored together with the gray scales of the corresponding pixels.

Contour code (CC)

The contour code (CC) is a method for storing one-dimensional contour data of (closed) object contours or edge data (not closed). Instead of storing the x and y coordinates of all contour pixels, the contour code stores a differential 3 bit information, indicating the direction of movement from one pixel to the next in the contour list. With this data structure only the x and y coordinates of the starting point must be given in order to reconstruct the contour.



different types of contours



0	up
1	up_right
2	right
3	down_right
4	down
5	down_left
6	left
7	up_left

contour code values (0 - 7) and the four major directions

example of the CC data format:

byte offset:	**CC**	**Remark**
0	00000004	length (4 contour pixels)
4	00000020	CC status (32: space exhausted)
8	00000018	starting pixel x coordinate
12	00000025	starting pixel y coordinate
16	00	CC: up
17	07	CC: up_left
18	00	CC: up
19	01	CC: up_right

CC status:

The generation of contour code may terminate due to different stop conditions. If an object contour is followed, the code generation will usually stop when the starting pixel is reached in the same direction. (Pixels may be in the contour list more than once, but only once for each direction). The code generation will also stop, if a corner of the image variable is reached or if the space for the contour list is exhausted.
The stop condition is stored in the CC status word in the header of the contour code according to the following table:

1 : closed contour (end pixel = starting pixel / same direction)
2 : contour stops at left corner of image variable
4 : contour stops at right corner of image variable
8 : contour stops at upper corner of image variable
16 : contour stops at lower corner of image variable
32 : space exhausted (CC lenght > lng)

Some of the conditions could be true at the same time. (example: contour stops at the left upper corner of the image variable) In this case the individual codes will be added (example: 2+8 = 10)

Connectedness

When dealing with binary objects, the principle of how objects are connected is important.

Some people consider pixels to belong to the same object only if they have neighbors of that object in one of the 4 major directions. The object is then called 4-connected. If you allow all 8 directions, it is called 8-connected.

```
1 0 0
0 1 1
1 1 1
```

example: all white pixels (1 = white) are 8-connected but not 4-connected

JPEG data (JPG)

JPEG is a standard for still image compression. The images may be stored at an arbitrary compression rate. There is some loss of information: the higher the compression rate the higher the image degradation due to loss of information. We recommend using quality factors of 50% - 80% for high quality images at reasonable compression rates.

Since JPEG is a standard it may be used for exchanging image data with e.g. a PC. Standard PC programs comply with the format used in this library. Please be sure to store images as grey-level images since this color compression / decompression is not supported.

Image variables used in JPEG compression must have a format which is a multiple of 8 for both, dx and dy. When decompressing images image variables must have a size of at least the size of the JPEG image - otherwise no decompression will be preformed.

Overview of the library functions

Macros

The file `macros.h` contains macros that are useful for working with the library. It is not necessary to use these macros, but it may turn out to be convenient.
The following types of macros are available:

- definition of bits, bytes, words, pages
- aliases for video modi
- conversion macros
- image variable macros
- screen macros
- overlay macros
- utility macros

Some macros (screen macros) use conventions for physical and logical addresses. There is, again, no obligation to use these conventions and the according macros.

1. macros for bits, bytes, words, pages

```
#define BitsPerByte          8
#define BitsPerWord         32
#define BytesPerWord         4
#define BytesPerPage         1
```

2. aliases for video modi (for function vmode())

`#define vmLive`	`(0)`	live image (including DRAM update)
`#define vmStill`	`(1)`	still image
`#define vmLiveRefresh`	`(2)`	live image (including DRAM update)
`#define vmFreeze`	`(3)`	still image
`#define vmOvlLive`	`(4)`	live image + overlay
`#define vmOvlStill`	`(5)`	still image + overlay
`#define vmOvlLiveRefresh`	`(6)`	live image + overlay
`#define vmOvlFreeze`	`(7)`	still image + overlay

3. conversion macros

`#define BitsAsBytes (bits) ((bits)/(BitsPerByte))`	number of bytes per bits
`#define BitsAsWords (bits) ((bits)/(BitsPerWord))`	number of words per bits
`#define ByteAddrAsBitAddr (addr) (addr)`	Overlay address above screen address
`#define BitAddrAsByteAddr (addr) (addr)`	screen address below Overlay address

4. image variable macros

assignment of a whole image variable in just one statement

```
#define ImageAssign (a,newst,newdx,newdy,newpitch)
{(a)->st=(long)(newst);(a)->dx=(I32)(newdx);(a)->dy=(I32)(newdy);
 (a)->pitch=(I32)(newpitch);}
```

display the values of an image variable for debugging

```
#define ImagePrintMembers (text,a) print(text);
print("st=0x%lx(%ld),dx=%d,dy=%d,pitch=%d\n",(a)->st,
(a)->st,(a)->dx,(a)->dy,(a)->pitch)
```

address of pixel on image variable

```
#define ImageAddr (a,x,y) ((long)((a)->st+(x)+(y)*(a)->pitch))
```

set a pixel to value at coordinates relative to an image variable

```
#define ImageSetPixel (a,x,y,g)
(*((U8 *)(ImageAddr((a),(x),(y)))) = (U8)(g))
```

get the value of a pixel at coordinates relative to an image variable

```
#define ImageGetPixel (a,x,y)
(*((U8 *)(ImageAddr((a),(x),(y))))
```

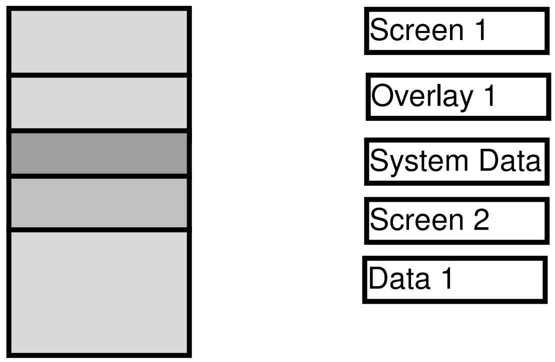
5. screen macros

```
#define ScrGetRows                number of screen rows
      getvar(VWIDTH)
#define ScrGetColumns             number of screen columns
      getvar(HWIDTH)
#define ScrGetPitch                pitch (in bytes)
      getvar(VPITCH)
#define SizeOfScreen              size of the screen in bytes
      (ScrGetPitch*ScrGetRows)

#define DispGetRows               number of display rows
      getvar(DVWIDTH)
#define DispGetColumns            number of display columns
      getvar(DHWIDTH)
#define DispGetPitch              pitch (in bytes)
      getvar(VPITCH)
```

logical and physical addresses

DRAM :



physical screen page: screen that is displayed
logical screen page: screen start page that is used for address calculations
physical overlay page: overlay that is displayed (if overlay video mode active)
logical overlay page: overlay start page that is used for address calculations

There's just one physical page but there may be multiple logical pages.

```
#define ScrGetPhysPage
      getvar(CAPT_START)
```
actual physical page being displayed right now

```
#define ScrSetPhysPage (phys)
      setvar(DISP_START,(addr)); setvar(CAPT_START,(addr));
```
display of screen page (set physical page)

```
#define ScrGetLogPage
      getvar(SCRLOGPAGE)
```
actual logical page being worked on

```
#define ScrSetLogPage (log)
      setvar(SCRLOGPAGE,(addr))
```
set logical page

```
#define ScrGetDispPage
      getvar(DISP_START)
```
actual display page

```
#define ScrSetDispPage (addr)
      setvar(DISP_START,(addr))
```
set display page

```
#define ScrGetCaptPage
      getvar(CAPT_START)
```
actual capture page

```
#define ScrSetCaptPage (addr)
      setvar(CAPT_START,(addr))
```
set capture page

```
#define ScrByteAddr (x,y)
      ((long)(ScrGetLogPage+(x)+(y)*ScrGetPitch))
```
(logical) screen address at coordinates (x,y)

```
#define ScrSetPixel (x,y,value)
      (*((U8 *)(ScrByteAddr(x,y))) = (U8)(value))
```
set pixel at (logical) coordinates (x,y) to value

```
#define ScrGetPixel (x,y)
      (*((U8 *)(ScrByteAddr(x,y)))
```
get value of pixel at (logical) coordinates (x,y)

```
#define ScrGetX (addr)
      ((U32)(addr)-(U32)ScrByteAddr(0,0))%ScrGetPitch
```
x-coordinate of (logical) address

```
#define ScrGetY (addr)
      ((U32)(addr)-(U32)ScrByteAddr(0,0))/ScrGetPitch
```
y-coordinate of (logical) address

6. overlay macros

```
#define OvlGetColumns
      ScrGetColumns
```
number of overlay columns

```
#define OvlGetRows
      ScrGetRows
```
number of overlay rows

```
#define OvlGetPitch
      ScrGetPitch
```
overlay pitch (in bits)

```
#define OvlGetPhysPage
      getvar(OVLY_START)
```
overlay page being displayed right now

```
#define OvlSetPhysPage (phys)
      setvar(OVLY_START,phys)
```
display of overlay page (set physical overlay page)

```
#define OvlGetLogPage
      getvar(OVLLOGPAGE)
```
overlay page being worked on

```
#define OvlSetLogPage (log)
      setvar(OVLLOGPAGE,(log))
```
set logical overlay page

```
#define OvlBitAddr (x,y)
      OvlByteAddr(x,y)
```
overlay address at the (logical) coordinates (x,y)

```
#define OvlByteAddr (x,y)
      ((long)OvlGetLogPage + (long)(x)+(long)(y)*OvlGetPitch)
```
overlay address at the (logical) coordinates (x,y)

```
#define OvlSetPixel (x,y,value)
      wovl(value,OvlBitAddr((x),(y)))
```
set/clear an overlay pixel at (logical) coordinates (x,y)

```
#define OvlGetPixel (x,y)
      rovl(OvlBitAddr((x),(y)))
```
get value of overlay pixel at (logical) coordinates (x,y)

```
#define OvlGetX (addr)
      ((long)(addr)-OvlBitAddr(0,0))%OvlGetPitch
```
x-coordinate of overlay pixel

```
#define OvlGetY (addr)
      ((long)(addr)-OvlBitAddr(0,0))/OvlGetPitch
```
y-coordinate of overlay pixel

```
#define OvlClearAll
      {image ovl;ImageSet(&ovl,BitsAsBytes(Overlay.st),
      BitsAsBytes(OvlGetColumns),OvlGetRows,BitsAsBytes(OvlGetPitch));
      set(&ovl,0);}
```
clear whole (logical) Overlay

7. utility macros

`#define getchar` `rs232rcv`	input a char via RS232
`#define putchar` `rs232snd`	output a char via RS232
`#define kbhit ()` `(-1 != rbempty())`	key pressed ?
`#define DRAMScreenMalloc ()` `((int)sysmalloc(((SizeOfScreen)+1024+BytesPerWord-1)/BytesPerWord,MIMAGE))`	allocates memory for one screen page (not aligned)
`#define DRAMDisplayMalloc ()` `((int)sysmalloc((DispGetPitch*DispGetRows+1024+BytesPerWord-1)/BytesPerWord,MIMAGE))`	allocates memory for one display page (not aligned)
`#define DRAMOvlMalloc ()` `DRAMScreenMalloc()`	allocates memory for one overlay page (not aligned)

Programs for processing gray images

set	set image variable to a constant value
copy	copy an image variable
histo	histogram of an image variable
img2	link two image variables
imgf	any 3 x 3 operator of an image variable
ff3	3 x 3 filter with any mask
ff5	5 x 5 filter for image variable
ff5y	5 x 5 filter for image variable horizontal / vertical separation
robert	robert's cross operator of an image variable
projh	horizontal projection of an image variable
projv	vertical projection of an image variable
look	look-up table function
focus	calculate the focal value of an image variable
mean	calculate the mean value of an image variable
variance	calculate the variance of an image variable
pyramid	pyramid filter for image variable
subsample	subsample image (image variable)
arx	calculate the number of pixels above threshold of image variable
arx2	calculate the number of pixels between two thresholds of image variable
bin0	fast binarization of an image variable
avg	moving average or unsharp masking of an image variable

Image variable

The image variable is a *struct* which summarizes all information required to characterize a gray-scale image or an image window.

