TIM for WINDOWS Help

TIM for Windows is a program for processing images. It can aquire and manipulate images and it can extract data from images.

Introduction

This section gives you a brief introduction into TIM and its usage.

How to ...

This section gives you "how to" information on several subjects.

Commands

This section helps you to find a command for a task.

Command Files

This section introduces you to TIM's command language and compiler.

Menus

This section describes the menus

Changes

A list of changes introduced since the last release

Introduction

TIM allows you to process images. Generally, the main goal will be extraction of information, which is hidden in the image. TIM has many operations and tools to achieve this.

Images can be acquired using a frame grabber and a video camera. TIM can also handle TIFF image files, as produced by scanners.

In TIM, images are data structures, and as such accessed by most functions. It is important that a user is aware of the properties of TIM's images. See: images and image use.

To further become familiar with TIMWIN, see procedures

How to ...

Auto update windows
Compile command files
Control directories
Control image cursor using the mouse
Control sub-image size & position using the mouse
Control windows
Convert grey value images into binary images
Cut & Paste
Draw in an Image
Edit command files
Establish a DDE link
Find a command
Install images
Make bitplanes visible
Perform real time image processing
Print images
Preview operations
Select images
Set debugger breakpoints
Set up the frame grabber
Use 16 bit images
Use Alias
Use bitplanes
Use colours
Use default parameters
Use FFT

Commands

TIM has almost 200 image processing and control commands. If you look for a certain command to perform a task, you have the following possibilities. You can:

Select a command based on the phase of the image processing process

- acquiring an image
- pre-processing an image
- image segmentation
- data extraction
- data processing

Select a command based on the division in related commands (families) like:

- Pixel operations
- Bitplane operations
- Windows (or neighbourhood) operations
- Control operations, etc.

See further: Entering a command

Command Files

A command file is a collection of statements. It can be used to write image processing programs and perform calculations on the derived data.

The language to write the statements in (TIMWIN's Command language) is a very powerful. It has many characteristics of a modern, structured language.

To achieve speed, programs are compiled before they can be executed. TIM has a built in compiler, that automatically compiles a source if it notices a change in the source file.

The language has many useful features, like:

- control structures: while, for - next, switch - case, repeat - until
- data types: strings, integer, floating point, arrays, files
- easy file IO: only reading, writing and positioning for both binary and text files.
- string manipulation and formatting

A debugger helps with making correct programs. Features include: a source window, single stepping, break points, watching variables.

See also:

- Command Files Reference
- Command Language Description

Changes

Status September 29, 1993, version 1.12

New

1.07 Faster windows image display; selection between fixed and scalable windows
1.08 Cursor in Windows images (see manual chapter 3)
Drawing in images redesigned (see manual chapter 3)
Windows LUT control simplified (see manual chapter 3)
Network awareness improved (see manual Appendix K)
DDE (Excel link) has been enhanced
1.09 XUSER added
1.10 `switch - endsw`: number lists introduced
1.12 updating windows images speeded up

Warning

Don't run command files with a time stamp prior to September, 29 1993 with this version without recompiling. Internal tables have been changed, so that wrong results are likely to occur.

Menus

File Edit Contr ImageProc Applic Graph FrGrabber CommFile... View

The top bar in in the TIM main window contains the menu entries. These entries allow you to select a specific command or function in a top - down fashion.

- File — File related functions and DDE
- Edit — Clipboard- and various Window Edit functions
- Control — Setting up your system
- Image Processing — Image processing functions, grouped familywise
- Applications — Image processing functions, grouped applicationwise
- Graphics — Graphic functions
- Frame Grabber — Image aquisition using the frame grabber
- Command File — Editing and compiling command file programs
- View — Organizing your screen

Auto update windows

Several windows which show image dependent information, can be told to update their content with each image operation. This regards:

- the graph window
- the ibuf window
- the image edit window
- the statistics window

To auto update a window, do one of the following:

- In the window, select the option menu and click the update mark. If it is checked, auto-updating takes place. If not, the window will not change automatically.
- Enter the **set update ...** command with the proper parameters. See also: set

---

Notice, that this does not affect the commands that deal specifically with a window. For example, the **hist** command always writes a graph if the graph window is on the screen.

Compile command files

TIM command files have to be compiled, before they can be executed. This is in contrast to elder TIM versions, where the text file was immediately executed.

Compiling the file results in a much more efficient execution.

Compiling can take place in the following ways:

- manually
- automatically, with prompts
- automatically

To compile manually, do the following:
1. Select the CommandFile dialog box
2. Select a command file to compile
3. Choose the Compile button

To compile automatically do the following
1. From the Control Menu, choose Installation
2. Choose the Compiler Options button
3. In the Update ... box check Atomatic or Prompt Yes/No

If you check Automatic, compiling takes place whenever a source file is newer than a compiled file. If you check Prompt Yes/No, a message box appears which allows you to prevent compilation.

Control directories

TIM uses several directories to find or store files. You can specify the following directories:

- image file directory
- source command file directory
- compiled command file directory

To set up directories, do the following:
1. From the Control Menu, choose Installation
2. Choose the Files button
3. Fill in the appropriate path

You can add more search paths manually by editing the file **timwin.ini** in the main Windows directory (usually c:\windows). These specifications should have the form:

```
CmdPath=c:\timwin\cmc\
CmcPath=c:\timwin\cmc\
ImagePath=c:\timwin\im\
CmcPath1=d:\tim\cmc\
ImagePath1=d:\tim\im\
CmcPath2=f:\myproj\cmc\
ImagePath2=f:\myproj\im\
```

The first 3 entries in this example are written in **timwin.ini** as a result of filling in the dialog box, as described above. The entries with a number must be added manually. There is no limitation in the number of entries.

When TIM looks for a file, it looks up the path specifications in timwin.ini in the numerical order. For images, this is: first `ImagePath`, then `ImagePath1`, `ImagePath2`, etc.

When TIM writes a file, it only uses the directory specified in the Files dialog box.
If you specify a complete path name, this path will be used.

Control Sub-Image Size & Position

To adjust sub-image size and position using the mouse, do the following:

Select the image in which the sub-image must be defined

1. Switch to image mode:
   - In the status bar, click the Cursor button.
   - Or, position the (windows) cursor over an image and click the right button
2. Click the left mouse button until the box cursor appears
3. During the following procedure, keep the Ctrl key pressed.
4. Press the left mouse button and move the cursor to the upper left position of the intended sub-image. Notice that the box shrinks to 3x3.
5. While keeping the left button pressed, move the cursor downwards and to the right. The box will grow, while the upper left position remains in place.
6. When the box has the correct size, release the mouse button.
7. If necessary, repeat the procedure from 3 to 6.
8. If ready, release the Ctrl key

Control windows

TIM exists of many windows. Most of them can be shown or removed, as you like. You can choose to remove a window entirely or reduce it to an icon.

The windows react according to the MS-Windows conventions. See the MS-Windows manual for details.

Convert grey value images into binary images

If you want to perform a binary image operation like bitplane or CLP operations, you'll have to provide for a binary image first.

The following operations convert a grey-value image into an image consisting of two grey values (0 and 255). This result, an 8-bit binary image, can be considered to contain equal binary images in each of the 8 bitplanes of the image.

You can select any of the 8 bitplanes to continue working in binary mode.
To make space for other binary images or intermediate results you may want to erase the obsolete binary images. See the commands era and keep.

To view any binary image in colour see How to ...

To operate on binary images see Bitplane and CLP operations

Cut & Paste

You can place the content of any window in TIM in MS-Windows' Clipboard by selecting the window and press ALT-PrintScreen. From the Clipboard you can insert the data in any other MS-Windows application.

See the MS-Windows' Clipboard documentation for details.

Draw in an Image

You can draw in an image using the mouse. To do so, you must first select an image to draw in. Then set up the display in order to make graphics visible. See How To ...

Next tell the system that the mouse no longer is used to control the program:

1. Switch to image mode:
   - In the status bar, click the Cursor button.
   - Or, position the (windows) cursor over an image and click the right button
2. To make the image cursor visible, click the left mouse button repeatedly. All cursor shapes except the box cursor will do. In this mode, the box cursor allows you to control sub-image size and position
3. To draw in the image, you must keep one of the following keys pressed down:
   - the Shift key to draw in drawing mode
   - the Ctrl key to draw in graphics mode
4. Click the left mouse button to start drawing. You'll see a rubber band line in the graphics bitplane that follows the cursor movements. Once the left mouse button is clicked, the line freezes.
5. To draw freehand, keep the left mouse button pressed. To draw vectors, click it.
6. To close a polygon, click the right mouse button
7. To stop drawing, release the Ctrl or Shift key.
8. To switch back to normal mode, click the right mouse button

Edit command files

TIMWIN comes with a simple editor EditCF, which offers various advantages over the standard Windows editor NotePad. These include:

- Visible line numbers
- Two files can be open at once
- Various fonts
- Easier and more powerful search and replace
- Keyword awareness: if you select a keyword and press **F1** , help on that keyword comes up.

You can specify any editor in the Install dialog box. Default is EditCF.

To edit a command file you can:

- From the Command File dialog box select the desired command file and click Edit. The editor will come up with the command file (if available) opened.
- When you finished editing, select Save in the File menu.
- If necessary, you can reduce EditCF to an icon. It will come up automatically with the new file opened when you select another file.

Establish a DDE link

TIM can exchange data with another application using Window's Dynamic Data Exchange protocol.

- When TIM sends data to another application, the data in Ibuf is used. Sending takes place automatically whenever a TIM command changes Ibuf.
- When TIM receives data from another application, it is considered a regular TIM command.

To establish a DDE link do the following

1. Start the other application

2. In TIM's Files menu click IO operations

3. If you want to
   - send data to the other application, click Ibuf Link
   - receive commands from the other application, click Command Link

4. Fill in the dialog box

You can also use the excelo, excels and excelc commands.

Currently only Microsoft Excel can be used for DDE communication

Find a command

To find a specific image processing function, you can:

- Approach the functions by type (or family)
- Approach the functions by application phase
- Approach the functions alphabetically - click the Search button in this window

Install images

Setting up images takes place outside TIM, by specifying the desired images in a file IMAGES.TIM. This file can be edited from within TIM, but the results will show up only after the next time TIM is invoked.

To modify IMAGES.TIM do the following:

1. From the Control menu select Images. The editor, specified in the Install dialog box, will come up with IMAGES.TIM opened.
2. Edit the file and close the editor.

Make bitplanes visible

To make bitplanes visible, you'll have to provide for an appropriate translation of the pixel values to screen intensities.

To display grey value images usually a linear translation is used: pixel value 0 is shown as black and pixel value 255 is shown as white.

To display a bitplane a special translation is used, such that an object in the bitplane causes the screen to show a bright colour. Notice, that this does not prevent the display of grey values: both translations can be mixed.

The translation takes place using look up tables. TIMWIN uses the convention that the default content of the frame grabber look up tables no. 3 and 4 is devoted to the display of bitplanes. Applying these tables is different for frame grabber images and Windows images:

To show bitplanes:
To show bitplane 1 in red, select look up table 3.
Or, to show bitplane 1 in red, bitplane 2 in green and bitplane 3 in blue, select look up table 4.

To show bitplanes in a frame grabber image:

- Initialize the system by running one of the command files `*ini` or `*init`

To select a look up table, you can:

- use the **lut** command: `lut 2 3` means: select *output* LUT no. *3*
- use the Output Lut function in the status bar

To show bitplanes in a windows image:

- use the winlut command file; `*winlut 4` means: load LUT pattern no. *4*
- use the Windows Lut function in the status bar

See also: ***TIMWIN*** command lut

Perform real time image processing

To perform real time image processing, you must have installed a frame grabber that has real time capabilities. Example: the Imaging Technology VFG frame grabber.

Real time image processing is processing the pixels while they flow into the system. Thus an entire image is processed in 20ms.

To perform this, the frame grabber is set up in a special mode, that is not fully compatible with normal mode. You have to take special mesures afterward, to put the system back to normal mode.

- the result of a real time operation is 6-bits wide, so scaling is necessary. This can be done using the shl command
- TIM uses the frame grabber's highest output LUT for display. Most of the time the previously selected LUT is switched back on after the operation, but not always.
- TIM uses the entire input look up table bank for conversion. Any previous content is lost and wil not be restored automatically.

See also: real time operations

Print images

Currently images cannot be printed directly from TIMWIN.
You can use the Windows Clipboard to copy an image to another application (e.g. a word processor or a paint program), and print it from that application.

To copy a TIM-Windows image to the Windows clipboard:

1. Select the image by clicking with the mouse on the caption bar
2. Press ALT+PrintScreen

See also the Clipboard documentation in the Windows manual.

Preview operations

Previewing is looking at the results of an operation without actually processing the image. It is possible on the following conditions:

- A frame grabber must be available
- The operation must be a table operation
- You must open the dialog box for the operation.

If Previewing is possible, the dialog box has a Preview button. Clicking this button will load the table into the frame grabber's output LUT, so that the result is immediately visible. The highest output LUT no. is used, which must be kept free for this purpose.

To stop this mode, close the dialog box in one of the following ways:

- Click Cancel to return without performing the operation
- Press OK to execute the command and return

Note: TIMWIN uses the highest frame grabber LUT as a scratch LUT for previewing.

Select images

In TIM you must select an image to be the destination of operations. Such an image is called: the default destination image. Selecting an image is done as follows:

- Click an image button on the status bar.
- Or enter the command dest x**,** where **x** is the name of the image to be selected

The status bar will reflect the new selection by showing a red border around the image's button.

The dis command has a similar function if the specified image is a frame grabber image or a windows image. If the image is a memory image (without display - either frame grabber or windows - connected to it), the **dis** command means: copy the image to the currently selected default destination.

Set Breakpoints

To set a breakpoint, do the following (assuming the debug window is opened and the source program is shown):

1. Position the cursor on the source line where you want to set the breakpoint
2. Double click this line or hit the Enter key

Or

1. In the Debug menu click Breakpoints
2. In the Breakpoints dialog box fill in the line number(s) of the source line(s) where you want a break to occur. Press Set after each number.
3. When ready, click Done

Set up frame grabber

Setting up the frame grabber for TIMWIN involves the following steps:

1. From the Control menu select Installation
2. In the Installation dialog box click Frame grabber
3. In the Frame grabber listbox select the frame grabber you are using
4. In the **Segment** edit control enter the frame grabber's memory base address
5. In the **I/O-address** edit control enter the frame grabber's IO-base address

Consult your frame grabber's manual for hardware details
When selecting addresses beware of conflicts with other devices in your system.

Setting up TIMWIN for a Frame Grabber

If there is a frame grabber in your system and if you specified frame grabber images in the image specification file images.tim, TIMWIN needs to know how to access the frame grabber. This is done by specifying two values:

- The memory base address
- The IO-base address

When ***TIMWIN*** starts up and discovers that these values are yet unspecified (e.g. when started for the first time), it invites you to fill in these values. ***TIMWIN*** will not run, unless proper values have been specified. After you specified correct values, ***TIMWIN*** will continue starting up. If it doesn't 'see' the frame grabber, additional diagnosis messages will appear.

Below some suggested values are given (values are shown and specified in hexidecimal). Additional values are shown in parentheses. When you specify a value, usually a range of values is reserved, depending on the frame grabber.

	Specification	Reserved range
Memory base address		
All frame grabbers	D000	D000 - DFFF
I/O-base address		
ITI-VFG	300 (320, ...)	300 - 320, (320 - 340, ...)
PCVisionPlus	300 (310, ...)	300 - 310, (310 - 320, ...)
CORTEX-I	230	230 - 234

As can be seen, the only possible memory base address for a frame grabber is usually D000. This is because this is the largest available area. Frame grabbers need a 64K memory base.
For the IO-base address more values are available.

Beware of using a value for more than one device. For example, if there is a network adapter in your system, it is likely to occupy memory and IO-base as well. Choose a separate value (range) for each device.

Notice that the reserved memory space must be specified as well in the following files:

- **system.ini**
- **config.sys**

For more information, see:

- The ***TIMWIN*** manual (chapter 4)
- The frame grabber manual

The frame grabber's memory base address is the starting point of the memory area that the frame grabber occupies. This channel is used to read and write pixels.

In most cases this base address is D000 (hexadecimal). Be sure to avoid different devices to share a memory area.

The frame grabber's IO-base adress is the starting point of the IO-addresses that the frame grabber occupies. These addresses are used to control the frame grabber functions.

In many cases 300 (hexadecimal) is a good value. The CORTEX-I is the easiest installed at 230 (hex). Be sure to avoid different devices sharing a single IO address range.

Use 16 bit images

Most image processing operations in TIMWIN use 8-bits pixels. Some operations need a larger pixel size, and some frame grabbers come with 12 bits pixels. Therefore TIMWIN allows you to define 8, 12, 16 and 64-bits images (see: How to ...)

You can use 12- and 16 bits images for 8-bits operations. In this case the lowest byte of each pixel word is used.

The following operations support 16 bits pixels:

era16	erase a 12- or 16 bits image
cp16	copy a 12- or 16 bits image into an 8-bits image
sum	repeatedly add 8-bits images in a 16-bits image (integrating)

Use Alias

Aliases are defined in a file **alias.tim**. TIMWIN comes with a file which contains some general definitions. You may add your favourite terms to this file. Instructions to do so are in the file header.

You can use aliases in interactive (command line) mode.
The alias definitions can also be used when compiling a command file. To put this option on:

1. In the main menu click Commfile
2. In the Commfile dialog box select the Aliases box.

Warning: be very cautious when defining terms. Possible sources of conflicts are: TIMWIN's keywords, variable names, etc.

Use bitplanes

Some TIM operations act on bitplanes. These are: bitplane operations and CLP operations. This is what you should do when preparing for operations on bitplanes:

1. Create a binary image, for example by executing the thre command. This operation fills all bitplanes with the same binary image.
2. Choose the bitplane number(s) to operate on. This can be 1, 2 and/or 3.
3. Display the content of this bitplane(s) by selecting a Look-Up table function that shows the bitplane in a contrasting colour. See also: How to ...

Use colours

Colours can be very handy with image processing tasks. Here a few examples follow:

- Working with bitplanes
- Estimating contrast
- Expressing special features (e.g. an overlay showing binary properties in grey value images)

Use Pseudo Colours

Pseudo colours appear if a regular black & white image is displayed using look up tables, that are filled with different tables. Then each pixel value is represented with a colour out of a set of 16.000.000.

TIM has some standard pseudo colour tables, that can be very useful in some situations

Use default parameters

TIM commands generally need some specification regarding the details of their operation. This is done using parameters.

In many cases it is not necessary to explicitly specify all parameters, because TIM uses sensible defaults where possible. The following rules determine the default values:

- If no source image is specified, the active image is used as a source
- If a parameter specifies a bitplane, bitplane 1 is used
- If a parameter specifies a drawing value , the drawing value is used.
- If a parameter specifies a graphics value , the graphics value is used.
- If a parameter specifies a position in the image, the image's cursor position is used.

For more information on default values refer to the individual command descriptions.

Use FFT

Fast Fourier operations an be used to create filters, that cannot be made in any other way. To be able use this type of filter, the image must be transformed into the frequency domain. In this representation the individual frequencies that constitute the image can be accessed. This allows frequencies to be selectively removed or enhanced.

See further FFT-operations

Use history

Use the image cursor

Use Ibuf

Use sub-images

Use the status bar

Watch Variables

Families of TIM commands

TIM image operations are ordered into the following categories or *families*

Bitplane Operations
Cellular Logic Operations
Control Operations
FFT Operations
Geometric Operations
Graphic Operations
Input/Output Operations
Miscellaneous Operations
Parameter Operations
Pattern Recognition Operations
Pixel Operations
Real Time Operations (VFG Only)
Transport Operations
Window Operations

Application Sequence

A typical image processing sequence contains the following activities:

Acquiring an image
Preprocessing
Segmentation
Data extraction

The following help screens give some suggestions for image processing in the various stages of an image processing task. Realize, that many advanced operations consist of a sequence of basic operations (e.g. in the form of a command file).

Notice, that the separation between subjects is usually not as rigid as this summary suggests.

IMAGE ACQUISITION OPERATIONS

The first task is to get a proper image to operate on. You can get an image in either of two ways:

1. grab an image using a frame grabber
2. copy an image from disk

copy	copy an image from disk
dig	get an image using a frame grabber
real time	perform real time image processing (processing while grabbing)

PREPROCESSING OPERATIONS

The purpose of the preprocessing phase is to enhance the image, and to prepare it for the following tasks. This includes: noise rejection, contrast operations. geometric corrections, etc.

Contrast operations

cstr	stretch contrast
ehis	re-distribute grey values

Linear filtering (convolution) and noise rejection

dgaus	gaussian noise filtering
gaus	fast gaussian noise filtering
filt	universal filter
qshrp	fast sharpening
shrp	sharpening
unif	uniform filter

Non linear filtering and noise rejection

kuwa	suppresses noise, keeps edges sharp
perc	removes extreme values

Geometric corrections

ct	universal geometric correction
dim	horizontal stretch

SEGMENTATION OPERATIONS

In the segmentation phase objects are localized and separated.

dt	distance transform
threshold	divide image into objects and background
label	label objects with grey values

DATA EXTRACTION OPERATIONS

Finally the segmented image is analyzed: the properties of the objects are recorded. You can measure shape, grey values, statistics, etc.

Shape and size measurements

fcont	measure the length of the perimeter
maxl	find the maximum diameter
mark	localize an individual object and measure its size

Aquire pixels from the image

rdln	get grey values under a line
rdvec	get grey values under a vector
rdpat	get grey values under a random line

Global grey value measurements and statistics

hist	get the grey value histogram
stat	calculate statistic values from the histogram

BITPLANE OPERATIONS

These operation perform their action on bitplanes. The result is written into the specified bitplane of *the same image*.

band	- ANDs two bitplanes
bbord	- sets/resets a border
bcop	- copies one bitplane to another
bdump	- fills an entire bitplane
binv	- inverts a bitplane
bor	- ORs two bitplanes
bsw	- exchanges two bitplanes
bxor	- XORs two bitplanes

Overview of all families . . .

CELLULAR LOGIC OPERATIONS

CLP operations perform morphologic operations on bitplanes.

lcon - keeps the contour pixels
ldi - dilates (grows a layer of pixels)
lenp - keeps end pixels (pixels with 1 neighbour)
ler - erodes (removes a layer of pixels)
life - game of life
link - keeps the link pixels (pixels with 2 neighbours)
lmaj - majory vote
lpr - propagation (controlled dilation)
lps - removes pepper and salt (single 1 or 0 pixels)
lsp - keeps only single pixels
lsk - produces the skeleton
lskz - produces the skeleton, removes endpixels
lska - produces the anchor-skeleton
lskz - produces the anchor-skeleton, removes endpixels)
lver - keeps vertex pixels (pixels with 3 or more neighbours)

Overview of all families . . .

CONTROL OPERATIONS

Control operations control many aspects of images, the TIM program, frame grabber or files

curlock	- controls cursor coupling between images
curs	- controls the image cursor (position, on/off, shape)
cursx	- controls the horizontal cursor position
cursy	- controls the vertical cursor position
dest	- changes default destination
dig	- controls frame grabbing (digitizing an image)
dig3	- controls frame grabbing in colour systems
dis	- shows the current image, changes default destination
exist	- allows to check existence of file
frmt	- controls the sub image format
frmtx	- controls the horizontal sub image format
frmty	- controls the vertical sub image format
lcset	- loads character set
lut	- fills and controls the look up tables
ovl	- controls use of upper/lower byte of 16-bit images
pan	- controls panning
set	- controls various system parameters
timer	- sets/reads timer
ver	- supplies version information
zoom	- controls zooming in and out

Overview of all families . . .

FAST FOURIER TRANSFORM OPERATIONS

FFT operations operate on images in the Fourier (frequency) domain. There are also operations for converting standard images into the Fourier domain and v.v.

fftb	transforms from Fourier domain to complex floating point
fftd	converts the floating point image to pixel format
fftm	multiplies the floating point image with a mask image
ffto	transforms from floating point to Fourier domain
fftr	converts pixel image to floating point format

A typical FFT sequence is:

1. fftr convert a grey value image into complex floating point
2. ffto transform a complex floating point image from the space domain into the frequency domain
3. fftd optionally display the frequency domain image
4. fftm optionally multiply the frequency domain image with a grey value image to enhance or suppress frequencies
5. fftb transform the frequency domain image into space domain
6. fftd convert the complex floating point image into integer pixel values

Overview of all families . . .

GEOMETRIC OPERATIONS

Geometric operations act on pixel *positions* instead of *grey values*

blow	- enlarges an image
ct	- coordinate transformation
dim	- modifies the horizontal dimension of an image
movx	- moves an image horizontally
movy	- moves an image vertically
redu	- reduces the size of an image
rotl	- rotates the image left 90ø
rotr	- rotates the image right 90

Overview of all families . . .

GRAPHIC OPERATIONS

Graphic operations perform actions on images based on line figures.

- bbord - draws a border in a bitplane around a (sub) image
- bord - draws a border around a (sub) image
- cirk - draws a circle
- drln - draws a line between two sets of coordinates.
- drpat - draws a line along a Freeman path
- drvec - draws a vector
- dvec - draws a vector in Freeman directions
- graf - plots data in Ibuf (draws a line)
- grav - plots data in Ibuf (draws vertical bars)
- incln - increments pixels 'under' line
- incpat - increments pixels 'under' Freeman path
- incvec - increments pixels 'under' vector
- orln - draws a line
- orpat - draws a line along a Freeman path
- orvec - draws a vector
- qplot - produces a surface plot (3D)
- sbln - scans along line until bitplane set
- sgln - scans along line until greyvalue Ø 0
- sbvec - scans along vector until bitplane set
- sgvec - scans along vector until greyvalue Ø 0
- text - writes text in an image
- textv - writes text vertically
- wcur - writes the actual cursor pattern in the actual display
- xorln - draws a line
- xorpat - draws a line along a Freeman path
- xorvec - draws a vector

Overview of all families . . .

INPUT/OUTPUT OPERATIONS

I/O operations perform input and output of images or other data, or control DDE or peripheral devices

dig	- controls frame grabbing (digitizing an image)
excelc	- closes a DDE-link with MS-Excel
excelo	- opens a DDE-link with MS-Excel
excels	- sends data to an open a DDE-link with MS-Excel
ps	- produces an image file in PostScript format
ribuf	- reads an Ibuf file
wibuf	- writes an Ibuf file

Overview of all families . . .

MISCELLANEOUS OPERATIONS

This category of operations perform various useful actions

bgm	- reads or writes a single pixel
cal	- multiplies return parameter for calibration purposes
dump	- fills an image with a grey value
edit	- edits an image part
editi	- edits Ibuf
era	- erases an image or selected bitplanes
era16	- erases a 16-bits image
fscan	- reads integer values from file
hist	- builds histogram data in Ibuf
ibuf	- reads/writes a byte in Ibuf
ihis	- copies the data from Ibuf in an image line
keep	- erases an image; keeps selected bitplanes
line	- fills Ibuf with data from horizontal image line
noise	- produces random noise image
wig	- produces various wedges
wcur	- writes current cursor in display
wrxy	- outputs data from another command in X/Y-format

Overview of all families . . .

PARAMETER OPERATIONS

Parameter operations produce data out of images, in addition to the image operation

comp	- compares two images or an image and a constant
dist	- returns (absolute) distance between points
fcont	- returns length of contour / produces Freeman code
label	- separates objects by assigning unique grey values
ibuf	- returns the specified entry from Ibuf
mark	- searches for a closed object having 1 pixel value
maxl	- finds maximum diameter
qorde	- orders fringes
qphas	- interpolates image consisting of ordered fringes
rdln	- reads pixels 'under' line
rdpat	- reads pixels 'under' Freeman pattern
rdvec	- reads pixels 'under' vector
stat	- computes various grey value properties from image histogram

Overview of all families . . .

PATTERN RECOGNITION OPERATIONS

Pattern recognition operations show image properties graphically

corr - correlates two images by producing a scatter plot
phis - plots data from two image lines (X and Y information)

Overview of all families . . .

PIXEL OPERATIONS

Pixel operations perform several actions on a pixel-by-pixel basis

add	- adds two images or an image and a constant
and	- ANDs two images or an image and a constant
bit	- shows a bitplane
cmpr	- compresses the contrast
comp	- compares two images or an image and a constant
conv	- converts one pixel value in another
cstr	- stretches the contrast of an image
div	- divides (arithmetically) two images or an image and a constant
ehis	- histogram equalisation
inv	- inverts an image
log	- logarithmic conversion
mul	- multiplies two images or an image and a constant
neg	- produces the 2's complement of an image
or	- ORs two images or an image and a constant
sel	- selects pixels from two images
shl	- shifts the bits of an image left
shr	- shifts the bits of an image right
strip	- removes a range of grey values from an image
sub	- subtracts two images or an image and a constant
tab	- converts an image using the content of Ibuf as a LUT
thre	- thresholds an image
val	- keeps pixels having grey value in a specified range
xor	- XORs two images or an image and a constant

Overview of all families . . .

REAL TIME OPERATIONS (Series 100 only)

These operations control frame grabber with proper hardware to do real time image processing

rt	- real time using current set up
rta	- real time average
rtc	- real time compare (abs (rt.image - ref. image))
rtd	- real time difference (abs (rt.image - prev.image))
rte	- real time edge detection
rtm	- real time minus (rt.image -prev.image) +32
rts	- real time subtract (rt.image -ref. image) +32

Overview of all families . . .

TRANSPORT OPERATIONS

Transport operations copy images from one place to another

copy	- copies images and/or files
dis	- copies an image to the actual display
save	- copies the actual display to an image
swap	- exchanges (swaps) two images
tcopy	- copies images and/or files with TIFF header
tdis	- copies a TIFF file into the actual display
tsave	- copies the actual display to a TIFF file

Overview of all families . . .

NEIGHBOURHOOD OPERATIONS

This family contains linear ond non-linear filters.
Window (or neighbourhood) operations calculate resulting pixel values by taking into account a neighbourhood around the central pixel.

Linear filters (convolutions)

dgaus - fast implementation of gaussian blur (dual scan)
dt - distance transform
filt - universal convolution filter
gaus - gaussian blur (convolution)
grad - gradient operator (1st derivative)
lapl - laplacian operator (2nd derivative)
qlap - quick running (3x3) laplace operation
qshrp - quick running (3x3) sharpening
shrp - sharpens an image
unif - uniform blur

Non-linear filters

kuwa - kuwahara filter
max - maximum filter
min - minimum filter
perc - percentile filter
robg - direction independent contour operator

Overview of all families . . .

TABLE OPERATIONS

Table operations are a subclass of the pixel opartion family. The implementation as table look up operations gives them some unique properties:

They can be forced to produce a table only. The table can be used by other operations.
See ibuf, lut
In preview mode the effect of operations can be observed non-destructively.

The following operations are table operations:

bit	- shows a bitplane
cmpr	- compresses the contrast
comp	- compares an image and a constant
conv	- converts one pixel value in another
cstr	- stretches the contrast of an image
ehis	- histogram equalisation
inv	- inverts an image
log	- logarithmic conversion
shl	- shifts the bits of an image left
shr	- shifts the bits of an image right
strip	- removes a range of grey values from an image
tab	- converts an image using the content of Ibuf as a LUT
thre	- thresholds an image
val	- keeps pixels having grey value in a specified range

The following bitplane operations are also implemented as table look up operations:

band	- ANDs two bitplanes
bbord	- sets/resets a border
bcop	- copies one bitplane to another
bdump	- fills an entire bitplane
binv	- inverts a bitplane
bor	- ORs two bitplanes
bsw	- exchanges two bitplanes
bxor	- XORs two bitplanes

Overview of all families . . .

add

Command syntax

```
1. add a b [/]
2. add [a] #
```

Return value:

number of overflows

Family

Pixel operation

Function

1. Adds two images.
2. Adds an image and a constant

Description . . .

Description

This operation adds two images or an image and a constant, pixel by pixel. If the result of an addition is greater than 255, it is considered an overflow. The result will then be set to 255, and an overflow counter is incremented. The content of this counter is the return parameter.

If - in the case of adding two images - the special parameter ('/') is entered, the result of each addition is divided by two, thus producing the mean of the two images as a result. Consequently, no overflow can occur in this situation.

Examples

`add a b` add **a** and **b**

`add a 11` add 11 to **a**

`add 11` add 11 to the default source

`add a a >a` add **a** and **a**, copy result to **a**

`add a b /` calculate the mean of **a** and **b**

`add a b / >c` as above, copy the result to **c**

and

Command syntax

```
1. and a b
2. and [a] #
```

Return value

none

Family

Pixel operation

Function

1. The pixels of two images are logically AND-ed
2. The pixels of an image are AND-ed with a constant.

Description . . .

Description

This is a bitwise binary operation.

This operation performs the logic AND-function of the pixels of two images, or the AND-function of an image and a constant. This means that the bit(s) of the resulting pixel will be 1 if the corresponding bits of both source pixels (or source pixel and constant) are 1.

Examples

`and a b`	**a** and **b** are AND-ed
`and a b >c`	as above; in addition the result is copied to **c**
`and 128`	the current image is AND-ed with 128 (binary: 1000 0000) - only the most significant bit remains
`and a 15`	**a** is AND-ed with 15 (binary 0000 1111); the four lower bits remain.

band

Command syntax
```
band [a] #b1 #b2 [#b3]
```

Parameters
#b1 - source bitplane no. 1 (1 - 8)
#b2 - source bitplane no. 2 (1 - 8)
#b3 - destination bitplane; (1 - 8; default: #b1)

Return value
none

Family
Bitplane operation

Function
Binary ANDS two bitplanes in a single image

Description . . .

Description

Bit no. #b3 of a pixel is set to 1 if, and only if, the bits #b1 AND #b2 of that pixel are 1. If no parameter #b3 is given, the bitplane specified by parameter #b1 is the destination bitplane.

If **'/'** is specified, only the look up table is computed.

Examples

`band a 1 2`	bitplanes 1 and 2 of **a** are AND-ed, the result is put into bitplane 1.
`band a 1 2 3`	as above, result in bitplane 3.
`band 1 2 >a`	as in 1; default source image and the result transported to **a**.
`band 1 2 /`	only produce a look-up table, and
`lut 2 1 4`	copy the table to frame grabber LUT for display

See also: table operations and preview

bbord

Command syntax
```
bbord [a] #b [#]
```

Parameters
#b - bitplane to write (1 - 8)
- action (1 = set border (default), 0 = remove border)

Return value
none

Family
Bitplane operation

Function
Writes or erases a border around a bitplane image.

Description . . .

Description

This command writes or removes border pixels in a bitplane of an image. The border pixels can be set or reset.

Examples

`bbord 1`	sets border in bitplane 1 of default image
`bbord pc red 0`	removes border pixels in bitplane red (alias for 1) of sub-image of **p**

bcop

Command syntax

```
bcop [a] #b1 #b2 [/]
```

Parameters

#b1 - source bitplane (1 - 8, no default)
#b2 - destination bitplane (1 - 8, no default)

Return value

none

Family

Bitplane operation

Function

Copies bitplane #b1 to bitplane #b2.

Description . . .

Description

This operation copies bitplane #1 to another bitplane #2. The two parameters must be specified; there are no defaults.

If **'/'** is specified, only the look up table is computed.
See also: table operations and preview

Examples

`bcop a 1 2`	bitplane 1 of **a** is copied into bitplane 2 of **a**.
`bcop 1 2`	as above; the operation takes place in the default image.
`bcop 1 2 /`	only produce a look-up table, and
`lut 2 1 4`	copy the table to frame grabber LUT for display

bdump

Command syntax:
`bdump [a] #b [#]`

Parameters

#b	specifies the bitplane number (1 - 8; default 1)
#	1: sets the bitplane to 1 (default) 0: resets the bitplane to 0

Return value
none

Family
Bitplane operation

Function
Sets the bits of a bitplane .

Description . . .

Description

This commands sets or clears the bits of a binary image.

Examples

`bdump 1 0`	resets bitplane 1 of default image
`bdump pc green`	sets the bits of sub-image of **p** in bitplane green (alias)

bgi

Command syntax:

```
1. bgi #1 [#2]
2. bgi <file>
```

Return value

1. Previous value
2. none

Function

1. Reads or sets an individual frame grabber register
2. Sets frame grabber registers according to the content of a FgRegs file

Description . . .

Description

This command allows control of frame grabber registers. Realize that ***TIMWIN*** may get disturbed if vital registers are modified. The nature of register access depends on the frame grabber: Imaging Technology's VFG grabber has 16 bits access; all others have 8-bits access

1. `bgi #1 [#2]`
   Command line read and write of a single register
   #1 - register address. If #1 < 20h, it is considered an offset to the register base address, else it is considered an absolute value
   #2 - new content (previous value is returned)

2. `bgi <file>`
   Writing registers through a command file. The lines in the file must have the following format (example):

   ```
   8 0ff00     ;comment.
   ```

   This writes 0ff00 (hex) to register offset 8 (the register base address is found in the installation data base). Notice that the values are considered hexadecimal.

The distribution disks contain some **bgi** files `(fgregs.*)` for specific environments.

Warning: this operation allows you to interfere directly with computer hardware. Wrong use of this operation may cause your computer to malfunction.

bgm

Command syntax:

```
1. bgm [a] #1 [#Y #X]
2. bgm [a] [#Y #X]
```

Return value

1. original pixel value
2. pixel value

Family

Miscellaneous operation

Function

This operation writes (1.) or reads (2.) single pixels into an image.

Description . . .

Description

This function reads or writes a single pixel. The meaning of the numerical parameters depends on their number (see under **No.**)

No.	Example	Function
0	`bgm`	Read pixel at cursor
1	`bgm 22`	Write value 22 at cursor
2	`bgm 11 22`	Read pixel value at 11 (Y), 22 (X)
3	`bgm 11 22 33`	Write 11 at 22 (Y), 33 (X)

binv

Command syntax:
`binv [a] #b`

Return value
None

Family
Bitplane operation

Function
Inverts bitplane no. #b (1 - 8)

Description . . .

Description

The indicated bitplane is logically inverted: bits having the value 1 are set to 0 and vice versa.

If **'/'** is specified, only the look up table is computed.
See also: table operations and preview.

Examples

`binv a 1`	bitplane 1 of **a** is inverted
`binv 5 /`	only produces a look-up table, that will invert bitplane 5, and
`lut 2 1 4`	copy the table to frame grabber LUT for display

bit

Command syntax:
`bit [a] #b`

Return value
none

Function
Copies bitplane #b (1 - 8) to allother bitplanes.

Family
Pixel operation

Description . . .

Description

In the resulting image, each pixel whose bit '#' was 1, will have value 255. The other pixels will have value 0.
To put it another way: all bitplanes will be made equal to the specified bitplane.

If **'/'** is specified, only the look up table is computed.
See also: table operations and preview

Examples

`bit a 1`	duplicates the least significant bit of image **a**
`bit p 8 >a`	shows the most significant bit of image **p**, copies the result to **a**
`bit 8 /`	makes a table in Ibuf, to be used to show the most significant bit of an image,
`lut 2 1 4`	and copy the table to frame grabber LUT for display

blow

Command syntax:
`blow a [#]`

Return value
none

Family
Geometric operation

Function
Enlarges source image by replacing each pixel with NxN pixels of the same value, where N = # (default 2).

Description . . .

Description

This function enlarges an image by replacing each pixel by a NxN array of pixels of the same value.

The size of the resulting image depends upon the size of the source image and the enlarging factor; however, the size can never exceed the destination image's size.

If the resulting image would exceed this size, the result is limited by skipping the pixels that fall outside the maximum format. The upper-left corner of the image is the starting point in the enlarging procedure.

Examples

`blow a1` enlarges sub-image **a1** 2x
`blow bc 3` enlarges sub-image **bc** 3x

Comment

There is no default image specification in this operation. The default display cannot act as a source; the pixels to be enlarged would be overwritten before they could be processed.

See also: ct, dim

bor

Command syntax:

```
bor [a] #b1 #b2 [#b3]
```

Parameters
#b1 - source bitplane no. 1 (1 - 8)
#b2 - source bitplane no. 2 (1 - 8)
#b3 - destination bitplane (1 - 8; default: #b1)

Return value
none

Family
Bitplane operation

Function
Binary OR of two bitplanes

Description . . .

Description

Bit no. #b3 of a pixel is set to 1 if the bits #b1 OR #b2 of that pixel are 1. If no parameter #b3 is given, the bitplane specified by parameter #b1 is the destination bitplane.

If **'/'** is specified, only the look up table is computed.
See also: table operations and preview

Examples

`bor a 1 2`	bitplanes 1 and 2 of **a** are OR-ed, result in bitplane 1.
`bor a 1 2 3`	as above, result in bitplane 3.
`bor 2 3 /`	only produces a look-up table that ORs bitplane 2 and 3; result in bitplane 2,
`lut 2 1 4`	and copy the table to frame grabber LUT for display

bord

Command syntax:
```
bord [a] [#b]
```

Parameters
#b - value of border pixels (0 - 255; default: 255)

Return value
none

Family
Graphic operation

Function
Writes a border around the image.

Description . . .

Description

This operation draws a border with the specified grey value in the outside pixels of the image.

Note, that the border is *in* the image. This is different from the box-cursor (which can be written using the wcur command), which is drawn at the outside of the (sub) image.

See also: bbord

Examples

`bord 0`	clears pixels at the border in the default image
`bord pc`	draws a border around sub image **pc**, with grey value 255

bsw

Command syntax:
`bsw [a] #b1 #b2`

Parameters
#b1 - no. 1 bitplane (1 - 8)
#b2 - no. 2 bitplane (1 - 8)

Return value
none

Family
Bitplane operation

Function
Swaps (exchanges) the bitplanes #b1 and #b2 (1 - 8).

Description . . .

Description

This operation exchanges two bitplanes.

If **'/'** is specified, only the look up table is computed.
See also: table operations and preview

Examples

`bsw a 1 2`	bitplanes 1 and 2 of **a** are swapped.
`bsw 1 2`	as above; default source involved.
`bsw 8 7 /`	a look up table is produced, which swaps bitplanes 7 and 8, and
`lut 2 1 4`	copy the table to frame grabber LUT for display

bxor

Command syntax:
```
bxor [a] #b1 #b2 [#b3]
```

Parameters
#b1 - source bitplane no. 1 (1 - 8)
#b2 - source bitplane no. 2 (1 - 8)
#b3 - destination bitplane (1 - 8; default: #1)

Return value
None

Family
Bitplane operation

Function
Binary exclusive OR (XOR) of two bitplanes

Description . . .

Description

Bit no. #b3 of a pixel is set to 0 if the bits in bitplane #b1 and #b2 of that pixel are equal, and set to 1 if they are different. If no parameter #b3 is given, the bitplane specified by parameter #b1 is the destination bitplane.

If **'/'** is specified, only the look up table is computed.
See also: table operations and preview

Examples

`bxor a 1 2`	bitplanes 1 and 2 of **a** are XOR-ed, result in bitplane 1.
`bxor a 1 2 3`	as above with result in bitplane 3.
`bxor 1 2 /`	as 1., but only a look up table is produced,
`lut 2 1 4`	copy the table to frame grabber LUT for display

cal

Command syntax:
```
cal <TIM-command>
```

Function
Modifies (calibrates) the return value of the specified TIM command

Family
Miscellaneous operation

Return Value
Corrected return value of <TIM command>

Description . . .

Description

cal executes the specified TIM command, and multiplies the return parameter with the value, specified with the set command or the set menu.

Thus return parameters (e.g. measured values) can be calibrated.

Examples

```
cal dist 11 22 33 44
```

executes **dist**, which calculates the distance between specified points in pixel distances, then modifies this value to represent user specified units.

```
label cermet -128
cal mark 10
```

modifies the area of object no. 10 in image **cermet**, as returned by the **mark** operation.

If the calibration factor is a linear measure (e.g. 1 pixel == 2.3 mm) you have to apply **cal** twice for correct calibration of area. The above example should then be:

```
cal cal mark 10
```

NOTE: The result is floating point, independent of the type of <TIM-command>'s return parameter.

cirk

Command syntax:

```
1. cirk [a] #r [#pd [#Y #X]]
2. cirk [a] #r #pd #XY
```

Return value

none

Family

Graphic operation

Function

Draws a circle.

Description . . .

Description

A circle is drawn at the specified (or default) position.

The meaning of the numerical parameters depends on their number:

No.	Example	Function
1	`cirk 22`	Draw a circle, radius 22, drawing value at the cursor
2	`cirk 11 22`	Draw a circle, radius 11, pixel value 22 at the cursor
3	`cirk 9 1 120034h`	Draw a circle, radius 9, pixel value 1 at 12h (Y), 34h (X)
4	`cirk 9 1 66 77`	Draw a circle, radius 9, pixel value 1 at 66 (Y), 77 (X)

Comment

With this operation the specified image is the destination image.

cmpr

Command syntax:
`cmpr [a] #1 #2 [/]`

Parameters
#1 - lower boundary value (0 - 255)
#2 - upper boundary value (0 - 255)

Return value
none

Family
Pixel operation

Function
Compresses contrast by mapping all gray values into the range #1 to #2 .

Description . . .

Description

The span of grey values in the image is compressed by defining a look up table starting with the value of #1 and ending with #2, and converting the pixels with this table. The minimum and maximum values may be specified in any order.

If you process an image with pixel values of 0 and 255, after this operation the minimum and maximum values in the image will be #1 and #2.

If **'/'** is specified, only the look up table is computed.
See also: table operations and preview.

Examples

`cmpr 100 200`	if the original image's grey value range was 0 to 255 it will be 100 up to and including 200 after processing.
`cmpr a 0 6`	make a an image with 7 grey values (0 up to and including 6 if the original had values 0 to 255)
`cmpr 10 20 /`	only produce a look-up table, and
`lut 2 1 4`	copy the table to frame grabber LUT for display

comp

Command syntax:

```
1. comp a <=!=> b
2. comp a <=!=> #
```

Return value

number of matching pixels

Family

Pixel operation

Function

1. Compares pixels of two images
2. Compares an image and a constant

Description . . .

Description

Applies a relational operation to an image and a constant, or to two images. If the relation is true, the resulting pixel value is 255; if it is false, the pixel is set to 0. The number of pixels, for which the relation is true, are counted and returned as return parameter.

Valid relations are:	
`==`	is equal
`!=`	is not equal
`>`	is greater than
`<`	is less than
`>=`	is greater or equal
`<=`	is less or equal

Examples

`comp a > 128` produces an image with pixels 255 where pixels in **a** are >128
`comp p == a` produces an image with pixels 255 where pixels in **a** and **p** are identical.

Comment

This operation has no defaults; all parameters must be specified.

conv

Command syntax:
`conv [a] #1 #2 [/]`

Parameters
#1 - original pixel value (0 - 255)
#2 - new pixel value (0 -255)

Return value
none

Family
Pixel operation

Function
Converts one gray value (#1) into another (#2.).

Description . . .

Description

This operation leaves the image unchanged except for pixels with value #1, which are changed to value #2.

If **'/'** is specified, only the look up table is computed.
See also: table operations and preview.

Examples

`conv a 255 0 >b`	converts pixel value 255 of **a** into 0, copies result to **b**
`conv 127 255 /` `lut 2 1 4`	produces a table, in which 127 is converted to 255, and copy the table to frame grabber LUT for display

copy

Command syntax:

```
1. copy a b
2. copy <file> a [#]
3. copy a <file>
4. copy a zz*
5. copy zz* b
```

Family

Transport operation

Function

Copies images.

Description . . .

Description

1. `copy a b`
   Copy source to destination

2. `copy <file> a [#]`
   Copy image file to image. If the user didn't specify a directory or an extension, TIMWIN adds them:
   - the default directory, as specified in Install, is used.
   - the extension is: .im

   A numerical parameter specifies an offset in the file. Image reading starts after the offset. This is useful if a file format with an unknown header has to be read. The default is: 0 (TIMWIN images dont't have headers).

3. `copy a <file>`
   Copy an image to a file
   This operation writes a pure binary file to disk. If the user didn't specify a directory or an extension, TIMWIN adds them:
   - the default directory, as specified in Install, is used.
   - the extension is: .im

4. `copy a zz*`
   Create a unique file name
   The string will be expanded into '`zzxXXXXX`', where x is '0, a - z' and XXXXX is a number, which remains the same during the session. Examples: `zz016335, zza16335`, etc. Note: no extension (.im) will be appended.

5. `copy zz* b`
   Read all matching files
   All matching files will successively be copied. If no matching file exists, an error message will be produced.
   For regular TIM files, specify '`zz*.im`'. For files specified using the construction in 4. above, specify '`zz*.`' This suppresses the addition of an extension.

corr

Command syntax:
`corr a b [#]`

Parameter
= 1: plotting takes place by incrementing present pixel values. Else: graphics value is plotted

Return value
none

Family
Pattern Recognition operation

Function
Plots dots in X-Y space of default image. The addresses come from the pixels of both source images: **a** delivers the Y-addresses, **b** the X-addresses.

Description . . .

Description

This operation produces a scatter plot of dots in the default source, which shows the correlation of two images. The pixel value of the dots is the graphics value (see **set** command).
If the two images specified are completely equal, the X- and Y-addresses of all dots are equal, which results in a straight line running from the upper left corner to the lower right corner.

Differences in the two images, even imperceptibly small, develop deviations from this straight line, which can be recognized easily. Also, properties of the images themselves can be observed: distribution and span of grey values, number of bits used in pixels, etc. Using the same image for both source specifiers shows these properties for one image alone.

Examples

corr a b	produce a scatter plot using pixels from **a** to supply the X-address, and from **b** to supply the Y -address.
`corr a a`	show various properties of the image in **a** (see Description)

Comment

Because of the sensitivity of this operation do not use the default source as a source image. Plotting dots would change one of the source images, resulting in random dots appearing as the operation continues.

cp16

Command syntax:
`cp16 a16 [b] [#] [/]`

Parameters

#	if **/** is specified: division factor
	if not, shift count
/	pixel scaling modifier

Function

Copies a 16-bits image to an 8-bit image while scaling the pixels.

Description . . .

Description

To bring a 16-bits image with arbitrary pixel values into an 8-bits range, the pixels must be scaled. This operation does so using two methods:

- by shifting the pixels binary. The relation between shifting and division is shown in the table below.
- by dividing the pixels using the specified divisor. This method if chosen if the '/' parameter is specified.

The relation between shift count and division factor is:

division by	shift count	division by	shift count
2	1	32	5
4	2	64	6
8	3	128	7
16	4	256	8

Examples

`cp16 m16 3 /`	divides the pixels of 16-bits image **m16** by 3 and writes the result to the default destination (8-bits)
`cp16 m16 p 8`	shifts the pixels of 16-bits image **m16** 8 positions to the right and writes to **p**

cstr

Command syntax:

```
1. cstr [a] [#] [/]
2. cstr [a] #1 #2 [/]
```

Return value

none

Family

Pixel operation

Function

Stretch the contrast of an image:

1. Automatic
2. Manual

Description . . .

Description

1. Automatic contrast stretch
The pixels belonging to the #% having minimum and maximum values are set to 0 and 255. The rest is scaled in proportion. Default percentage (#): 1.

2. Manual contrast stretch
All pixel values low #1 are replaced by 0, and all pixel values of #2 and above are replaced by 255. The rest is scaled in proportion.

If **'/'** is specified, only the look up table is computed.
See also: table operations and preview.

Examples

`cstr a 22 222`	**a**'s contrast is stretched using the values specified.
`cstr >b`	the default source's contrast is stretched automatically. 98% (=100-1-1) of the pixels count for the calculation of the stretching values. The result is copied to **b**.
cstr 5	stretch contrast so, that the 5% lowest and 5% highest pixels are set to minimum (0) resp. maximum (255)
`cstr /`	only produce a look-up table, and
`lut 2 1 4`	copy the table to frame grabber LUT for display

ct

Command syntax:

```
1. ct [a]
2. ct [a] #
3. ct [a] #Y1 #X1 #Y2 #X2 #Y3 #X3 [#Y4 #X4]
```

Family
Geometric operation

Function
Coordinate transform.

Description . . .

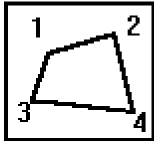
Description

This operation takes the image, which must be specified in one of the following ways, and maps it into the destination image by interpolating pixel values bilinearly.

```
1. ct [a]
```
The specified (sub) image is transformed to fit into the destination image.

```
2. ct [a] #
```
The indicated image is rotated over # degrees.

```
3. ct [a] #Y1 #X1 #Y2 #X2 #Y3 #X3 [#Y4 #X4]
```



The image part, bound by the specified coordinates, is transformed to fit in the destination image. If the lower right coordinates (#Y4 - #X4) are not specified, the Y-value is that of the lower left corner (#Y4 = #Y3) and the X-value is that of the upper right corner (#X4 = #X2).

Examples

`ct ac`	blow image **ac** to the current image format
`ct a 22`	rotate image **a** 22 degrees
`ct a 0 0 0 255 255 127 255 128`	
	map triangular part of image **b** onto destination image

curlock

Command syntax:
```
curlock a [b] [#]
```

Parameter
- 0: remove lock, 1: set lock (default)

Return value
previous lock status

Family
Control operation

Function
This command controls cursor locking: the cursor of image **a** is locked to the cursor of image **b**.
That is, the cursor of **a** follows the cursor of **b**.

Description . . .

Description

Locking cursors of images means that the sub image position of one image is locked to another. If the cursor of **a** is locked to **b**, **a**'s cursor will follow **b**'s when **b**'s cursor is changed for any reason.
If an image is locked to more than one image, its cursor will not be updated as a result of switching active images. Only if the image's cursor changes as a result of an action in the image, the corresponding cursors will be updated.

Example

`curlock a p`	lock **a**'s cursor to **p**'s
`curlock a p 0`	remove the lock

curs

Command syntax:

```
1. curs [a] #Y #X
2. curs [a] #XY
3. curs [a]
4. curs a b
5. curs [a] <SpecPar> [#]
```

Return value

1. previous cursor position (packed format)
2. as 1.
3. cursor position (packed format)
4. cursor position (packed format)
5. previous type

Family

Control operation

Function

This operation controls the image cursors

1. Positions the cursor of the indicated image
2. Positions the cursor of the indicated image (packed parameter).
3. Returns the cursor position of the specified image.
4. Positions the cursor of image **b** to that of image **a**.
5. Controls cursor type and optionally activates the image cursor
   - **of** (off)
   - **cr** (cross)
   - **ch** (crosshair)
   - **bo** (box)
   - **ar** (arrow)

Description . . .

Description

This operation controls the cursor position, attributes or type of the specified image(s).

1. `curs [a] #Y #X`
If two numerical parameters are entered, the cursor position of the specified image is set. The first numerical parameter specifies the Y-value, the second specifies the X-value.

2. `curs [a] #XY`
If one numerical parameter is entered, it will be considered a packed address.

3. `curs [a]`
If no numerical parameter is entered, the routine reads the cursor position of the specified image. The cursor position is returned in packed format.

4. `curs a b`
If two images are specified, the cursor position of the second image is copied to the first.

5. `curs [a] <SpecPar> [#]`
If one of the mentioned special parameters is specified, the cursor type is selected. If the optional numerical parameter is 1, the mouse will control the cursor of the indicated image. If 0, mouse control is returned to Windows.

Examples

`curs 44 55`	set curs.pos. of default source to 44(Y) - 55(X)
`curs 800020H`	set curs.pos. to 128(Y)-32(X) (80H = 128; 20H = 32)
`curs s`	get cursor position of image **s**.
`wrxy curs s`	return the cursor position of **s** in a readable format
`curs p q`	set cursor position of **p** to that of **q**.
`curs q bo`	set cursor type of **q** to: box
`curs cr 1`	set cursor type the default image to a cross and activate mouse control

cursx

Command syntax:
`cursx [a] [#]`

Return value
If no numerical parameter: X - cursor position
Else: previous X - cursor position

Family
Control operation

Function
Sets/reads the horizontal (X) position of the cursor.

Description . . .

Description

This operation reads or sets the horizontal cursor position. It offer a subset of the curs operation.

Examples

`cursx 0`	position the cursor of the default source in the leftmost column
`cursx a 128`	position the cursor of image a to the 128th column
`cursx x`	read the horizontal cursor position of image x

cursy

Command syntax:
`cursy [a] [#]`

Return value
If no numerical parameter: Y-cursor position
Else: previous Y- cursor position

Family
Control operation

Function
Sets/reads the vertical (Y) position of the cursor

Description . . .

Description

This operation reads or sets the vertical cursor position. It offer a subset of the curs operation.

Examples

`cursy 0`	position the cursor of the default source in the top row
`cursy a 128`	position the cursor of image a to the 128th row
`cursy x`	read the vertical cursor position of image x

del

Command syntax:
`del <file>`

Function
Deletes specified file.

Comment
The file specification is interpreted literally; this command cannot handle wildcards (e.g. *.*), nor does it provide automatic paths or extensions.

dest

Command syntax:

```
1. dest [p]
2. dest #
```

Return value
previous destination

Family
Control operation

Function
1. Changes the default destination (frame grabber display does not change).
2. as 1, using encoded numerical return parameter. Use this version only with values derived from the **dest** command itself.

Description . . .

Description

The **dest** operation controls the default destination image. This operation will not influence the display. Al image types may be selected as destination images.

The numerical parameter may be used in command files, when the destination image needs to be changed, and restored to its original value afterward.

Examples

`dest q`	make **q** the destination image
	Command file usage:
`olddest = dest a`	Read present destination image, store encoded parameter, set destination to **a**
`. . . .`	
`dest olddest`	restore original destination after finishing command file

dgaus

Command syntax:
`dgaus [a] [#]`

Parameters
- filter size (3 (default), 5, 7, 9)

Family
Neighbourghood operation

Function
Gaussian filter, implemented in a fast (dual scan) mode.

Description . . .

Description

The Gaussian filter has a blurring effect. It uses a coefficient scheme, whose values approximate a Gaussian curve. With such a Gaussian filter a less dramatic effect is produced than just taking the mean value of all pixels in the window, as performed by the unif uniform filter operation.

The dual scan implementation yields a greater speed than the standard **gaus** operation (especially with large filter sizes), by separating the operation in two linear operations, horizontal and vertical.

Examples

`dgaus`	produces a 5x5 gaussian blur of default source
`dgaus a 9`	produces a 9x9 gaussian blur of **a**

See also: gaus

dig3

Command syntax:
```
dig3  #
```

Family
Control operation

Function
Controls digitizing of 3 frame grabbers in colour system.

Description . . .

Description

This operation allows control of more than 1 (up to 3) frame grabbers the same time. This is useful where grabbing the same frame with several cameras is essential. For more information regarding control of the frame grabbers please consult the description of the **dig** command.

Comment

dig3 is not capable of changing display in each of the frame grabbers, so you have to set the frame grabbers into the desired status before issuing the **dig3** command (therefore no image specifier is allowed in the command, although supplying one wil not produce an error).

Examples

`dig3 1`	grab an image in each of 3 frame grabbers the same time
`dig3`	start digitizing in each of the 3 frame grabbers; the ***TIMWIN*** prompt returns immediately

For details see dig

dig

Command syntax:
`dig [p] [#]`

Parameter
#	delay (number of frames)
# = 0:	grabbing continues until OK button in dialog button is pressed (default)
# = -1:	start frame grabbing, return control to TIM without freezing. To stop grabbing: enter `dig 1`

Family
Control operation

Function
Controls image aquisition by the frame grabber

Description . . .

Description

Grabbing is started by this command.and optionally finished by freezing the image after the specified period. **#** specifies the number of frames by which the actual freezing of the image has to be delayed. One frame takes 40ms (CCIR Europe) or 32ms (RS170 USA).

If the specified image differs from the current active image (position, size), the display window is adjusted for the duration of the command. The original display window is restored, except when no duration is specified.

Series 100 frame grabbers (including VFG) are capable of grabbing an image while displaying another. However, this is only the case with zoom value 0 (no zoom in).

If you do not specify a numerical parameter, frame grabbing starts, but the TIMWIN prompt returns immediately. In this mode, you can enter all TIMWIN commands while frame grabbing continues. You may manipulate the input or output Look Up Tables, the host- and video mask settings, gain and offset (see the set command), or take histograms of the images being digitized. Note, that other commands may cause grabbing to stop as a side effect.

If an image is specified, the frame grabber will switch to that display (zoom in, if necessary - depending on the grabber), perform grabbing, and switch back (zoom out again, if necessary). Switching back will not occur, however, with the continuous version (# = -1) of the command.

The window, in which grabbing occurs, is determined by frame grabber hardware. It may not match entirely the user defined images, as specified in **images.tim**. Usually the entire area, visible on the monitor screen, will be written.

Examples

`dig 1`	grab one frame (digitize the first complete video frame)
`dig`	start FG-mode; ***TIMWIN*** prompt will return immediately, while digitizing continues
`dig 100`	start digitizing mode, stop after 100 frames (4s); grab 100th frame
`dig p 1`	zoom into **p**, grab 1 frame in **p** and zoom out (zooming in and out will only occur if necessary)
`dig p 1 >a`	as above; image is copied to **a**
`dig 0`	start grabbing; pressing a key will end grabbing

dim

Command syntax:
```
dim [a] #
```

Parameters
- percentage of horizontal stretching; default: 100.

Return value
none

Family
Geometric operation

Function
Enlarges or reduces the image horizontally.

Description . . .

Description

The image's horizontal dimension is changed. The numerical parameter specifies the percentage of the width change.

This operation is useful to correct images made with non-square pixel frame grabbers.

The pixels are assigned new values by a protocol based upon interpolation of source pixel values. This gives good results with 'normal' grey images, but with special images (graphics, color coded images, information in bit planes) undesirable results can be produced.
If the percentage is less than 100 (reducing the image), the result is centered, leaving two columns in the destination image, left and right, unmodified.

If the image is enlarged, a sub image should be specified as a source. The enlargement could produce an image that exceeds the destination image format. For constructing the result, as many pixels are used as possible. If the number of pixels necessary for the requested enlargement exceeds this maximum, a smaller part is enlarged. The starting point for enlargement is the upper left corner of the specified (sub) image.

Examples

`dim a 70`	reduce the image in a to 70% of its original width. This value can be used to correct for standard (10MHz) non-square pixel frame grabbers.
`dim pc 200`	stretch sub image pc to become twice its original width

See also ct, redu, blow

dis

Command syntax:

```
1. dis a
2. dis p
3. dis [#]
4. dis <file>
5. dis zz*
```

Parameters

- image code as produced by **dis** and **dest**

Return value

1, 2, 3. (previous) image code
3, 4. none

Family

Control operation
Transport operation

Function

1. Copies an image to the default image
2,3. Makes the indicated display the default image
4. Reads the specified file from disk into the default image
5. All matching files will successively be copied.

Description . . .

Description

`1. dis a`
dis offers a subset of the general **copy** command: **dis** copies the specified image or the disk file to the default display.

`2. dis p`
If the specified image is a display image (frame grabber or windows) no image copying takes place: the specified display will become the active image and will be made visible.

`3. dis [#]`
The numerical parameter may be used in command files, when the destination image needs to be changed, and restored to its original value afterward (see the example below)

`3. dis <file>`
Copy image file to the active image. If the user didn't specify a directory or an extension, TIMWIN adds them:

- the default directory, as specified in Install, is used.
- the extension is: .im

`4. dis zz*`
Read all matching files
All matching files will successively be copied. If no matching file exists, an error message will be produced.
For regular TIMWIN image files, specify '`zz*.im`'. For files specified using the wildcard construction with copy or save, specify '`zz*.`'. This suppresses the addition of an extension.