Here is the definition of the image variables:

```
typedef struct
  {
  long st;    /* start address         */
  int dx;     /* horizontal width      */
  int dy;     /* vertical width        */
  int pitch; /* memory pitch          */
  } image;
```

st is start address of the image or image window in the memory (byte address).
dx and dy are the horizontal and vertical size of the image/image window.
pitch is the of the address difference of two vertically adjacent pixels (above one another).

With the current version of the library, it is no longer necessesary that start address and horizontal width of an image variable must be even numbers. All parameters of an image variable may be arbitrary numbers.

Sample image variables

1. The pattern of a part is to be stored in a gray image with the size 256(h) x 128(v).

```
#include <vclib.h>

main()
{
image a =   {0L,                   /* start address              */
             256,                  /* dx                         */
             128,                  /* dy                         */
             256};                 /* pitch                      */

a.st = (long)(getvar(CAPT_START)); /* assign start of image      */
                                   /* to address of capture      */
                                   /* memory buffer              */
...
```

Selecting 256 for pitch produces a *tight* version of the image in memory, without gaps. This is not always the case. When pictures are taken, the resulting image sometimes contains gaps, meaning that `pitch` is greater than dx. However, `pitch` may never be smaller than `dx`.

2. A full frame (a) is assumed to have a size of 640(h) x 480(v) with a pitch of 640. Two partial images (b, c) with a size of 128(h) x 128(v) are to be defined in this full frame. The partial images will later be used to evaluate the image.

```
#include <vclib.h>
#define PITCH_A   640

main()
{
image a, b, c;

ImageAssign(a,(long)(getvar(CAPT_START)),  640, 480, PITCH_A);
ImageAssign(b, a.st + 100L*PITCH_A + 200L, 128, 128, PITCH_A);
ImageAssign(c, a.st + 200L*PITCH_A + 300L, 128, 128, PITCH_A);
```

The upper left corner of the image window b is located at position (200,100) of the full frame a. The upper left corner of the image window c is located at position (300,200).

If it is desired, for example to set the contents of the image variable c to the constant value 255 (white), this can be done with the following function call:

```
set(&c,255);
```

3. You may also use `pitch` with a value twice as large as normal in order to access half images. The start address will then determine which half image is processed.

4. The `pitch` for the capture and display memory of the camera can be retrieved from a system variable called `VPITCH` (video pitch):

```
#include <sysvar.h>

pitch = getvar(VPITCH);
```

set — **set image variable to a constant value**

synopsis

```
void set(image *a, int x)
```

description

The function `set()` sets all pixels of an image variable to the constant value x.

memory

none

copy — **copy an image variable**

synopsis

```
void copy(image *a, image *b)
```

description

The function `copy` copies the contents of the image variable a to b.

If the format of the image variable (dx, dy) is not identical, the format of the result variable b is used. In particular, this means that the result of the operation is not defined if the image format of a is smaller than that of b. (a->dx < b->dx or a->dy < b->dy)

You are recommended to work with identical image formats, i.e.
a->dx = b->dx and a->dy=b->dy

memory

none

histo — **histogram of an image variable**

synopsis

```
void histo(image *a, U32 hist[256])
```

description

The function `histo` calculates the histogram of the image variable a.
The histogram is the absolute frequency of the 256 different gray scales in an image/image window.
In addition to the image variable a, a pointer to an array with 256 values is passed to the function. After calling the function, the result can be taken from this array.

memory

none

see also

`mean(), variance()`

img2 — link two image variables

synopsis

```
void img2(image *a, image *b, image *c,
                              void (*func)(),int q)
```

description

The function img2() makes it possible to calculate any links of the two image variables a and b. The result is stored in the image variable c, which can be identical with a or b or both.

If the format of the image variables (dx, dy) is not identical for all three image variables, then the format of the result variable c is used. In particular, this means that the result of the operation is not defined if the image format of a or b is smaller than that of c.
(a->dx < c->dx or a->dy < c->dy or b->dx < c->dx or b->dy < c->dy)

You are recommended to work with identical image formats, i.e.
a->dx = b->dx = c->dx and a->dy=b->dy=c->dy

q is a parameter which is passed to the basic function func().

The nature of the link is specified by providing a pointer to the basic function to be executed. For the available basic functions there are macros (#define instructions), which make it easier to call the function.

The following macros are available:

Call	operation	function
add2(a,b,c,sh)	sh>0: c = (a + b)<<sh; sh<0: c = (a + b)<<-sh	add2f()
sub2(a,b,c)	c = abs(a-b)	sub2f()
max2(a,b,c)	c = max(a,b)	max2f()
min2(a,b,c)	c = min(a,b)	min2f()
and2(a,b,c)	c = a AND b	and2f()
or2 (a,b,c)	c = a OR b	or2f()
xor2(a,b,c)	c = a XOR b	xor2f()
subx2(a,b,c,offs)	c = (a - b + offs); clipping if c>255 or c<0	sub2x()
suby2(a,b,c)	c = (a - b)>0 ? 255 : 0	sub2y()

Of course, you can write your own basic functions. Pass their address (function pointer) to img2().

example

The following example subtracts two image variables from one another. The result is stored in the image variable b.

```
#include <vclib.h>
#include <flib.h>

main()
{
image a, b;
ImageAssign(a,(long)(getvar(CAPT_START)),256,256,768);
ImageAssign(a, a.st + 256L,256,256,768);
sub2(&a, &b, &b);
```

memory

none

imgf **arbitrary 3 x 3 operator of an image variable**

synopsis

```
void imgf(image *a, image *b, void *func())
```

description

The function `imgf()` makes it possible to calculate any arbitrary 3 x 3 filter operation of the image variable a. The result is stored in the image variable b, which may be identical with a.

If the format of the image variables (dx, dy) is not identical, then the format of the result variable b is used. In particular, this means that the result of the operation is not defined if the image format of a is smaller than that of b. (a->dx < b->dx or a->dy < b->dy)

It is recommended to work with identical image formats, i.e. a->dx = b->dx and a->dy=b->dy

The nature of the filter operation is specified by providing a pointer to the basic function to be executed.

For the available basic functions there are macros (#define instructions), which make it easier to call the function.

The following macros are available:

Call	Basic function
`sobel(a, b)`	`sobelf()`
`laplace(a, b)`	`laplacef()`
`mx(a, b)`	`maxf()`
`mn(a, b)`	`minf()`

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

example

The following example calculates the Sobel operator of the image variable. The result is also stored in a and overwrites the original contents of a.

```
#include <vclib.h>
#include <flib.h>

main()
{
image a;
ImageAssign(a,(long)(getvar(CAPT_START)),256,256,768);

sobel(&a, &a);

...
```

memory

none

see also

`ff3()`, `ff5()`, `robert()`

sobel

Sobel filter routine (macro)

synopsis

```
void sobel(image a, image b)
```

description

The function `sobel()` calculates the Sobel filter.

The Sobel filter is calculated with the following masks:

1	0	-1
2	0	-2
1	0	-1

1	2	1
0	0	0
1	2	1

The convolution with both masks is executed, the absolute values of both results are added, and the result is divided by 4.

`sobel()` is a macro which calls `imgf()` with basic function `sobelf()` as an argument.

laplace

Laplace filter routine (macro)

synopsis

```
void laplace(image a, image b)
```

description

The function `laplace()` calculates the Laplace filter.

The Laplace filter is calculated with the following mask:

0	1	0
1	-4	1
0	1	0

The convolution with the mask is executed, the magnitude is calculated, and the result is divided by 4.

`laplace()` is a macro which calls `imgf()` with basic function `laplacef()` as an argument.

mx

maximum filter routine (macro)

synopsis

```
void mx(image a, image b)
```

description

The function `mx()` calculates the maximum filter.

The maximum filter is calculated as follows:
The pixel with the maximum gray scale in a 3x3 window is found.
The value of this pixel is used as the result of the filter.

`mx()` is a macro which calls `imgf()` with basic function `maxf()` as an argument.

mn **minimum filter routine (macro)**

synopsis
```
void mn(image a, image b)
```

description

The function `mn()` calculates the minimum filter.

The minimum filter is calculated as follows:
The pixel with the minimum gray scale in a 3x3 window is found.
The value of this pixel is used as the result of the filter.

`mn()` is a macro which calls `imgf()` with basic function `minf()` as an argument.

ff3 **3 x 3 filter with arbitrary mask**

synopsis
```
void ff3(image *a, image *b, int c[3][3], int sh)
```

description

The function `ff3()` makes it possible to calculate 3 x 3 filter operations of the image variable a. In contrast to `imgf()`, this is always a convolution with a 3x3 mask.

The two-dimensional array `c[3][3]` contains the coefficients for the convolution.

Mask:

c11	c12	c13
c21	c22	c23
c31	c32	c33

The convolution with the mask is executed, the magnitude is calculated, and the result is shifted `sh` bits.
`sh=0` means no shift, `sh=1` means multiply by 2, `sh=-1` is equivalent to dividing by 2, etc.

memory none

see also `imgf()`, `ff5()`, `robert()`

ff5 — 5 x 5 filter for image variable

synopsis

```
void ff5(image *a, image *b, int c[5][5], int sh)
```

description

The function `ff5()` performs the 5 x 5 filter operation of image variable `a` with arbitrary mask `c[5][5]`. The result is stored in image `b`.

The two-dimensional array `c[5][5]` contains the coefficients for the convolution mask.

Mask:

c00	c01	c02	c03	c04
c10	c11	c12	c13	c14
c20	c21	c22	c23	c24
c30	c31	c32	c33	c34
c40	c41	c42	c43	c44

The convolution with the mask is executed, the magnitude is calculated, and the result is shifted `sh` bits. `sh=0` means no shift, `sh=1` means multiply by 2, `sh=-1` is equivalent to dividing by 2, etc.

example

```
static int c[5][5];

/* Mask: */

c[0][0]=1; c[0][1]=0; c[0][2]=0; c[0][3]=0; c[0][4]=-1;
c[1][0]=1; c[1][1]=0; c[1][2]=0; c[1][3]=0; c[1][4]=-1;
c[2][0]=1; c[2][1]=0; c[2][2]=0; c[2][3]=0; c[2][4]=-1;
c[3][0]=1; c[3][1]=0; c[3][2]=0; c[3][3]=0; c[3][4]=-1;
c[4][0]=1; c[4][1]=0; c[4][2]=0; c[4][3]=0; c[4][4]=-1;

ff5(&a,&a,&c[0][0],-2);
```

memory

none

see also

`ff3()`, `ff5y()`, `imgf()`, `robert()`

ff5y

5 x 5 filter for image variable horizontal / vertical separation

synopsis

```
void ff5y(image *a, image *b, int pm *h,
                              int pm *v, int sh)
```

description

The function `ff5y()` performs a 5 x 5 filter operation of image variable `a`.
The filter consists of a 1x5 and a 5x1 mask which are executed in sequence.
The horizontal 5x1 mask is specified with array h[5], the vertical 1x5 mask is specified with array v[5].
The result of the operation will be stored in image `b`.
Horizontal mask:

h0	h1	h2	h3	h4

Vertical mask:

v0
v1
v2
v3
v4

The convolution with both mask is executed, the magnitude is calculated, and the result is shifted `sh` bits. `sh=0` means no shift, `sh=1` means multiply by 2, `sh=-1` is equivalent to dividing by 2, etc.

example

```
static int h[5];
static int v[5];

/* Masks: */

h[0]=1; h[1]=1; h[2]=1; h[3]=1; h[4]=1;

v[0] = -1;
v[1] =  0;
v[2] =  0;
v[3] =  0;
v[4] =  1;

ff5y(&a,&a,h,v,-2);
```

memory

`20*((b->dx)/2 + 1)` bytes of DMEM heap

see also

`ff3()`,`ff5()`,`f5hf()`,`f5vf()`,`imgf()`,`robert()`

robert **robert's cross operator of an image variable**

synopsis

```
void robert(image *src, image *dest)
```

description

The function `robert()` calculates the robert's cross filter operator of image variable `src` and outputs the result in image variable `dest`.