Examples

`dis a`	copies the content of image a to the default source
`dis image`	copies the content of disk file <path>image.im to the default image
`dis im*`	finds matching file; copies it if found.
`dis q`	make **q** the active image
Command file usage:	
`olddis = dis a`	Read present destination image, store encoded parameter, set destination to **a**
`. . . .`	
`dis olddis`	restore original destination after finishing command file

Comment

The numerical code can be used in command files to read the status of the default source on entrance, keep it in a variable and restore the setting on exit.

See also: dest, copy, tdis

Default image search path: see How to control directories
Default extension: '.im'

dist

Command syntax:

```
1. dist #XY1 #XY2
2. dist #Y1 #X1 #Y2 #X2
```

Return value

Calculated distance

Family

Parameter operation

Function

Returns the Euclidian distance between two points in the image, in floating point.

Description . . .

Description

The absolute Euclidian distance is calculated using floating point arithmetic.
Correction for non-square pixels is performed using the correction factor specified in the Install menu.
The unit of distance can be modified by using the calibration factor.

Examples

`dist 0 0 3 4`	returns the distance between points specified using X and Y values (result is 5, if correction for non-square pixels is disabled)
`cal dist 0 0 3 4`	as above; the return value is modified using the calibration factor
`dist 0 100080h`	returns the distance between points using packed addresses

div

Command syntax:

```
1. div [a] #
2. div a b [#]
```

Parameter

1. divisor (1 - 256)
2. multiplier (1 (default) - ...)

Return value

1. none
2. number of overflows

Family

Pixel operation

Function

Divides two images or an image and a constant

Description . . .

Description

This function divides an image by a constant or another image.

`1. div [a] #`
If one image is specified: divides an image and a constant (formula: `a/#`)

`2. div a b [#]`
If two images are specified: divides two images. To keep the result in the range 0 - 255 a multiplication factor may be specified (formula: `a * # / b`).

Division by 0 is prohibited. However, this situation might occur when an image is divided by another image. When a pixel is divided by 0 the result is set to 255 and the overflow counter is incremented. The combined division/multiplication action in **div a b #** may create a result that is higher than 255. This also increments the overflow counter.
In 2. multiplication takes place before division to avoid round off errors.

Examples

`div 3`	divides the default source by 3
`div a 5 >b`	divides a by 5, copies the result to b
`div a a 255`	divides a by itself, multiplies the result (=1 for all pixels) by 255. This operation will set all pixels to 255.

dot

Command syntax:

```
dot [a [b]] [#]
```

Parameter

- bitplanes to copy without dithering: 0 (no, default), 1, 2 or 3

Return value

none

Function

Makes a grey value image by dithering the specified source.

Description . . .

Description

This operation converts a gey value image into a dithered binary image.

In this operation the image which is specified second is the destination image (as an exception to the general rule). The default destination is the active image, as usual.

To be able to combine a grey value image with (coloured) bitplane information in the bitplanes 1, 2 or 3, you can specify a value with this command.

Value	bitplanes to be copied literally
0 (default)	none
1	1
2	1 and 2
3	1, 2 and 3

You must load an appropriate look up table in order to see the content of the copied bitplanes. See how to ...

Examples

`dot a`	dither image a, result to default destination
`dot a 1`	as above; copy least sign. bit without dithering
`dot a h`	dither image a, result to h

drln

Command syntax:

1. `drln [a] #YX1 [#YX2 [#pd]]`
2. `drln [a] #Y1 #X1 #Y2 #X2 [#pd]`

Parameters

See line drawing

Family

Graphic operation

Return value

number of pixels on line

Function

Draws a line in an image

Description . . .

Description

A line is drawn in the specified image by writing the specified bits into the image. The meaning of the parameters depends upon their number:

For details on line drawing and parameter interpretation, see line drawing
See also: graphics concepts

Examples

`drln 11 22 33 44`	draws a line (pixel value = drawing value) in the default source
`drln a 0 0ff00ffh 1`	draws a diagonal line with value 1 in a

drpat

Command syntax:
```
1. drpat [a] [#YX] [/]
2. drpat [a] #Y #X [#pd] [/]
```

Parameters
See pattern drawing

Family
Graphic operation

Function
Draws a figure along a path, specified by a Freeman string in Ibuf.

Description . . .

Description

This function draws a figure from a Freeman contour string. The string consists of bytes having values between 0 and 7, and has to end with 255 (0ffh).

If performed immediately after the fcont operation, the exception parameter **'/'** can be specified. Then the Freeman codes are read from the internal buffer, where ~~fcont~~ stores them, and the default starting position is that of the original image. Otherwise:starting position is the cursor position.

For details on pattern drawing and parameter interpretation, see pattern drawing
See also: graphic concepts

Examples

Command	Description
`drpat`	draws a figure, specified by a Freeman string in Ibuf, from the cursor position
`drpat /`	as above, but reads Freeman string from **fcont** buffer and starts at original position
`drpat a 100 200`	draws into a, reads from ibuf, starts at 100 (Y), 200 (X)
`drpat a 100 200 /`	as above, but reads Freeman string from **fcont** buffer

drvec

Command syntax:
```
drvec [a] #a [#l [#Y #X] [#pd]]
```

Parameters
See vector drawing

Family
Graphic operation

Return value
number of pixels on vector

Function
Writes pixels along the imaginary vector, specified by the numerical parameters.

Description . . .

Description

This function draws vectors in any direction. The length is specified in real length units. If no length is specified (or the length value is 0), then the vector runs to the image edge.

Angle direction is interpreted as usual (e.g. 90 degr. is up).

For details on vector drawing and parameter interpretation, see vector drawing
See also: Concepts of graphic operations

Examples

`drvec 222`	draws a vector from the cursor position with an 222 degr. angle to the image edge
`drvec a 33 10`	draws a vector in image a from the cursor position angle 33, length 10
`drvec 10 0 128 128 1`	draws a vector from 128, 128 to the image edge, angle 10, pixel value 1

dt

Command syntax:
`dt [a]`

Return value
none

Family
Neighbourghood operation

Function
Distance transform

Description . . .

Description

This operation produces the distance transform of **a**.
In the resulting image the pixel values represent the nearest distance to the edge of the object.

The distance is calculated with more accuracy with operations, that depend on the more common 4- and 8-connected schemes.

The source of this operation should be a binary or grey value image, in which the objects are represented by pixels > 0, and the background by pixels having a greyvalue 0.

Examples

`dt` perform a distance transform on the default source

dump

Command syntax:
`dump [a] #`

Return value
none

Family
Miscellaneous operation

Function
Fills the image with gray value #.

Description . . .

Description

The pixels of the specified image are set to the specified value.

Examples

`dump 55`	fills the default source with grey value 55
`dump ac 0`	clears ac.

dvec

Command syntax:
```
dvec [a] #1 [#2 [#3 [#4 [#5]]]]
```

Return value
Position of the end vector (packed)

Family
Graphic operation

Function
Draws a vector with length #1 in freeman direction #2 Default direction: 0.
(This command is obsolete)

Description . . .

Description

The following table shows the meaning of the parameters:

Number	#1	#2	#3	#4	#5
1	length				
2	length	direction			
3	length	direction	start pnt. (XY)		
4	length	direction	start pnt.Y	start pnt.X	
5	length	direction	start pnt. Y	start pnt.X	bitmask

Default gray value: graphic value

The bit mask (default 255) is ANDed with the graphic value to determine the actual bitplanes to be written.

Note: 'length' is expressed in number of pixels, so in diagonal directions the 'real' length is $\sqrt{2}$ times as large.

edit

Command syntax:
`edit [a]`

Return value
none

Family
Miscellaneous operation

Function
Opens the image edit window.

Description . . .

Description

The image edit window shows an area out of the selected image in a numerical format. You can edit the values by selecting a cell and entering a new value.

The initial area is determined by the image's cursor position. You can move to another point in the image using the keyboard's arrow keys, or by using the window's scroll bars.

Examples

`edit`	opens image edit window for default image
`edit a`	as 1, but reading from image **a**

editi

Command syntax:
`editi [a]`

Return value
none

Family
Miscellaneous operation

Function
Opens the Ibuf edit Window.

Description . . .

Description

The Ibuf edit window shows an area from the ibuf buffer in a numerical format. You can edit the values by selecting a cell and entering a new value.

You can move to another point in the image using the keyboard's arrow keys, or by using the window's scroll bars.

See also: Ibuf

Examples

`editi`	opens Ibuf edit window

ehis

Command syntax:
`ehis [a] [/]`

Return value
none

Family
Pixel operation

Function
Performs a histogram equalization.

Description . . .

Description

Histogram equalizing is a way of distributing the pixels in an image equally over the available grey values. This is done by calculating the cumulative histogram of the image and using it, after scaling, as a look-up table to convert the image.

As a result, the distribution of the gray values over the pixels is equalized. This is, areas in the histogram that contain many pixels are stretched, others are compressed.

If **'/'** is specified, only the look up table is computed.
See also: table operations and preview.

Examples

`ehis a` equalize the histogram of a
`ehis p >a` equalize histogram of p, transport result to a
`ehis p /` only produce the look up table, necessary for equalizing p

era

Command syntax:
`era [a] [#1 [#2 [#3 ..]]]`

Parameters

#1 - bitplane to be erased (1 - 8; default: all bitplanes)
#2, #3 - more bitplanes (1 - 8)

Return value

none

Family

Miscellaneous operation

Function

Erases an image.

Description . . .

Description

This command erases an image enntirely (default) or partially (if one or more bitplanes are specified).

See also: keep, era16

Examples

`era 1 2 3`	erase bit plane 1, 2 and 3 of the default source
`era xc 8`	erase the most significant bitplane of image xc
`era a`	erase image a entirely

era16

Command syntax:
```
era16 [a16]
```

Return value
none

Family
Miscellaneous operation

Function
Erases an entire 16-bits image.

Description . . .

Description

This command erases a complete 16-bits image. Since TIMWIN by default operates on 8-bits images, the era command can only erase either the lower or the upper part of a 16-bits image.

Examples

`era16 m16` erases image m16

excelc

Command syntax:
`excelc`

Return value
none

Family
I/O operation

Function
Closes the link with the selected spreadheet

Description . . .

Description

This operation is used to control a DDE-link with an Excel (R) spreadsheet.

Only one link can be active at a time. This operation closes an existing link, that once was opened with the excelo command.

See also: excelo, excels

excelo

Command syntax:
```
excelo "name_sheet"
```

Parameter
"name_sheet" - name of the worksheet to link to.

Family
I/O operation

Function
Opens a data link with a Microsoft Excel sheet

Description . . .

Description

This operation opens a DDE-link with MS-Excel.
Excel must be running, and the named sheet must be active, otherwise the link will fail.

If the link is open, sending data can take place in the following ways:

- using the excels command
- if the DDE-update flag is active: automatically whenever Ibuf changes

See also: excelc, excels, set

Example

`excelo "sheet1"`	open link with Excel sheet1
`excels 2 1`	send the content of Ibuf to a column, that start with cell 2 (Y), 1 (X)
`excelc`	close the link when ready

excels

Command syntax:
`excels [#y #x [#]]`

Parameter
#y row number of 1st cell
#x column number of 1st cell
0: write in rows; 1: write columns

Family
I/O operation

Function
Sends the content of lbuf to the selected position in the spreadsheet opened by excelo

Description . . .

Description

This command sends the content of Ibuf to an Excel spreadsheed cell. The sheet is specified with the excelo command.

See also: excelo, excelc

Example

`excelo "sheet1"`	open link with Excel sheet1
`excels 2 1`	send the content of Ibuf to a column, that start with cell 2 (Y), 1 (X)
`excelc`	close the link when ready

exist

Command syntax:
`exist <file>`

Return value
1 (file exists), 0 (file doesn't exist)

Family
Control operation

Function
Tests for existence of file.

Description . . .

Description

This command tests the existence of the indicated file. It returns a value, which makes it suitable for use in command files to make decisions.

This operation can also be used to synchronise between systems in a network.
If a TIMWIN system is used as a frame grabbing front end for another system, the other system can pass commands to the TIMWIN front end by writing a command file to a shared disk. In the TIMWIN system a loop containing the **exist** command should be executed to test if a command file arrived already.

```
begin:                  test for command file on external system
 avail = exist n:xt.cmd
 if avail == 1
    *n:xt               if present, execute it
    del n:xt.cmd        remove it to be able to receive a new order
  endif
  wait 1                poll once a second for new command file
goto begin
```

fcont

Command syntax:

```
1. fcont [p] [#1 [#2] [#Y #X]]
2. fcont [p] [#1] [#Y #X] ib
3. fcont [p] [#1] [#Y #X] <file> ["format"]
4. fcont app|ovw|incl|excl
5. fcont #1 #2 #3 #4 #5 #6 #7 #8
```

Return value
Length along line

Parameters

#1	bit, which holds the line (1 - 8; default: 1)
#Y, #X	starting point. If not specified, fcont will scan the image from upper-left until suiting pixel found. If a sub-image is specified, following starts from the cursor position.

Family
Parameter operation

Function
Follows a contour and stores it as a Freeman string into a special buffer.

Description . . .

Description

This operation breaks a contour into a string of Freeman codes, and calculates the length along the contour very accurately. It optionally stores the string in a file or in Ibuf.

During following **fcont** 'eats' the contour, i.e. pixel bits belonging to the contour are removed. This prevents contour points to be detected twice. The contour must be an 8 or 4-connected line, as produced by the **lcon** operation. **fcont** has many options, which are described below.

1. Returns the real distance along the contour. #2 is a qualifier. If it is:
   0: the distance is measured using optimal correction constants, including correction for non square pixels (default)
   1: as 0, without correction for non square pixels
   2: distance is measured using 1 unit for hor. & vert. steps, and √2 units for 45 degr. steps; incl. correction as in 0,
   3: as 2, without correction for non square pixels.

2. Writes (the first 1000) Freeman codes of the contour into Ibuf.

3. Writes Freeman codes of contour into file in ASCII format. The optional format specifier is a string and has the form: `n%#dxxx`, where:
   `n` a decimal number that specifies the number of codes on a line
   `%#d` decimal number occupying # positions
   `xxx` commas, white space etc. for layout purposes

   If no format specifier is given, 50 freeman codes are printed on a line without spaces, etc.
   The default extension is: **.cnt**

4. Specifies file properties for subsequent **fcont** operations (select one or more specifiers):
   ovw overwrite the content of the file (present information will be lost; default)
   app append information to file
   excl omit the starting address (default)
   incl add the address of the starting pixel to the file

5. Order of preference for searching directions (Freeman codes) Default: 0, 7, 1, 6, 2, 5, 3, 4

Examples

`fcont p 1`	searches for contour in bitplane 1 of p, returns length along contour
`fcont pc 1`	follows contour in p, starting at the cursor position (!).
`fcont 8 fcode "3%2d,"`	searches for contour in bitplane 8 of the default source; stores the Freeman string in file **fcode.cnt**. Example layout: `1, 2, 3` (3 codes on a line; 2 pos./code; comma separator)
`fcont 1 111 222 ib`	starts following a contour in bitplane 1 of the default source at Y=111, X=222; stores the Freeman chain codes in Ibuf.
`fcont app`	make the type of file writing: appending
`fcont 0 2 6 4 0 0 0 0`	force **fcont** to follow 4-connected

fftb

Command syntax:
`fftb f`

Family
FFT operation

Function
Transforms a Fourier transformed image back to space domain (in complex floating point).

Description . . .

Description

This operation transforms Fourier transformed data back to the spatial domain. After transformation the image may be displayed using fftd. This transformation is performed in place.

The specified image must be a complex floating point image.

Examples

`fftb f`		perform the conversion

fftd

Command syntax:
`fftd f [#1 [#2]]`

Family
FFT operation

Function
Display complex floating point image.

Description . . .

Description

This function displays a complex floating point image.

Display of the floating point image can serve two goals: to visualize the Fourier transformed image, or to display an image, that has been operated on in the frequency domain, and has been transformed back to space domain.

The latter requires no further modification: the (floating point) values in memory represent exactly the pixel values. To visualize the frequency spectrum, however, it is necessary to compress the contrast, since the dynamic range of the values in the Fourier domain is larger than 1:256.

#1 specifies which part of the complex data will be displayed:

- 1: display real part of floating point image
- 2: display imaginary part of floating point image
- 4: display modulus of complex fp. image (default).Use this mode to display an fft-image.

When displaying a frequency spectrum in modulus mode (#1 = 4), you can specify parameter #2 (1 - 255) to display

```
#2*log(modulus).
```

Examples

`fftd`	display modulus of complex FP image in default display; no correction.
`fftd f 4 128`	display modulus of complex FP image in **f**, perform correction.
`fftd f p 1`	display real part of complex FP image **f** into **p**

fftm

Command syntax:
```
fftm f [p]
```

Family
FFT operation

Function
Multiplies the Fourier transformed image with a standard (8-bits) mask image that represents the frequencies in the Fourier image that must be kept.

Description . . .

Description

This image has to be symmetric to the center.

The mask image can be obtained by observing the Fourier transformd image using the fftd operation.

The result will be the filtering of the selected frequencies in the Fourier image.

The FFT image must be a complex floating point image, the mask image is a standard 8-bits grey value image.

Examples

`fftm f r`	multiply using the content of image r
`fftm f`	multiply using the content of the default source

ffto

Command syntax:
`ffto f`

Family
FFT operation

Function
The complex floating point image is transformed to Fourier domain.

Description . . .

Description

This operation transforms spatial image data, which must be in complex floating point format (as produced by fftr) to the frequency domain. In this format it may be displayed using fftd or operated on using fftm.
The image must be a complex floating point image.

Examples

`ffto f`	no variations possible

fftr

Command syntax:
`fftr [a] f`

Family
FFT operation

Function
The content of a grey value image (a) will be transformed to complex floating point, to prepare it for conversion into the frequency domain.

Description . . .

Description

This function converts the specified image from pixel format to complex floating point.
This operation is a necessary intermediate step to transform the image to the frequency domain with the ffto command.

In this operation a destination image which has the floating point pixel type must be specified.

Examples

`fftr f`	the image in the default source is reduced and converted to complex floating point.
`fftr q f`	as above; q is the source image.

filt, filta, filtp

Command syntax:

```
1. filt [a] [#]
2. filt [a] [#] <file>
```

Parameter
- gain (1 - 8; default: 1)

Return value
number of overflows

Family
Neighbourhood operation

Function
Performs a convolution operation, using a user specified coefficient array (convolution kernel).

Description . . .

Description

This function performs a user definable convolution operation. The convolution kernel can be created using the **Filter** function in the Edit menu, or read from a disk file.

The **filt** command is available in the following formats:

filt	calcutates the result according to the coefficients
filta	adds 128 to the resulting pixel value.
filtp	adds the original pixel value to the result. # = gain (2**(#-1), default: 1)

Examples

`filt a`	performs a convolution on the image in a, using the coefficients in ibuf
`filta a cfile.flt`	as above, but the coefficients from cfile.flt are used and 128 is added to the result
`filtp 4`	performs a convolution using the coefficients in ibuf. The result is multiplied by 8 (= 2**(4-1))

frmt

Command syntax:

```
1. frmt [a] [#Y #X]
2. frmt [a] [#XY]
```

Return value

(previous) sub image format (in packed format)

Family

Control operation

Function

Sets/reads the sub-image format (horizontal & vertical)

Description . . .

Description

TIMWIN has several standard image sizes; each of them has a sub image format which is common for all images of the same size class. The **frmt** operations control the size of the sub images of the class involved.

See also: sub images

Examples

`frmt`	ask for the current sub image format
`frmt 44 55`	set the sub image format of the current image class
`frmt 80008H`	as above, using a packed parameter (8x8)
`frmt p 44 55`	set the sub image format of the image class, to which **p** belongs (256x256)
`frmt x 44 55`	set the sub image format of the image class, to which **x** belongs (512x512)

frmtx

Command syntax:
`frmtx [a] [#]`

Return value
horizontal sub-image size

Family
Control operation

Function
Sets/reads the horizontal sub-image format.

Description . . .

Description

TIMWIN has several standard image sizes; each of them has a sub image size which is common for all images of the same type. The **frmtx** operation controls the horizontal size of the sub images of the class involved.

See also: sub images, frmt

Examples

`frmtx`	ask for the current horizontal sub image size
`frmtx 55`	set the horizontal sub image format of the current image class
`frmtx p 44`	set the horizontal sub image format of the image class, to which **p** belongs (256x256)
`frmtx x 44`	set the horizontal sub image format of the image class, to which **x** belongs (512x512)

frmty

Command syntax:
`frmty [a] [#]`

Return value
vertical sub-image size

Family
Control operation

Function
Sets/reads the vertical sub-image format

Description . . .

Description

TIMWIN has several standard image sizes; each of them has a sub image size which is common for all images of the same type. The **frmty** operation controls the vertical size of the sub images of the class involved.

See also: sub images, frmt

Examples

`frmty`	ask for the current vertical sub image size
`frmty 55`	set the vertical sub image format of the current image class
`frmty p 44`	set the vertical sub image format of the image class, to which **p** belongs (256x256)
`frmty x 44`	set the vertical sub image format of the image class, to which **x** belongs (512x512)

fscan

Command syntax:
`fscan <file>`

Return value
number of values read

Family
Miscellaneous operation

Function
Reads an ASCII file into lbuf.

Description . . .

Description

This command reads the content of an ASCII file, which is supposed to contain integer values. The values are read and stored into ibuf in the long integers format (32 bits). 256 such values can be stored in Ibuf.

The values in the file may be separated by any non-numeric delimiter: space, comma, newline, etc. Non numericals (a string that does not begin with a digit) are ignored

The return value allows you to check the successful interpretation of the file's content.

Examples

`fscan numbers` reads file 'numbers'; stores content into Ibuf

Comment

This command differs from ribuf in that **ribuf** reads a binary file, which is an exact image of Ibuf's content, whereas **fscan** reads a free-format ASCII file, which may contain up to 256 items.

gaus

Command syntax:
```
gaus a [#w [#g]]
```

Parameters
#w - window size (3 (default) - 5 - 7 - 9)
#g - gain (default: 1)

Return value
overflow pixels

Family
Neighbourghood operation

Function
Convolution filter, type Gaussian blur.

Description . . .

Description

The Gaussian filter is a blur operation. It uses a coefficient scheme whose values approximate to a Gaussian curve, so that the contribution to the mean decreases when the distance to the central pixel increases.

With the Gaussian filter a less dramatic effect is produced than just taking the mean value of all pixels in the window, as performed by the unif operation. The manual appendix C lists the coefficient arrays used with convolutions.

Comment

A faster version of this operation using separate horizontal and vertical smoothing is: dgaus.

See also: dgaus, unif

getim

Command syntax:
`getim #1 [#2]`

Return value
name of requested image

Function
Returns the name of an image with the requested properties (if any).

~~Description . . .~~

Description . . .

This operation provides you with the name of the image, that has the combinations of parameters that you request. You can use this name in subsequent operations. If the requested combination does not exist, an error is generated, that can be handled using the `on error ...` procedure.

This operation is especially useful in command files running on systems with an unknown image set-up, for example TIMWIN's own `*demo1` set. It allows you to get a valid and consistent image name, regardless the names chosen by the user of the system.

The requested properties must be defined using bit patterns, which must be combined using the OR operator. To ease the use of this operation, aliases have been defined for the properties.

The following table shows the properties, their bit values and their alias.

Image property	alias	bit pattern (hex)
memory image	MEM_BIT	2
windows image	WIN_BIT	4
frame grabber image	DIS_BIT	0x10
mask for pixel properties field	PIXMASK	0xE0
8-bit pixels	PIX8	0
12-bit pixels	PIX12	0x40
16-bit pixels	PIX16	0x20
32-bit pixels	PIX32	0x60
64-bit pixels	PIX64	0x80
256 size class	I2	0x100
384 size class	I3	0x200
512 size class	I5	0x400
768 size class	I7	0x800
1024 size class	IK	0x1000
sub-image	SUBIM	0x2000
overlay image (read)	OV_READ	0x4000
overlay image (write)	OV_WRITE	0x8000

In the property specifier, all subfields **must** be specified, with the exception of the sub-image and overlay fields.

Examples

```
char imname[10]
IMAGE MyIm
 . . .
fprint imname 0 "%s",  getim (DIS_BIT + PIX8 + I2)
MyIm = imname

copy miss MyIm
```

grad

Command syntax:
```
grad a [#F]
```

Parameter
#F = direction of gradient (Freeman chain code: 0 - 7; def. = 1).

Family
Neighbourghood operation

Function
Produces the gradient (1e derivative) of an image.

Description . . .

Description

Produces the gradient (1e derivative) of an image. The gradient is determined by subtracting from a pixel the value of one of its eight neighbours and adding 128 to the result to simulate a signed value. The neighbour is specified by a Freeman chain code.

Examples

`grad a 6 >b`	determines the gradient of **a** in 'south' direction, copies result to **b**

graf

Command syntax:
```
graf [a] [#]
```

Parameters
= scaling factor. Default: autoscaling

Return value
scaling factor

Family
Graphic operation

Function
This function plots the data in Ibuf in the specified image.

Description . . .

Description

This command write a line-plot of Ibuf's content into an image. It assumes up to 256 values in Ibuf (any type). These values are scaled by repeated division (or multiplication) by 2, so that the maximum value will fit into the specified (sub-) image. Horizontally the available values are sampled so, that the resulting graph corresponds to the specified image.
Graf draws a line plot, using the drawing value.

Examples

`graf`	draws a graph into the default destination
`graf pc 3`	as above; writes into the sub-image, divides data by 8 (=2**3)

grav

Command syntax:
```
grav [a] [#]
```

Parameters
= scaling factor. Default: autoscaling

Return value
scaling factor

Family
Graphic operation

Function
This functions plots the data in Ibuf in the specified image.

Description . . .

Description

This command write a vector-plot of Ibuf's content. It assumes up to 256 values in Ibuf (any type). These values are scaled by repeated division (or multiplication) by 2, so that the maximum value will fit into the specified (sub-) image. Horizontally the available values are sampled so, that the resulting graph corresponds to the specified image.
Grav draws a bar plot (consisting of vertical bars), using the graphics value.

Examples

`grav`	draws a graph into the default destination
`grav pc 3`	as above; writes into the sub-image, divides data by 8

hist

Command syntax:
```
hist [a] [#1 [#2]]
```

Parameters
#1 - lower boundary of interesting values (0 - 255)
#1 - upper boundary of interesting values (0 - 255)

Return value
None

Family
Miscellaneous operation

Function
Collects the pixel values of the image into a histogram array.

Description . . .

Description

The histogram operation builds the histogram of the specified image in Ibuf: each Ibuf entry will contain the number of pixels in the image whose grey value correspond to that entry.

If the graphic window (See the View menu) is active, the histogram will appear there. To place the content of Ibuf as a graph into an image, use graf and grav.

To exclude ranges of values from this operation use numerical parameters (default is: no exclusion):

- #1 excludes grey values 0 - (#1)
- #2 excludes grey values (#2) - 255

Exclusion is useful when only a grey value region of the image is of interest, for example to suppress extreme values (0 and 255) after an operation that produces overflow. or to suppress large numbers of background pixels (0), that would cause a graph to be unattractively scaled.

Examples

`hist a`	Collects histogram of **a**
`hist a 0`	Collects histogram in the range 1 to 255 (suppresses 0)
`hist a 5 250`	Collects histogram in the range 6 to 249 (suppresses 0 - 5 and 250 - 255)

ibuf

Command syntax:

```
1. ibuf #
2. ibuf #1 #2
3. ibuf er|cb|cw|cl|by|wo|lo
```

Return Value

1. Ibuf value
2. Previous Ibuf value
3. None

Family

Miscellaneous operation

Function

Reads/writes Ibuf values

Description . . .

Description

IBUF is the general exchange buffer, used by many commands. This function allows you to access individual data elements, or to change Ibufs data type or content.

`1. ibuf #`
Returns the content of location '#' of 'ibuf.

`2. ibuf #1 #2`
Writes #2 to location #1 of 'ibuf. Returns the original value.

`3. ibuf er|cb|cw|cl|by|wo|lo`
Changes Ibufs data type, or erases Ibuf, or converts data or display. Only the first two characters of the string are relevant:

er	erase Ibuf
cb	convert to byte (8 bits units)
cw	convert to word (16 bits uinits)
cl	convert to long words (32 bits units)
by	display as bytes (regardles of current content)
wo	display as words (regardles of current content)
lo	display as long words (regardles of current content)

The difference between *convert* and *display* is that convert takes the original values and changes them to the new format , thereby changing the content of Ibuf (e.g. changing bytes into words), whereas display only displays the content of ibuf in another mode.

To work with Ibuf interactively, use the Ibuf dialog box in the Edit menu.

Examples

`ibuf 10`	reads ibuf[10]
`ibuf 10 255`	sets ibuf[10] to 255
`ibuf er`	erase ibuf
`ibuf cb`	convert data in ibuf into bytes

ihis

Command syntax:
```
ihis a #1 [#2]
```

Parameters
#1 = number of image line (default: 0)
#2 = scalling factor (default: autoscaling)

Return value
none

Family
Miscellaneous operation

Function
Writes data in Ibuf into line #1 of image **a**.

Description . . .

Description

This operation makes the content of Ibuf visible and offers the opportunity to collect and compare data from successive operations. Since image lines can only contain bytes, scaling can be necessary. If you specify a scaling value, that it too low to bring all Ibuf values in the range 0 - 255, the values above 255 will be clipped to 255.

Examples

`ihis s 11`	the content of Ibuf is auto- scaled and copied to line 11 of s.
`ihis 0 0`	the content of Ibuf is not scaled and copied to line 0 of the default source.

improp

Command syntax:
`improp [p]`

Return value
pattern read

Function
Returns an integer value representing various image properties

~~Description . . .~~

Description

This operation returns a combination of bit values, that represent properties of the specified image. To find out if a given property is available in an image, you can perform an AND operation on `improp`'s return value and the bit pattern representing that property.

This operation is especially useful in command files running on systems with an unknown image set-up, for example TIMWIN's own `*demo1` set. It allows you to test if a given configuration is available. To ease the use of this operation, aliases have been defined for the properties.

The following table shows the properties, their bit values and their alias.

Image property	alias	bit pattern (hex)
memory image	MEM_BIT	2
windows image	WIN_BIT	4
frame grabber image	DIS_BIT	0x10
mask for pixel properties field	PIXMASK	0xE0
8-bit pixels	PIX8	0
12-bit pixels	PIX12	0x40
16-bit pixels	PIX16	0x20
32-bit pixels	PIX32	0x60
64-bit pixels	PIX64	0x80
256 size class	I2	0x100
384 size class	I3	0x200
512 size class	I5	0x400
768 size class	I7	0x800
1024 size class	IK	0x1000
sub-image	SUBIM	0x2000
overlay image (read)	OV_READ	0x4000
overlay image (write)	OV_WRITE	0x8000

Notice that, in the pixel property field, combinations of bits can occur. This requires a special test procedure, as the examples show.

Examples

(The examples are in command file syntax, because in interactive mode this command has little use)

```
int property
  . . .
property = improp f
              ;test if 64-bits image is present:
if property & PIX64
              ;test procedure using mask:
if (property & PIXMASK) == PIX8
              ;immediate use without variable:
if (improp p) & DIS_BIT
```

in

Command syntax:

```
in [#1] [#2]
```

Return value

pattern read

Function

Reads data from the upper 5 bits of the parallel (printer) port.

~~Description . . .~~

Description

Waits, until the input matches the mask, specified in #1, or #2 video frames (each 1/25th of a second) have expired, whichever is first.

#1 = 0F8(hex) (default): any value at the port returns
#2 = 0 (default): waiting period infinitely. Waiting may be interrupted by pressing 'ESC'.

Pins of the printer port:	11	10	12	13	15	X	X	X
Bits read:	7	6	5	4	3	2	1	0

Notes

- the lower 3 bits carry no information.
- this command is not implemented in TIMWIN (Windows version)

incln

Command syntax:
```
1. incln [a] #YX1 [#YX2]
2. incln [a] #Y1 #X1 #Y2 #X2
```

Parameters
See line drawing

Family
Graphic operation

Return value
number of pixels on line

Function
Increments the pixel values along imaginary line, specified by the numeric parameters.