The operator uses the following masks:

1	0
0	-1

and

0	1
-1	0

The sum of the absolute values of each mask operation is calculated, the result is right-shifted by 1 (divided by 2) and output to the destination image.

memory none

see also `sobel()`, `imgf()`

projh — horizontal projection of an image variable

synopsis

```
void projh(image *a, U32 result[dy])
```

description

The function `projh` calculates the horizontal projection of a image variable.
Here, projection means the sum of all pixels in one line.
The result is stored in the array `result[dy]`. `result[0]` is the projection of the first line, `result[1]` the projection of the second line, etc.

Earlier VCLIB version had the restriction: dx<256. **This is no longer the case.**

example

```
#include <vclib.h>
#include <flib.h>
#define A_DY 512

main()
{
image a;
ImageAssign(a,(long)(getvar(CAPT_START)),640,A_DY,768);
U32 x[A_DY];
projh(&a, x);
...
```

see also `projv()`

memory none

projv — vertical projection of an image variable

synopsis

```
void projv(image *a, U32 result[dx])
```

description

The function `projv` calculates the vertical projection of an image variable.
Here, projection means the sum of all pixels in one column.
The result is stored in the array `result[dx]`. `result[0]` is the projection of the first column, `result[1]` is the projection of the second column, etc.

Earlier VCLIB version had the restriction: dy<256. **This is no longer the case.**

example

```
#include <vclib.h>
#include <flib.h>
#define A_DX 512

main()
{
image a;
ImageAssign(a,(long)(getvar(CAPT_START)),A_DX,480,768);
U32 x[A_DX];
projv(&a, x);
...
```

see also `projh()`

memory none

look — look-up table function

synopsis

```
void look(image *a, image *b, U32 table[256])
```

description

The function `look` transforms the image variable a with the aid of a look-up table function. The result of the operation is stored in image variable b, which may be identical to a. `table` is the transformation table, which must have been created beforehand.

If the format of the image variable (dx, dy) is not identical, the format of the result variable b is used. In particular, this means that the result of the operation is not defined if the image format of a is smaller than that of b.

(a->dx < b->dx or a->dy < b->dy)

You are recommended to work with identical image formats, i.e.
a->dx = b->dx and a->dy=b->dy

If a pixel initially had the value 0 <= x < 256, then after the transformation with the function `look()` its value will be `table[x]`.

memory

none

focus — calculate the focal value of an image variable

synopsis

```
U32 focus(image *a, I32 sh)
```

description

The function `focus` calculates the focal value of the image variable a.
For details of how the focal value is calculated, please refer to the description of the basic function `focusf()`.
`sh` is a shift value to suppress noise.
`sh=0` means no shift, `sh=-1` is equivalent to dividing by 2,
`sh=-2` is equivalent to dividing by 4, etc.

The focal value is calculated according to the following procedure:

a1 a2
b1 b2

a1 and a2 are adjacent pixels in the upper line,
b1 and b2 are adjacent pixels in the lower line.

We have: $f := \sum ((|a1 - a2| + |a1 - b1|) >> (-sh))$

The summation is performed for all ((dx-1)*(dy-1)) pixels of the image variable.

Problem:
The focal value calculated by the above formula depends upon the mean brightness of the image field used.
Thus, the formula is only recommended for evaluating the focus if the individual images used for the comparison have similar values for the mean brightness.

memory

none

mean — calculate the mean value of an image variable

synopsis

```
U32 mean(image *a)
```

description

The function `mean()` calculates the mean value of an image variable. The value is rounded and returned to the calling function.
There is no restriction for the format of the image like for earlier VCLIB versions. The mean is the sum of all pixel values in the image divided by the image area.

memory

none

see also

`variance()`, `histo()`, `projh()`, `projv()`

variance — calculate the variance of an image variable

synopsis

```
U32 variance(image *a)
```

description

The function `variance()` calculates the statistical variance of an image variable. The value is rounded and returned to the calling function.
There is no restriction for the format of the image like for earlier VCLIB
The variance is the sum of all pixel values squared divided by the image area.
The variance can be used to measure the contrast, e.g the presence or absence of high contrast structures like printing, etc.

memory

none

see also

`mean()`, `histo()`, `projh()`, `projv()`

pyramidx — pyramid filter for image variable

synopsis

```
void pyramidx(image *a, image *b, void (*func)())
```

description

The function `pyramidx()` computes the pyramid filter operation of an image defined by image variable `a`. The result is stored in image `b`.

a1	a2
a3	a4

Four values of the source image are combined to one pixel of the destination image. The nature of the operation is defined by the basic function `func()`.

The following macros are available:

Call	Basic function
`pyramid(a, b)`	`FL_2x2_Mean_U8P_U8P()`
`pyr_max(a, b)`	`FL_2x2_MAX_U8P_U8P()`
`pyr_min(a, b)`	`FL_2x2_MIN_U8P_U8P()`

Please note that the result image is smaller by the factor of two in both the horizontal and vertical direction.

The operation may be performed in-place, i.e. `a` and `b` may be equal.

memory

none

see also

`pyramid()`, `pyr_max()`, `pyr_min()`, `subsmpl()`

pyramid **pyramid filter for image variable (macro)**

synopsis

```
void pyramid(image *a, image *b)
```

description

The function `pyramid()` computes the pyramid filter operation of an image defined by image variable `a`. The result is stored in image `b`.

a1	a2
a3	a4

Four values of the source image are combined to one pixel of the destination image according to the following formula:

result = (a1+a2+a3+a4)/4

`pyramid()` is a macro which calls `pyramidx()` with basic function `FL_2x2_Mean_U8P_U8P()` as an argument.

pyr_max **pyramid filter maximum for image variable (macro)**

synopsis

```
void pyr_max(image *a, image *b)
```

description

The function `pyr_max()` computes the pyramid filter operation of an image defined by image variable `a`. The result is stored in image `b`.

a1	a2
a3	a4

Four values of the source image are combined to one pixel of the destination image according to the following formula:

result = max(a1, a2, a3, a4)

`pyr_max()` is a macro which calls `pyramidx()` with basic function `FL_2x2_MAX_U8P_U8P()` as an argument.

pyr_min **pyramid filter minimum for image variable (macro)**

synopsis

```
void pyr_min(image *a, image *b)
```

description

The function `pyr_min()` computes the pyramid filter operation of an image defined by image variable `a`. The result is stored in image `b`.

a1	a2
a3	a4

Four values of the source image are combined to one pixel of the destination image according to the following formula:

result = min(a1, a2, a3, a4)

`pyr_min()` is a macro which calls `pyramidx()` with basic function `FL_2x2_MIN_U8P_U8P()` as an argument.

subsample **subsample image (image variable)**

synopsis

```
void subsample(image *a, image *b, I32 rh, I32 rv)
```

description

The function `subsample()` copies the image defined by image variable a into image variable b and reduces its size.

`rh` and `rv` specify the horizontal (`rh`) and vertical (`rv`) subsampling ratio.

constraints:

```
rh, rv > 0
```

`rh=2` means that every other pixel of an original image line will be taken and stored to the result image. The horizontal image witdh of the result image will be half the source image width.

example

```
image a = {sta, 512, 512, 768};
image b = {stb, 128, 128, 768};

subsample(&a, &b, 4, 4);
```

memory none

see also `pyramid()`

arx **calculate the number of pixels above threshold of image variable**

synopsis

```
U32 arx(image *a, I32 thr)
```

description

The function `arx()` calculates the number of pixels above the threshold `thr` according to the following equivalent c program:

```
int i, cnt=0;

for(i=0;i<n;i++)
  if(*p++ > thr) cnt++

return(cnt);
```

memory none

see also `arx2()`

arx2 **calculate the number of pixels between two thresholds of image variable**

synopsis

```
U32 arx2(image *a, I32 th1, I32 th2)
```

description

The function `arx2()` calculates the number of pixels between the thresholds `th1` and `th2`.

memory none

see also `arx()`

bin0 — **fast binarization of an image variable**

synopsis

```
void bin0(image *src, image *dest, I32 thr,
                  I32 bl, I32 wt, void *(*fc)());
```

description

bin0() binarizes image src and writes the result to dest, which may be equal to src. thr is the threshold, bl the grey value for the display of binary „black“, wt the greyvalue for the display of binary „white“.

fc is a function pointer to the basic binarisation function.

The following macros are available:

Call	I/O function
binarize(s, d, t, b, w)	FL_SetBin_U8P_U8P()
PaintWhite(s, d, t, w)	FL_SetIfGE_U8P_U8P()
PaintBlack(s, d, t, b)	FL_SetIfLT_U8P_U8P()

binarize(): If the pixel value is < thr, the resulting pixel will have the value given by b, otherwise the value will be w.
PaintWhite (): If the pixel value is < thr, the pixel value is not changed, otherwise the value will be w.
PaintBlack (): If the pixel value is < thr, the resulting pixel will have the value specified by b, otherwise the value will not be changed.

Of course, you can write your own basic binarisation functions. Pass their address (function pointer) to bin0().

memory none

see also look()

avg, avg2 — moving average or unsharp masking of an image variable

synopsis

```
I32 avg(image *a, image *b, I32 kx, I32 ky,
                              void (*func)(), I32 v)

I32 avg2(image *a, image *b, I32 kx, I32 ky,
                              void (*func)(), I32 v)
```

description

The function `avg()` calculates the moving average filter of image variable `a` and stores the result in image variable `b`.
The size of the moving average is specified with the values `kx` (horizontal kernel size) and `ky` (vertical kernel size).
Images specified by image variables a and b **must be different**.

If `void (*func)()` is zero, the function will calculate the moving average. If a function address is given, the original image will be subtracted from the moving average, performing an „unsharp masking“ operation.

For `avg()` the result image will be centered according to the kernel size `(kx, ky)`, i.e. the (smaller) result image will start at location

```
b->st +  kx/2 + (ky/2) * b->pitch
```

For `avg2()` the result will be placed in the left upper corner of b.

The function pointer passed specifies the type of subtraction being performed.

The return value is negative, if an error is encountered.

The following macros are available:

Call	subtract function
`avgm(a, b, kx, ky)`	`void (*)()0`
`maskx(a, b, kx, ky, offset)`	`sub2x()`
`masky(a, b, kx, ky)`	`sub2y()`
`avgm2(a, b, kx, ky)`	`void (*)()0`
`maskx2(a, b, kx, ky, offset)`	`sub2x()`
`masky2(a, b, kx, ky)`	`sub2y()`

`avgm(a,b,kx,ky)`	`moving average`
`maskx(a,b,kx,ky,offs)`	`b = (a - avg + offs); clipping if c>255 or c<0`
`masky(a,b,kx,ky)`	`b = (a - avg)>0 ? 255 : 0`

Of course, you can write your own subtract functions. Pass their address (function pointer) to `avg()`.

memory

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

see also

`ff3(), ff5()`

zoom_up **enlargement of an image variable**

synopsis `void zoom_up (image *a, image *b, I32 factor)`

description The function `zoom_up()` enlarges the pixels in image variable `a` by `factor` and stores the result in image variable `a`.

If the size of `b` is not sufficient for this operation the maximum size will be truncated to the size of `b`.

memory none

Gray scale correlation routines

vc_corr0	small kernel correlation routine / extended search area
vc_corr1	same as `vc_corr0()`
contrast	contrast calculation

vc_corr0 — small kernel correlation routine / extended search area

synopsis

```
I32 vc_corr0 (image *a, image *b, I32 mcn,
                        int mcr, I32 *x0, I32 *y0)
```

description

The function `vc_corr0()` 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_corr0`() is intended for use with small kernels and small images.