Description . . .

Description

This command increments the pixel values laying on an imaginary line determined by the numeric parameters.
For details on line drawing and parameter interpretation, see line drawing
See also: graphics concepts

Examples

`incln 110022h`	increments pixels positioned on line between 11H, 22H (Y,X) and cursor
`incln p 11 22 33 44`	increments pixels positioned in p on line between 11,22 and 33,44 (Y,X)

incpat

Command syntax:
```
incpat [a] [#YX]
incpat [a] #Y #X
```

Parameters
See pattern drawing

Return value
none

Family
Graphic operation

Function
Increments the pixel values along a path, specified by a Freeman chain.

Description . . .

Description

This command increments the pixel values along a path, which is specified by a Freeman chain.
If performed immediately after the fcont operation, the exception parameter **'/'** can be specified. Then the Freeman codes are read from the internal buffer, where **fcont** stores them, and the default starting position is that of the original image. Otherwise:starting position is the cursor position.

For details on pattern drawing and parameter interpretation, see pattern drawing
See also: graphic concepts.

Examples

`incpat`	increments a figure, specified by a Freeman string in Ibuf, from the cursor position
`incpat /`	as above, but reads Freeman string from **fcont** buffer and starts at original position
`incpat a 100 200`	increments a, reads from Ibuf, starts at 100 (Y), 200 (X)
`incpat a 100 200 /`	as above, but reads Freeman string from **fcont** buffer

incvec

Command syntax
`incvec [a] #a [#l [#Y #X]]`

Parameters
See vector drawing

Family
Graphic operation

Return value
number of pixels on vector

Function
Increments the pixel values along the imaginary vector, specified by the numeric parameters.

Description . . .

Description

This command increments the pixel values laying on an imaginary vector, determined by the numeric parameters.
See also: Concepts of graphic operations

Examples

`incvec 222`	increments pixels on vector, running from the cursor position to the image edge with an angle of 222 degrees.
`incvec a 33 10`	increments pixels on vector in image a; angle 33 degr., length 10 pixels
`incvec 10 0 128 128`	increments pixels on vector running from 128, 128 to the image edge, angle 10

inv

Command syntax
`inv [a] [/]`

Return value
none

Family
Pixel operation

Function
Inverts the pixels of **a** bitwise

Description . . .

Description

The image is inverted logically. The inversion is performed by table look up; the result is that all bits of each pixel are logically inverted.

If **'/'** is specified, only the look up table is computed.
See also: table operations and preview.

Examples

`inv`	inverts the default source
`inv a >a`	inverts **a**, copies result to **a**
`inv /`	produce an inverting table only

keep

Command syntax
`keep [a] [#1 [#2 [#3 ....]]]`

Parameters
- bitplanes to keep. Default: keep nothing (erase all).

Return value
none

Family
Miscellaneous operation

Function
Erases in an image all bitplanes, except those specified.

Description . . .

Description

This command erases an image, and is as such comparable to **era**. With **keep** however, you can specify bitplanes that have to be protected from erasure. This is in contrast to **era**, where you specify the bitplanes that must be erased.

See also: era

Examples

`keep`	erases entire image
`keep pc 1 2 3`	keeps bitplanes 1, 2 and 3 of sub-image of **p**
`keep 1 2 3`	keep bitplanes 1, 2 and 3 (erase 4, 5, 6, 7 and 8)
`era 1 2 3`	erase bitplanes 1, 2 and 3 (keep 4, 5, 6, 7 and 8)

kuwa

Command syntax:
`kuwa [a] [#w]`

Return value
none

Parameters
#w - window size: 3 (default) - 5 - 7 - 9

Family
Neighbourghood operation

Function
Performs a kuwahara filtering: produces a smoothed image, but preserves sharp edges

Description . . .

Description

The Kuwahara filter is a non-linear window filter. It suppresses noise as the uniform filter does, but sharp edges are not affected. The effect is, more or less (depending upon window size), that the image is divided into relatively smooth clusters. As such, this operation offers a useful preprocessing step for detecting contours.

Examples

`kuwa a 9`	performs the kuwahara filter to image a, window size 9x9.
`kuwa`	performs the kuwahara filter to the default source, window size 3x3

label

Command syntax
`label [a] [#]`

Return value
Number of objects found. If negative: more than 255 objects in the image (absolute value represents number of labeled objects in this pass).

Parameter
- Threshold value. If negative: image is inverted. Default: 1

Family
Parameter operation

Function
Labeling of objects.

Description . . .

Description

Labels the objects in image **a** 8-connected
Expects '0' as background value between the objects, unless '#' is specified. In that case the pixels are thresholded using value '#. A negative value inverts the result.

The objects are numbered in order of encounter by assigning them increasing grey values. If more than 255 labels have to be assigned the remainder of the objects are labeled '0'. If this happens the return parameter is negative, but the absolute value represents the number of objects actually labeled. Objects thus set apart can be labeled in a second pass. See the description of command file **label&** in Appendix H of the manual.

See also
mark, label4

Examples

`label`	the (thresholded) image in the default source is labeled
`label a -128`	the image in a is thresholded by the value of 128, inverted and the result is labeled

label4

Command syntax
`label4 [a] [#]`

Parameter
- Threshold value. If negative: image is inverted. Default: 1

Return value
number of objects found

Function
4-connected labeling. Pixels are supposed connected only if they are 4-connected.

Description . . .

Description

Labels the objects in image **a** 4-connected
Expects '0' as background value between the objects, unless '#' is specified. In that case the pixels are thresholded using value '#. A negative value inverts the result.

The objects are numbered in order of encounter by assigning them increasing grey values. If more than 255 labels have to be assigned the remainder of the objects are labeled '0'. If this happens the return parameter is negative, but the absolute value represents the number of objects actually labeled. Objects thus set apart can be labeled in a second pass. See the description of command file **label&** in Appendix H of the manual.

See also
mark, label

Examples

`label4`	the (thresholded) image in the default source is labeled
`label4 a -128`	the image in a is thresholded by the value of 128, inverted and the result is labeled (4-connected)

lapl

Command syntax:
`lapl a [#w] [#g]`

Return value
number of overflows

Parameters
#w - window size: 3 (default), 5, 7 or 9
#g - gain

Family
Neighbourghood operation

Function
Produces the 2nd derivative ("laplacian") of an image

Description . . .

Description

This convolution operation produces the "Laplacian" (= 2nd derivative) of the image.

To be able to represent negative values, 128 is added to the result. Thus:

positive values:	>128
0:	128
negative:	<128

See also: qlap

Examples

`lapl a`	3x3 laplace filter applied to a
`lapl a 9 2`	performs a 9x9 laplace filter, multiplies the result by 4 (2**2)

Icon

Command syntax:
`lcon [a] #b`

Parameters
#b - bitplane number (1 - 8; default: 1)

Return value
Number of changed pixels

Family
Cellular logic operation

Function
Produces the 4-connected contour in bitplane # (removes non-border pixels).

Description . . .

Description

The contour operation removes all but the pixels that are at the borders of the objects, leaving the contour pixels of the objects. The remaining contour is 4-connected.

For more details regarding Cellular Logic Operations see CLP.

Examples

`lcon`	leaves the contour of the default source, bitplane 1
`lcon p green`	as above, image is p, bitplane is green (2).

lcset

Command syntax:
`lcset <file>`

Return value
none

Family
Control operation

Function
Loads character set <file>.fnt (the extension .fnt is appended by default)

Description . . .

Description

This operation installs a character set, to be used with the text and textv commands. These character sets allow you to write text into images.

There are several character sets available, offering several sizes and shapes, as well as proportional and non-proportional fonts. The following fonts belong to the standard set:

Name	Font type	Size (pixels)
chic09.fnt	chicago	9
cour11.fnt	courier	11
mona09.fnt	monaco	9
time15.fnt	times	15
time15p.fnt	times (prop)	15
upper.fnt	upper case	6

In addition, the STANDARD.FNT is a copy of one of the other fonts. It is installed in some command files (e.g. **init**) as a default. This allows you to have your favourite font loaded automatically by just copying it to STANDARD.FNT.

Examples

`lcset standard` loads character font file standard.fnt

ldi

Command syntax:

```
ldi [a] #b [#n]
ldi4 [a] #b [#n]
ldi6 [a] #b #n
ldi8 [a] #b [#n]
```

Parameters
#b - Bitplane (1 - 8; default 1)
#n - repeating factor (0 = infinity). Repeating stops when no further changes occur. Default: 1.

Return value
Number of changed pixels

Family
Cellular logic operation

Function
Dilates objects in bitplane #1 (adds a layer of pixels to the borders).

Description . . .

Description

Dilation grows a layer of pixels around the objects. The decision whether to add a pixel to an object depends upon connectivity.

This operation can be performed 4 and 8-connected, and alternating. In the latter case the result is a simulated 6-connectivity.

For more details regarding Cellular Logic Operations see CLP.

Examples

`ldi`	dilates objects in bitplane 1 of the default source
`ldi4 p 3 2`	dilates (4 connected) the objects in the 3rd bitplane of **p**, 2 times.

ler

Command syntax:
```
ler  [a] #b [#n]
ler4 [a] #b [#n]
ler6 [a] #b #n
ler8 [a] #b [#n]
```

Parameters
#b - Bitplane (1 - 8; default 1)
#n - repeating factor (0 = infinity). Repeating stops when no further changes occur. Default: 1.

Return value
Number of changed pixels

Family
Cellular logic operation

Function
Erodes objects in bitplane #1 (removes a layer of pixels).

Description . . .

Description

Erosion is the opposite of dilation. Pixels laying at the border of an oject will be removed with this operation. This operation can be performed 4 and 8-connected, and alternating. In the latter case the result is a simulated 6-connectivity.

For more details regarding Cellular Logic Operations see CLP.

Examples

`ler`	erodes one layer of pixels from the objects in bitplane 1 of the default source
`ler8 p 3 4`	erodes 4 layers of pixels from the objects in bitplane 3 of p.

lenp

Command syntax:
`lenp [a] #`

Parameter
- Bitplane (1 - 8; default 1)

Return value
Number of changed pixels

Family
Cellular logic operation

Function
Removes all but the end pixels in bitplane #.

Description . . .

Description

End pixels are pixels that have only one (1) neighbour. This operation is used typically after the **lsk** (skeleton) operation; **lsk** reduces objects to lines. Performing the **lenp** operation after a skeleton operation, gives the end pixels of those line figures which may tell us something about the character of the object.

This is a Cellular Logic Operation (CLP)

Examples

`lenp`	keeps end pixels of objects in bitplane 1 of the default source
`lenp p 3`	keeps end pixels of the objects in bitplane 3 of p.

life

Command syntax:
`life [a] #b [#n]`

Parameter
#b - Bitplane (1 - 8; default 1)
#n = repeating factor (0 = infinity; default). Repeating stops when no further changes occur.

Return value
Number of changed pixels

Family
Cellular logic operation

Function
Game of life

Description . . .

Description

Generates or removes pixels in bitplane #1 according to the rules:

- A pixel is generated if there are three neighbours.
- It is removed if it has less than two or more than three neigbours,

For fun only. For more details regarding Cellular Logic Operations see CLP.

Examples

`life 1`	perform life in bitplane 1 of the default source
`life q 3 55`	perform life in bitplane 3 of q 55 times

Comment

This function runs until no pixels change anymore, which may never happen. In that case, stop the operation by pressing the ESC key.

line

Command syntax:
```
line [a] [#]
```

Parameter
- line number (default: vertical part of the image cursor position)

Return value
None

Family
Miscellaneous operation

Function
Reads the pixels of the specified image line into Ibuf.

Description . . .

Description

Collects gray values found along line # , and stores the data in Ibuf.

Examples

`line`	copies the image line at the cursor position from the default source to ibuf.
`line a 44`	copies image line 44 of a to ibuf.

link

Command syntax:
`link [a] #b`

Parameter
#b - bitplane (1 - 8; default 1)

Return value
Number of changed pixels

Family
Cellular logic operation

Function
Removes all but the link pixels in bitplane #.

Description . . .

Description

Link pixels are pixels that have two (2) neighbours. This operation is typically used after the **lsk** (skeleton) operation; **lsk** reduces objects to lines. Performing the **link** operation after a skeleton operation keeps only those pixels which are part of single lines; branch pixels and end pixels are removed.

For more details regarding Cellular Logic Operations see CLP.

Examples

`link 1`	remove all but the link pixels in bitplane 1 of the default source
`link p 3`	keep link pixels in bitplane 3 of p.

lmaj

Command syntax:
`lmaj [a] #b`

Parameter
#b - Bitplane (1 - 8; default 1)

Return value
Number of changed pixels

Family
Cellular logic operation

Function
"Majory vote" in bitplane #:

Description . . .

Description

This operation will set the center pixel in a neighbourhood equal to the majority in the window.
If a pixel is 1, and fewer than 4 of its neighbours are 1, it will be set to 0. If it is 0, and more than 4 of its neighbours are 1, it will be set to 1. Thus this operation will remove binary noise.

For more details regarding Cellular Logic Operations see CLP.

Examples

`lmaj 3`	performs majority vote on bitplane 3 of the default source
`lmaj q 2`	performs majority vote on bitplane 2 of q.

log

Command syntax:
`log [a] [/]`

Return value
none

Family
Pixel operation

Function
Logarithmic conversion

Description . . .

Description

Converts pixel value 'p' to:

```
C*log (p+1),
```

where 'C' is a constant that scales the result into the range 1 - 255.

If **'/'** is specified, only the look up table is computed.
See also: table operations and preview

Examples

`log p`	Perform a log function to image **p**
`log /`	produce a log table in Ibuf

lpr

Command syntax:

```
lpr  [a] #b1 #b2 [#n]
lpr4 [a] #b1 #b2 [#n]
lpr6 [a] #b1 #b2 #n
lpr8 [a] #b1 #b2 [#n]
```

Parameters
#b1 - bitplane to be propagated (1 - 8)
#b2 - mask bitplane (1 - 8)
#n - repetition factor (0 = infinity; default)

Return value
Number of changed pixels

Family
Cellular logic operation

Function
Propagation of seeds in bitplane #b1, masked by bitplane #b2, repeated #n times

Description . . .

Description

Propagation is dilation (growing of objects), controlled by a mask. Dilation takes place within boundaries, set by second bitplane.
This operation can be performed 4 and 8-connected, and alternating. In the latter case the result is a simulated 6-connectivity.
If the repetition factor is infinity (propagating until no further changes occur in the image), propagating takes place in a 'brute force'way, which offers the greatest speed, but does not keep topology.

For more details regarding Cellular Logic Operations see CLP.

Examples

`lpr 1 2`	propagates seeds in bitplane 1 (mask bitplane 2) in the default source
`lpr8 s 4 5 33`	propagates seeds in bitplane 4 of **s** (mask bitplane 5) 33 times.

lps

Command syntax:
`lps [a] #b`

Parameter
#b - bitplane (1 - 8; default: 1)

Return value
Number of changed pixels

Family
Cellular logic operation

Function
Removes 'pepper & salt' noise in bitplane #:

Description . . .

Description

Pepper & salt noise are single '1' and '0' pixels in an opposite neighbourhood. In this operation they are replaced by their neighbourhood.

For more details regarding Cellular Logic Operations see CLP.

Examples

`lps`	removes p&s in bitplane 1 of the default source
`lps r 3`	removes p&s in bitplane 3 of r

lsk

Command syntax:
```
lsk [a] #b [#n]
```

Parameters
#b - bitplane (1 - 8; default: 1)
#n - number of iterations (default: infinity).

Return value
Number of changed pixels

Family
Cellular logic operation

Function
Produce the skeleton while keeping end pixels

Description . . .

lskz

Command syntax:

```
lskz [a] #b [#n]
```

Parameters

#b - bitplane (1 - 8; default: 1)
#n - number of iterations (default: infinity).

Return value

Number of changed pixels

Family

Cellular logic operation

Function

Produce the skeleton while removing end pixels

Description . . .

Description

Bitplane #1 is skeletonized (eroded, under the condition that object topology is kept). This means that objects are thinned to single lines.
In addition, **lskz** keeps removing end pixels, until a single pixel or a closed contour remains.

For more details regarding Cellular Logic Operations see CLP.

Examples

`lsk 1`	skeleton of objects in bitplane 1 of the default source, end pixels will be kept.
`lskz 1`	as above, end pixels will also be removed.
`lsk p 2 10`	skeleton of objects in bitplane 2 of p; 10 iterations are performed, end pixels are kept.
`lska 1 2`	skeleton of objects in bitplane 1 with anchor in bitplane 2.

lska

Command syntax:
`lska [a] #b1 #b2 [#n]`

Parameters
#b1 - bitplane (1 - 8)
#b2 - anchor bitplane (1 - 8)
#n - number of iterations (default: 0 = infinity).

Return value
Number of changed pixels

Family
Cellular logic operation

Function
Skeletonizing of image in bitplane #b1 using an 'anchor' in bitplane #b2

Description . . .

lskza

Command syntax:
`lskza [a] #b1 #b2 [#n]`

Parameters
#b1 - bitplane (1 - 8)
#b2 - anchor bitplane (1 - 8)
#n - number of iterations (default: 0 = infinity).

Return value
Number of changed pixels

Family
Cellular logic operation

Function
Skeletonizing of image in bitplane #b1 using an 'anchor' in bitplane #b2

Description . . .

Description

Bitplane #b1 is skeletonized, except where pixels in bitplane #b2 are present. The skeleton is "achored" to the object in bitplane #b2.

lskza also keeps removing end pixels (with the exception of pixels masked by pixels in bitplane #2) until a single pixel or a closed contour remains

Note that this operation has no default bitplanes.

See also: lsk
For more details regarding Cellular Logic Operations see CLP.

Examples

`lska 1 2`	make the skeleton of objects in bitplane 1 with an anchor in bitplane 2 of the default source, end pixels will be kept.
`lskza 1 2`	as above, end pixels will also be removed.

lsp

Command syntax:
`lsp [a] #b`

Parameters
#b - bitplane (1 - 8; default: 1)

Return value
Number of changed pixels

Family
Cellular logic operation

Function
Keeps single pixels in bitplane #.

Description . . .

Description

Pixels completely surrounded by background pixels are kept by this operation, all others are removed.
For more details regarding Cellular Logic Operations see CLP.

Examples

`lsp 8`	keeps single pixels in bitplane 8 of the default source.
`lsp q`	keeps single pixels in bitplane 1 of q

lut

Command syntax:
`lut [#1] #2 [#3 [#4] ... ]]`

Family
Control operation

Function
Controls the frame grabber's and display window's look-up tables

Description . . .

Description

#1 - type table:
- 1 - Frame grabber Input,
- 2 - Frame grabber Output
- 3 - Windows image

#2 - selection of table: 1 to 4, 8 or 16 depending on frame grabber. This parameter has no meaning with Windows images (where #1 = 3)

#3 - Function for Input LUT, Output LUT and Windows display)

#	Input/Output/Windows	Function
1	(I/O/W)	linear table ...
2	(I/O/W)	inverse table ...
3	(I/O/W)	logarithmic table ...
4	(I/O/W)	load content of Ibuf...
5, 6	(O/W)	pseudo colours ...
7, 8, 9	(O/W)	shows bitplanes in colours ...
10	(O/W)	real colours ...
14	(O/W)	spectrum psudo colours ...
15	(O/W)	sinus pseudo colours ...
105, 106	(O)	12-bit pseudo colours ...
110	(O)	12-bit real colours ...

lut (Black & White LUT functions)

These functions produce various tables for black and white display of images.

Command syntax:

`lut #1 #2 1 [#4]`	produce a linear table
`lut #1 #2 2 [#4]`	produce a inverse table
`lut #1 #2 3 [#4]`	produce a logarithmic table
`lut #1 #2 4 [#4]`	load the data which is currently in Ibuf

In addition you may select one of the colour channels, in which case only this channel is filled and the others keep their original content. This can be done using #4. Default is: load all tables

#4 - 1: load red table
#4 - 2: load green table
#4 - 3: load blue table

Examples:

`lut 2 1 3`	load a logarithmic table in frame grabber's output LUT no. 1
`lut 1 3 1`	load a linear table in frame grabber's input LUT no. 3
`lut 3 2`	load an inverse table for displaying windows images
`lut 2 8 1 1`	load a linear table in the red table of FG output LUT no. 8
`lut 2 8 2 2`	load an inverse table in the green table of FG output LUT no. 8

Pseudo colour LUT functions

These functions produce various tables for pseudo colour display of images. This is done by connecting three consecutive bitplanes to the three colour channels.

Command syntax:

`lut #1 #2 5 [#4]`	show the colours in full intensity
`lut #1 #2 6 [#4]`	show the colours in an intensity, determined by the underlaying grey value.

#4 - the bitplane where the red channel is connected to. Default: 6

Examples:

`lut 2 9 6`	show 'soft' colours in bitplane 6, 7 and 8 (default) of LUT 9
`lut 2 1 5 4`	show 'hard' colours in bitplane 4, 5, and 6 of LUT

Depending on the capabilities of the display system, it is possible to use either 8 or 12 bits

Bitplane colour LUT functions

These tables show a bitplane in one of the primary colours red, green or blue. If applied repeatedly for different colours in different bitplanes, you create display 'layers'. A top layer (i.e. a layer which is created last) will cover lower layers.

These tables are often used to show the effect of binary operations. See CLP-operations and bitplane operations

Command syntax:

`lut #1 #2 7 #4`	make bitplane #4 red
`lut #1 #2 8 #4`	make bitplane #4 green
`lut #1 #2 9 #4`	make bitplane #4 blue

Examples:
The following sequence is used in the standard LUT set-up, as used in `*ini`

`lut 2 4 7 1`	make bitplane 1 in output table 4 red
`lut 2 4 8 2`	make bitplane 2 in output table 4 green
`lut 2 4 9 3`	make bitplane 3 in output table 4 blue

Real Colour LUT functions

These tables allow you to display a real colour image. There are two versions:

For 8-bits images

Command syntax:
`lut #1 #2 10 #4 #5 #6`

The parameters #4, #5 and #6 specify the number of levels of colours for red, green and blue, respectively. Since the total amount of colours is 256, the product of these values must be below this value.

For 12-bits images

Command syntax:
`lut #1 #2 110 #4 #5 #6`

The parameters #4, #5 and #6 specify the number of levels of colours for red, green and blue, respectively. Since the total amount of colours is 4096, the product of these values must be below this value. Usually 16, 16, 16 is used.

Special Colour LUT function

The following special LUTs exist:

`lut #1 #2 14` spectrum table
`lut #1 #2 15 #4 #5 #6` sine table

The spectrum table maps the pixel values from 0 = blue via green to 255 = red, thus simulating a spectrum.

The sine table maps the pixel values to a sine, with the extra parameter #4 - #6 specifying a phase shift for red, green and blue, respectively.

Example:
`lut 2 10 15 0 120 240` 120 degrees shift between colours

12 Bits LUT functions

12 bits frame grabbers allow you to dedicate overlay colours to special bitplanes, while keeping the full 8 bits free for grey value display

`lut #1 #2 105`	hard colours in bitplanes 9, 10 and 11
`lut #1 #2 106`	soft colours in bitplanes 9, 10 and 11

In addition, bitplane 12 displays white

Since these tables use all of the LUT memory, specifying a LUT to fill (#2) doesn't make sense. In this case, this parameter is a placeholder to remain compatible with other LUT functions.

lver

Command syntax:
`lver [a] #`

Parameters
#b - bitplane (1 - 8; default: 1)

Return value
Number of changed pixels

Family
Cellular logic operation

Function
Keeps the vertex pixels in bitplane #.

Description . . .

Description

The vertex operation keeps pixels that have three or more neighbours. This operation is meant to analyze line figures, as produced by the lsk operations. lver removes all pixels but the branch pixels.

See also: lsp, link
For more details regarding Cellular Logic Operations see CLP.

Examples

`lver`	vertex operation in bitplane 1 of the default source
`lver p 4`	vertex operation in bitplane 4 of p

mand

Command syntax:
`mand [a] [#f1 #f2 #f3 #4]`

Function
This operation creates Mandelbrot fractal images in the specified image.

Parameters
#f1 real (X) coordinate of middle of the image (floating point value)
#f2 imaginary (Y) coordinate of middle of the image (floating point value)
#f3 size of square part (horizontal & vertical)
#4 iteration step size (1 - 128; default: 2). This value determines the number of iterations (256/stepsize), and thus the grey value or colour resolution.

Description . . .

Description

The Mandelbrot fractal generator produces images of fascinating shapes and colours. Their calculation is based upon an iteration process; the pixel's coordinates act as a seed for the calculations. The number of iterations necessary to get the desired result (convergence) is used as the actual grey value. The number of iterations is limited and specified by parameter #4 (256/#4).

In large parts of the Mandelbrot figure (shown black in the figure) the calculation does not converge; there the maximum computation time is spent. At the border of the figure, rapid changes in the result occur. Further away convergence occurs quickly. The small area between no convergence and fast convergence is the most interesting part, and there you can zoom in infinitely.

Examples

`lut 2 6 6 6`	select an appropriate LUT with nice colours
`mand`	default: zoomed into interesting area
`mand -0.5 0.0 2.0 2`	complete Mandelbrot set

Comment

Recommended literature:
Computer Recreations - A.K.Dewdney, Scientific American, August 1985
Geometrics of Nature - B.B.Mandelbrot, Wiley - New York
Fractals - Hans Lauwerier, Aramith Uitgevers, Amsterdam ISBN/NUGI 90 6834 031X/819 (in Dutch)

mark

Command syntax:
```
mark [a] [#1 [#2]] [/]
```

Parameters
#1 - grey value to search for (0 - 255; default: 1)
#2 - grey value to replace search value with (0 - 255; default: 255)
/ - search modifier

Return value
number of pixels

Family
Parameter operation

Function
Searches image for an object having a unique grey value '#1'. Replaces pixel value with: #2

Description . . .

Description

This function searches in the image for an object having a particular grey value. It is designed to be executed after the label operation, which assigns objects different grey values.

The object is considered found completely, when no more pixels of the desired value are encountered in an entire image line.

mark adjusts the image's sub image format so, that the object will fit completely in the sub image without touching the borders (a single pixel will result in a 3x3 sub image).
The cursor is positioned as close to the centre of the surrounding box as possible.

If **'/'** is specified, searching will start at the image line where a previous call to **mark** found an object. **mark** is meant to be used after label.

Examples

`mark 23`	searches for an object with grey value 23 in the default source, overwrites it with grey value 255.
`mark a 13 1`	searches in a for object with grey value 13, writes it into the default source with grey value 1.

max

Command syntax:
```
max [a] [#w]
max [a] #wy #wx
```

Parameters
#w - window (neighbourhood) size (3 - <image size>; default: 5)
#wy - vertical size of window (3 - <image size>)
#wx - horizontal size of window (3 - <image size>)

Return value
none

Family
Neighbourghood operation

Function
Replaces each pixel with the maximum pixel value found in a window of #Y*#X.

Description . . .

Description

In the **max** operation, each pixel is replaced by the maximum pixel value that is found in the specified window around it. This operation is also known as grey level dilation.

This two dimensional operation is programmed to be executed as dual scan operation: one horizontal scan and one vertical. This speeds up the operation and makes the execution time less dependent of window size.

Examples

`max`	creates the maximum image of the default source, window is 5x5.
`max a 37 39`	creates the maximum image of the image in a, window is 37 (v) and 39 (h).

maxl

Command syntax:
`maxl [a] [#1 [#2]] [/]`

Parameters
#1 - action to be performed:
  1 = write pixels with value #2 (0 - 255; default: drawing value)
  2 = XOR pixels with value #2 (0 - 255; default: graphic value)
  3 = OR pixels with value #2 (0 - 255; default: graphic value)
  5 = read pixels on line and store them in lbuf

Return value
Maximum distance

Family
Parameter operation

Function
Finds the position of the longlest diagonal that fits in an object.

Description . . .

Description

From a Freeman string of a closed contour (produced by fcont) this operation finds the two points on the contour with the largest mutual distance. The distance is returned.

The coordinates of the points and the arc of the connecting line are stored in IBUF:

Ibuf	item
0	Y1
1	X1
2	Y2
3	X2
4	arc*100 (radians)
5	arc (degrees)

Specifying **'/'** forces the calculation to take place in integer. This makes it much faster, but prevents correction for non-square pixels.

Example:

```
dis p
frmt 22 44
bord pc 255         ;make a rectangle
fcont 8             ;make Freeman string from contour
maxl 3 255          ;calculate maximum distance
```

min

Command syntax:
```
min [a] [#w]
min [a] #wy #wx
```

Parameters
#w - window (neighbourhood) size (3 - <image size>; default: 5)
#wy - vertical size of window (3 - <image size>)
#wx - horizontal size of window (3 - <image size>)

Return value
none

Family
Neighbourghood operation

Function
Replaces each pixel with the minimum pixel value found in the specified window.

Description . . .

Description

Each pixel is replaced by the minimum pixel value that is found in a specified window around it. This operation is also known as grey level eroding.

This two dimensional operation is programmed to be executed as dual scan operation; one horizontal scan and one vertical. This speeds up the operation and makes the execution time less dependent of window size.

Examples

`min`	creates the minimum image of the default source's image, window is 5x5.
`min a 37 39`	creates the minimum image of a, window is 37 (v) and 39 (h).

movy

Command syntax:
`movy [a] #`

Parameters
- number of pixels to shift (from -<image size - 1> to +<image size - 1>)

Return value
none

Family
Geometric operation

Function
This operation moves the image the specified amount of pixels in the Y-direction. Positive Y direction is: down.

Description . . .

Description

This operation moves the image vertically (**movy**) or horizontally (**movx**). Pixels that shift out of the image at either side are shifted into the image at the opposite side (the image 'rotates').

The moving direction depends upon the sign of the parameter:

	movx	**movy**
+	right	down
–	left	up

Examples

`movx -10`	moves the default source 10 pixels to the left
`movy a 10`	moves the content of image **a** 10 pixels down

movx

Command syntax:
`movx [a] #`

Parameters
- number of pixels to shift (from -<image size - 1> to +<image size - 1>)

Return value
none

Family
Geometric operation

Function
This operation moves the image the specified amount of pixels in the X-direction. Positive X direction is: right.

Description . . .

mul

Command syntax:

```
1. mul [a] #1
2. mul a b
```

Return value

none

Family

Pixel operation

Function

Multiplies two images (2.) or an image and an unsigned constant. The result is divided by 256, to keep the result in the 8-bits range.

Description . . .

Description

An arithmetic, unsigned multiplication is performed for each pixel. The multiplier may be a constant or the pixels of another image.

The result of the multiplication is divided by 256. This limits the result of the multiplication of two images to the range 0 - 255.

Examples

`mul 3`	the image in the default source is multiplied by 3/256
`mul p q >a`	the images in p and q are multiplied and scaled back by division by 256; the result is copied to a.

neg

Command syntax:
`neg [a]`

Return value
none

Family
Pixel operation

Function
Negates the pixels

Description . . .

Description

Pixels are replaced by their 2's complement value:

0 -> 0
1 -> 255
2 -> 254 etc.

Examples

`neg`	produces the 2's-complement of the default source
`neg a >b`	produces the 2's complement of **a** and copies the result to **b**.

noise

Command syntax:
`noise [a] [#]`

Parameters
- seed (0 - 65535; default: 0)

Return value
none

Family
Miscellaneous operation

Function
Creates image, consisting of pseudo random pixel values. '#' is a seed value.

Description . . .

Description

This operation will generate pseudo random pixel values. The seed value determines a starting point for the generator. Using the same seed value, two noise operations result in two equal, though (pseudo) random, images.

This operation can be used to encode information by XOR-ing an image produced by noise with another image. The product of this is an unrecognizable image which can easily be decoded by XOR-ing it once more with the same noise image. After coding an image, it is sufficient to remember the seed value to be able to decode it again.