Valid kernel sizes must comply to `kx*ky <= 256`, e.g. `16x16` or `10x25`. 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_corr0()` returns the correlation coefficient for the pattern found. If no pattern is found (due to low contrast or low correlation) it will return –1.
The function also returns the `x0` and `y0` coordinates of the closest match.

`vc_corr0` is quite fast. The following table gives some benchmark values for a VC2038 with 150MHz:

Kernel size (kx*ky)	Image size (dx*dy)	processing time
16x16	64x64	25 msec
16x16	120x120	100 msec
16x16	160x120	130 msec

memory

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

see also

vc_corr1 **small kernel correlation routine / extended search area**

synopsis

```
I32 vc_corr1 (image *a, image *b, I32 mcn,
                       I32 mcr, I32 *x0, I32 *y0)
```

description

The function `vc_corr1()` has been replaced by `vc_corr0()` for VCLIB 3.0 since the latter provides an extended search area.

see also

`vc_corr0()`

contrast **contrast calculation**

synopsis

```
I32 contrast(image *a)
```

description

The function `contrast()` calculates the average contrast in image variable `a` Contrast is defined as the standard deviation of the grey values of all pixel. in the image variable. The maximum allowable size of the image variable is 65536 pixles (e.g. 256 x 256)

see also

`vc_corr0()`

Programs for JPEG compression / decompression

fwrite_jpeg	write image variable to JPEG image file / flash EPROM
cjpeg	encode image variable to JPEG image file
fread_jpeg	read JPEG image file / flash EPROM into image variable
djpeg	decode JPEG image file into image variable

fwrite_jpeg — write image variable to JPEG image file / flash EPROM

synopsis

```
I32 fwrite_jpeg(image *a, char *path,
                  I32 quality, U32 maxlng)
```

description

The function `fwrite_jpeg()` compresses image variable a to JPEG format according to the JPEG standard.
`quality` is a value between 0 and 100 indicating the resulting image quality.
A value near 0 indicates a low quality image (with high compression rate),
a value of 100 indicates a high quality image (with low compression rate).
In general, a compression rate of 10 - 20 can be expected, depending on the input image.

A file is created and all JPEG data will be stored in this file.
The path of the file is specified by the string `path`.

If a filesize of `maxlng` is reached and the JPEG generation process did not finish, the file is deleted afterwards, since it containes no useful information. In this case the function will return -1, otherwise 0. The function also returns –1 if the specified file could not be opened.

`maxlng` must be 22 at minimum, this is the size of the file-header and -trailer. It is recommended to use much larger values for `maxlng`, e.g. several kilobytes.

example

The following example compresses the image given by image variable a and stores the data in a JPEG file with name „jpeg“.

```
#include <vclib.h>
#include <flib.h>

main()
{
image a={sta, 256, 256, 768};
int err;

err=fwrite_jpeg(&a,"jpeg",80,0x10000);

if(err!=0) pstr("memory overrun\n");

...
```

memory

768 bytes of heap memory

see also

`fread_jpeg()`

cjpeg — encode image variable to JPEG image file

synopsis

```
U8 *cjpeg(image *a, I32 quality, U8 *addr,
                        U32 maxlng, I32 (*func)())
```

description

The function `cjpeg()` compresses image variable a to JPEG format according to the JPEG standard.
`quality` is a value between 0 and 100 indicating the resulting image quality. A value near 0 indicates a low quality image (with high compression rate), a value of 100 indicates a high quality image (with low compression rate). In general, a compression rate of 10 - 20 can be expected, depending on the input image.

The JPEG data output is passed to the I/O function `func()` which specifies the destination of the data and how these data are stored or transmitted. A pointer to this function must be passed to `cjpeg()`.

For the available I/O functions there are macros (#define instructions), which make it easier to call the function.

The following macros are available:

Call	I/O function
`cjpeg_d(img, qual, addr, maxlng)`	`wr_dram()`
`cjpeg_f(img, qual, addr, maxlng)`	`wr_flash()`
`cjpeg_a(img, qual)`	`wr_ascii()`
`cjpeg_b(img, qual)`	`wr_binary()`

Of course, you can write your own I/O functions. Pass their address (function pointer) to `cjpeg()`.

`cjpeg_d()` writes JPEG data to memory starting at address `addr`.

`cjpeg_f()` writes JPEG data to Flash Eprom starting at address `addr`. Since this is a raw write (no file information is provided) care must be taken to use this function. If you want to write a flash Eprom file with JPEG data, use function `fwrite_jpeg()` instead.

`cjpeg_a()` sends JPEG data to the serial RS232 interface as ASCII Hex characters. The data flow can be controlled by XON/XOFF handshaking by the receiving computer system.

`cjpeg_b()` sends JPEG data to the serial RS232 interface as binary (8 bits) data. The data flow can be controlled by XON/XOFF handshaking by the receiving computer system.

For all macros the variable `img` is the image variable a of the `cjpeg()` function call, `qual` is the corresponding quality factor.
`addr` and `maxlng` are start address and maximum length, the functions transferring data via the serial link do not need such variables.

`cjpeg_d()` and `cjpeg_f()`: If a data size of `maxlng` is reached and the JPEG generation process did not finish, the function will return 0L, otherwise it returns the next available address behind the JPEG data.

`cjpeg_a()` and `cjpeg_b()`: the function will return 1L if sucessfully finished. It may return 0L on occurence of some error in the I/O functions.
This is, however, not used with the I/O functions supplied.

example 1

The following example compresses the image given by image variable a and stores the data in memory.

```
#include <vclib.h>
#include <flib.h>

main()
{
image a={sta, 256, 256, 768};
U8 *next;

next=cjpeg_d(&a, 80, addr, 0x10000);

if(next==NULL) pstr("memory overrun\n");

...
```

example 2

The following example compresses the image given by image variable a and transmits it via RS232 as ASCII hex data.

```
#include <vclib.h>
#include <flib.h>

main()
{
image a={sta, 256, 256, 768};

cjpeg_a(&a, 80);

...
```

example 3

This following I/O function writes data bytes from memory and updates the emit-control *struct* . It can be used as an example to create your own I/O functions.

```
typedef struct
{
int put_bits;      /* number of bits already sent      */
long cde;          /* actual bit code                  */
U8 *ptr;           /* pointer to memory                */
U8 *last;          /* last available memory address    */
int  err;          /* error flag                       */
int  (*fc)();      /* write byte function pointer      */
} emit_ctrl;
```

```
void dr_dram(emit_ctrl *emc, int val)
{
U8 *addr=emc->ptr;

if(addr-emc->last <= 0)
  {
  *addr++ = val;
  emc->ptr=addr;
  }
else
  {
  emc->err=-1;
  }
}
```

memory 768 bytes of heap memory

see also `djpeg()`

fread_jpeg — read JPEG image file / flash EPROM into image variable

synopsis `I32 fread_jpeg(image *a, char *path)`

description The function `fread_jpeg()` decompresses a JPEG file and displays the result in image variable a.

A file with a path given by `path` is searched, the data is decompressed and the image is written into image variable a.

The function returns -1 if the file could not be opened, it returns -2 if the image could not be displayed. That may be the case, if the JPEG image size is larger than the image variable size. Under normal conditions, the function will return 0.

example The following example searches a file with name „jpeg“, decompresses the image and stores the result in image variable a.

```
#include <vclib.h>
#include <flib.h>


image a = {sta, 740, 574, 768};
int err;

err=fread_jpeg(&a,"jpeg");

if(err != 0) pstr("jpeg error\n");

...
```

memory 768 bytes of heap memory

see also `fwrite_jpeg()`

djpeg — decode JPEG image file into image variable

synopsis

```
U8 *djpeg(image *a, U8 *addr, I32 (*func)())
```

description

The function `djpeg()` decompresses a JPEG file and displays the result in image variable a.

The JPEG data input is provided by the I/O function `func()` which specifies the source of the data and how these data are read or transmitted.
A pointer to this function must be passed to `djpeg()`.

For the available I/O functions there are macros (#define instructions), which make it easier to call the function.

The following macros are available:

Call	I/O function
`djpeg_d(img, addr)`	`rd_dram()`
`djpeg_f(img, addr)`	`rd_flash()`
`djpeg_a(img)`	`rd_ascii()`
`djpeg_b(img)`	`rd_binary()`

Of course, you can write your own I/O functions. Pass their address (function pointer) to `cjpeg()`.

`djpeg_d ():` The JPEG data are read from memory starting at address addr, the resulting image is stored in image variable a.
The function returns the next memory address behind the JPEG code. If a format error occurs it will return 0L. That may be the case, if the JPEG image size is larger than the image variable size.

`djpeg_f ():` The JPEG data are read from flash eprom starting at address addr, the resulting image is stored in image variable a.
Since this is a raw read (no file information is used) care must be taken to use this function. If you want to read a flash Eprom file with JPEG data, use function `fread_jpeg()` instead.
The function returns the next flash eprom address behind the JPEG code. If a format error occurs it will return 0L. That may be the case, if the JPEG image size is larger than the image variable size.

`djpeg_a ():` The JPEG data are read from the RS232 serial interface in ASCII hex format, the resulting image is stored in image variable a.
If a format error occurs it will return 0L. That may be the case, if the JPEG image size is larger than the image variable size. If the function exectutes correctly, it will return 1L.
The function uses XON/XOFF handshaking to control the data flow.

`djpeg_b ():` The JPEG data are read from the RS232 serial interface in binary format (8 bits), the resulting image is stored in image variable a.
If a format error occurs it will return 0L. That may be the case, if the JPEG image size is larger than the image variable size. If the function exectutes correctly, it will return 1L.
The function uses XON/XOFF handshaking to control the data flow.

example 1

The following example decodes the JPEG data in memory starting at address `addr` and displays the image in image variable a.

```
#include <vclib.h>
#include <flib.h>

main()
{
image a = {sta, 256, 256, 768};
U8 *addr, *next;

next=djpeg_d(&a, addr);

if(next==NULL) pstr("jpeg error\n");

...
```

example 2

This following I/O function reads data bytes from memory and updates the fct-control *struct* . It can be used as an example to create your own I/O functions.

```
typedef struct
{
U8 *ptr;           /* memory address                  */
int bits_left;     /* # of unused bits in it          */
long buffer;       /* bit buffer                      */
int (*fc)();       /* read byte function pointer      */
} fill_ctrl;

int rd_dram(fill_ctrl *fct)
{
return(rpix((fct->ptr)++));
}
```

memory

768 bytes of heap memory

see also

`cjpeg()`

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

rlcmalloc	allocate RLC memory
rlcfree	free RLC memory
rlcmk	create run length code for an image variable
rlcout	output run length code
parse_rlc	parse RLC and return next available address
rlc_inv	in-place inversion of RLC
rlc2	logical link of two images in run length code
erxdi	erosion / dilation of run length code / square typ
erxdi2	erosion / dilation of run length code / diagonal typ
testrlc	create RLC test image (chess-board)
rlc_mf	horizontal „median filter“ for RLC
fwrite_rlc	write RLC to flash EPROM
fread_rlc	read RLC from flash EPROM
rlc_move	move RLC
rlc_area	calculate area in run length code
rlc_feature	determine features in unlabelled RLC
sgmt	segment run length code (object labeling)

rlcmalloc — allocate memory for RLC (macro)

synopsis

```
U16 *rlcmalloc(U32 size)
```

description `rlcmalloc` returns a pointer to a heap memory area for size RLC items (U16)

rlcfree — free RLC memory (macro)

synopsis

```
void rlcfree(U16 *rlc)
```

description `rlcfree()` releases RLC memory previously allocated with `rlcmalloc()`.

rlcmk **create run length code for an image variable**

synopsis `U16 *rlcmk(image *a, I32 thr, U16 *rlc, I32 size)`

description The function `rlcmk()` creates run length code for the image variable a and stores it in memory. `thr` is the threshold value used for binarization `0 <= thr < 256.` A pixel with a gray scale g >= thr 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.

Four data words (U16) are placed before the RLC itself, which are required for image reconstruction and for labelled RLC.
The address of the segment label code comes first. `rlcmk()` enters (void *)0 here, to show that the RLC is not yet labelled.
The horizontal (a->dx) and vertical (a->dy) image size follows, in order to later reconstruct the image format.

Address	Value	
rlc	0	SLC address (0, if unlabelled)
rlc+1	0	SLC address (0, if unlabelled)
rlc+2	dx	
rlc+3	dy	
rlc+4	first change in the first line	
...		
rlc+n	dx / end of the first line	
rlc+n+1	first change in the second line	
...		

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 `rlcout()`

memory none

parse_rlc **parse RLC and return next available address**

synopsis `U16 *parse_rlc(U16 *rlc)`

description `parse_rlc` parses the RLC specified by pointer `rlc` and returns the next available memory address (integer aligned) right behind the RLC.

memory none

rlcout — output run length code

synopsis

```
I32 rlcout(image *a, U16 *rlc, U32 dark, U32 bright)
```

description

The function `rlcout()` makes it possible to convert run length code to a gray image.
This is mostly used to display the run length code on the screen.
However, this function can also be used to perform image processing operations which are not possible directly in the run length code, or which would be difficult in it.

The image variable a provides the start address and the pitch for the output. The function immediately aborts with error code -1 if the image format (dx, dy) implicitly contained in the run length code does not agree with a->dx and a->dy of the image variable.

`rlc` is the start address of the run length code in memory, dark is the gray scale for the black areas of the RLC, bright is the gray scale for the white areas - here, values between 0 and 255 are possible.

return values:

0: no error
-1 format error

see also

`rlcmk()`

memory

none

rlc_inv — in-place inversion of RLC

synopsis

```
U16 *rlc_inv(U16 *rlc)
```

description

The function `rlc_inv()` performes the in-place inversion of **RLC** stored at address `rlc` in memory.
Inversion means, that black segments are changed to white and vice versa. The inversion is obtained by negating the color information at the start of each line.
`rlc_inv()` returns the address of the next item to follow the **RLC** code (integer aligned).

memory

none

rlc2 **logical link 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 links between two run length codes.
`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`.
The RLCs to be linked must have the identical format (dx, dy). If this is not the case, then the function returns NULL.
Otherwise, 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 link.

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
`rlcnand(a, b, dest)`	`rlc_nandf()`	NAND
`rlcnor(a, b, dest)`	`rlc_norf()`	NOR
`rlcequiv(a, b, dest)`	`rlc_equivf()`	EQUIV

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

memory none

erxdi — erosion / dilation of run length code / square typ

synopsis

```
U16 * erxdi(U16 *src, U16 *dest, U16 *(*fc1)(),
                                  U16 *(*fc2)())
```

description

The function `erxdi()` erodes/dilates the image by one pixel. Erosion means that all white areas in the RLC become one pixel wider in each direction, while all black areas are narrowed one pixel.
Thus, black areas which are 1 or 2 pixels in diameter disappear completely.

`src` is a pointer to the source RLC, `dest` to the destination RLC.

The function pointers passed specify the type of operation being performed. `fc1` is the function pointer for the horizontal erosion/dilation, `fc2` is the function pointer for the vertical erosion/dilation.

The following macros are available:

Call	horizontal function	vertical function
`erode(a, b)`	`rlc_xero`	`rlc_orf`
`dilate(a, b)`	`rlc_xdil`	`rlc_andf`

Of course, you can write your own horizontal and vertical functions. Pass their address (function pointer) to `erxdi()`.

memory

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

see also

`erxdi2(), rlc_mf(), mx(), mn()`

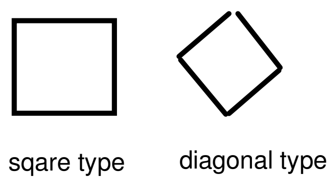
erxdi2 **erosion / dilation of run length code / diagonal typ**

synopsis

```
U16 * erxdi2(U16 *src, U16 *dest, U16 *(*fc1)(),
                               U16 *(*fc2)())
```

description

The function `erxdi2()` erodes/dilates the image by one pixel. It is most similar to the `erxdi2()` function, but instead of a square as structuring element, it uses a diagonal (diamond-shaped) structuring element.
The influence of the structuring element becomes apparent, if the function is called several times on the data of the previos erosion / dilation.



A round shaped structuring element can be approximated by alternating the calls of `erxdi`() and `erxdi2`(), this procedure will produce an octagonal shaped structuring element which is much closer to a circle.

`src` is the source RLC, `dest` is the destination RLC.

The function pointers passed specify the type of operation being performed. `fc1` is the function pointer for the horizontal erosion/dilation, `fc2` is the function pointer for the vertical erosion/dilation.

The following macros are available:

Call	horizontal function	vertical function
erode2(a, b)	rlc_xero	rlc_orf
dilate2(a, b)	rlc_xdil	rlc_andf

Of course, you can write your own horizontal and vertical functions. Pass their address (function pointer) to `erxdi()`.

memory

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

see also

`erxdi()`, `rlc_mf()`, `mx()`, `mn()`

testrlc **create RLC test image (chess-board)**

synopsis

```
U16 *testrlc(U16 *rlc, I32 dx, I32 dy, I32 size)
```

description

The function `testrlc()` creates a testimage in RLC format.
`rlc` is the start address of the RLC, where the testimage is written to.
`dx` and `dy` is the horizontal and vertical size of the image.
`size` is the size of the individual chess-board squares.
`dy` must be a multiple of `size`, it will be rounded off otherwise

The function returns a pointer to the next available memory word (U16) to follow the RLC code (integer aligned).

The function can be used to test functionality and execution timing of RLC functions including object labelling.

A test image of size 640 x 480 with a chess-board square size of 32 pixels needs 10084 words (U16) of RLC, which is a good approximation of the average information amount of a „real life“ image of that format.

memory 4*(dx+1) bytes of heap memory

example

```
U16 *r0;
r0=(U16 *)rlcmalloc(12000);
testrlc(r0,640,480,32);
rlcout(&a, r0, 0, 255);
```

rlc_mf **horizontal „median filter“ for RLC**

synopsis

```
U16 *rlc_mf(U16 *src, U16 *dest, I32 col, I32 lng)
```

description

`rlc_mf()` performs the horizontal median filter for RLC.
The median filter purges all structures of color `col` with length less than `lng`.

The operation will create less data at address `dest` than the original RLC at address `src`. Moreover, the operation may be performed in-place, i.e. `src` and `dest` be be the same.

The function is valuable to reduce the amount of useless information in noisy images in an early stage of RLC processing.

The function returns a pointer to the next available memory word (U16) to follow the RLC code (integer aligned).

memory none

see also `erxdi()`, `erxdi2()`, `mx()`, `mn()`

fwrite_rlc **write RLC to flash EPROM**

synopsis `I32 fwrite_rlc(char *path, U16 *rlc)`

description `fwrite_rlc()` creates a flash EPROM file and writes the RLC starting at address `rlc` to this file.

The full path of the file is specified by the string `path` .

If the function is unable to open the specified file, it returns –1, otherwise 0.

memory none

see also `fread_rlc()`

fread_rlc **read RLC from flash EPROM**

synopsis `U16 * fread_rlc(char *path, U16 *rlc)`

description `fread_rlc()` opens a flash EPROM file and writes the RLC of this file to meory at address `rlc`.

The full path of the file is specified by the string `path` .

The function returns a pointer to the next available memory word to follow the RLC code (integer aligned).

memory none

see also `fwrite_rlc()`

rlc_move **move RLC**

synopsis `U16 *rlc_move(U16 *src, U16 *dest, I32 mx, I32 my)`

description `rlc_move()` reads RLC line at memory address `src`, moves all RLC items horizontally by `mx` pixels (`mx` negative: move left), vertically by `my` lines (`my` negative: move up) and outputs the result to `dest`.
`src` and `dest` must be different for this operation.

Black space (color=0) is added for the regions outside the original window.

memory none

rlc_area **calculate area in run length code**

synopsis

```
U32 rlc_area(U16 *rlc, I32 color)
```

description

The function `rlc_area()` calculates the area of all pixels of a given color (black: color = 0, white: color = -1) in the unlabelled RLC.
All pixels of a given color (black or white) are included. There is **no** differentiation according to **objects**.

`rlc` is the start address of the run length code in memory.
The return value of this function is the area.

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

memory none

rlc_feature **determine feature in unlabelled RLC**

synopsis

```
void rlc_feature(feature *f, U16 *rlc, I32 color)
```

description

The function `rlc_feature()` calculates features in the unlabelled RLC.
The parameter `color` can be used to specify if the features for all black (color = 0) or all white (color = -1) pixels of the RLC should be calculated.
All pixels of a given color (black or white) are included. There is **no** differentiation according to **objects**.

The following features are calculated:

area:	area
x_center:	x-coordinate of the center of gravity
y_center:	y-coordinate of the center of gravity
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

The maximum and minimum values of x and y define the bounding box around the pixels chosen with `color`.
The point with coordinates (x_lst,y_max) is a point which can serve as the initial point of the object's contour, if the chosen pixels are contiguous.

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

`f` is a pointer to the feature list stored in the following *struct*.

```
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                     */
  } feature;
```

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

see also `rlc_area()`, `rl_ftr2()`

memory none

sgmt — **segment run length code (object labelling)**

synopsis

```
U16 *sgmt(U16 *rlc, U16 *slc)
```

description

The function `sgmt()` 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, is also passed to the function - enough memory must be available for the memory needs of the SLC. (The SLC needs (size_of_rlc – 6) bytes of memory)
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).
It returns NULL in case of object number overrun. In this case, the number of objects field in the SLC is also set to zero.

memory

256000 bytes of heap memory (= 8 * 32000)

Programs for processing binary images in labelled run length code

dispobj	output labelled run length code
rlc_cut	cut individual objects out of the labelled RLC
rl_area2	calculate object area in the labelled RLC
rl_ftr2	calculate object features in the labelled RLC
chkrlc	check RLC

dispobj — output labelled run length code

synopsis

```
int dispobj(image *a, U16 *rlc)
```

description

The function `dispobj()` serves to output the labelled RLC for test purposes.
The various objects contained in the labelled RLC are displayed with different gray scales.
Otherwise, the output is basically equivalent to the function `rlcout()`.

memory

none

rlc_cut — cut individual objects out of the labelled RLC

synopsis

```
U16 *rlc_cut(U16 *src, U16 *dest, I32 objnum)
```

description

The function `rlc_cut()` copies individual connected pixel areas (objects) out of the labelled RLC.
`src` is the *labelled* source RLC, `dest` is the *unlabelled* target RLC. `objnum` is the number of the object to be copied.
All objects copied out (including black ones) are stored white on a black background in the target RLC.
The return value of this function is the address of the next available memory space immediately after the target RLC.
The function is aborted and NULL is returned if `src` is unlabelled or if `objnum` is larger than the number of objects contained in the RLC.

memory

no heap space required

rl_area2 — calculate object area in the labelled RLC

synopsis

```
I32 rl_area2(U16 *rlc, U32 *area, U32 n)
```

description

The function `rl_area2()` calculates the area of all objects in the labelled RLC.

`rlc` is the start address of the labelled RLC in memory.
`area` is an array for the object areas, and `n` is the maximum number of objects, i.e. usually the dimension of the array.

After the function `rl_area2()` is called, the object areas for all objects (independent of their colors) are available in the array `area`.

The function returns the number of objects in the labelled RLC.

see also `rlc_area()`

memory none

example

```
U16 *next, *rlc;
U32 area[2048];
I32 nobj;

next = rlcmk(&a, 128, rlc, 0x40000);
next = sgmt(rlc, next);
nobj = rl_area2(rlc, area, 2048);
```

rl_ftr2 — calculate object features in the labelled RLC

synopsis

```
I32 rl_ftr2(U16 *rlc, ftr *f, U32 n)
```

description

The function `rl_ftr2()` 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_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.