Using a seed value will result in an identical image. If the seed is omitted, the noise pattern will depend upon previous activities.

Examples

`noise`	produces a random image in the default source
`noise p 2`	produces a random image in p, using seed value 2.

or

Command syntax:
```
1. or a b
2. or [a] #
```

Return value
none

Family
Pixel operation

Function
ORs bitwise two images (1.), or an image and a constant (2.). See also: binary

Description . . .

Description

OR performs the logic OR-function of the pixels of two images, or the OR-function of an image and a constant. This means that the bits of the result are set to 1, if the corresponding bits of either of the source pixels (or constant) are 1.

Examples

`or a b`	a and b are OR-ed
`or a b >c`	as above; the result is copied to c
`or a 15`	a is OR-ed with 15 (binary: 0000 1111): the 4 lowest bitplanes are set.

orpat

Command syntax:

```
1. orpat [a] [#YX] [/]
2. orpat [a] #Y #X [#pg] [/]
```

Parameters
See pattern drawing

Family
Graphic operation

Function
Draws a figure along a path, specified by a Freeman string in Ibuf. Drawing takes place by ORing a bit pattern (default: graphics value).

Description . . .

Description

This function draws a figure, of which the Freeman contour string is available, by OR-ing the pixels along the path with a bit pattern. The string consists of bytes, and has to end with 255 (0ffh).

The Freeman string is read from Ibuf. If the string is produced by **fcont**, and **orpat** follows immediately, then **orpat** can be instructed to read directly from the internal **fcont** buffer, by specifying the exception parameter (/). In this case the figure's default starting position is not the image's cursor position, but the original starting position.

Valid Freeman codes are: 0, 1, ...7. The following numbers have a special meaning:

Freeman code + 128 skip this pixel
255: end of string

If performed immediately after the fcont operation, the exception parameter **'/'** can be specified. Then the Freeman codes are read from the internal buffer, where fcont stores them, and the default starting position is that of the original image. Otherwise: cursor position.

For details on pattern drawing and parameter interpretation, see pattern drawing
See also: Concepts of graphic operations

orvec

Command syntax:
`orvec [a] #a [#l [#Y #X] [#pg]]`

Parameters
See vector drawing

Family
Graphic operation

Return value
number of pixels on vector

Function
Writes pixels (OR-wise) along the imaginary vector, specified by the numeric parameters.

Description . . .

Description

This function draws vectors in any direction by OR-ing a bit pattern into the pixels of the vector. The length is specified in real length units. If no length is specified (or the length value is 0), then the vector runs to the image edge.

Angle direction is interpreted as usual (e.g. 90 degr. is up).

For details on vector drawing and parameter interpretation, see vector drawing
See also: Concepts of graphic operations

Examples

`orvec 222`	draws a vector from the cursor position with an 222 degr. angle to the image edge, using the graphics value
`orvec a 33 10`	draws a vector in image a from the cursor position angle 33, length 10
`orvec 10 0 128 128 1`	draws a vector from 128, 128 to the image edge, angle 10, bitplane value 1

orln

Command syntax:

1. `orln [a] #YX1 [#YX2 [#pg]]`
2. `orln [a] #Y1 #X1 #Y2 #X2 [#pg]`

Parameters

See line drawing

Return value

number of pixels on line

Family

Graphic operation

Function

Writes pixel values (ORwise) along imaginary line, specified by numeric parameters.

Description . . .

Description

For details on line drawing and parameter interpretation, see line drawing
See also: graphics concepts

ovl

Command syntax:
`ovl [a] [#1 [#2]]`

Return value
previous status

Family
Control operation

Function
Controls reading & writing in overlays

Description . . .

Description

Controls whether the lower or upper byte of a 12-16 bits image memory acts as a source and/or destination of operations.

Parameter	Read	Write
0	Lower	Lower
1	Upper	Lower
2	Lower	Upper
3	Upper	Upper

Note: parameter values are added, so: `ovl 3` is equivalent to `ovl 1 2`. This allows the use of aliases as in:

```
ovl p read_lower write_upper
```

+ TIMCOM:outK out$ TIM Command out# T_OUT

Command syntax:

```
out #1 [#2]
```

Return value

none

Function

Ouputs a pattern, specified by #1, to the parallel (printer) port.

Description . . .

Description

If #2 is given, the system will wait for the first (double) frame to start, and then count #2 image lines, before the output is sent to the port. The duration of the signal is 1 image line time (64 microseconds).
Pins of the printer port: `9 8 7 6 5 4 3 2`
Bits no.: `7 6 5 4 3 2 1 0`

pan

Command syntax:
`pan [#]`

Return value
previous pan status

Parameters
- 0: pan on
- 1: pan off

Family
Control operation

Function
Enables/disables panning

Description . . .

Description

If the image to be displayed is larger than the display page, display is automatically adjusted such, that the cursor is in the center of the image. With 'pan' you can toggle this on and off.
No parameter: toggle panning. See also: zoom.

perc

Command syntax:
```
perc a [#1 [#2]]
```

Return value
none

Family
Neighbourghood operation

Function
Percentile filter.

Description . . .

Description

Removes extreme pixel values (e.g. shot noise) and makes a good guess for the substituted value.

#1 - window size: 3 - 5 (def) - 7 - 9
#2 - percentile value (entry in histogram of window where substituted value is taken from)

phis

Command syntax:
```
phis [a] [#1 [#2]]
```

Parameters
#1 = image line supplying Y-addresses (def. 0)
#2 = image line supplying X-addresses (def. #1+1)

Return value
none

Family
Pattern Recognition operation

Function
Plots dots at coordinates, specified by the pixels in image lines #1 and #2

Description . . .

Description

This operation plots dots in the default source image, using X- and Y-addresses coming from the pixels of the image lines #1 & #2 of the specified image. The pixel values, pointed to by these addresses, are incremented.

Coordinate 0,0 is top-left.

Examples

`hist p`	make histogram of image
`ihis a 1`	store it into an image line
`hist q`	make histogram of another image
`ihis a 2`	store it into another image line
`phis a 1 2`	make an XY plot of the two histograms

pl

Command syntax:
`pl [a] [#1 [#2 #3]] [<file>]`

Return value
none

Function
Converts a Freeman string, as produced by `fcont` into HP-GL plotter code. The size of the plotted figure is related to the size of the indicated image.
#1 - pen number
#2, #3 - vertical and horizontal offset of the drawing.

Description . . .

Description

If a file name is specified, printer data is sent to that file or device. If not, data is sent to the device specified in 'I'(nstall 8).

prb

Command syntax:
`prb [a] [#1 [#2 #3]] [<file>]`

Return value
none

Function
Prints a bitplane on a matrix printer: 1 dot per pixel.

Description . . .

Description

#1- bitplane to be printed: 1-8 (def. 8) If a negative value is supplied, the inverse in printed.
#2- multiplication factor vertical size (default: 1)
#3- multiplication factor horizontal size (default: #2) Vertical image size should be a multiple of 8.

If a file name is specified, printer data is sent to that file or device. If not, data is sent to the device specified in 'I'(nstall 16).

prc

Command syntax:
```
prc [a] [#1 [#2 [#3]]]
```

Return value
none

Function
Prints 3 bitplanes on the IBM Colorjet 3852 printer.
Colors used:

upper bitplane: blue
lower bitplane: green
lowest bitplane: red

Description . . .

Description

#1 - (abs. value) upper of the 3 bitplanes (3 - 8; def. 8 (msbit)) If negative (e.g. -3) black is rejected
#2 - multiplication factor vertical size (def. 1)
#3 - multiplication factor horizontal size (def. #2)

If a file name is specified, printer data is sent to that file or device. If not, data is sent to the device specified in 'I'(nstall 16).

pri

Command syntax:
```
pri [a] [#1 [#2]] [<file>]
```

Return value
none

Function
Prints a gray value image on a matrix printer. Gray values are simulated by dithering (varying the distance between dots).
#1 - vertical multiplication factor (def.: 1).
Some printers (e.g. HP-LJ) require a power of 2 (1, 2, 4, 8)
#2 - idem horizontal (default: #1)

Description . . .

Description

Vertical image size should be a multiple of 8.
If a file name is specified, printer data is sent to that file.

ps

Command syntax:
`ps [a] #1 [#2] [/] [<file>]`

Family
I/O operation

Function
Produces file of image in PostScript format.

Description . . .

Description

Produces file of image in PostScript format.

#1 width of image in cm. Height = width * V/H ratio (See Non square pixels)
#2 number of bits: 1, 2, 4, 8 (default)
If less than 8: top bitplanes are used
Negative values (-1, -2, -4, -8): image is inverted. In this case lines will be printed black on white
/ Encapsulated PostScript file is produced (extension: **.eps**) If no '/' is specified, the extension is **.ps**.

If no file is specified, the result is sent to a file: TIMWIN.PS (TIMWIN.EPS)

qlap

Command syntax:
`qlap [a] [#]`

Parameter
- gain. Default: 1

Return value
Number of overflow pixels

Family
Neighbourghood operation

Function
Fast 3x3 implementation of convolution filter, type Laplace

Description . . .

Description

This convolution operation produces the "Laplacian" (= 2nd derivative) of the image with a fixed 3x3 coefficient scheme. The implementation is optimized for speed.

To be able to represent negative values, 128 is added to the result. Thus:

positive values:	>128
0:	128
negative:	<128

Examples

`qlapl a`	3x3 laplace filter applied to a
`qlapl a 2`	performs a laplace filter operation, multiplies the result by 4 (2**2)

See also: lapl. (The result of this operation differs from **lapl** due to differences in the coefficients)

qorde (not available)

Command syntax:
```
qorde [a] [#Y1 #X1 #Y2 #X2 [#]]
```

Return value
higest fringe

Function
Fringes crossing an imaginary line are assigned an increasing or decreasing grey value. The line is specified by the coordinates #X, #Y (defaults: upper left to lower right).
The source image must consist of 8-connected fringes in the least significant bitplane.

Description . . .

Description

- the start value for the fringes, where positive means increasing and negative means decreasing. Order values are even, since the least significant bit is used for internal purposes.

See also: qphas

qphas

(not available)

Command syntax:
`qphas [a] [#]`

Return value
points outside figure

Function
Produces a grey value image out of a ordened fringe image (as produced by qorde) by interpolating and scaling fringe values.

Description . . .

Description

= the stitch of the grid (default: 8)

The resulting image can be enlarged to full size by blow (fast but coarse) or ct (smooth but slow)

qplot

Command syntax:
```
qplot a [#1 [#2 [#3]]]
```

Return value
none.

Parameters
#1 - arc of horizontal view (default: 60)
#2 - arc of vertical view (default: 30)
#3 - Plotting grey value (default: drawing value)

Family
Graphic operation

Function
Plots a grey value image as a semi 3D-plot.

Description . . .

Description

This operation creates a surface plot of a 256x256 grey value image. The plot is drawn in a 512x512 image.

Example

The sequence below creates a 256x256 test image in **p**, and plots this image in a 512x512 image **x**. Change image names according to your set-up, if necessary.

```
dis p          ;create test image:
dump 255
bgm 0
dt
inv            ;grey value cone
dis x          ;select image for plot
qplot p        ;perform plot
```

qshrp

Command syntax:
`qshrp [a] #`

Parameter
- gain. Default: 1

Return value
number of pixels having overflow

Family
Neighbourghood operation

Function
Quick sharpening (adds laplacian to original)

Description . . .

Description

This convolution operation produces the "Laplacian" (= 2nd derivative) of the image with a fixed 3x3 coefficient scheme, and adds it to the original image. Thus the image is sharpened: the high frequencies are enhanced. The implementation is optimized for speed.

Examples

`qshrp a`	3x3 sharpening filter applied to **a**
`qshrp a 2`	performs a sharpening filter operation, multiplies the result by 4 (2**2)

See also: shrp. (The result of this operation differs from **shrp** due to differences in the coefficients)

rdpat

Command syntax:

```
1. rdpat [a] [#YX] [/]
2. rdpat [a] #Y #X [/]
```

Parameters

See pattern drawing

Family

Parameter operation

Function

Reads the pixel values along a path, specified by a Freeman string in Ibuf. The pixel values are written into Ibuf

Description . . .

Description

This operation follows a path through an image, specified by a Freeman string.
The path can be 'learned' using the fcont command, or created otherwise.
The starting point can be user specified, or be the same as the learned curve's (use the **'/'** (exception) parameter).

If this operation is performed immediately after an fcont operation, then the **'/'** (exception) parameter can be specified. Then the default starting address is the starting address of the original contour; else it is the image's cursor position.

For details on pattern drawing and parameter interpretation, see pattern drawing

Examples

`rdpat`	read pixels specified by a Freeman string in Ibuf. Start at the cursor position. Store values in Ibuf
`rdpat /`	as above, but read Freeman code from internal buffer. Start at original position
`rdpat a 100 200`	read from **a**, start at specified position
`rdpat a 100 200 /`	as above; read Freeman codes from internal buffer.

See also

Concepts of graphic operations

rdln

Command syntax:
1. `rdln [a] #YX1 [#YX2]`
2. `rdln [a] #Y1 #X1 #Y2 #X2`

Parameters
See line drawing

Return value
number of pixels on line

Family
Parameter operation

Function
Reads the pixel values along the imaginary line, specified by the numeric parameters, and stores them into Ibuf

Description . . .

Description

This operation follows a line through the image, and copies the pixel values it encounters into Ibuf.

For details on line drawing and parameter interpretation, see line drawing

Examples

`rdln 0`	reads the pixels on a line between the current cursor postion and the upper left position in the image
`rdln 11 22 33 44`	reads the pixels on a line running from 11, 22 and 33, 44 (Y,X)

See also

graphics concepts

rdvec

Command syntax:
```
rdvec [a] #a [#l [#Y #X]]
```

Parameters
See vector drawing

Return value
number of pixels on vector

Family
Parameter operation

Function
Reads the pixel values along the imaginary vector, specified by the numeric parameters, and stores them into Ibuf.

Description . . .

Description

This operation follows a vector through the image, and copies the pixel values it encounters into Ibuf.

For details on vector drawing and parameter interpretation, see vector drawing

Examples

`rdvec 222`	reads the pixels on a vector running from the current cursor postion under an angle of 222 degr. to the border of the image
`rdvec a 33 10`	reads the pixels on an angle of 33 degr. from the current cursor postion with a length of 10 pixels
`rdvec a 33 10 2 3`	reads the pixels on an angle of 33 degr. from postion 2 (Y), 3 (X) with a length of 10 pixels

See also

Concepts of graphic operations

redu

Command syntax:
`redu [a] [#]`

Parameter
- reduction factor (default: 1)
  0 - fast 2x2 reduction
  1 - rearranging the pixels 2*2, producing 4 reduced images
  2 - reducing by calculating the mean in a 2*2 window.
  3 - idem 3*3, etc.

Return value
none

Family
Geometric operation

Function
Reduces the size of image with a discrete value.

Description . . .

Description

This operation has two forms:

1. Reduces an image by rearranging pixels.

With parameter # = 1 this operation does not change pixel values, but shuffles them such that reduction occurs. Of every 2x2 window the upper-left pixel is placed in the upper-left sub-image, the upper-right pixel in the upper-right sub-image, etc. Thus, four subimages appear, that are not completely equal. Consequently, after reducing eight times this way, the original image is build up again.

This version does not operate correctly in place.

2. Reduces by averaging

With parameter # >= 2 it performs straightforward reduction, producing one reduced image in the upper left corner of the destination image. Each pixel value of the reduced image is calculated as the mean value of the # x # reduction window.

ribuf

Command syntax:
`ribuf <file>`

Parameter
<file>   the file to be read. Default extension: **.ibf**

Return value
none

Family
I/O operation

Function
Reads the specified file and writes its content into Ibuf.

Description . . .

Description

Reads the specified file and writes its content into Ibuf. The file must have the correct format (i.e. it must have been created by wibuf).
If no path is specified, TIMWIN looks for the file in the TIMWIN directory.
If no extension is specified, TIMWIN appends **.ibf** to the base name. If you don't want that, specify an empty extension (**.**)

By reading an Ibuf file, you can restore Ibuf to the state that it had immediately before executing **wibuf**, including data type, number of elements, etc.

Examples

`ribuf test`	reads a file test.ibf into Ibuf
`ribuf f:\test\data.`	reads a file f:\test\data into Ibuf

See also

fscan, wibuf

robg

Command syntax:
`robg [a]`

Return value
none

Family
Neighbourghood operation

Function
Roberts gradient operation

Description . . .

Description

This operation finds the edges in an image. Edges indicate changes in grey value of the image. The strength of the edge depends of the slope of the grey value change.

This operation is independent of the direction of the gradients in the image since it calculates the absolute value of the differences: in a 2x2 window, the sum of the absolute values of the two diagonal differences is calculated.

Examples

`robg a`	make an edge image of **a**

See also

grad, and the **vedge** command file

rotl

Command syntax:
`rotl [a]`

Return value
none

Family
Geometric operation

Function
Rotates the image left 90 degrees. This operation cannot be performed in place.

Description . . .

Description

This operation rotates the image left (anti-clockwise) by reading lines and writing back columns. Thus it cannot be performed in place without side effects.

Examples

`rotl a`	rotate image **a**
`rotl b >b`	rotate image **b** and write it back immediately

See also

ct, rotr

rotr

Command syntax:
`rotr [a]`

Return value
none

Family
Geometric operation

Function
Rotates the image right 90 degrees. This operation cannot be performed in place.

Description . . .

Description

This operation rotates the image right (clockwise) by reading columns and writing back lines. Thus it cannot be performed in place without side effects.

Examples

`rotr a`	rotate image **a**
`rotr b >b`	rotate image **b** and write it back immediately

See also

ct, rotl

rt

Command syntax:

```
rt [#]
```

Parameter

#1 - Grabbing mode:

-1 - Continuous grabbing (default).

0 - Grabbing until key pressed.

>0 - Number of frames to grab

Family

Real Time operation

Function

Controls real time operation

Description . . .

Description

This function performs a real time operation without modifying the Look Up Tables. This command is very useful to end a continuous grabbing sequence, that was started using another RT-command.

Examples

`rta p`	zoom into p, then start real time average and return control
`.....`	do whatever needed
`rt 1`	to end, use the **rt** command as shown

`rte 20`	start real time edge detection for 20 frames
`shl 2`	make an 8-bits image out of it
`lut 1`	display it using a standard LUT

`rta 20`	produce reference image
`rtc`	do real time compare

rta

Command syntax:
`rta [#1] [#2] [#3]`

Parameter
#1 Grabbing mode
-1 - Continuous grabbing (default).
0 - Grabbing until key pressed.
>0 - Number of frames to grab

#2 - Weightfactor accumulated content (default 15)
#3 - Weightfactor current image (default 1)

Return value
none

Family
Real Time operation

Function
Real-time average.

Description . . .

Description

Calculates and displays the average of the grabbed frames. The result of this operation is a 6-bits image.

To keep the process of continuous integration going, each frame time a fraction of the accumulated values is removed, and the value of the current image is added. These fractions can be specified:

(#2 + #3) / #3 specifies the number of frames that it takes to fully refresh the image

All input LUTS and output LUT 16 will be modified.

Examples

`rta p`	zoom into **p**, then start real time average and return control
.....	do whatever needed
`rt 1`	to end, use the **rt** command as shown
`shl 2`	make an 8-bits image out of it
`lut 1`	display it using a standard LUT
`rta 20`	produce reference image
`rtc`	do real time compare

rtc

Command syntax:
`rtc [#]`

Parameter
#1 - Grabbing mode:
- -1 - Continuous grabbing (default).
- 0 - Grabbing until key pressed.
- >0 - Number of frames to grab

Family
Real Time operation

Function
Real time compare.

Description . . .

Description

Calculates and displays the absolute difference between the grabbed image and the image which was already stored in the upper six bits of the frame memory at the start of this operation (e.g. produced by rta).

This operation produces a 6-bits image. All input LUTS and output LUT 16 will be modified.

Examples

`rta 20`	produce reference image
`rtc`	do real time compare
`rt 1`	to end, use the **rt** command as shown
`shl 2`	make an 8-bits image out of it
`lut 1`	display it using a standard LUT

rtd

Command syntax:
`rtd [#]`

Parameter
#1 - Grabbing mode:
- -1 - Continuous grabbing (default).
- 0 - Grabbing until key pressed.
- >0 - Number of frames to grab

Return value
none

Family
Real Time operation

Function
Real-time difference.

Description . . .

Description

Calculates and displays the absolute difference between two consecutive images. This operation produces a 6-bits image. All input LUTS and output LUT 16 will be modified.

Examples

`rtd`	do real time difference
`rt 1`	to end, use the **rt** command as shown
`shl 2`	make an 8-bits image out of it
`lut 1`	display it using a standard LUT

rte

Command syntax:
`rte [#]`

Parameter
#1 - Grabbing mode:
- -1 - Continuous grabbing (default).
- 0 - Grabbing until key pressed.
- >0 - Number of frames to grab

Return value
none

Family
Real Time operation

Function
Real time edge detection.

Description . . .

Description

Calculates and displays the absolute difference between two consecutive images, which feature a mutual displacement of 2 pixels in x- and y direction. This operation produces a 6-bits image. All input LUTS and output LUT 16 will be modified.

Examples

`rte`	start real time edge detector
`rt 1`	to end, use the **rt** command as shown
`shl 2`	make an 8-bits image out of it
`lut 1`	display it using a standard LUT

rtm

Command syntax:
`rtm [#]`

Parameter
#1 - Grabbing mode:
-1 - Continuous grabbing (default).
0 - Grabbing until key pressed.
>0 - Number of frames to grab

Return value
none

Family
Real Time operation

Function
Real-time minus.

Description . . .

Description

Calculates and displays the signed difference between two consecutive images.

This operation produces a 6-bits image (0 - 63). No difference: value = 32.
All input LUTS and output LUT 16 will be modified.

Examples

`rtm`	do real time minus
`rt 1`	to end, use the **rt** command as shown
`shl 2`	make an 8-bits image out of it
`lut 1`	display it using a standard LUT

rts

Command syntax:
`rts [#]`

Parameter
#1 - Grabbing mode:
-1 - Continuous grabbing (default).
0 - Grabbing until key pressed.
>0 - Number of frames to grab

Return value
none

Family
Real Time operation

Function
Real-time subtraction.

Description . . .

Description

Calculates and displays the signed difference between the grabbed image and the image which was already stored in the upper six bits of the frame memory at the start of this operation (e.g. produced by rta).

This operation produces a 6-bits image. (0 - 63; no difference = 32).
All input LUTS and output LUT 16 will be modified.

Examples

`rta 20`	produce reference image
`rts`	do real time compare
`rt 1`	to end, use the **rt** command as shown
`shl 2`	make an 8-bits image out of it
`lut 1`	display it using a standard LUT

rub (currently not available)

Command syntax:
`rub [a] [#]`

Return value
1 (left button), ASCII code

Family
Graphic operation

Function
Draws a 'rubber band' line.

Description . . .

Description

A non destructive line is drawn (using the graphics value) between the initial cursor position, and wherever the cursor is positioned during the command. The command is aborted by pressing the left mouse button or any keyboard key (encoded in return parameter).

specifies the action to take afterwards:

1 the resulting line is written into the image using the drawing value
2 the resulting line is written into the image by XOR-ing using the graphics value
3 the resulting line is written into the image by OR-ing using the graphics value
5 the grey values laying 'under' the line are stored in Ibuf

save

Command syntax:

```
1. save a
2. save <file>
3. save zz*
```

Parameter

`<file>`	image file to save
`zz*`	file name with wildcards

Return value

none

Family

Transport operation

Function

Copies the default image to the specified destination

Description . . .

Description

The **save** operation is equivalent to the **copy** operation, except that it uses the active image as the default source image. It can copy to an image as well to a disk file. In the latter case you can specify a partial file name; **save** will crate a unique file name then.

In case of a file name **zz***, the string will be expanded into `zzxXXXXX`, where

- x is a changing character `0, a - z`
- XXXXX is a number, which remains the same during the session. Examples: `zz016335, zza16335`, etc. Note: no extension (.im) will be appended.

Examples

`save a`	copies the contents of the default source to **a**
`save fname`	copies the contents of the default source to a file on disk named **<path>fname.im**
`save a:fname.pic`	copies the image in the default source to the specified file (the defaults are overruled).
`save xy*`	copies default source image to a file with a name like **xy031871**
`save xy*`	copies default source image to a file with a name like **xya31871**
`save xy*`	copies default source image to a file with a name like **xyb31871**
`...`	etc.

See also

copy

sbln

Command syntax:

1. `sbln [a] #YX1 [#YX2 [#]]  [/]`
2. `sbln [a] #Y1 #X1 #Y2 #X2 [#] [/]`

Parameters

See line drawing

Return value

grey value of last pixel.

Family

Graphic operation

Function

Scans along imaginary line, specified by the numeric parameters, until a pixel with bitplane # set is encountered (default: 1). Then scanning stops. The cursor is moved to that position, unless the **'/'** parameter is specified.

Description . . .

Description

This operation draws an imaginary line through the image. When it encounters a pixel on this line, whose bitplane # is set {e.g. is not 0), it stops. The image cursor is moved to that position, except when the exception parameter (/) is specified to avoid this. The return parameter indicates if this happened: if 0, no appropriate pixel was encountered.

Examples

`sbln 33 44 2`	scan along line that runs from the current cursor position to 33,44 (Y,X); stop when bit set in bitplane 2 is encountered
`sbln 0 128 255 128 /`	scan along line; stop when pixel with bitplane 1 (default) set is encountered; don't move cursor to that position.

For details on line drawing and parameter interpretation, see line drawing
See also: graphics concepts

sbvec

Command syntax:
```
sbvec [a] #a [#l [#Y #X] [#b]] [/]
```

Parameters
See vector drawing

Return value
grey value of last pixel.

Family
Graphic operation

Function
Scans along the imaginary vector, specified by the numeric parameters, until a pixel with the indicated bitplane set is encountered. Then scanning stops. The cursor is moved to that position, unless the **'/'** parameter was specified.

Description . . .

Description

This operation draws an imaginary vector through the image. When it encounters a pixel on this vector, whose bitplane # is set (e.g. is not 0), it stops. The image cursor is moved to that position, except when the exception parameter (*/*) is specified to avoid this. The return parameter indicates if this happened: if 0, no appropriate pixel was encountered.

Examples

`sbvec 11 8`	scan from the cursor position to the image edge, angle: 11 degr., bitplane 8
`sbvec 0 222 333 /`	scan from 222,333 (Y,X) to the image edge, angle: 0. Don't move the cursor

For details on vector drawing and parameter interpretation, see vector drawing
See also: Concepts of graphic operations

sgln

Command syntax:
1. `sgln [a] #YX1 [#YX2] [/]`
2. `sgln [a] #Y1 #X1 #Y2 #X2 [/]`

Parameters
See line drawing

Return value
grey value of last pixel.

Family
Graphic operation

Function
Scans along a line,   until a pixel having a grey value different from '0' is encountered.

Description . . .

Description

This operation draws an imaginary line through the image. When it encounters a pixel on this line, that has a grey value that differs from 0, it stops. The image cursor is moved to that position, except when the exception parameter (**/**)   is specified to avoid this. The return parameter indicates if this happened: if 0, no appropriate pixel was encountered.

Examples

`sgln 0`	scan along line that runs from the current cursor position to packed position 0 (=0,0); stop when greyvalue >0 is encountered; mover cursor to that position
`sbln 0 128 255 128 /`	scan along line; stop when greyvalue >0 is encountered; don't move cursor to that position.

For details on line drawing and parameter interpretation, see line drawing
See also: graphics concepts

sgvec

Command syntax:
```
sgvec [a] #a [#l [#Y #X] [#pg]] [/]
```

Parameters
See vector drawing

Return value
grey value of last pixel.

Family
Graphic operation

Function
Scans along a vector until a pixel having a grey value different from '0'   is encountered.

Description . . .

Description

This operation draws an imaginary vector through the image. When it encounters a pixel on this vector, whose greyvalue is not 0, it stops. The image cursor is moved to that position, except when the exception parameter (/) is specified to avoid this. The return parameter indicates if this happened: if 0, no appropriate pixel was encountered.

Examples

`sgvec 11`	scan from the cursor position to the image edge, angle: 11 degr., stop when pixel value >0 is encountered; move cursor to that position.
`sgvec 0 222 333 /`	scan from 222,333 (Y,X) to the image edge, angle: 0. Stop if pixel value >0 is encountered; don't move the cursor.

For details on vector drawing and parameter interpretation, see vector drawing
See also: Concepts of graphic operations

sel

Command syntax:
`sel a <=> b`

Return value
number of valid relations

<=> compare parameter

Family
Pixel operation

Function
Selects pixels from two images, according to the result of the comparison

Description . . .

Description

Produces an image, consisting of pixels of the specified images. If, for one pixel, the relation is true, then the pixel value of the first image is used. If it is false, the pixel value of the second image is used.

Valid relations are:

Relation	Meaning
>	greater than
<	less than
>=	greater or equal
<=	less or equal

Examples

`sel a > b`	selects the largest pixel values of **a** and **b** (where equal, **b** is selected)
`sel p <= q`	selects the smallest pixel values of **p** and **q**

set

Command syntax:

```
1. set #1
2. set #1 [#2]
```

Return value

previous set value

Family

Control operation

Function

1. Shows the values of set parameters.
2. Adjusts a set parameter.

Description . . .

Description

This command manages several system settings.
The **set** function has the following values:

#1	Variable to set	#2 - value range
1	graphics value	0 - 255
2	cursor value	0 - 255
4	drawing value	0 - 255
5	~~clock~~ Frame Grabber	0 (crystal), 1 (PLL)
9	calibration factor	(floating point value)
10	stepsize histogram	1 - 10
11	bit mask for host access	0 - 255
12	bit mask for video access	0 - 255
13	video input channel	1 - ...
14	frame grabber no.	1 - 3
15	update window	-64 - 64
20	video gain	
21	video offset	

See also: alias, Install Menu

shl

Command syntax:
`shl [a] [#] [/]`

Parameter
- shift count (1 - 8; default: 1)

Return value
none

Family
Pixel operation

Function
Shifts the pixel bits # positions left (or up; default: 1)

Description . . .

Description

Shifting numbers up or down is an efficient way to multiply or divide by 2.

Examples

`shl`	shift the bits in the pixels of the default source left one position (multiply by 2)
`shl a 3`	shift left the pixels of a 3 positions (i.e. multiply by 8)
`shl 3 /`	only produce a look-up table, don't change pixels.

If **'/'** is specified, only the look up table is computed. See also: table operations and preview.

shr

Command syntax:
`shr [a] [#] [/]`

Parameter
- shift count (1 - 8; default: 1)

Return value
none

Family
Pixel operation

Function
Shifts the pixel bits # positions right (or down; default: 1).

Description . . .

Description

Shifting numbers up or down is an efficient way to multiply or divide by 2.

Examples

`shr`	shift the bits in the pixels of the default source right (down) one position (divide by 2)
`shr a 3`	shift right the pixels of a 3 positions (i.e. divide by 8)
`shr 3 /`	only produce a look-up table, don't change pixels.

If **'/'** is specified, only the look up table is computed. See also: table operations and preview.

shrp

Command syntax:
`shrp a [#1 [#2]]`

Parameters
#1 = filter size: 3 (def) - 5 - 7 - 9
#2 = gain

Return value
 # overflow pixels

Family
Neighbourghood operation

Function
Sharpens the image

Description . . .

Description

This convolution operation sharpens an image by adding the 'laplacian' (= second derivative) to the image.
Sharpening an image is achieved by adding the result of a laplace filter to the original image. By setting the gain, the amount of sharpness to be added can be defined.

Examples

`shrp a`	sharpens using 3x3 window
`shrp a 9 3 >a`	sharpens using 9x9 window, increase the effect by setting gain to 4 (2**(3-1)), copies result back to a.