`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     */
  } 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 return value of the function is the number of objects in the labelled RLC.

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

memory no heap space required

example

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

next = rlcmk(&a, 128, rlc, 0x40000);
next=sgmt(rlc,next);
nobj=rl_ftr2(rlc, f, 100);
```

chkrlc **check RLC**

synopsis

```
int chkrlc(U16 *rlc)
```

description

Problems usually occur with functions which use run length code *if the RLC contains errors*. For example, even the function `rlcout()`, which outputs the RLC on the screen, may crash if called with faulty RLC.
The functions in the library have been checked for correctness, so problems are not to be expected. If the user, however, develops an own function for RLC processing, it is recommended to check the RLC during the development process. The function `chkrlc()` may be used for this purpose.
The function should not be used for the final program, since

- it was not optimized for speed
- it outputs debug information via the serial communication link

In particular, this function checks the following:

- labelled or unlabelled RLC
  for labelled RLC the SLC address and the number of objects are printed
- dx and dy are printed
- negative RLC values
- nonincreasing, i.e., constant or decreasing RLC values per line
- RLC values greater than dx

For labelled RLC, the following is also checked:

- two identical sequential SLC values
- SLC values greater than the maximum number of objects

In case of error, the function returns -1, otherwise 0. Debug information is printed during execution.

Programs for processing contour code(CC)

contour8	Contour following / 8-connected
cdisp	display contour
ccxy	convert contour code into xy-array

contour8 — Contour following / 8-connected

synopsis

```
I32 contour8(image *a, I32 x0, I32 y0, I32 dir, I32 thr,
                        U32 lng, U32 **dst)
```

description

This function generates the contour code (CC) of an object or an edge starting at (x0,y0) in image variable a.
`dir` has two meanings: The value of `dir` indicates the direction in which the contour has been found (according to the contour code values from 0=up to 7=upper left). In other words: in the reverse direction must be at least one white pixel.

The second meaning of `dir` is the direction of movement for the contour-following algorithm.

`dir > 0`	positive direction (counter-clockwise)
`dir < 0`	negative direction (clockwise)

negative values of `dir` are the logical NOT of the corresponding positive value (0..7).

`thr` is the threshold for the underlying binarisation. If `(pix < thr)` the pixel `pix` is assumed to be black.

`lng` is the maximum code length allowed (number of bytes in memory for contour). Since additional information is stored, there must be at least (16 + `lng`) bytes of memory available.

`**dst` is a handle for the destination address in memory. It will be updated to the next available address when the function finishes.

The function will only take black pixels as contour pixels. For this reason the starting pixel (x0,y0) must be black. It must also be a contour pixel which means, that it must have at least one white neighbor. If all neighbors are white, it is a so called isolated pixel - no contour code will be generated.

Although the pointer to the destination is a **U32 ***, the contour code itself is stored as byte values.

return values:

the function will return the following values:

`-1`	:	invalid starting pixel (not black) - no contour code generated
`0`	:	isolated pixel or pixel inside object - no contour code generated
`1`	:	closed contour (end pixel = starting pixel / same direction)
`2`	:	contour stops at left corner of image variable
`4`	:	contour stops at right corner of image variable
`8`	:	contour stops at upper corner of image variable
`16`	:	contour stops at lower corner of image variable
`32`	:	space exhausted (CC lenght > `lng`)

Some of the conditions could be true at the same time. (example: contour stops at the left upper corner of the image variable) In this case the individual codes will be added (example: 2+8 = 10)

memory: no heap space required

see also `cdisp()`

cdisp — display contour

synopsis

```
void cdisp(image *a, U32 *src, I32 col, void (*func)())
```

description

This function displays the contour code data (CC) of an object or an edge starting at address `src` in image variable `a`.
The contour will be displayed with color `col`.

The nature of drawing is specified by passing the pointer `(*func)()` to `cdisp()`.

The following macros are available:

Call	Drawing function	Remark
`cdisp_d(a, src, col)`	`wpix()`	DRAM write
`cdisp_x(a, src, col)`	`xorpix()`	DRAM XOR
`cdisp_o(a, src, col)`	`wovl()`	Overlay write
`cdisp_z(a, src, col)`	`xorovl()`	Overlay XOR

memory no heap space required

see also `contour8()`

ccxy — convert contour code (CC) into xy array

synopsis

```
I32 ccxy(I32 *src, I32 *xy, I32 *tbl, U32 maxcount)
```

description

This function converts contour code data (CC) of an object or an edge into a list of x/y-values stored beginning at address `xy`.
`src` is the source contour code (CC), `xy` the x/y coordinate list where the function places its output and `maxcount` is the maximum number of coordinates to be written to `xy` .

The function returns the number of contour pixels or –1 on overflow for `xy`.

`tbl` is a pointer to the following table:

```
int tbl[16] = { 0, 1, 1, 1, 0,-1,-1,-1,
               -1,-1, 0, 1, 1, 1, 0,-1};
```

Graphics functions

chprint	Output a string to an image variable
linexy	basic line creation routine
line	draw line
frame	draw frame
dframe	draw double-width frame
marker	draw marker
dmarker	draw double-width marker
EllipseXY	basic ellipse creation program (quarter)
ellipse	draw ellipse

chprint **Output a string to an image variable**

synopsis `void chprint(char *s, image *a, I32 cx, I32 cy)`

description `chprint()` outputs the string passed by `s` to the image variable `a`.
An 8 x 8 matrix is used for the character set.
`cx` and `cy` are the width and height of the characters in multiples of 8 pixels.

The characters are displayed in white (gray scale 255) on a black background (gray scale 0).

If the passed string cannot be displayed in the specified image variable, then it will be truncated to the displayable length.

No such check is made in the vertical direction.

This function is mostly used for displaying information on the screen.

see also `chprint_ov()`

memory no heap memory required

linexy **basic line creation routine**

synopsis

```
I32 linexy(I32 dx, I32 dy, I32 *xy)
```

description

The function `linexy()` creates a list of coordinates which creates the (x,y) coordinates for all pixels on a line.
The line begins at the origin `(0,0)` and ends at the point `(dx,dy)`.
This routine creates a list of (x,y) coordinates which are stored starting at the memory address specified by the pointer `xy`. The list contains first each x-coordinate and then the y-coordinate, respectively.
This kind of access is faster than other kinds. However, the storage type is selected such that you can also select other access types.
The function returns the number of generated line points minus 1.

1. Access as a two-dimensional array

```
I32 xyarr[1024][2];
I32 dx = 100;
I32 dy = 100;
I32 count, x, y;

count = linexy(dx, dy, xyarr) + 1;

x = xyarr[0][0];  /* x-coordinate of 1st pixel */
y = xyarr[0][1];  /* y-coordinate of 1st pixel */
```

2. Access as a struct array

```
typedef struct
  {
    int x;
    int y;
  } vcpt;

...

vcpt *xy;
I32 dx = 100;
I32 dy = 100;
I32 count, x, y;

xy = (vcpt *)vcmalloc(1024);

count = linexy(dx, dy, (I32 *)xy) + 1;

x = xy->x;        /* x-coordinate of 1st pixel */
y = xy->y;        /* y-coordinate of 1st pixel */

xy++;             /* address next coordinate */
```

The return value of the function is the number of coordinates created.

see also `line()`

line — draw line

synopsis

```
void line(image *a, I32 x1, I32 y1,
               I32 x2, I32 y2, I32 col, void (*func)() )
```

description

The function `line()` draws a line in video memory, or more precisely in the image variable `a`.
The line begins at coordinate `(x1,y1)` and ends at coordinate `(x2,y2)`, whereby both coordinates relate to the origin (upper left corner) of image variable `a`.
The line can be drawn normally, or as XOR in the gray image or in the overlay.

Caution: No check is made if the line to be drawn partially or entirely leaves the memory area of the image variable(s).

Therefore, you should make sure the following is true:

0 < x1 < a->dx	or	0 < x2 < a->dx
0 < y1 < a->dy	or	0 < y2 < a->dy

If the image variable `a` is part of a larger image variable, then of course going beyond the bounds of the memory area does not cause a problem.

`col` is the gray scale to be drawn.

The nature of drawing is specified by passing the pointer `(*func)()` to the drawing function itself.

The following macros are available:

Call	Drawing function
`lined(a, x1, y1, x2, y2, col)`	`wp_set32()`
`linex(a, x1, y1, x2, y2, col)`	`wp_xor32()`
`lineo(a, x1, y1, x2, y2)`	`wp_set32()`
`linez(a, x1, y1, x2, y2)`	`wp_xor32()`

memory

`8*(max{abs(x2-x1),abs(y2-y1)}+1)` bytes of heap memory

see also

`linexy()`

frame — draw frame

synopsis

```
void frame(image *a, I32 col, void (*func)())
```

description

The function `frame()` draws a frame in video memory, or more precisely in the image variable `a`.
The frame is drawn precisely on the margin of the image variable, i.e., in the first and last lines, and in the first and last columns of the image variables.

The frame can be drawn normally, or as XOR in the gray image or in the overlay.

The nature of drawing is specified by passing the pointer `(*func)()` to the drawing function itself.

The following macros are available:

Call	Drawing function
`framed(a, col)`	`wp_set32()`
`framex(a, col)`	`wp_xor32()`
`frameo(a)`	`wo_set32()`
`framez(a)`	`wo_xor32()`

memory

`8*(max{a->dx,a->dy}+1)` bytes of heap memory

see also

`dframe()`

dframe — draw double-width frame

synopsis

```
void dframe(image *a, I32 col, void (*func)())
```

description

The function `dframe()` draws a frame in video memory, or more precisely in the image variable `a`. The frame is drawn with a width of 2 pixels. With cameras based on the CCIR or EIA standard, this eliminates most of the half-image flicker.
The frame is drawn precisely on the margin of the image variable, i.e., in the lines 0, 1, dx-1 and dx-2, as well as in columns 0,1,dy-1 and dy-2.

The frame can be drawn normally, or as XOR in the gray image or in the overlay.

The nature of drawing is specified by passing the pointer `(*func)()` to the drawing function itself.

The following macros are available:

Call	Drawing function
`dframed(a, col)`	`wp_set32()`
`dframex(a, col)`	`wp_xor32()`
`dframeo(a)`	`wo_set32()`
`dframez(a)`	`wo_xor32()`

memory

`8*(max{a->dx,a->dy}+1)` bytes of heap memory

see also

`frame()`

marker **draw marker**

synopsis

```
void marker(image *a, I32 col, void (*func)())
```

description

The function `marker()` draws a marker in video memory, or more precisely in the image variable `a`.
The marker is drawn centered at the image variable.

The marker can be drawn normally, or as XOR in the gray image or in the overlay.

The nature of drawing is specified by passing the pointer `(*func)()` to the drawing function itself.

The following macros are available:

Call	Drawing function
`markerd(a, col)`	`wp_set32()`
`markerx(a, col)`	`wp_xor32()`
`markero(a, col)`	`wo_set32()`
`markerz(a, col)`	`wo_xor32()`

memory

`8*(max{a->dx,a->dy}+1)` bytes of heap memory

see also

`dmarker()`

dmarker **draw double-width marker**

synopsis

```
void dmarker(image *a, I32 col, void (*func)())
```

description

The function `dmarker()` draws a marker in video memory, or more precisely in the image variable `a`.
The marker is drawn with a width of 2 pixels. With cameras based on the CCIR or EIA standard, this eliminates most of the half-image flicker.
The marker is drawn centered at the image variable.

The marker can be drawn normally, or as XOR in the gray image or in the overlay.

The nature of drawing is specified by passing the pointer `(*func)()` to the drawing function itself.

For the available basic functions there are macros (#define instructions), which make it easier to call the function.

The following macros are available:

Call	Drawing function
`dmarkerd(a, col)`	`wp_set32()`
`dmarkerx(a, col)`	`wp_xor32()`
`dmarkero(a, col)`	`wo_set32()`
`dmarkerz(a, col)`	`wo_xor32()`

memory

`8*(max{a->dx,a->dy}+1)` bytes of heap memory

see also

`marker()`

EllipseXY **basic ellipse creation program (quarter)**

synopsis