See also

filt, qshrp, lapl

stat

Command syntax:
```
stat [a] [#]
```

Return value
Calculated value (see Description)

Family
Parameter operation

Function
Calculates several statistical values, calculated from the grey value histogram of the specified image.

Description . . .

Description

This operation produces a grey value histogram of the indicated image and then calculates the following:

Index	Description
0	the minimum pixel value
1	the maximum pixel value
2	the gray value below which are 1% of the pixels
3	the gray value above which are 1% of the pixels
4	the number of gray values in the image
5	the mean gray value
6	idem, without pixels having gray value 0
7	the median gray value
8	idem, taking the percentile borders (see 2 en 3) into account
9	the mean of abs. differences of mean pixel value and pixel values
10	idem, without pixels having gray value 0
11	relative mean of abs. difference (as 9, divided by mean (5))
12	idem, without pixels having gray value 0 (10/6)
13	standard deviation
14	idem, without pixels having gray value 0

If a numerical parameter is specified, the corresponding value is returned. If not, all values are copied to Ibuf, in the position specified with **index**. Since Ibuf is an integer buffer, the floting point values are rounded.

See also the statistics window

strip

Command syntax:
`strip [a] #1 [#2] [/]`

Parameters
#1 - grey value range
#2 - center of range (0 - 255; default: 128)

Return value
none

Family
Pixel operation

Function
Removes a specified range of grey values from an image.

Description . . .

Description

Leaves the image as it is, except for a range of grey values #1 around a central value #2, where 0 is written.

This operation is useful after `sub a b /`, where the difference between 2 images appears around the grey value 128. Small difference (e.g. due to noise) can be suppressed using **strip** by defining a 'band' of grey values that is set to 0.

Examples

`strip 10`	strips 10 grey values (123 - 133) off the image in the default source
`strip a 10 120 >b`	strips the grey values 115 to 125 off the image in **a**, copies the resulting image to b.

If **'/'** is specified, only the look up table is computed. See also: table operations and preview.

sub

Command syntax:

```
1. sub [a] #1
2. sub a b [/]
```

Return value

underflow pixels

Family

Pixel operation

Function

1. Subtracts a constant value from an image
2. Subtracts an image from another

Description . . .

Description

Subtracts two images (2.) or an image and a constant (1.). If underflow occurs (b > a or # > a) the result is set to 0.

If the **'/'** parameter is specified the subtraction is carried out in the form:
`(a - b)/2 + 128`,
to be able to represent negative results. In this case, no underflow can happen.

Examples

`sub a b`	subtract **b** from **a**
`sub a 11`	subtract 11 from **a**
`sub 11`	subtract 11 from the default source
`sub a b /`	calculate the signed difference of **a** and **b**

sum

Command syntax:
`sum a16 [b] [#n]`

Parameter
#n - repetition number

Return value
none

Function
Adds images (repeatedly) into a 16-bits image

Description . . .

Description

This operation performs a repeated 16-bits addition on an 8-bits image.
It can be used to integrate an image being grabbed, to reduce noise. To do so, start grabbing and perform this operation on the image being grabbed. To bring the 16-bits result to an 8-bits image, use the cp16 command

Example

`era16 a16`	erase the destination image
`dig`	start grabbing continuously
`sum a16 32`	add 32 times the content of the default source image into **a16**
`cp16 a16 p 32 /`	divide the resulting image by 32 and store the 8-bits result into **p**

See also: era16 and cp16

swap

Command syntax:
`swap a [b]`

Return value
none

Family
Transport operation

Function
Exchanges (swaps) images.

Description . . .

Description

This operation exchanges images. Only swapping between memory- or display images of the same size are allowed.

Examples

`swap a`	exchange the default source with **a**
`swap p1 p2`	exchange sub images **p1** and **p2**

tab

Command syntax:
`tab [a]`

Return value
none

Family
Pixel operation

Family
Pixel operation

Function
Converts an image using the content of Ibuf as a look up table.

Description . . .

Description

This operation performs a table look up operation on the specified image, using the present content of ibuf as look-up table data. This operation does not change the data in ibuf, it only makes use of it. **tab** will use the data in Ibuf on a byte basis, regardless of its actual type and content. The presence of correct values in Ibuf are the responsibility of the user. See also: Ibuf

Examples

`tab`	performs table look up (using existing table) on the default source
`tab a`	performs a conversion on **a** with existing data in the look up table

tcopy

Command syntax:

```
1. tcopy a <file> [#]
2. tcopy <file> a [#]
```

Function

Copies images using TIFF image format.

Parameters

1. # - number of bits to copy: 1, 2, 4 or 8 (default)
2. # - colour image (each pixel has a red, green and blue component):
    1 = red
    2 = green
    3 = blue

Family

Transport operation

Function

This operation reads and writes images having a TIFF-file header and the `.tif` file extension.

Description . . .

Description

1. Writes an image to a disk file using the TIFF format.
2. Reads a TIFF-image from a disk file

The TIFF header guarantees that special formats are recognized and handled correctly. This concerns:

- Image size
- number of bits per pixel
- RGB coded colour images
- Motorola (Apple) and Intel (PC) byte ordering

Examples

`tcopy miss p`	reads image <path>\miss.tif into **p**
`tcopy x result`	writes the content of **x** into a file <path>\result.tif

TIFF images - size

TIMWIN adjusts the dimensions of the destination image according to the specification in the TIFF header.

If the destination image is smaller than the TIFF header requires, then the upper left corner of the image is copied.

If the destination image is larger than the TIFF header requires, then the lower left of the image remains unchanged.

TIFF images - Number of bits

When writing a TIFF file, without further specification standard 8-bits pixels are written. The numerical parameter specifies the number of bits per pixel. If this is 1, 2 or 4, the 1, 2 or 4 top bitplanes are copied. In this case more pixels are packed into one byte.

#	bitplanes copied	pixels/byte	image size (%)
1	8	8	12.5
2	8, 7	4	25
4	8, 7, 6, 5	2	50
8	all	1	100

TIFF images - reading colour images

An RGB colour TIFF image consists of sequences of 3 bytes per pixel: one byte per colour. You can address the individual colour components by specifying the colour to be read in:

1 = red
2 = green
3 = blue

Example:

`tcopy col_imag p 1`	read red image
`tcopy col_imag q 2`	read green image
`tcopy col_imag r 3`	read blue image

TIFF images - byte ordering

Due to differences in the architecture of microcomputer's CPU, 16 bits numbers can be stored differently. For example, Motorola (Apple) and Intel (PC) based microprocessors use different storage schemes. As a result, a TIFF image created on a Macintosh differs from a TIFF image created on a PC.

TIMWIN automatically corrects for this specific byte ordering, so that images should be recognized correctly, regardless their origin.

tdis

Command syntax:
```
tdis <file>
```

Family
Transport operation

Function
Reads images files using TIFF image format.

Description . . .

Description

This operation reads images having a TIFF-file header and the `.tif` file extension.

The TIFF header guarantees that special formats are recognized and handled correctly. This concerns:

- Image size
- number of bits per pixel
- RGB coded colour images
- Motorola (Apple) and Intel (PC) byte ordering

Examples

`tdis image` read TIFF file **image.tif** into display

text

Command syntax:

```
text [a] [#p] [#Y #X] "Text [<*>] ", [variables]
```

Parameters

#p	pixel value (Default: graphics value)
#Y, #X	starting position text (Default: cursor position)
variables	variables whose value is to be included in the text

Family

Graphic operation

Function

Writes text into an image.

Description . . .

Description

This command writes text into an image. **text** and **textv** are similar, except that **textv** prints vertically.
To use this function, a character set must be available (see lcset).

Static text can be mixed with dynamic parts, e.g. variables. To do so, the text string must contain format specifiers (see formatting). After the string, the variables to be printed must be given.

Examples

`text "print this text"`
text written in graphics value bits at cursor position

`text a 1 20 10 "print value %d at position" 123`
text starts line 20 column 10, written in bitplane 1 of **a**. The printed text is: `print value 123 at position`

`text "Cursor position: X=%d, Y=%d" curs`
the return value of the curs command (a packed value) is printed as:
`Cursor position: X=111, Y=222` (the actual value may be different)

Notice, that **curs** produces a long integer (32-bits) value, and that the **%d** format specifier formats a short (16-bits) integer. In this case, two **%d** specifiers each take a 16-bits part of the 32-bits total. Intel byte ordering makes that the X-value comes first.
To print a single value, the format specifier should have been: **%ld**

textv

Command syntax:

```
textv [a] [#p] [#Y #X] "Text [<*>] ", [variables]
```

Parameters

#p pixel value (Default: graphics value)
#Y, #X starting position text (Default: cursor position)
variables variables whose value is to be included in the text

Family

Graphic operation

Function

Writes text into an image vertically.

Description . . .

thre

Command syntax:
```
1. thre [a] # [/]
2. thre [a] [/]
```

Return value
threshold value

Family
Pixel operation

Function
1. Thresholds an image with a specified threshold value
2. Thresholds an image with a automatically calculated threshold value

Description . . .

Description

The threshold operation separates an image in 'foreground' and 'background', by comparing the pixel values with a threshold value. If this value is positive, then pixels below that value will become 0, and pixels equal to or greater than that value will become 255. If the value is negative the same happens, but 0 (black) and 255 (white) will be exchanged.

If no threshold value is specified, a value is calculated from the grey value histogram of the image using the isodata algorithm. The calculated value is returned as return parameter. Note: the calculated value is positive.

Examples

`thre`	threshold the default source with 1
`thre a 200 >a`	threshold **a** with 200; copy the resulting image to **a**
`thre a -200`	threshold **a** with 200; inverts the result

If **'/'** is specified, only the look up table is computed.
See also: table operations and preview.

timer

Command syntax:

```
1. timer #
2. timer
```

Return value
current value

Family
Control operation

Function
1. Presets timer to # seconds. This value is decremented each second.
2. Reads value of timer.

Description . . .

Description

This operation starts a timer and presets it to the specified value. Each second the timer value decreases. Subsequent calls to the **timer** function return the value that has been reached by then.

In addition, absolute timing information is stored in lbuf (long integers) in the following order :

0. Seconds,
1. Minutes,
2. Hours (0-24),
3. Day of month (1-31),
4. Month(0-11; January = 0),
5. Year (current year minus 1900),
6. Day of week (0-6; Sunday = 0),
7. Day of year (0-365; January 1 = 0).

tsave

Command syntax:
`tsave <file>`

Family
Transport operation

Function
Writes the active image to a disk file and adds a TIFF image header.

Description . . .

Description

Writes an image to a disk file using the TIFF format.

The TIFF header guarantees that special formats are recognized and handled correctly. This concerns:

- Image size
- number of bits per pixel
- RGB coded colour images
- Motorola (Apple) and Intel (PC) byte ordering

Examples

`tsave spec` writes default source image to a disk file **<path>\spec.tif**

unif

Command syntax:
`unif [a] [#Y [#X]]`

Parameters
#Y - vertical filter size (default 5)
#X - horizontal filter size (def. #Y).

Return value
none

Family
Neighbourghood operation

Function
Uniform (blur) filter. The pixels are replaced by the mean value from a window of size #Y * #X.

Description . . .

Description

This operation replaces each pixel value with the mean value of the specified window.

This operation is implemented as two linear scans, one horizontally and one vertically. This enables the operation to be performed fast: speed is virtually independent of window size.

Window size is limited only by the image dimensions.

Examples

`unif`	blurs the image in the default source using a 5x5 window
`unif a 66 55 >b`	blurs the image in **a** using a 66x55 window, copies the result to **b**.

val

Command syntax:
`val [a] #1 [#2] [/]`

Return value
none

Family
Pixel operation

Function
This operation converts pixels having grey value #1 (or laying in the range #1 - #2) to 255, and all others to 0.

Description . . .

Description

This operation produces an image in which the pixels with a specified grey value, or range of values, are marked with pixel value 255. The remaining pixels are set to 0. This operation can be considered a threshold operator with two boundaries, an upper and a lower.

Examples

`val 128`	pixels having of value 128 are set to 255, others to 0.
`val a 100 200`	pixels from **a**, that have a grey value between 100 and 200, become 255, all others become 0.

If **'/'** is specified, only the look up table is computed.
See also: table operations and preview.

ver

Command syntax:
`ver [#]`

Parameters
= 1: return code number of frame grabber supported by the program (in hex)
= 2: return current version number of TIMWIN (in hex)

Return value
With parameter: version number
Without parameter: version string

Family
Control operation

Function
Supplies version information.

Description . . .

Description

This command gives informnation about version number and supported frame grabber.
If no parameter is supplied: version information is printed in text. This text is also available as return parameter.
If a parameter is supplied, the same information is giver numerically.

Frame Grabber	dec. code	hex. code
PCVision	1	1
PCVisionPlus	2	2
Cortex-I	3	3
FG100	257	101
VS100	258	102
VFG	259	103

Examples

`ver` print version information
`ver 1` return current frame grabber (e.g. 257 = 101H = FG100)

wcur

Command syntax:
`wcur`

Return value
none

Family
Graphic operation

Function
Writes the cursor (as visible on the screen) into the image.

Description . . .

Description

This operation writes the cursor, as it is before invoking this command, into the image permanently.

Note: After this operation the cursor (which was removed during the operation) pops up again as usual, but is then XORed out due to the cursor pattern just written. It will show up when the cursor is (re)moved. Repeating this operation restores the image.

wibuf

Command syntax:
`wibuf <file>`

Return value
none

Family
I/O operation

Function
Writes the content of Ibuf into the indicated file. See also:ribuf

Description . . .

Description

Writes the content of Ibuf into the indicated file. See also:ribuf. This allows you to restore Ibuf to its current state at a later moment.

Examples

`wibuf cfile1`	content of Ibuf is written to file **cfile1.ibf**

wig

Command syntax:
```
wig [a] [#1 [#2]]
```

Parameters
#1 - wedge function
#2 - modifier

Return value
none

Family
Miscellaneous operation

Function
Draws various grey value wedges

Description . . .

Description

Produces a test image, whose pixel values consist of the result of one of the operations below, carried out on the X- and Y-address of every pixel:

Parameter #1	function
1	divide*
2	multiply**
3	OR
4	AND
5	XOR
6	subtract
7	add (default)
8	y - address
9	x - address

* Default result: the quotient. If #2 = 1: the remainder
** Result is 16 bit. The upper byte is used as a result, if #2 = 1 the lower byte is used.

Examples

`wig`	generates a wedge in the default source
`wig a 1`	generates a pattern in **a**

wrxy

Command syntax:
`wrxy <TIM-command>`

Family
Miscellaneous operation

Function
Writes the return value of <TIM-Command> in Y-X format

Description . . .

Description

This function executes the specified TIMWIN command, and writes the return parameter to the system area in the form:

`Y-value: ..., X-value: ...`

Doing so a packed return value (as produced by frmt and curs) can be made readable.

Examples

`wrxy curs a`	show the cursor position of a in a readable format.

xdemo1

Command syntax:

```
xdemo1 [a] [#1 [#2]] XUSER example
```

Parameters

#1 - vert. size (default 10),
#2 - hor. size (default #1)

Function

Pixel operation demo - produces chessboard.

Description . . .

Description

This operation demonstrates how a pixel operation can be written using the XUSER system. Its goal is to spread a chessboard over the source image, and invert the areas covered by the white fields. The black fields are left alone.
The size of the fields is individually adjustable for X and Y.

Example

`xdemo1`	perform the operation on the default image, default field size (10)
`xdemo a 30`	perform the operation on **a**, field size 30
`xdemo x 10 40`	perform the operation on **x**, field size 10 (Y) and 40 (X)

Listing ...

Listing

```
/***************************************************************************
 * 2nd example: pixel operation
 * Chessboard image - the source image will be periodically inverted.
 * If one parameter is specified, this value determines the X- and Y-values
 * of the grid.
 * If 2 parameters are specified, the first determines the vertical size of
 * the grid and the second determines the horizontal size.
 * Syntax: xdemo1 [a] [#1 #2]]
 * #1, #2 - the size if the chessboard grid (Y, X) (Default: 10 for both)
 ***************************************************************************/
long xdemo1 (void)
{
LPTIMPARMS lpParm = (LPTIMPARMS) &Parm;
LPIMPTR lpImPtr = (LPIMPTR) &ImPtr;
LPPIXEL lpDest, lpSrc;
LPBINW lpBinW = (LPBINW) &BinW;

short xloop, yloop;
short cnt;                              // loop counter
UCHAR patrn = 0;                        // XOR pattern
short nLine = 0;                        // variable
short destincr;

   LPAR1 = 10L;                         // install a default value in 1st parm.
                                        // extract parameters, all types allowed
   cnt = XtractParms (lpParm, ALL_SRC+ONE_SRC);
   if (cnt < 0)                         // if something is wrong, 'cnt' has a
      return (cnt);                     // negative (error) value && error is set

   if (cnt <= 1)                        // if 0 or 1 param. specified:
      LPAR2 = LPAR1;                    // fill #2

                                        // test boundaries of parms
   xloop =  GetOpData (OP_XLOOP);       // get horizontal image size
   yloop =  GetOpData (OP_YLOOP);       // get vertical image size
   destincr = GetOpData (OP_DESTINCR);  // get destination ptr. increment value

   if (SPAR1 < 0 || SPAR1 >= yloop)
      return (SetError (E_PARM, ES_PARM1, "Must be smaller than image (vert)"));
   if (SPAR2 < 0 || SPAR2 >= xloop)
      return (SetError (E_PARM, ES_PARM2, "Must be smaller than image (hor)"));

   do                                // start of image processing loop
   {                                 // determine initial value of pattern:
      if ( (nLine++ / SPAR1) & 1)
         patrn = 255;                   // if nLine / parm
      else                              // is odd: patrn = 255
         patrn = 0;                     // is even: patrn = 0

      xloop = CopyImLine (lpImPtr);     // get image line
      lpSrc =  lpImPtr->src[0];         // get buffer of image line
      lpDest = lpImPtr->dest;           // get pointer to destination

      while (--xloop)
      {
         if (! (xloop % SPAR2))         // if cnt = multiple of parm
            patrn ^= 0xff;              // invert pattern
         *lpDest = *lpSrc++ ^ patrn;    // read pixel and perform operation
         lpDest += destincr;
      }
   } while (NextLine (0));              // decrease vert. counter, test if ready
   return ((long) patrn);               // return last value of pattern
}
```

xdemo2

Command syntax:
`xdemo2 [a] [#]`   XUSER example

Function
Window operation demo.
Central pixel = max - min (in window)

~~**Description . . .**~~

Description

- window size (3 (def), 5, 7 or 9)

xdemo3

Command syntax:
`xdemo3 [a] [#]`   XUSER example

Function
Random pixel access demo - draw a spiral; add randomness

~~**Description . . .**~~

Description

- spiral factor; the larger, the better (default: 32198)

xdemo4

Command syntax:

`xdemo4 [a]` XUSER example

Histogram & IBUF manipulation - calculate mean grey value of image

xdemo5

Command syntax:
`xdemo5 #1 [#2 [#3]]` XUSER example

Description
LUT & IBUF demo - load sinusoid in LUT

#1 - LUT no. (1 - <max.LUT>)
#2 - colour (1 = red, 2 = green, 3 = blue, 4 = input, 0 = RGB (default))
#3 - phase shift (0 - 359; default: 0)

xor

Command syntax:

```
1. xor a b
2. xor [a] #
```

Parameters

- constant

Return value

none

Family

Pixel operation

Function

1. XORs the pixels of two images
2. XORs the pixels of an image and a constant

Description . . .

Description

XOR performs the bitwise logic exclusive OR-function of the pixels of two images, or the exclusive OR-function of an image and a constant. Bits of a pixel are set to 0 if the corresponding bits of both source pixels (or source pixel and constant) are equal and set to 1 if they differ, see table below.

Examples

`xor a b`	a and b are XOR-ed, result is copied to the default source
`xor a b >c`	as 1., the result is copied to c
`xor 128`	the default source is XOR-ed with 128 - the most significant bit is inverted
`xor a 15`	a is XOR-ed with 15, result is copied to the default source.

xorpat

Command syntax:
```
1. xorpat [a] [#YX] [/]
2. xorpat [a] #Y #X [#pg] [/]
```

Parameters
See pattern drawing

Family
Graphic operation

Function
Draws a figure along a path, specified by a Freeman string in Ibuf.

Description . . .

Description

This function draws a figure, of which the Freeman contour string is available, by XOR-ing the pixels along the path with a bit pattern. The string consists of bytes, and has to end with 255 (0ffh).

The Freeman string is read from Ibuf. If the string is produced by **fcont**, and **xorpat** follows immediately, then **xorpat** can be instructed to read directly from the internal **fcont** buffer, by specifying the exception parameter (/). In this case the figure's default starting position is not the image's cursor position, but the original starting position.

Valid Freeman codes are: 0, 1, ...7. The following numbers have a special meaning:

Freeman code + 128	skip this pixel
255:	end of string

Examples

`xorpat`	draws a figure, specified by a Freeman string in ibuf, from the cursor position
`xorpat /`	as above, but reads Freeman string from fcont buffer and starts at original position
`xorpat a 100 200`	draws into a, reads from ibuf, starts at 100, 200
`xorpat a 100 200 /`	as above, but reads Freeman string from fcont buffer

For details on pattern drawing and parameter interpretation, see pattern drawing
See also: Concepts of graphic operations

xorvec

Command syntax:
```
xorvec [a] #a [#l [#Y #X] [#pg]]
```

Parameters
See vector drawing

Return value
number of pixels on vector

Family
Graphic operation

Function
Changes pixels (XOR-wise) along the imaginary vector.

Description . . .

Description

This function draws a vector in any direction, by XORing the pixels on the vector with a bit pattern. The length is specified in number of pixels. If no length is specified (or the length value is 0), then the vector runs to the image edge. If no starting position is specified, the cursor position is used. The default bit pattern is the graphics value

Examples

`xorvec 222`	draws a vector from the cursor position with an 222ø angle to the image edge
`xorvec a 33 10`	draws a vector in image a from the cursor position angle 33, length 10
`xorvec 10 0 128 128 1`	draws a vector from 128, 128 to the image edge, angle 10, XOR pattern 1

For details on vector drawing and parameter interpretation, see vector drawing
See also: Concepts of graphic operations

xorln

Command syntax:

1. `xorln [a] #YX1 [#YX2 [#pg]]`
2. `xorln [a] #Y1 #X1 #Y2 #X2 [#pg]`

Parameters

See line drawing

Family

Graphic operation

Return value

number of pixels on line

Function

Changes pixel values (XORwise) along an imaginary line.

Description . . .

Description

This function writes a line by XOR-ing the pixels laying on it with a pattern. XOR-ing a bit with 1 means: if the bit's value was 0, it becomes 1 and if it was 1 it becomes 0. XOR-ing with 0 doesn't change anything.

Drawing lines using the XOR function has the advantage that a line can be removed by drawing it a second time.

Examples

`xorln 110022h`	writes line between 11H,22H (Y,X) and cursor using graphics value
`xorln p 11 22 33 44 128`	writes line in most significant bit (128 = 2**7) of **p** from 11,22 to 33,44 (Y,X)

For details on line drawing and parameter interpretation, see line drawing
See also: graphics concepts

zoom

Command syntax:
```
1. zoom [p] #
2. zoom [p]
```

Parameter
1. # - set zoom to absolute value (0 = zoom out)
2. toggle zoom (increase zoom value until maximum reached, that back to 0, etc.)

Return value
(previous) zoom value

Family
Control operation

Function
Controls hardware zoom of frame grabber

Description . . .

Description

The image must be a frame grabber image (defined in `images.tim` as `dis`)

zoom controls the hardware zoom option of the frame grabber. When zooming in only display changes; the destination image will not change, as can be seen on the status line.

zoom depends upon the frame grabber's hardware properties. See Appendix F for the zoom capacities of your frame grabber, or consult the frame grabber's manual.

When the image is larger than the display window, display is centered around the cursor. This can be controlled using the pan command.

If zoom is entered without a numerical parameter, the zoom value is incremented until the maximum value is reached. Then the zoom value is reset to 0.

If the zoom factor is such, that the specified image is larger than the display window, panning is enabled.

Active Image

The active image is the currently selected image. It is the image where the results of image processing operations go to, and it is the default source image if you don't specify a source image with the command.

You can select the active image

- by clicking the image button in the Status Window
- by command dis or dest

See also images

Alias

Aliases allow you to substitute one string for another. This can be done to make a command more descriptive. See the following example (dilate bitplane 1 5 times):

`ldi 1 5` and `ldi red 5`

are equivalent, if the alias **red** is specified for **1** (bitplane 1 often appears red in the default colour setting).

Aliases are specified in a file **alias.tim**, which must be in the ***TIM*** home directory.
See also: How to ...

The TIM compiler can also be instructed to use the alias definitions. In addition, you can define local aliases (using the `#define` directive) and specify local alias files (using the `#include` directive).

Aspect Ratio

Aspect Ratio is the ratio of the vertical and the horizontal dimensions of a pixel. In ideal cases, this ratio is 1.0. Practical frame grabbers often deviate from this ideal, having Aspect Ratios of as much as 1.4.

In the European Video standard (CCIR), the pixel clock for producing square pixels should be 14MHz.
The pixel's aspect ratio can be calculated by dividing the theoretical pixel clock by the applied pixel clock.
If the latter is 10MHz, the ratio is:
14/10 = 1.4

Bit

A bit is the smallest unit of storage in a computer. It represents a binary value: 1 or 0, true or false, etc.

Bit mask

A bit mask is a numerical value, whose bit pattern represents a mask to be applied with another value. For example, take 7:

value	binary repr.
7	00000111

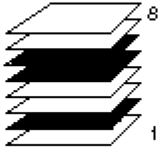
This pattern, applied to an image, would leave the 3 lowest bits of the pixels of the image, reducing the range of values in the image from 256 (0 - 255) to 8 (0 - 7)

A frame grabber may have hardware mask registers, so that bitplanes can be protected from being overwritten. This can be done both while grabbing and during host access.

Byte

A byte is a standard unit of storage in a computer. It consists of 8 bits. A byte can represent a value range of 0 to 255.

Bitplane



A pixel generally consists of one byte. A byte contains 8 bits.
In grey value images, these 8 bits are used to encode a value between 0 and 255.
In binary images (where a pixels represents a one bit value, e.g. "background" or "object") we can use the 8 bits of a grey value pixel to store as many as 8 binary pixels.

TIM has no special *binary image* type. A binary image is stored in one layer of bits of a regular grey value image. These layers are numbered from 1 (lowest bit) to 8 (highest bit).

See also the following sources of related information:

How to convert grey value images to binary images
How to make bitplanes visible

Operations that deal with bitplanes:

- Bitplane Operations,

- CLP Operations

Binary Image

A binary image contains 1-bits information, for example object/background.

TIM does not have a separate binary image type. Instead, binary images are stored in a bitplane of a standard (grey value) image.

Bitwise Binary operators

Bitwise binary oparators are: AND, OR and Exclusive OR (XOR).
These operators operate on single bits, which can have a value 0 and 1. The following definitions apply:

- the result of an AND is 1 if both the operands are 1
- the result of an OR is 1 if one or more of the operands is 1
- the result of an XOR is 1 if the operands differ

See also the following table:

Input		Result		
Operand 1	Operand 2	AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Numerical example:

```
        13          0000 1101
         5          0000 0011
-----------------------------
        AND         0000 0001
        OR          0000 1111
        XOR         0000 1110
```

Breakpoint

Breakpoints are markers in a program that cause a running command file program to stop. In the Debug window lines with a breakpoint are coloured red.

Calibration Factor

The calibration factor is a constant, that can be used to scale a return value of a TIMWIN command.

- To specify the constant, see the **Set** menu or the set command
- To perform the calibration, use the cal command in combination with the command, that produces the result to be corrected.

Clock

The frame grabber uses an internal clock for producing the video signal. This clock can come from the following sources, depending upon the configuration:

- internal, locked to a crystal reference. This is for stand alone configurations.
  Command: `set clock xtal`
- internal, locked to an external video source. This is for digitizing images.
  Command: `set clock pll`
- external. This is for special configurations (non standard video)

CLP

Cellular Logic operations are morphologic operations on binary images. These operations deal with shape properties of the object in the image. See also the family of CLP-operations.

CLP operations can:

- erode objects (remove pixels from the border)
- dilate objects (grow pixels onto the border)
- propagate objects (fill the object, starting with a seed)
- produce the contour (remove all but the border)
- skeletonize objects (thinning the object until a line or single pixel remains)
- find special pixels (see below)
- remove binary noise

The following terms are used with CLP operations:

single pixel	a pixel with no neighbours
end pixel	a pixel with one neighbour (the end of a string of pixels)
link pixel	a pixel with two neighbours (the pixels in a string of pixels)
vertex pixel	a pixel with three or more neighbours (the pixel in a branchpoint)
neighbourhood	the 3x3 pixel area that determines the outcome of the operation
connectivity	the neighbours that count in an operation (see Connectivity)

Command File

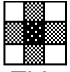
A TIM command file consists of TIM commands and other statements in the TIM language. A command file must be compiled to be executable.

Other related topics:

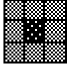
- TIM program
- TIM program module
- Creating TIM programs

Connectivity

The term connectivity specifies the number of neighbours that are involved in an operation



This figure represents 4-connectivity



This figure represents 8-connectivity

Convolution

A convolution operation is a weighted average of a pixel and its neighbourhood. The convolution coefficients determine the result of the operation.

TIM has several predefined convolution operations, as well as a user definable convolution operation (see filt)

Cursor

The image cursor represents a position in an image. Each image has an individual cursor position.

The cursor position of the active image is shown in the status bar.
The cursor position is used with many operations. It determines the location of image's sub image.

In frame grabber images the cursor can be made visible. See the curs command.

The cursor can be moved through the image by pressing the mouse button in the status bar. In this case the mouse no longer controls the program windows, but the images instead. Control is returned by pressing the right mouse button.

Image cursors can be locked between images of the same size class. This is, when the cursor of an image moves, the cursors of locked images move with it. See curlock.

Examples of cursor usage:

- defining the position of sub-images
- editing pixels
- viewing parts of the image
- drawing

Cursor value

The cursor value is a system value setting, used for drawing the image cursor in images. The cursor is drawn by inverting the specified pixel bits of the pixels belonging to the cursor shape. The default value is 128, which causes the most significant bit to be inverted.

The user can control this value as follows:

- using the Installation menu
- using the set menu
- using the set command

Destination

The destination image is the place where the results of an image processing operation go to. Usually you don't specify the destination image with the command. In TIM, the destination image is selected by special commands (dis and dest) and remains the same until a new specification is given.

Some commands write their results into the specified (source) image. These include: graphics operations, bitplane operations, CLP operations and some miscellaneous operations.

Disk Images

Disk images are images, located on disk in a file. Many TIM operations accept a disk image as a source. However, the destination can never be a disk image.

Disk images can be created using the commands copy and ~~save~~. Disk images are stored to and read from a default directory, specified in the Installation dialog box in the Contr menu.

Display Window

The display window is the part of frame grabber memory that is visible on the frame grabber monitor. Its size depends on the zoom factor.

Images may and may not have sizes that correspond with a display window. If smaller, more than one image can be seen. If larger, a part of the image is visible. See also pan

Dithering

In a dithered image only black and white pixels exist, while the grey values are simulated by varying the distance between the white pixels.
This gives the effect of a continuous grey value range in an image, that has no grey value capacities.
Examples of use:

- a display that has insufficient levels of grey for displaying grey value images (e.g. a standard VGA display)
- a printed image

Drawing value

The drawing value is a system value setting, used for drawing in images. The user can control this value as follows:

- using the Installation menu
- using the set menu
- using the set command

Operations that use the drawing value replace the entire pixel value. This is in contrast to the behaviour of operations, that use the graphic value

Frame Grabber

A frame grabber is a special hardware device, which can be placed in one of the computer's insertion slots. With a video camera connected, it is capable of reading in images by digitizing the camera's video signal.

Also a video monitor can be connected, which allows you to watch the content of the frame grabber's memory directly. Usually frame grabbers offer special display options, which go beyond the possibilities of. standard computer's display adapters.

Some frame grabbers offer processing capabilities in real time.

See also: zoom, LUT, pan, real time operations, How to . . .

Frame Grabber Look Up Tables

The frame grabber's look up tables (LUTs) offer facilities for converting pixels values while the pixels flow in or out.

Input LUTs convert pixel values while grabbing an image. This allows you to perform pixel operations like contrast stretch or thresholding while the image is captured.

Output LUTs convert pixel values before they are sent to the display monitor. This allows you to view results of operations without modifying pixels (see preview).
There are independent output LUTs for each of the output colours: red, green and blue.This allows you to obtain special colour effects.

You can control the frame grabber's LUTs using the lut command. Several commands produce tables, that can be loaded into the frame grabber's LUTs: see ~~table~~. Output LUT functions are also available for Windows images. Since this is achieved in software, speed is much lower.

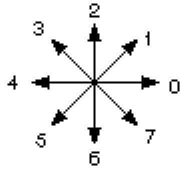
Frame Grabber No.

TIMWIN supports more than one frame grabber in one computer. Two or three frame grabbers allow you to grab images concurrently, in stead of sequentially.

Although only one of the frame grabbers can be active at a time, you can select them one after each other, and set them up for grabbing. On the next video frame, they will start grabbing together.

The Frame grabber no. setting allows you to make one of the frame grabbers active.

Freeman



Freeman codes are numbers from 0 to 7 that specify a direction.
TIM commands that have direction dependency use a freeman code to specify it.

Freeman codes can also be chained to indicate a path in an image. To produce a Freeman chain from an image contour, use the fcont command.

Gain

In some image processing operations gain is expressed exponentially:

parameter	gain
1	1
2	2
3	4
4	8
5	16
6	32
7	64
8	128

Graphics

It is often desirable to present data in Ibuf graphically. TIMWIN can produce graphics in two ways:

- By writing a graph in a special graph window.
- By writing a graph in an image

To display the graph window:

1. From the View menu click Graphics
   The present content of Ibuf will be shown graphically. You can size and position the graphics window.

To refresh the graph automatically whenever the data it represents changes, check the appropriate item in the graphics window's **Update** menu:

- **Ibuf** if you want to see the content of Ibuf (except histograms) whenever it changes
- **Histogram** if you want to see the grey value histogram after each image processing operation.

To write a graph in an image:

See the commands graf and grav

Graphic value

The graphic value is a system value setting, used for writing graphics in images. The user can control this value as follows:

- using the Installation menu
- using the set menu
- using the set command

Operations that use the drawing value replace only the specified bitplanes in the pixels, using an XOR technique. For example:

if youspecify	bitplane to be written:
1	1 (least significant)
2	2
3	1 and 2
4	3
....	
128	8 (most significant)
255	all

This is in contrast to the behaviour of operations, that use the drawing value

Hexadecimal

The hexadecimal numerical base is often used in technical systems, because of its direct relationship with binary information: bits and bytes.

The hexadecimal system has 16 digits: 0 - 9 and a - f. A hexadecimal number must be interpreted as follows:

2A8 = 2x256 + Ax16 + 8x1 =   2x256 + 10x16 + 8x1 = 680

In TIMWIN hexadecimal numbers are distinguished from normal decimal numbers by preceding them by 0x or appending an H or h. So the above number should be written: 0x2A8 or 2A8h

Histogram

A histogram is plot, that shows the frequency of occurrence of an event. For example, the histogram of an image is a plot of the numbers of all pixel values in the image.

IBUF

Ibuf is TIM's internal cut & paste buffer. Several operations put data into Ibuf, while others read data from it. By executing commands in a clever order, you can move data in your processing sequence, and thus take advantage of these properties.

To view the data in Ibuf, open the Ibuf window:

- See the Edit Ibuf Menu
- Or, on the command line, enter the command `editi`

To view the data in Ibuf graphically, open the graphic window:

- See the View Graph menu

Commands that interact with IBUF are, for example:

- All table operations (they create their table in IBUF)
- The rdln command, which reads an image line and stores it into IBUF
- The ribufand wibuf file read/write commands, that load/store data from/to files
- The graphic functions, that read plot data from IBUF.

Images

An image is an amount of data, organized in 2 (or more) dimensions, generally representing a scene from the real world. In TIMWIN, the term **image** is used to indicate storage space in memory or on disk, where *image data* is stored.

An image is usually the source and destination of an image processing operation. The available images are user specified in a file **images.tim** (see How to . . ..)

The following image classes exist:

- Frame grabber images — images located in the frame grabber
- Memory images — images located in computer memory.
- Windows images — memory image which are visible on the computer screen

Images can also be located on disk. Disk images can be accessed by file name. See Install to set the default image directory.

Images have properties:

- a name (usually a single letter)
- type
- size (horizontal & vertical)
- a cursor (pointing to a position in the image)

Each image has a set of sub-images associated with it.

Usually, in a session you use images of the same size, to be able to copy images back and forth without resolution loss. Therefore, only images of the currently selected size are visible in the status bar.

Image mode vs. Windows mode

In **Image mode**, the mouse controls image attributes: image cursor, sub image format, image drawing, etc.
In **Windows mode**, the mouse controls the Windows cursor.

To switch from Windows mode to Image mode:

- Click the Cursor button on the Status bar
- Or, position the Windows cursor on the (Windows) image and the click the right mouse button.

Then, click the left mouse button to select an image cursor

To switch from Image mode to Windows mode:

- Click the right mouse button.

Image size

The number of pixels necessary to represent an image, depends upon the desired spatial resolution. The greater the number of pixels, the greater the accuracy in describing the image spatially. However, bigger images use more memory and greater processing time.

TIM images are devided in the following size classes, based on the horizontal image size:

Class	Hor. size range	standard size
1	1 - 256	256x256
2	257 - 384	256x384 *)
3	385 - 512	512x512
4	513 - 768	512x768 *)
5	769 - 1024	1024x1024

*) These sizes can be used with square pixel frame grabbers
To switch from one image size class to another, just select an image of the desired size.

Once you select an image size class to work with, TIM shows only images of that size. The other images (smaller and bigger) are also available, but using them may introduce side effects due to size mismatch. See the Geometric Operations for operations that convert image sizes

The current image, its size and the other family members are shown in the status bar.

In addition to the standard image formats, TIM offers the possibility to define inside an image any smaller sub image.

Image Type

The pixels in an image are usually represented by bytes. For special purposes other representations are possible. The following table shows the current pixel representations in TIMWIN

pixel size	name	description
8	byte	standard pixels
12	word	pixels in 12 bits frame grabbers
16	word	pixels in 16 bits images
64	complex	pixels in complex floating point images

An image processing operation determines which way it accesses the pixels.
A byte-oriented operation accesses bytes, independent of the image type. E.g., in a 16-bits image only the least significant byte is used with these operations.

Examples of 16-bits operations: era16 , cp16 , sumExamples of 64-bits operations: fft

Line drawing

Drawing lines takes place by specifying two coordinate sets. TIM has a flexible way of interpreting this data, which depends upon the number of parameters specified. For example, if only one parameter is specified, it is interpreted as a packed XY value. Also, if a second coordinate set is not part of the specification, TIM uses the image's cursor position.

The following table shows how the specified parameters are interpreted.

Number	#1	#2	#3	#4	#5
1	XY1		(XY2 = cursor pos.)		
2	XY1	XY2			
3	XY1	XY2	grey value		
4	Y1	X1	Y2	X2	
5	Y1	X1	Y2	X2	grey value

Defaults:

- line end point: the cursor position
- grey value: ~~drawing value~~ for draw (dr..) commands, ~~graphics value~~ for other (or.., xor..) commands. The rdln function does not use the grey value parameter

Long integers

In computer lingua, a long integer is a 32-bits value. In memory , a long integer takes 4 bytes. A long integer can represent a value range of 0 to 4.294.967.296, or -2.147.483.648 to +2.147.483.647

Another term for long integer is: **double word**. Don't confuse this with **double**, which means double precision floating point.

Look Up Tables

Look up tables offer a powerful way for converting data. In TIM, look up tables are used in many places for several purposes. For more information, see:

Table look up image processing operations

Look Up Tables in the Frame grabber

The Preview function

IBUF

Numerical parameters

Numerical parameters are used to further specify the function of a command. Sometimes a numeric property is meant, as in:

`add a 10`	add 10 to the pixel values in image a

Sometimes an enumerated option is meant, as in

`set 5 0`	set the frame grabber clock in crystal mode

In this case, the use of aliases can make the command more legible:

```
set clock xtal
```

Overflow

The result of a pixel operation may be too big for a pixel's storage space (generally a byte). In this case overflow occurs. The general action is:

- If the result was < 0 (negative): the result is set to 0
- If the result was > 255 : the result is set to 255

Also, an overflow counter is incremented. Some operations return this overflow counter, to indicate that overflow occurred.

Packed value

Packing a value is a method to 'pack' two integer values into one value. Several TIMWIN commands (e.g. curs) return a packed X and a Y value.

Packed values can be useful in command file programs, to reduce the number of statements and variables. See the following examples, that copy the cursor position of one image to another:

Packed	Not packed
`int curpos`	`int cx, cy`
`curspos = curs a`	`cx = cursx a`
`curs b curpos`	`cy = cursx a`
	`cursx b cx`
	`cursy b cy`

If you want to specify a packed value manually, the use of the hexidecimal format may be handy, since this format allows recognition of the separate X- &Y-part in a packed value.

The formula used for packing Y and X is: 65536Y + X
Many commands accept packed parameters.

Pan

Panning means: moving around the display window in an image. This is possible when the display window is smaller then the image is:

- when zoomed in
- with images which are larger than the maximal display window.

See also pan, zoom

Pattern Drawing

Pattern drawing takes place by following a line pattern, which is determined by a Freeman chain code.
The path can be 'learned' using the fcont command, or created otherwise.
The starting point can be user specified, or be the same as the learned curve's.

Pattern drawing functions read the Freeman chain from one of two places:

- from lbuf. In this case you have to specify the starting point for the drawing action. This is the default situation
- from an internal buffer (where fcont stores the Freeman chain). In this case the starting point corresponds to the starting point of the original line or contour. Specify the exception parameter (/) for this function. Notice, that this buffer may be overwritten by subsequent operations.

The following table shows how the command parameters are interpreted:

Number	#1	#2	#3	
1	XY			(packed starting point)
2	Y	X		
3	Y	X	grey value	

Defaults:

- starting point: the cursor position
- grey value: ~~drawing value~~ for draw (dr..) commands, ~~graphics value~~ for other (or.., xor..) commands. The rdpat function does not use the grey value parameter

See fcont for information how to obtain a Freeman chain.

Pixel

A pixel (picture element) is a single image element. The size of an image is expressed in pixels.
A pixel represents a vital property of the image. This can be:

- grey value
- colour
- object/background, etc.

In the computer, a pixel is represented by one or more bytes.

PostScript

PostScript is a graphic description language, that is recognized by many printers and text processors. If you make a PostScript file of an image, you are able to export it to other devices.

The PostScript file itself has all image details; it depends on the quality of the output device how the result looks.

TIMWIN is able to produce the following PostScript formats:

1. **.PS** files, that can be sent to a printer imeediately.
2. **.EPS** files, that are suited for importing in text processors.

PostScript is a trade mark of Adobe Systems Incorperated

Post Transport

After an image processing operation is finished, the resulting image can be copied to any other image (including disk images). Post transport is specified by appending the post transport character (**>**)and an image name after the command. Example:

```
add a 10 >b
```

Preview

The preview option allows you to see the result of table operations immediately, without modifying the image. Thus you can interactively adjust parameters, until the result satisfies you. Then you can actually perform the operation by selecting the OK button.

Operations that allow previewing have a Preview button in the dialog box.

Real Time

Some frame grabbers have facilities to perform real time operations. This means that pixels are converted while being acquired.

The following classes of real time image processing exist

- *table look up conversion*
  This type of processing takes place when the digitized data is fed through a look up table before stored in memory. Thus you can perform all table operations in real time. All frame grabbers support this mode.
- *RT-processing*
  This type of processing requires some kind of hardware processor in the frame grabber. You can combine current and previous images in this mode. Thus you can integrate, differentiate or produce real time edge images. Frame grabbers of the Series 100 (VS100, VFG) support RT mode.

Scaling

Scaling takes place with graphic operations. When data in Ibuf is plotted, the values are converted to bytes, and the size is brought within the range 0 - 256. If you want to control scaling, you can supply a parameter. Its meaning is:

parameter	division by
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

Autoscaling takes care of correct scaling automatically.

Set Update

Several TIMWIN windows that display data, can be forced to automatically update their display when the corresponding data changes. For this purpose the windows have an Options menu.

In order to be able to control this facility from the command line, the set command has a function update (15). In the parameter each window is specified with a bit:

bit	value	update window
0	1	Ibuf edit
1	2	image edit
2	4	Ibuf graphics
3	8	statistics
4	16	histogram in Ibuf edit
5	32	DDE transaction on Ibuf
6	64	histogram graphics

Specifying one of the above values (e.g. `set 15 1`) starts updating, but leaves settings of other windows undisturbed.
To remove a setting, specify a negative value (e.g. `set 15 -1`).
To remove all settings, specify 0 (e.g. `set 15 0`)
You can also use an alias (e.g. `set update ibuf_on`)

Source

A source image is the image from which an operation reads the pixels. The available source images can be seen in the status bar. In many cases also disk images can act as a source image.

Special parameters

Special parameters control options in certain commands. The following special parameters exist:

/	exception parameter. See the command description for details

Cursor shape parameters. Only the first two characters are significant

bo(x)
cr(oss)
ch(crosshair)
ar(row)
of(f)

Compare parameters, used with the **comp** and **sel** commands

==	equal
!=	not equal
<=	less than or equal
>=	greater than or equal
<	less than
>	greater than

Square pixels

A pixel is said to be square if its horizontal and vertical size are equal. Depending upon the frame grabber's properties, pixels can be square or not.

Performing measurements on images having non-square pixels is difficult. TIM has facilities for correction of the effects of non-square pixels. See also: Install and the dim operation.

String

A string is an array of characters. Generally a string is used where a piece of text must be handled. To indicate a text string, encose it in quotation marks (").
Example: "This is a text string"

Status bar

The status bar is a window that gives information about images and frame grabber properties. You can control the shown properties by pressing buttons, filling in numbers, etc.

To make the status bar visible do one of the following:

- Press ALT+S
- In the Windows menu, press Status

Stepsize histogram

When the grey value histogram is produced, the number of occurrence of each grey value is counted. To make this prcedure faster, pixels can be skipped from this procedure. The histogram stepsize setting allows you to define this.

Histogram stepsize	Meaning
1	take all pixels
2	take every other pixel
3	take 1, skip 2, etc.

Sub Images

A sub image is an image inside a standard image.

TIM has two types of sub images:

1. standard sub images, that have a size of one quarter of the standard image, and are located in the four quadrants of the image
2. variable sub images, that can have any size and position inside the standard image. Its location is determined by the cursor position. The images of a certain size group (see image size) share a sub image size.

Sub images have the same name as the standard image. Standard sub images have a number appended (1, 2, 3 or 4 depending upon the quadrant), variable sub images have a **c** appended:

a1, a2, a3, a4 are 4 standard sub images of image **a**
ac is the variable sub image of image **a**

See also: frmt, curs

Syntax

The TIM command syntax description shows how to enter a TIM command in the edit window or in command files. A symbolic notation is used, like:

```
comm [a] #1 [#2] [/]
```

Command
The first item is the command name.

Images
The second item `[a]` is the source image. The brackets `[]` indicate that the image specification may be omitted. If this is the case, the default source image will be used. Any available image is valid.
Often disk images are also allowed.
In special occasions, properties for different types of images vary. The following general names are used to indicate image types:

a, b for general images. If 16 is added, it must be a 16-bits image
p for images located in a frame grabber
w for images with a windows display attached

Numerical parameters
Items containing the pound character (#) indicate numerical parameters (the pound character is often used to represent a numerical value). When enclosed in brackets a specification may be omitted. If this is done, a default value will be used.

A specification looks like: `#1 [#2 [#3]]`
This means: you may specify either `#1,` or `#1` *and* `#2,` or `#1` *and* `#2` *and* `#3`
To distinguish numerical parameters, notational conventions are defined.

Special parameters
Other parameters are possible (for example: `[/]`). They are explained with the command.

Strings
Some commands accept a text string as an input. To be able to distinguish a text string from a disk file, the text string must be enclosed in quotation marks.

Syntax (notational conventions)

In the command descriptions numerical parameters may be shown using the following abbreviations:

#YX	packed address
#Y, #X	address, Y- or X-coordinate
#a	angle
#b	bitplane
#d	delay (number of frames @ 40ms)
#f	floating point value
#F	Freeman code
#g	gain
#h	horizontal multiplication factor
#l	length
#m	bitplane mask
#n	repetition factor
#p	pixel value (0 - 255)
#pd	pixel value (default: drawing value)
#pg	pixel value (default: graphics value)
#r	radius
#s	scaling (or shift) factor
#t	threshold value
#v	vertical multiplication factor
#w	window size (square)
#wx, #wy	window size (hor, vert)

Table

A table is an array of data, that is used to convert one value into another. For example, to convert a value into its square root, you can make a table of square roots:

```
        sqrt (0)
        sqrt (1)
        sqrt (2)
--->    sqrt (3)
        . . .
```

To find a value (say, sqrt 3), just look at the corresponding position in the table (using the value to convert as an index).

Using a table for conversion is advantageous when the same value must be calculated many times (as in image processing).

Threshold

Thresholding a grey value image is making it binary, e.g. dividing it into object and background pixels. This is done by making the pixels having a grey value below the threshold level black, and above that level white.

TIFF

TIFF is a specification for a header of bitmapped images. The header contains information about the image.

TIFF images are produced by scanners and various software products, and can be read by many programs.

An advantage of using TIFF images internally in TIMWIN is that the image's properties: (size and pixel format) are available.

TIFF images have the `.tif` file extension

TIM Commands

TIM commands specify an action for image processing or control. TIM commands are divided in categories:

- the family category, which combines operations based on algorithm.
- the application category, which combines operations based on a typical application sequence.

TIM Program

A TIM program consist of one or more compiled command files. It can be a single module, or consist of several modules that call each other.

A TIM program module is a single compiled command file.

A command file is a source file, from which a TIM program module can be made.

TIM Program Module

A TIM program module is a single compiled command file. It can be a stand alone program or part of a larger TIM Program.

True & False

In TIM programs the terms TRUE and FALSE are used for binary values. Since there is no binary data type, an integer is used to represent TRUE and FALSE in the following way:

> FALSE is 0
> TRUE is any other value

Vector Drawing

Drawing a vector takes place by specifying a starting point, an angle and a length. TIM has a flexible way of interpreting this data, which depends upon the number of parameters specified.

The following table shows how the specified parameters are interpreted. The first column indicates the number of parameters:

Number	#1	#2	#3	#4	#5
1	angle				
2	angle	length			
4	angle	length	start.pos. Y	start.pos X	
5	angle	length	start.pos. Y	start.pos X	grey value

Parameter range:

- angle: 0 - 360

Defaults:

- starting position: the cursor position.
- length: to the image border (parameter value: 0)
- grey value: drawing value for draw (dr..) commands, graphics value for other (or.., xor..) commands. The rdvec function does not use the grey value parameter

Video Gain & Offset

The AD-converter in the frame grabbers converts analog voltage levels into digital numbers between 0 and 255. There is one distinct level that will be converted to 0, and one other that will be converted to 255.

The video gain and offset adjustment in a frame grabber allow you to adapt these levels to the levels of your video source. If this adjustment is software controllable, you can use TIMWIN's **set** function for this purpose.

The offset setting allows you to match the 0-level of frame grabber and input.
The gain setting allows you to match the 255-level of frame grabber and input.

Video Input channel

A frame grabber can have more than one video inputs. A TIMWIN setting, the video input channel, determines which input is actually used.

Word

In computer lingua, a word is a 16-bits value. In memory, a word takes 2 bytes. A word can represent a value range of 0 to 65536, or, when signed numbers are needed: -32768 to 32767.
In some cases, TIMWIN uses words to represent pixels.

XUSER

The TIMWIN XUSER system is a facility to add user written code to TIMWIN. It is available to those, who want to write specific image processing functions, and use them in the context of an existing program.

For XUSER programs a set of library functions are available, that free the programmer of the tedious work like writing the user interface, dealing with the frame grabber hardware, etc.

An XUSER function behaves like any other TIMWIN function. For example, it can be used in command file programs.

The standard TIMWIN program has a few functions (xdemo1 - xdemo5). Each of them demonstrate an aspect of TIMWIN. See the description of the commands for details.

Zoom

Zooming is concentrating the frame grabber's display window onto a smaller area of the image. Depending upon the frame grabber's properties, zoom factors of 1, 2 and 3 are possible.

If the display window is smaller than the image, then a part of the image is shown, such that the image cursor is in the middle of the window.

The current zoom factor is visible in the status bar.

See also: the commands zoom, pan

TIM Command File Program

A TIM program consists of one or more compiled command files, that can be run in the TIM environment.

Procedures

- Building a program
- Compiling a program
- Executing a program
- Debugging a program

Anatomy of a command file

- Introduction
- Declarations
- Statements
- Flow control
- Modules
- Comments

Reference Information

- Language reference

Procedures for creating TIM programs

- Building a program
- Compiling a program
- Executing a program
- Debugging a program

Editing a source

A command file program is written using an editor. ***TIMWIN*** has its own editor, EditCF. It has several useful features for editing command files.

To write a program, do the following:

1. Choose the CommFile menu
2. In the Command File dialog box, fill in the name of the command file you want to create
3. Press the Edit button. An empty file is created, and the editor you selected in the Install menu is invoked.
4. Write the command file
5. When done, save the file.

Compiling a command file

Before a command file can be executed, it must be compiled. Once a compiled version of a command file exists, the source file is not necessary anymore, except for debugging purposes, or to make changes.

Compiling can take place using either of the following methods:

1. Compiling using the Command File Dialog box
2. Compiling automatically
3. Compiling using the stand alone compiler

Compiling using the Command File Dialog box

To compile a command file using the Command File Dialog box:

1. Open the dialog box by clicking CommFile on the menu bar
2. Select a command file in the file list box
3. Click the Compile button

The Compiler messages appear in the main windows' message area.

Compiling automatically

If you specify a command file for running, which is not ready for execution, TIM does one of the following:

- it refuses to run the command file
- it prompts you with a message box, asking "Do you want to compile this command file?"
- it compiles the command file silently

Which of the above is chosen depends on the selected properties in the Installation menu. To set the desired mode:

1. In the Contr menu, select Install
2. Click the Compiler Options button
3. In Update Compiler Options check one of the boxes

Compiling using the stand alone compiler

You can run a stand alone version of the compiler, outside TIM. This can be useful if you want to recompile all of your command files. See the description of the stand alone compiler for details.

Executing a TIM program

A TIM program can be executed by typing its file name in the work top. To distinguish a command file from a regular TIM command, precede the name by a / or a *. Thus:

```
*mycfile
/mycfile
```

are both correct commands.

If the command file requires arguments, just add them as with a regular TIM command. If the number or type of the parameters is not correct, the command file is aborted and an error message appears.

You can run a command file in standard mode and in debug mode. To run a command file in debug mode, add 'debug' to the command string as in:

```
*mycfile debug
```

See also the following sources of related information:

- User interaction with a command file program
- Interrupting a command file program
- Handling errors
- Issuing TIM commands while a command file program is running

User interaction with a TIM program

A running TIM command file is halted in the following situations:

- When it is waiting for user response (mouse or keyboard)
- When a user interrupts the program deliberately

In this situation a user has limited facilities to deal with other aspects of TIM. For example:

- issue TIM commands
- change display
- compile command files

etc.

Interrupting a command file program

To interrupt a running command file you can:

- Press the Esc key. This will finish the current instruction
- Press Ctrl-Break (holding down the Ctrl key while pressing the Break key). This will immediately interrupt whatever action takes place.

Both actions will bring up the Break dialog box, in which you can specify how to continue:

Break Dialog box

The Break Dialog box comes up when you interrupt a running command file program by pressing Esc. You can specify how to continue by checking one of the following radio buttons and then pressing OK:

Debug	to continue in debug mode.
Quit	to stop the running command file. This brings you one step higher in the calling tree - after the point where the running command file was invoked
Quit all	to stop executing all command files. This brings you back at the TIMWIN prompt
Continue	to continue in the current executing mode

Handling errors

Issuing TIM commands while a command file program is running

Since Windows is multi-tasking, you can perform other tasks when a TIM program is running. You could, for example, enter a TIM command in the work top.
This is not a good idea, because this will certainly interfere with TIM commands executed by in the command file. However, when a command file is halted, it can be very useful to enter TIM commands manually, for example to correct for a minor malfunctioning.

To do so:

1. Put focus to the TIM work top by clicking in the window
2. Perform the necessary action. You can only run single commands in this mode, no TIM programs .
3. When you are done, bring focus back to the TTY window and continue the TIM program

Realise, that interfering in a running program is potentially dangerous.

Debugging a program

Debugging is removing errors in a program using special debugging tools. To debug a TIM command file program the following conditions must be met:

- The source file must be available
- The program must be compiled with the debug flag set.

To compile in debug mode do either of the following:

1. In the Control menu select Installation
2. Click the Compiler Options button
3. Check the Debug Version check box

Or

In the CommFile menu, activate the Debug check box

To learn more about debugging, see the following topics:

- The Debug window
- Breakpoints
- Watch variables

Debug window

The debug window contains a fragment of the source code of the running TIM program. It shows the current instruction in inverse video.

The debug window comes up automatically whenever a TIM program is started in debug mode. It contains the following menu items:

Debug	View	Step!	Trace!	Animate!	Go!	Quit

Debug	control of breakpoints and variable watch
View	controls the visibility of the watch window
Step!	Step through program (step over subroutines and modules)
Trace!	Trace through program (trace into subroutines and modules)
Animate!	Automatic trace (approximately. 3 instructions/second)
Go!	Run program at full speed (end debug mode)
Quit	Stop executing program

Note: the exclamation mark (!) in some menu items indicates: immediate action

Breakpoints

Breakpoints are markers in a program that cause the program to stop when executing at full speed. In the Debug window source lines with a breakpoint appear in red.

Breakpoints can be set in either of the following ways:

1. Open the breakpoint dialog box by clicking Debug in the main menu of the Debug window
2. In the Line number edit control, enter the line number where you want to install a breakpoint
3. Click the Set button
4. Repeat this procedure for other breakpoints
5. When done, click Done

Or

In the Debug window, double click the source line where you want to install a breakpoint

See also:

- removing breakpoints
- disabling breakpoints

Removing Breakpoints

A breakpoint can be removed when it is not needed any longer. To remove a breakpoint, do the following:

1. In the Breakpoint dialog box select the item to be removed by clicking it
2. Click the remove button
3. Repeat the procedure for other breakpoints
4. When done, click Done

Disabling Breakpoints

Disabling a breakpoint is temporarily making it inactive, without removing it from the administration. To disable a breakpoint, do the following:

1. In the Breakpoint dialog box select the line number to be disabled by clicking it
2. Click the Disable button
3. Repeat the procedure for other breakpoints
4. When done, click Done

Watch variables

The watch window allows you to watch variables in a running program in debug mode. To set up the Watch window, do the following:

1. In the main menu of the Debug window, click Debug
2. Open the Watch dialog box by clicking Watch
3. In the Variable name edit control, fill in the name of the variable to be watched
4. If the variable is an array, enter the index where you want to start watching
5. Click the Set button
6. Repeat this procedure for other variables
7. When done, click Done
8. In the main menu of the Debug window, click View
9. In the View menu check Watch. This will bring up the Watch window

See also:

- removing variable watches
- disabling variable watches

Removing Watch variables

To remove a watch variable, do the following:

1. In the Watch dialog box select the item to be removed by clicking it
2. Click the Delete button
3. Repeat the procedure for other variables
4. When done, click Done

Disabling Watch variables

Disabling a watch variable is temporarily stopping with updating it, without removing it from the administration. To remove a watch variable, do the following:

1. In the Watch dialog box select the item to be disabled by clicking it
2. Click the Disable button
3. Repeat the procedure for other variables
4. When done, click Done

ANATOMY OF A TIM PROGRAM

A TIM program generally consists of the following main parts:

- Introduction
- Declarations
- Statements
- Flow control
- Modules
- Comments

Comments

Comments are separated from the program text by a semicolon (;). Everything between a semicolon and the end of the line is considered a comment and is ignored by the compiler.

It is advised to comment your programs liberally. Add a good description of the program in the header, containing information about :

- what the program does
- which conditions it expects to be present
- which parameters it needs
- which value it returns, if any

Example:

```
; sample  --  example command file header
;
;Function:  Shows how a command file header should look
;Expects:   system initialised using /init, camera ready
;Syntax:    sample number image print
;           number - number of iterations
;           image - destination image
;           print print results (1 = yes, 0 = no)
;Returns:   nothing
;*******************************************************
```

Declarations

You must "declare" every variable in a TIM program by stating its name and type before it is used. If you refer to an undeclared variable, the compiler displays an error message when you compile the program.

If the program accepts parameters, they must also be declared in this phase of the program.

A declarations consists of a type, a name and optionally an initialising value. After the variable is declared, it is used in the program by its name.

Below a few typical declarations follow.

```
int value
float accval = 1.0
file workf = "c:\\tmp\\workfile.tmp"
string text = "This is a static string"
char space[100]
parms            ;declaration of parameters
  int counter
  float f_values[100]
  file filename
endparms
```

See also:

- pre-processor directives
- constants
- variables
- data types
- Arrays
- Visibility
- Conversion

Preprocessor Directives

It is often handy to be able to use names and constants without declaring a variable for them. This can be done using the #define preprocessor directive. Example:

```
#define FALSE 0
#define TRUE 1
```

Wherever the string FALSE occurs in the program, the compiler substitutes the value 0. This construction can improve your program's readability. Be aware, however, of substituting TIM keywords, as this may cause strange behaviour of your program. Example:

```
#define max 10
```

Since 'max' is a TIM command, replacing each occurrence of the string 'max' in the program by '10' will prevent the program of executing the 'max' command.

It is good practice to keep preprocessor directives in upper case, to distinguish the from keywords and variables. Preprocessor directives must be placed in the beginning of the program, immediately after the program header.

Constants

Constants - values that don't change during the life time of a program - can be numbers, characters or strings. Your program can also define "symbolic constants", which are names that represent constant values.

This section describes:

- numeric constants
- string constants
- symbolic constants

Numeric constants

A numeric constant can have any basic data type and can be specified in decimal, hexadecimal or octal notation. The following table shows how to specify numeric constants.

`255`	decimal int
`0xff`	hexadecimal int
`0ffh`	hexadecimal int (compatible with previous versions of TIM)
`377`	octal int
`12.34E2`	floating point (scientific notation)
`-.1234`	floating point

A numeric constant always starts with a numeric character or a sign character. Hexadecimal constants use the alphabetic characters **a - f** to indicate the values 10 - 15.

A floating point constant contains either a decimal point or an exponent preceded by `e` of `E`.

String constants

A string constant is 0 or more characters enclosed in double quotes:

```
"This is a string constant"
""
```

The second example is an empty string: 0 characters between the double quotes.

Notice: Since the TIM language is based on the C-language, some C-conventions have to be obeyed. These regard the exception characters, which are indicated by preceding them with a backslash character (\). This interferes with the backslash as a directory separator in path names. To produce a correct path name, you must use two backslashes instead of one:
`"c:\\tim\\cmd\\ini.cmd"`

Symbolic constants

A symbolic constant is a user-defined name that represents a constant. Symbolic constants are usually typed in upper case. For instance, the directive

```
#define PI 3.14
```

declares a symbolic constant named `PI`.

Using symbolic constants can make your program more legible by replacing magic numbers with meaningful descriptions. They are more efficient than using variables, because they are entirely handled by the compiler.