```
int EllipseXY(I32 a, I32 b, I32 *xyc)
```

description

The function `EllipseXY()` creates a list of coordinates which creates the (x,y) coordinates for a quarter of an ellipse.
`a` and `b` are the two half axes of the ellipse, the ellipse is centered at the origin (0,0). The function only outputs positive values for x and y.
This routine creates a list of (x,y) coordinates which are stored starting at the memory address specified by the pointer `xyc`. The list contains first each x-coordinate and then the y-coordinate, respectively. The list should have a size of max(a,b) to assure proper operation.
The return value of the function is the number of coordinates created.

See documentation of linexy() function for examples.

see also

`ellipse()`, `linexy()`

ellipse — draw ellipse

synopsis

```
void ellipse(image *a, I32 col, void (*func)())
```

description

The function `ellipse()` draws an ellipse in video memory, or more precisely in the image variable `a`.
The ellipse fills the image variables, i.e. the ellipse is centered and the horizontal and vertical diameter are equal to the horizontal and vertical size of the image variable.

The ellipse can be drawn normally, or as XOR in the gray image or in the overlay.

Caution: No check is made if the ellipse to be drawn partially or entirely leaves the memory area.

`col` is the gray scale to be drawn.

The nature of drawing is specified by passing the pointer `(*func)()` to the drawing function itself.

The following macros are available:

Call	Drawing function
`ellipsed(a, c)`	`wp_set32()`
`ellipsex(a, c)`	`wp_xor32()`
`ellipseo(a)`	`wo_set32()`
`ellipsez(a)`	`wo_xor32()`

memory

`8*(max{(a->dx),(a-<dy)}+1)` bytes of heap memory

see also

`EllipseXY()`, `line()`

Programs for processing pixel lists

ad_calc32	address calculation for an array with x/y-coordinates
wp_list32	write video memory/access via address list
wp_set32	write video memory with constant/access via address list
wp_xor32	XOR video memory with constant/access via address list
rp_list32	read video memory/access via address list

ad_calc32 **address calculation for an array with x/y-coordinates**

synopsis

```
void ad_calc32(U32 count, I32 *xy,
              U8 *ad_list[], U8 *start, I32 pitch)
```

description

This function calculates the corresponding memory addresses for an array with x/y pairs.
It is especially efficient to combine `ad_calc32()` with functions such as `wp_list32(),rp_list32(),wp_set32(),` etc.

The addresses are calculated in accordance with the following C program:

```
for(i=0; i<count; i++)
  ad_list[i] = (U8 *)((U32)start + x[i] + y[i] * pitch);
```

The prototype for the two-dimensional array `xy[][2]` is specified as `I32 *xy`. This allows various types of access (see also the examples of the function `linexy()`).
The arrays xy[ ][2] and ad_list[ ] are allowed to be identical. The values for x and y are then replaced by the corresponding addresses.

example

```
I32 pitch=getvar(VPITCH);
I32 i,x,y;
U8 v_list[200];
U8 *ad_list[200];
U8 *start = (U8 *)getvar(DISP_START);
I32 *xy;

xy = (I32 *)ad_list;     /* same array */

for(i=0;i<200;i++)
 {
 x=y=i;
 xy[i][0] = x;
 xy[i][1] = y;
 v_list[i] = 255;
 }

ad_calc32(200, xy, ad_list, start, pitch);
wp_list32(200, ad_list, v_list);
```

wp_list32 **write video memory/access via address list**

synopsis

```
void wp_list32(U32 count, U8 *ad_list[], U8 v_list[])
```

description

This function writes an array of values (`v_list[]`) to the video memory. The corresponding video memory addresses are taken from the array `ad_list[]`. Both arrays should be the same size, and should contain at least `count` elements. `count` is the number of pixels which are written. It must be greater than or equal to 1.

example

```
I32 pitch=getvar(VPITCH);
I32 i,x,y;
U8 v_list[200];
U8 *ad_list[200];
U8 *start = (U8 *)getvar(DISP_START);

for(i=0;i<200;i++)
  {
  x=y=i;
  ad_list[i]     = (U8 *)((U32)start + x + y * pitch);
  v_list[i]      = i;
  }

wp_list32(200, ad_list, v_list);
```

Note:
It is more efficient to use the function `ad_calc32()` to calculate the addresses, instead of the above `for` loop.

wp_set32 **write video memory with constant/access via address list**

synopsis

```
void wp_set32(U32 count, U8 *ad_list[], I32 value)
```

description

This function writes `value` to the video memory. The corresponding video memory addresses are taken from the array `ad_list[]`.
This array should contain at least `count` elements. `count` is the number of pixels which are written. It must be greater than or equal to 1.

see also `wp_list32()`

wp_xor32 — XOR video memory with constant/access via address list

synopsis

```
void wp_xor32(U32 count, U8 *ad_list[], I32 value)
```

description

This function XORs the video memory with `value` and writes the result back to the video memory.
The corresponding video memory addresses are taken from the array `ad_list[]`.
This array should contain at least `count` elements.
`count` is the number of pixels which are written.
It must be greater than or equal to 1.

see also

`wp_list32()`, `wp_set32()`

rp_list32 — read video memory/access via address list

synopsis

```
void rp_list32(U32 count, U8 *ad_list[], U8 v_list[])
```

description

This function reads a number of pixels from the video memory and writes the corresponding values to the array `v_list[]`.
The corresponding overlay addresses are taken from the array `ad_list[]`.
Both arrays should be the same size and should contain at least `count` elements. `count` is the number of pixels which are written. It must be greater than or equal to 1.

example

```
I32 pitch=getvar(VPITCH);
I32 i,x,y;
U8 v_list[200];
U8 *ad_list[200];
U8 *start = (U8 *)getvar(DISP_START);

for(i=0;i<200;i++)
  {
  x=y=i;
  ad_list[i]    = (U8 *)((U32)start + x + y * pitch);
  }

rp_list32(200, ad_list, v_list);

for(i=1; i<200; i++) print(“value: %d\n“,v_list[i]);
```

Note:
It is more efficient to use the function `ad_calc32()` to calculate the addresses, instead of the above `for` loop.

see also

`wp_list32()`

Appendix A: Description of the example programs

adjust

The program „adjust“ is a simple way of adjusting VC cameras.

This program works in live mode with an overlay. In the middle of the image, a window and marker are displayed in the overlay. Only this portion of the image is evaluated.

Two displays are visible to the left and to the right in the image. Minimum, average and maximum brightness levels are shown in the right display. A relative focal value is shown in the left display.

When you start the program, a text message is displayed for a while and then disappears.

The library function `focus()` is used to create the display for focusing. The value is standardized for mean brightness, to make the displayed value basically independent of the shutter setting or the image’s brightness.

track - object tracking

This program implements a simple technique for tracking objects. Bright objects on dark backgrounds are viewed, such as small bright sources. (The program can be changed to the opposite by modifying a *define* statement.)

Object tracking uses a binary image. The required threshold value is automatically created as the mean value of the maximum and minimum gray scales in the image window ((max+min)/2).

First, the entire image is examined. If an object is found, then the search for the picture taken next is limited to a much smaller image window. (This can be set via *define*.)

Movement blur is always possible with object tracking. Therefore, the search is limited to a half image.

puzzle - sample program for the use of image variables

This program simulates a simple puzzle, in order to illustrate how image variables are used.

For the puzzle, the image is divided into 16 (4 x 4) image areas (image variables). Simultaneously, the sequence for the image areas is displayed in the overlay, also with image variables.

Based on the original sequence ranging from 1 to 15 (an empty field), the program copies the empty field, creating a “random” arrangements of the “stones”.

The user must make keyboard entries to restore the original sequence. When he has done so, the overlay is cleared and the game is over.

Through the use of image variables, it was possible to make the program very compact. In particular, the number of parameters which work with image variables was reduced considerably. This program also illustrates how image variables can be used to implement the overlay display.

The supplied source text is included for illustrative purposes.

flaw - flaw detection using unsharp masking

Flaw detection e.g. on a web requires contrasting a small brightness change with respect to a relatively homogeneous surface.
With the function `avg()` moving average filters of arbitrary size can be selected which run at the same speed regardless of the filter size.
This may be used for flaw detection. The original image is subtracted from the low-pass filtered. The remaining image, which contains the flaws only, is converted into run length code.
For noise-reduction a combination of erosion and dilation is used. The resulting RLC is labelled and all object features are printed out.

compare - binary object comparison (backlight recommended), fast !

Many problems in machine vision can be solved using backlight, giving images which may easily be binarised. „compare“ is a program for teach-in, manual and automatic comparison of objects. The program adjusts for a translation of the object to be checked, but not for rotation.

This is the main menu of „compare“:

```
Binary Object Compare using RLC Vs. 1.0
Copyright Vision Components       1998
press ESC to abort or any other key to continue

compare: Binary Object Compare Vs. 1.0
main menu               Copyright 1998

define object  ......................... (1)
compare ................................ (2)
set test mode  .........................(3)
set automatic mode  ....................(4)
exit  .................................. (e)
```

Menu item #1

You first must define the object to be compared. This is done using the following steps:
1. live image - make sure object is clearly visible in the middle of the image
2. threshold selection for binarising
3. object selection (you may select all objects including the background, all beeing displayed as white object on black background)

Menu item #2

This is the object comparison which consists of the following steps:
1. take a picture
2. binarize image with given threshold
3. run length encoding, object labelling, calculation of object area
4. take first object with area withing +/- 10% tolerance
5. move object so that centroid fits stored sample object
6. exclusive OR revealing difference between objects
7. count number of difference pixels and display result in percent

You may change the comparison from test mode (requiring manual interaction) to automatic mode by menu items #3 and #4

tdr - time delay recorder using JPEG compression

tdr is an example on how to use JPEG compression. Images may be taken in sequence with a time delay between pictures of approximately 1 sec (even fractions of a second are possible).
The images are compressed using the JPEG algorithms and stored in DRAM in a circular manner, i.e images that have been stored first will be the first to be overwritten after some time.
The images may be retrieved in the same order they have been stored in a sequence.

This is the main menu of the function:

```
tdr
tdr: Time Delay Recorder Vs. 1.0
Copyright Vision Components 1998
press ESC to abort or any other key to continue

tdr: Time Delay Recorder Vs. 1.0
main menu         Copyright 1998

set delay time constant  ............... (1)
set image quality  ..................... (2)
set image resolution  .................. (3)
recording  ............................. (4)
display  ............................... (5)
exit  .................................. (e)
```

The menu item #3 is currently not available.
Menu item #4 starts recording - this may be stopped pressing ESC and waiting some time (depending on the time constant you have selected)
Menu item #5 will display the stored images starting with the oldest image available in memory.

dbnce - debouncing of I/O signals

This is an example on how to debounce a (noisy) input signal

lamp - controlling a lamp with output signal / PWM brightness ctrl

This is an example on how to control a lamp (or any other device) with the PLC outputs of the camera. The lamp is switched on and off some times with some delay inbetween. Then the brightness of the lamp may be controlled by typing „+" (brighter) or „-" (darker). The brightness control is performed using pulse-width modulation (PWM)

corr - normalized grey scale correlation, sample size = 16x16 pixels

corr 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 (64x64 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 B: List of library functions

Programs for processing gray images

Name	Type	Description
`void set (image *a, int x)`	C	Write constant to image variable
`void copy(image *a, image *b)`	C	Copy image variable
`void histo(image *a, U32 hist[256])`	C	Histogram
`void img2(image *a, image *b, image *c, void (*func)(),int q)`	C	Link 2 image variables
`add2(image *a, image *b, image *c, int sh)`	M	Add two image variables
`sub2(image *a, image *b, image *c)`	M	Subtract two image variables (abs)
`max2(image *a, image *b, image *c)`	M	Maximum of two image variables
`min2(image *a, image *b, image *c)`	M	Minimum of two image variables
`and2(image *a, image *b, image *c)`	M	AND two image variables
`or2 (image *a, image *b, image *c)`	M	OR two image variables
`subx2(image *a, image *b, image *c, int offset)`	M	Subtract two image variables with offset and clipping
`sub2y(image *a, image *b, image *c)`	M	Subtract two image variables and binarize
`void imgf(image *a, image *b, void *func())`	C	any 3x3 operator
`sobel(image *a, image *b)`	M	Sobel operator
`laplace(image *a, image *b)`	M	Laplace operator
`mx(image *a, image *b)`	M	Maximum operator
`mn(image *a, image *b)`	M	Minimum operator
`void ff3(image *a, image *b, static int pm c[3][3], int sh)`	C	3 x 3 filter for image variable
`void ff5(image *a, image *b, static int pm c[5][5], int sh)`	C	5 x 5 filter for image variable
`void ff5y(image *a, image *b, int pm *h, int pm *v, int sh)`	C	5 x 5 filter for image variable horizontal / vertical separation
`void robert(image *src, image *dest)`	C	robert's cross operator
`void projh(image *a, U32 result[dy])`	C	Horizontal projection
`void projv(image *a, U32 result[dx])`	C	Vertical projection
`void look(image *a, image *b, U32 table[256])`	C	Look-up table
`U32 focus(image *a, I32 sh)`	C	focal value of an image variable
`U32 mean(image *a)`	C	mean value of an image variable
`U32 variance(image *a)`	C	variance of an image variable
`void pyramidx(image *a, image *b, void (*func)())`	C	general pyramid function
`void pyramid(image *a, image *b)`	M	pyramid filter for image variable
`void pyr_max(image *a, image *b)`	M	pyramid maximum for image variable
`void pyr_min(image *a, image *b)`	M	pyramid minimum for image variable
`void subsample(image *a, image *b, I32 rh, I32 rv)`	C	subsample image (image variable)
`U32 arx(image *a, I32 thr)`	C	number of pixels > threshold
`U32 arx2(image *a, I32 th1, I32 th2)`	C	number of pixels `th1 < x < th2`

Name	Type	Description
`void bin0(image *src, image *dest, I32 thr, I32 bl, I32 wt, void *(*fc)())`	C	fast binarization of an image variable
`binarize(image s, image d, I32 t, I32 b, w)`	M	binarizing
`PaintWhite(image s, image d, I32 t, I32 w)`	M	binarizing / dark pixels not changed
`PaintBlack(image s, image d, I32 t, I32 b)`	M	binarizing / bright pixels not changed
`I32 avg(image *a, image *b, I32 kx, int ky, void (*func)(), I32 v)`	C	moving average or unsharp masking of an image variable output centered
`I32 avg2(image *a, image *b, I32 kx, int ky, void (*func)(), I32 v)`	C	moving average or unsharp masking of an image variable - not centered
`avgm(a, b, kx, ky)`	M	moving average
`maskx(a, b, kx, ky, offset)`	M	subtract original + offset
`masky(a, b, kx, ky)`	M	unsharp masking + binarize
`void zoom_up(image *a, image *b, I32 factor)`	C	enlargement of an image variable

Programs for gray scale correlation

Name	Type	Description
`I32 vc_corr0(image *a, image *b, I32 mcn, I32 mcr, I32 *x0, I32 *y0)`	C	small kernel correlation routine extended search area

Programs for JPEG compression / decompression

Name	Type	Description
`I32 fwrite_jpeg(image *a, char *path, I32 quality, U32 maxlng)`	C	write image variable to JPEG image file / flash EPROM
`U8 *cjpeg(image *a,I32 quality, U8 *addr, U32 maxlng, I32 (*func)())`	C	encode image variable to JPEG image file
`cjpeg_d(img, qual, addr, maxlng)`	M	write JPEG data to DRAM
`cjpeg_f(img, qual, addr, maxlng)`	M	write JPEG data to Flash Eprom
`cjpeg_a(img, qual)`	M	send JPEG ASCII data to RS232
`cjpeg_b(img, qual)`	M	send JPEG binary data to RS232
`int fread_jpeg(image *a, char *path)`	C	read JPEG image file / flash EPROM
`U8 *djpeg(image *a,U8 *addr, I32 (*func)())`	C	decode JPEG image file into image variable
`djpeg_d(img, addr)`	M	read JPEG data from DRAM
`djpeg_f(img, addr)`	M	read JPEG data from flash eprom
`djpeg_a(img)`	M	read JPEG ASCII data from RS232
`djpeg_b(img)`	M	read JPEG binary data from RS232

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

Name	Type	Description
`U16 *rlcmalloc (U32 size)`	M	allocate RLC memory
`void rlcfree (U16 *rlc)`	M	deallocate RLC memory
`U16 *rlcmk(image *a, I32 thr, U16 *rlc, I32 size)`	C	Create RLC
`U16 *parse_rlc(U16 *rlc)`	C	parse RLC and output next address
`I32 rlcout(image *a, U16 *rlc, U8 dark, U8 bright)`	C	Output RLC
`U16 *rlc_inv(U16 *rlc)`	C	in-place inversion of RLC
`U16 *rlc2(U16 *rlca, U16 *rlcb, U16 *dest, U16 * (*func)() )`	C	Link any 2 RLCs
`rlcand(U16 *a, U16 *b, U16 *dest)`	M	AND RLCs
`rlcor(U16 *a, U16 *b, U16 *dest)`	M	OR RLCs
`rlcxor(U16 *a, U16 *b, U16 *dest)`	M	XOR RLCs
`U16 *erxdi(U16 *src, U16 *dest, U16 *(*fc1)(), U16 *(*fc2)())`	C	erosion / dilation of RLC / square type
`U16 *erxdi2(U16 *src, U16 *dest, U16 *(*fc1)(), U16 *(*fc2)())`	C	erosion / dilation of RLC / diag. type
`erode(U16 *src, U16 *dst)`	M	RLC erosion / square type
`dilate(U16 *src, U16 *dst)`	M	RLC dilation / square type
`erode2(U16 *src, U16 *dst)`	M	RLC erosion / diamond type
`dilate2(U16 *src, U16 *dst)`	M	RLC dilation / diamond type
`U16 *testrlc(U16 *rlc, I32 dx, I32 dy, I32 size)`	C	create RLC test image - chess-board
`U16 *rlc_mf(U16 *src, U16 *dest, I32 col, I32 lng)`	C	horizontal „median filter“ for RLC
`I32 fwrite_rlc(char *path, U16 *rlc)`	C	write RLC to flash EPROM
`U16 *fread_rlc(char *path, U16 *rlc)`	C	read RLC from flash EPROM
`U16 *rlc_move(U16 *src, U16 *dest, I32 mx, I32 my)`	C	move RLC
`U32 rlc_area(U16 *rlc, I32 color)`	C	Calculate area in RLC
`void rlc_feature(feature *f, U16 *rlc, I32 color)`	C	Determine features, unlabelled RLC
`U16 *sgmt(U16 *rlc, U16 *slc)`	C	Label RLC

Programs for processing binary images in labelled run length code

Name	Type	Description
`I32 dispobj(image *a, U16 *rlc)`	C	Output labelled RLC
`U16 *rlc_cut(U16 *src, U16 *dest, I32 objnum)`	C	Cut objects from RLC
`I32 rl_area2(U16 *rlc, U32 *area, U32 n)`	C	Object areas in labelled RLC
`I32 rl_ftr2(U16 *rlc, ftr *f, U32 n)`	C	Object features in labelled RLC
`I32 chkrlc(U16 *rlc)`	C	Check RLC

Programs for processing contour code(CC)

Name	Type	Description
`I32 contour8(image *a, I32 x0, I32 y0, I32 dir, I32 thr, U32 lng, U32 **dst)`	C	Contour following / 8-connected
`void cdisp(image *a, U32 *src, I32 col, void (*func)())`	C	display contour
`cdisp_d(a, src, col)`	M	DRAM write
`cdisp_x(a, src, col)`	M	DRAM XOR
`cdisp_o(a, src, col)`	M	Overlay write
`cdisp_z(a, src, col)`	M	Overlay XOR
`I32 ccxy(I32 *src, I32 *xy, I32 *tbl, U32 maxcount)`	C	convert **CC** into xy-array

Graphics functions

Name	Type	Description
`void chprint(char *s, image *a, I32 cx, I32 cy)`	C	Output a string to an image variable
`int linexy(I32 dx, I32 dy, I32 *xy)`	C	Basic line creation routine
`void line(image *a, I32 x1, I32 y1, I32 x2, I32 y2, I32 col, void (*func)() )`	C	Draw line
`lined(image *a, I32 x1, I32 y1, I32 x2, I32 y2, col)`	M	Draw line in video memory
`linex(image *a, I32 x1, I32 y1, I32 x2, I32 y2, col)`	M	Draw line in video memory/XOR
`lineo(image *a, I32 x1, I32 y1, I32 x2, I32 y2)`	M	Draw line in overlay
`linez(image *a, I32 x1, I32 y1, I32 x2, I32 y2)`	M	Draw line in overlay/XOR
`void frame(image *a, I32 col, void (*func)())`	C	Draw frame
`framed(image *a, I32 col)`	M	Draw frame in video memory
`framex(image *a, I32 col)`	M	Draw frame in video memory/XOR
`frameo(image *a)`	M	Draw frame in overlay
`framez(image *a)`	M	Draw frame in overlay/XOR
`void dframe(image *a, I32 col, void (*func)())`	C	Draw wide frame
`I32 EllipseXY(I32 a,I32 b,I32 *xyc)`	C	basic ellipse creation program
`void ellipse(image *a, I32 col, void (*func)())`	C	draw ellipse
`ellipsed(a, c)`	M	draw ellipse / image memory
`ellipsex(a, c)`	M	draw ellipse / image mem. XOR
`ellipseo(a)`	M	draw ellipse / overlay
`ellipsez(a)`	M	draw ellipse / overlay XOR

Name	Type	Description
`dframed(image *a, I32 col)`	M	Draw wide frame in video memory
`dframex(image *a, I32 col)`	M	Draw wide frame in vid. memory/XOR
`dframeo(image *a)`	M	Draw wide frame in overlay
`dframez(image *a)`	M	Draw wide frame in overlay/XOR
`void marker(image *a, I32 col, void (*func)())`	C	Draw marker
`markerd(image *a, I32 col)`	M	Draw marker in video memory
`markerx(image *a, I32 col)`	M	Draw marker in video memory/XOR
`markero(image *a, I32 col)`	M	Draw marker in overlay
`markerz(image *a, I32 col)`	M	Marker, overlay/XOR
`void dmarker(image *a, I32 col, void (*func)())`	C	Draw wide marker
`dmarkerd(image *a, I32 col)`	M	Draw wide marker in video memory
`dmarkerx(image *a, I32 col)`	M	Draw wide marker in video mem./XOR
`dmarkero(image *a, I32 col)`	M	Draw wide marker in overlay
`dmarkerz(image *a, I32 col)`	M	Wide marker, overlay/XOR

Pixellist functions

Name	Type	Description
`void ad_calc32(U32 count, I32 *xy, U8 *ad_list[], U8 *start, I32 pitch)`	C	Calculate an address list from a coordinate list
`void rp_list32(U32 count, U8 *ad_list[], U8 *v_list)`	C	Read pixel list
`void wp_list32(U32 count, U8 *ad_list[], U8 *v_list)`	C	Write pixel list
`void wp_set32(U32 count, U8 *ad_list[], I32 value)`	C	Set pixels in pixel list to constant
`void wp_xor32(U32 count, U8 *ad_list[], I32 value)`	C	XOR pixels in pixel list with constant

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

INDEX