Variables

A variable is a data item that can be modified. It has the following properties:

- a type
- a user defined name
- it must be declared in the beginning of the program
- it may be initialised. If not, it is automatically set to 0.

Examples:

```
int value
int start = 100
float calibr = 123.45
file myfile = "c:\\tim\\cmd\\ini.cmd"
```

Data types

There are different representations of data. To be able to handle this TIM uses several data types. The following table shows the TIM data types and their properties.

Type	size	range
char, byte, pixel	1 byte (8 bits)	0 to 255
int	4 bytes (32 bits)	-2.10+6 to 2.10+6
float	8 bytes (64 bits)	2.2E-308 to 1.8E+308
short int	2 bytes (16 bits)	-32768 to 32767
short float	4 bytes (32 bits)	1.2E-38 to 13.4E+38
string	undetermined	
file	undetermined	

Integer variables are used for normal numerical work. Floating point variables are used when fractional values must be represented, or when the range of an integer falls short.

Numerical (integer and floating point) variables may be initialised when defined. If not, TIM assigns them the value 0.

The character data type is used most to store ASCII characters. Sometimes integer values are also stored in the char type. Notice that char is always considered unsigned.

A string variable consists of a set of characters, and must be initialised when declared. It is read-only, so you cannot later assign another value to a string variable. String variables are used to print standard text, etc.

A file variable is used to access files. It may be initialised when declared, but it can also be assigned a name later.

The keyword `short` can be added to the declaration, but only when declaring an array. This is usually done to save space.

Arrays

An array is a group of data items that share the same type and a common name. You can make an array of any data type. An array is declared as follows:

```
int my_array[100]
```

This is a declaration of an array of 100 integers. Arrays are initialised to 0, unless you specify another value:

```
float this_array[10] = 1.1, 2.2, 3.3, 4.4, 5.5
```

Here the first 5 items of `this_array` are initialised.

See also:

- accessing array elements
- multidimensional arrays
- organisation of multidimensional arrays
- the Ibuf array

Multidimensional Arrays

In TIM arrays can have as many as 10 dimensions.

A multidimensional array is declared as follows:

```
int multi_arr[3][4][5]
```

This is a 3-dimensional integer array with 3x4x5=60 elements.

Accessing array elements

An array element is accessed by specifying its name and an index expression. The index expression must contain as many index elements as the number of dimensions of the array.
Indexes **start with 0**, and the highest index number is 1 less than the size of the array (or dimension).

Example:

```
float this_array[10]          ;definition of an array of 10 floats
this_array[0] = 0.0           ;first array element
this_array[9] = 0.0           ;last array element
val = object[(num_objects - ill_objects)*2]
arr[thre]++
```

As the example shows, in index expression can be any expression, including TIM image processing commands, provided that they return a proper value.

Accessing an array element with an index which is outside the valid range results in a run time error.

Organisation of multi dimensional arrays

In a multidimensional array(`multi_array[3][4][5]`), the elements are ordered as follows (the numbers in parentheses indicate the linear order of the element):

`multi_array[0][0][0]`	first array element (0)
`multi_array[0][0][1]`	next array element (1)
...	
`multi_array[0][0][4]`	last element of first dimension (4)
`multi_array[0][1][0]`	first element of second dimension (5)
`multi_array[0][1][1]`	next element (6)
...	
`multi_array[0][1][4]`	(9)
...	
`multi_array[0][3][4]`	(19)
`multi_array[1][0][0]`	first element of third dimension (20)
...	
`multi_array[2][3][4]`	last array element (59)

Array names can also be used in a program without an index. In this case the array is referenced as a whole. Example: when passing an array as a parameter to called command file.

The Ibuf array

In command file programs you can access the Ibuf array as a declared array, for example:

```
value = ibuf[10]
```

This method is much more efficient than using the TIMWIN ibuf command method.
To do this, the compiler automatically creates an array declaration in the form:

```
int ibuf[256]
```

TIMWIN's image processing operations regularly read and write this array, and change its data type. Whenever you access Ibuf using the array method, Ibuf's data type is set to **long** , and Ibufs content will be converted to **int**, if necessary (the long data type in Ibuf corresponds to int in command files). Thus, after accessing Ibuf in a command file using the array method, Ibuf's content may be changed (the type, not the values).

Visibility

Variables and arrays are visible in the entire program. A variable cannot be defined local within a part of the program. However, variables in a TIM program that is called from another program are invisible to the calling program.

Conversion

Variables from related types are converted from one type into another when necessary. Examples:

```
int counter
float newvalue
newvalue = counter + 1.1

char vartext[30]
file piet
string nonsense = "tvas brillig"
vartext = nonsense
piet = vartext
```

In the first example `counter` is converted to floating point before the calculation takes place. In the second example the string is copied to the character array `vartext`, which is then assigned to the file variable `piet`.
Note, that numerical types cannot be converted to string types and vv.

Statements

The main part of a program consists of statements. Statements tell the program what to do, one statement per line. Examples:

```
thre p                                  ;image processing
value = (value_old + 10) / 3            ;assignment and calculation
mean_area = (comp p > 100)/(label p)
                                        ;assignment and complex expression
```

In TIM a statement occupies a single program line. A statement may consist of a single expression, or a combination of more expressions.

Combining expressions into complex statements leads to a more efficient program, but can make your program more difficult to read and maintain. When assembling statements consisting of one or more image processing expressions, be sure to:

- define the image processing expression using parentheses.
  This is necessary because of TIM's variable number of arguments
- realise the order of evaluation of the expression.
  In the 3th example the comp operation must take place before the label operation. This is the case because the statement is evaluated left to right.

Expressions

An expression is an entity, which evaluates to a single value.

Expressions can consist of arithmetics, standard language functions, subroutine calls, TIM (image processing) commands, command files, etc.

Expressions can be combined to statements. Examples:

```
if (thre p) > 2**count
```

This statement consists of the following expressions:

`thre p`	(result: tmp1)
`2**count`	(result: tmp2)
`tmp1 > tmp2`	(result: true or false)

TIM commands in a Command file

TIM commands can be mixed with expressions at will. Generally, to avoid ambiguities, it is advised to enclose a TIM command in parentheses if it is a part of an expression.

After execution, the command expression is replaced by the value it returns.

Note: TIM commands are different from other language constructions. They have a variable number of arguments, and they use a special parameter type 'image' (not used anywhere else).

Images in a command file

An image is an important parameter in a TIM command. In the TIM language it is not a recognised item; it only has a meaning in the context of an image processing task. Therefore it is the responsibility of the programmer to take care that his image names do not interfere with keywords and variable names.

This is important when writing programs, that should run on other systems. In this case follow the following guidelines:

- use the image organisation as defined in the standard `images.tim` files
- add your version of `images.tim` to the distributed command files. Notice the differences between the supported frame grabbers!

Input and output

In a computer program it is often necessary to communicate with the outside world. TIM has the following facilities to import or export data and events:

- Screen & Keyboard IO
- Disk fileIO

Screen & Keyboard IO

- **Screen output**
  TIM has a screen area (the TTY-window), where the output of command files is written to. The TTY window pops up when a command file writes to it. See the print and fprint commands for details about formatting, etc.

- **Keyboard input**
  TIM has several methods to monitor key presses:
  inkey
  pause
  mouse

Formatting numeric values

The fprint command can be used for file output and formatting of ASCII strings. It can be used to format strings for files, for character arrays, for the TTY window and for printing.

Disk IO

Reading and writing of files is easy in TIM. You just write from or read to a file, which is referenced by name. To operate on a file, you must declare a variable of the file type in the front of your program.

There are two methods of file IO:

- character based
- binary

Character based file IO

Character based file IO writes readable (ASCII) strings to files. TIM's `fprint` command has formatting capacities based on the C-language, that give you a flexible way to construct ASCII strings. In the following program fragment:

```
string name = "Piet"
age = 33
 . . .
fprint myfile 1 "Name %s is %d years old", name, age
```

the string "`Name Piet is 33 years old`" is appended to the file `myfile`.

See also: fprint

Binary file IO

Binary file IO reads and writes binary data from or to files. Values from the program are written in their internal representation, without any formatting. Binary files usually cannot be read with a text editor.

A file is referenced with a variable of the file type
The source or destination of a file operation is an array.

TIM has the following operations for binary file IO

- rfile	read from a file
- wfile	write to a file
- pfile	position a file

Example:

```
file temp = "file01.tmp"
int realdata[1000]
int  tmpdata[10000]
.
.
.
wfile temp realdata 1000
pfile temp 0
rfile temp tmpdata 500
rfile temp tmpdata[2000] 500
```

In this example the content of array realdata is written to a file temp. If this file exists, it is reset and used; if it doesn't exist it will be created.
Then the file is repositioned to the beginning and the array tmpdata is filled with 500 integers. Then the remaining 500 integers are written in the same array tmpdata, but starting at position 2000

Operators

TIM has the standard set of operators, known from many other programming languages. As a bonus the set of operators that makes the C-language so powerful is also included in the TIM language.

- Arithmetic operators
- Relational operators
- Assignment operators
- Increment & Decrement operators
- Bitwise operators
- Logical operators

Arithmetic operators

Operator	Description
+	Addition
–	Subtraction
*	Multiplication
/	Division
**	Exponentiation
%	Modulus

The modulus operator returns the remainder of a division. For example, `20 % 3 = 2`. The modulus operation can only be performed on integer data.

Relational operators

Operator	Description
`<`	Less than
`<=`	Less than or equal
`>`	Greater than
`>=`	Greater than or equal
`==`	Equal
`!=`	Not equal

Example:

```
if count < maximum
if 1 == 2          ;always false
```

Be careful not to confuse the equality operator `(==)` with the assignment operator `(=)`.

Assignment operators

The assignment operator (=) sets one value to another. Following the C language, TIM allows you to combine the assignment operator and arithmetic and bitwise logic operators.

<table>
<tr><th>Operator</th><th>Expression</th><th>Equivalence</th><th>Description</th></tr>
<tr><td>+=</td><td>x += 1</td><td>x = x + 1</td><td>Addition</td></tr>
<tr><td>-=</td><td>x -= 1</td><td>x = x - 1</td><td>Subtraction</td></tr>
<tr><td>*=</td><td>x *= 2</td><td>x = x * 2</td><td>Multiplication</td></tr>
<tr><td>/=</td><td>x /= 2</td><td>x = x / 2</td><td>Division</td></tr>
<tr><td>%=</td><td>x %= 3</td><td>x = x % 3</td><td>Modulus</td></tr>
<tr><td><<=</td><td>x <<= 2</td><td>x = x << 2</td><td>Shift left</td></tr>
<tr><td>>>=</td><td>x >>= 1</td><td>x = x >> 1</td><td>Shift right</td></tr>
<tr><td>&=</td><td>x &= 1</td><td>x = x & 1</td><td>Bitwise And</td></tr>
<tr><td>|=</td><td>x |= 1</td><td>x = x | 1</td><td>Bitwise Or</td></tr>
<tr><td>^=</td><td>x ^= 1</td><td>x = x ^ 1</td><td>Bitwise Exclusive Or</td></tr>
</table>

Increment and Decrement operators

The increment and decrement operators increment or decrement an expression by 1.

Operator	Description
++	Increment expression by 1
--	Decrement expression by 1

The following expressions are equivalent:

```
val++
val += 1
val = val + 1
```

These operators can precede or follow an expression. Placed before an expression, the operation changes the expression before its value is used. Placed after an expression, the operator changes the expression's value after it is used.

In the following program examples, the first time the loop is executed once, and the second time it is skipped.

```
int cnt = 1
while cnt-- > 0    ;this statement is executed once
  print cnt
endw

int cnt = 1
while --cnt > 0    ;this statement is never executed
  print cnt
endw
```

Bitwise operators

Bitwise operators manipulate bits in data of the integer type.

Operator	Description
&	AND
|	OR
^	Exclusive OR
<<	Shift left
>>	Shift right
~	Complement

Example:

```
lsbit = counter & 1
value = 1 << count      ;equals 2**count
```

Logical operators

<table>
<tr><th>Operator</th><th>Description</th></tr>
<tr><td>&&</td><td>Logical AND</td></tr>
<tr><td>||</td><td>Logical OR</td></tr>
<tr><td>!</td><td>Logical NOT</td></tr>
</table>

Example:

```
if counter > 10 && debug == 1
if !error
```

Operator precedence

The table below shows the order of precedence in which expressions are evaluated. If in doubt, use parentheses to force a specific order of evaluation.

<table>
<tr><th>Operator</th><th>Name or Meaning</th><th>Associativity</th></tr>
<tr><td>[ ]</td><td>Array element</td><td></td></tr>
<tr><td>++</td><td>Increment</td><td>Right to left</td></tr>
<tr><td>--</td><td>Decrement</td><td></td></tr>
<tr><td>!</td><td>Logical NOT</td><td>Right to left</td></tr>
<tr><td>~</td><td>Bitwise complement</td><td></td></tr>
<tr><td>-</td><td>Arithmetic negation</td><td></td></tr>
<tr><td>+</td><td>Unary plus</td><td></td></tr>
<tr><td>**</td><td>Exponentiation</td><td></td></tr>
<tr><td>*</td><td>Multiplication</td><td>Left to right</td></tr>
<tr><td>/</td><td>Division</td><td></td></tr>
<tr><td>%</td><td>Remainder</td><td></td></tr>
<tr><td>+</td><td>Addition</td><td>Left to right</td></tr>
<tr><td>-</td><td>Subtraction</td><td></td></tr>
<tr><td><<</td><td>Left shift</td><td>Left to right</td></tr>
<tr><td>>></td><td>Right shift</td><td></td></tr>
<tr><td><</td><td>Less than</td><td>Left to right</td></tr>
<tr><td><=</td><td>Less than or equal to</td><td></td></tr>
<tr><td>></td><td>Greater than</td><td></td></tr>
<tr><td>>=</td><td>Greater than or equal to</td><td></td></tr>
<tr><td>==</td><td>Equality</td><td>Left to right</td></tr>
<tr><td>!=</td><td>Inequality</td><td></td></tr>
<tr><td>&</td><td>Bitwise AND</td><td>Left to right</td></tr>
<tr><td>^</td><td>Bitwise exclusive OR</td><td>Left to right</td></tr>
<tr><td>|</td><td>Bitwise inclusive OR</td><td>Left to right</td></tr>
<tr><td>&&</td><td>Logical AND</td><td>Left to right</td></tr>
<tr><td>||</td><td>Logical OR</td><td>Left to right</td></tr>
<tr><td>=</td><td>Simple assignment</td><td>Right to left</td></tr>
<tr><td>*=</td><td>Multiplication assignment</td><td></td></tr>
<tr><td>/=</td><td>Division assignment</td><td></td></tr>
<tr><td>%=</td><td>Modulus assignment</td><td></td></tr>
<tr><td>+=</td><td>Addition assignment</td><td></td></tr>
<tr><td>-=</td><td>Subtraction assignment</td><td></td></tr>
<tr><td><<=</td><td>Left-shift assignment</td><td></td></tr>
<tr><td>>>=</td><td>Right-shift assignment</td><td></td></tr>
<tr><td>&=</td><td>Bitwise-AND assignment</td><td></td></tr>
<tr><td>^=</td><td>Bitwise-exclusive-OR assignment</td><td></td></tr>
<tr><td>|=</td><td>Bitwise-inclusive-OR assignment</td><td></td></tr>
</table>

Flow control

Flow control is a powerful tool for structured programming. TIM programs use two types of flow control:

- Looping statements. These loops repeat while a condition is true, or for a set number of times.
  for - next
  while - endw
  repeat - until

- Decision-making statements. These statements transfer control based on the outcome of a logical test.
  if - elseif - else - endif
  switch - case - endsw

- Unconditional flow control
  goto

The while statement

A while loop repeats as long as a given condition remains true.

It consists of the while keyword, followed by a test expression. In the following lines the loop body is specified, which is ended with an **endw** statement.

The test expression can be any TIM expression which evaluates to either true or false. When the program encounters the **while** statement, the test expression is evaluated and if it evaluates true, the while loop is entered. If not, execution resumes after the **endw** statement.

Example:

```
while counter-- > 0
  ...
endw
```

The repeat statement

The repeat ... until loop is similar to the while loop, but the loop body is executed before the test expression is evaluated. Thus, this kind of loop is always executed at least once.

Another difference is that the test expression has to evaluate *false* for the loop to continue. Example:

```
repeat
  ...
until counter-- == 0
```

The for statement

The for statement in TIM is often used to repeat statements a set number of times.

In the statement a variable is initialised, and an end value and a step size are defined. The for loop ends with an endfor statement. Example:

```
for counter = 0 to 100 step 2
  ...
endfor
```

In the example a variable, `counter`, is assigned the value 0, 2, 4 etc. in each iteration of the program loop. When `counter` becomes 100, the loop ends.

Step sizes can also be negative.

The if - elseif - else - endif statement

if can be used for simple and complicated constructions. A simple , one-line instruction is:

```
if value < 20 thre p 100
```

Here a single action is performed if the expression evaluates true.

The more complicated construction consists of the **if - elseif - else** keywords, separated by one ore more lines containing statements. The keywords are followed by a test expression, except **else**.

The statements, belonging to the first keyword whose test expression evaluates true, are executed. The entire consecution is concluded with an **endif** statement. Example:

```
if select == 1
  . . .
elseif select == 2
  . . .
elseif select == 3
  . . .
else
  . . .
endif
```

The switch statement

The switch statement offers an elegant option in situations that require multiple branches.

It tests a single expression that can have several values, providing a different action for each value or range of values.

```
switch select
case 1
  . . .
case 2 to 9
  . . .
case >=10, <100
  . . .
default
  . . .
endswitch
```

- In the **switch** statement an expression is evaluated.
- The **case** statements enclose the program part that must be executed for a given value or range.
- The **default** part is executed when the expression has any other value than the **case** statements supply.

Note that a range can be specified using comma separated lists `(1, 2, 3)`, using the `to` keyword and using a logical expression (`<10`). Combinations ar also allowed.

The goto statement

The goto statement performs an unconditional jump to another part of a program. The target of the goto statement is a label, which you supply. The label must end with a colon.

Example:

```
if (value == -1) goto err_handler
.
.
.
err_handler:
.
```

Modules

A typical TIM program does not always consist of one program, starting at the beginning and running to the end. Subroutines and calls to other TIM programs give the programmer the opportunity to bring structure in the program. The special character of TIM programs makes it necessary to distinguish between subroutines and calls to other TIM programs.

- subroutines
- TIM programs

Subroutines

A subroutine is a separate part of a program. It is defined in the same program source as the main part of the program, but it is located outside the main program loop.

- It can be invoked from any point in a program.
- It starts with a label
- It ends with a return statement.
- It has no local variables
- No parameters can be passed with a subroutine call
- A subroutine doesn't produce a return value

A subroutine is called as follows:

```
call label
```

where `label` stands for the label statement somewhere in the program file. When in a subroutine the `return` statement is encountered, executing resumes after the original `call` statement.
A subroutine can operate on any data defined in the module.

Calling TIM program modules

Calling a TIM program module has the same effect of leaving the current program and start executing another , returning when things are done.

- It can be invoked from any point in a program.
- It has its own variables
- Parameters can be passed: both single values and arrays
- It returns a value

Example:

```
int mvalues[100]
float result
.
.
.
result = *other_program 10 mvalues
```

- Program module `other_program.cmc`. is executed
- Integer value 10 and array `mvalues` is passed
- On return, the return value is stored in variable `result`

Note: it is the responsibility of the programmer to supply the correct parameters (number, order). Also the type of return variable must match the type of value that the module returns. The compiler cannot check this. Illegal types or parameter mismatches lead to run time errors or unexpected program behaviour.

Passing parameters

To pass parameters to a TIM program module, use the following procedure

In the called program prepare the passed parameters by specifying them between the `parms ... endparms` directives:

```
parms
  int counter
  float f_values[100]
  file filename
endparms
```

In the calling program specify the parameter values in the correct order:

```
*prog 10 data "data.tmp"
```

In this example the variables are initialised with the following values:

`counter`	10
`f_values`	data
`filename`	"data.tmp"

- Single variables are passed by value: their value is passed. Whatever is done with this value is invisible to the calling program.
- Arrays are passed by reference: their address is passed. This means that modifications, made by the called program, are seen by the calling program.

See also: ~~declaration~~ of variables.

Differences between subroutines and TIM programs

- A subroutine is a part of a main program. All variables, defined in a program are 'visible' to all subroutines in the same program.
- A TIM program module, which is called from another TIM program module, is entirely isolated from it. It has its own variables, even if they have the same name. To be able to let a program use values, defined in another program, you can pass values from one program to another, and if necessary pass them back when done.

Limitations in TIM programs

A compiled TIM program module must be smaller than 64KB. This is not much of a limitation, because good programming practices keep module sizes well below this maximum.

The maximum level of program nesting is 32. This is, a program module can call another module, which calls another module, etc., up to 32 levels deep.

- the total of single variables and strings must be smaller than 64KB. To calculate the maximum number of variables and strings:
  - A single variable takes 8 bytes, regardless its type.
  - A string (and a file name) takes its number of characters + 9
- An array must be smaller than 64KB. Since the `int` data type takes 4 bytes, the maximum size of an integer array is 16.383. The following table shows the maximum number of elements of arrays:

int	16.383
float	8.192
short int	32.767
short float	16.383
string	depends upon individual string size

PROGRAMMING PITFALLS

Confusing Assignment and Equality Operators
Confusing Operator Precedence
Array problems
Omitting an array subscript
Overrunning an array boundary
Mismatching if and else statements
Omitting double backslashes in DOS path specs

Keywords and names

TIM's functions are specified by keywords. Keywords are reserved; you may not use these words for any other purpose than performing the function they stand for.

Keywords are:

- TIM's image processing command names
- TIM's language command names

Definitions

In the syntax descriptions of command file commands the following terms are used:

#	a constant value, either integer or floating point
<variable>	a valid variable
<value>	a constant value or a variable
<label>	a valid program label
<name>	a string of characters, to be used as a name.
<file>	standard DOS file description: [path\]filename.ext
<attr>	a special attribute, that is described in detail with the command
<timcommand>	a valid **TIM** command
<command>	a valid CFE command, including **TIM** commands
. . . .	more of the same (depending on context)
"string"	a character string, enclosed in quotation marks (")
<arithmetic expr>	arithmetic expression
<boolean expr>	Boolean expression

Overview Command File Keywords

Below is a list of command file key words. The mathematical functions can be found under math.

Preprocessing directives like #define can be found under preprocessor directives.

In the description of these commands notational conventions are used.

beep	sounds alarm
call	goto subroutine
case	conditional statement for **switch** structure.
char	defines a character array
chk	checks errors generated by commands
cls	clears console, cursor HOME
debug	enables debug mode
default	default statement for **switch** structure
dos	enters a DOS command
else	conditional execution of statements
elseif	conditional execution of statements
endfor	end of **for** loop
endif	end of **if** structure
endparms	end of parameter block
endsw	end of **switch** structure
endw	end of **while** loop
exist	checks if file exists
file	defines a file variable
float	defines a floating point variable
for	**for ... endfor** loop
fprint	writes text and/or data to a file
fscan	reads in numeric data from ASCII file
goto	absolute jump to a label
if	conditional execution of statements
inkey	checks keyboard status
int	defines an integer variable
math	various mathematical functions
mouse	reads mouse buttons, keyboard keys
next	end of **for** loop
on error	jump to label if error occurs
parms	starts parameter declarations block
pause	prints text, waits for pressing of key
pfile	position binary file

preadkb	prints string, read keyboard
preprocessor dir.	#ifdef, #include, #define etc.
print	prints (formatted) text, variables to console
readkb	reads keyboard
repeat	**repeat ... until** loop
return	last statement in subroutine
rfile	reads a binary file
run	runs a command file
scrs	positions console cursor
set error	set error message string
short	variable type modifier
step	step value in **for** loop
stop	ends command file
string	defines a string variable
switch	**switch ... endsw** structure
timer	timer function
until	end of **repeat** loop
wait	adjustable delay
wfile	writes to binary file
while	**while ... endw** conditional loop

Mathematical functions

abs	absolute value of argument
acos	arc cosine
asin	arc sine
atan	arc tangent
atan2	arc tangent, 2 variables
ceil	rounding off
cos	cosine
exp	exponentiation
floor	rounding off
ln	natural logaritm
log10	logaritm
rest	remainder
sin	sine
sqrt	square root
tan	tangent
todegr	conversion to degrees
torad	conversion to radians

Preprocessor directives

#define	define an immediate value
#include	include a file containing pre-processor directives
#ifdef	compilation if condition is TRUE
#elseif	compilation if condition is TRUE
#else	compilation if condition is FALSE
#endif	end of conditional compilation

ABS

Type

Mathematical function

Syntax

```
abs(<value>)
```

Parameters

<*value*> - an integer or float value

Function

Calculates the absolute value of <*value*>

Description ...

Description

This function produces the absolute value of an argument. It returns a value with the same type as the argument.

Examples

```
print abs(-3)
```

result:

```
3
```

ACOS

Type

Mathematical function

Syntax

```
acos(<value>)
```

Parameters

<*value*> - a floating point value in the range: -1 to +1

Function

Calculates the arc cosine of <*value*> in the range 0 to pi.

Description . . .

Description

This function produces the arc cosine result. If the argument is out of range, an error is generated. See on error for how to handle run-time errors.

Mathematical functions operate on floating point values. However, if you specify an integer, type conversion takes place automatically.

Example

```
print acos(0.5)
```

result:

```
1.04719
```

See also

sin, cos, tan, asin, atan.

ASIN

Type

Mathematical function

Syntax

```
asin(<value>)
```

Parameters

<*value*> - a floating point value in the range: -1 to +1

Function

Calculates the arc sine of <*value*> in the range pi/2 to pi/2.

Description . . .

Description

This function produces the arc sine result. If the argument is out of range, an error is generated. See on error for how to handle run-time errors.

Mathematical functions operate on floating point values. However, if you specify an integer, type conversion takes place automatically by rounding.

Example

```
print asin(0.5)

result: 0.52360
```

See also

sin, cos, tan, acos, atan.

ATAN2

See atan

ATAN, ATAN2

Type

Mathematical function

Syntax

```
1.  atan(<value>)
2.  atan2(<value1>, <value2>)
```

Parameters

<value>, *<value1>*, *<value2>*	floating point values

Function

1. Calculates the arctangent of <*value*> in the range -pi/2 to pi/2.
2. Calculates the arctangent of <*value1*> and <*value2*> in the range pi/2 to pi/2.

Description . . .

Description

These functions calculate the arc tangent of their respective argument(s).

Atan2 uses the sign of both arguments to determine the quadrant of the return value.

If both arguments in **atan2** are 0, an error is generated. See on error for how to handle run-time errors.

Mathematical functions operate on floating point values. However, if you specify an integer, type conversion takes place automatically.

Examples

```
int length
int width
arcval = atan(length/width)
arcval = atan2(length, width)
```

See also

sin, cos, tan, asin, acos.

BEEP

Type

TIM command

Syntax

```
beep
```

Parameters

None

Function

Produces an audible alarm.

Description . . .

Description

This command enables the user to produce an alarm during the execution of a command file, to emphasize special situations.

Windows' system beep is used, which does not allow control of pitch and duration.

Examples

```
beep            ; alarm when command file is ready
stop

if error == 1 beep   ; signal on error
```

CALL

Type

Flow-control

Syntax

```
call <label>
```

Parameters

<label> a label, defined somewhere in the program.

Function

Starts execution of a subroutine

Description . . .

Description

A subroutine consists of a number of commands preceded by a **label** and concluded with a **return** statement. Within a subroutine, all kinds of TIM and CF commands may occur, even calls to the subroutine itself.

A subroutine should be ended by a **return** statement.

The **call** statement and **return** statement must occur in pairs.

Examples

```
call procs
  <statement> ; returning from the subroutine
    . . .
stop

procs:  ; start of subroutine
  <statement>
    . . . .
return
```

Comment

Variables are global, so no local variables can be created within a subroutine. If this is necessary, consider using a command file.

Although it is not recommended programming style, it is possible to jump to a subroutine with a **goto**. It is the user's reponsibility to avoid the **return** statement in that case, as executing one would cause a run-time error.

See also

goto, return, stop.

CASE

See switch.

CEIL

Type

Mathematical function

Syntax

```
ceil(<value>)
```

Parameters

<*value*> - a floating point value

Function

This function calculates the smallest integer that is greater than or equal to <value>

Description . . .

Description

This function calculates the smallest integer that is greater than or equal to <value>. For a negative <value> this leads to a result that is closer to zero than <value>.

Example

```
print "Next higher integer of ", 10.5 "is: ", ceil(10.5)
print "Next higher integer of ", -10.5 "is: ", ceil(-10.5)
```

result:

```
Next higher integer of 10.5 is: 11
Next higher integer of -10.5 is: -10
```

See also

floor.

CHAR

Type

`declaration keyword`

Syntax

`char <array> [= #1, #2, ...]`

Parameters

<*array*>	- an array name followed by an dimension between brackets ('[]').
#	- optional char values to be assigned; default: 0

Function

Declares a char array

Description . . .

Description

This command declares a character array for use within the command file. In contrast to the other declaration keywords, this type can not be used to declare single variables. The type of the array item is an 8 bits signed char. Arrays must be declared before usage; recommended is to declare all arrays at the beginning of the command file. One or more arrays can be declared on a line, separated by comma's. Each array can be initialised with a string. If no initialisation value is given the default values are 0.

Examples

Declare character array of 32 characters:

```
char arr1 [32]
```

declare character array and initialise with a string:

```
char arr1 [32] = "Hello, world"
```

Comment

Arrays can be declared without dimension, the dimension is then deduced from the number of initialisation elements.

See also

char, float, variable, string, file, short.

CHK

Type

TIM command

Syntax

```
chk <timcommand>
```

Parameters

<*timcommand*> a valid **TIM** command

Function

Handles errors caused by a **TIM** command

Description . . .

Description

chk invokes the **TIM** command that is specified with it. If this command generates an error, **chk** notices this and resets the error flag. This flag would normally end the execution of the command file. If an error occurred, **chk** returns the error code value; if not, it returns 0. This value can be checked by the user to start an error handling routine.

This facility is useful to check conditions. For instance, if an image file should be available to run a command file, its absence can be detected, and the program can jump to a routine that deals with this exception. See the example.

Example

check if image is available, if not, handle error:

```
err = chk dis image

if err > 0 goto errmess


or more compact:


if (chk dis image) > 0 goto errmess
```

Comment

This operation is **not** capable of handling the following situations:

- run time errors (except the *file not found* category)
- unknown command files as in **chk** **non_xst*, where *non_xst* is the name of a non-existing command file.

These types of errors still cause the command file to end prematurely. To prevent this, the function **on error** can be used to install an error handling routine.

See also

exist, on error.

CLS

Type

TIM command

Syntax

```
cls
```

Parameters

None

Function

Erases the computer screen (console).

Description . . .

Description

cls is one of the format operations, that enables the user to arrange the output of his command file on the console (TTY window). After the **cls** command, the console is blank and the cursor is positioned at 1,1. Without further commands, writing to the console starts from the HOME (upper left) position.

Example

```
cls
```

Comment

This operation fills the TTY screen with the last background colour attribute that is used in a print statement. A second **cls** also resets all attributes.

See also

scrs, scroll.

COS

Type

Mathematical function

Syntax

```
cos(<value>)
```

Parameters

<*value*> - a floating point value

Function

Calculates the cosine of <value> in the range -1 and +1.

<*value*> must be -pi/2 to pi/2.

Description . . .

Description

This function produces the cosine result. Mathematical functions operate on floating point values. However, if you specify an integer, type conversion takes place automatically by rounding.

Example

```
print cos(1.0)
```

result:

```
0.540302
```

See also

sin, cos, tan, asin, acos, atan.

DEBUG

Syntax

```
debug
```

Parameters

None

Function

Starts debug mode

Description . . .

Description

Debug activates the debug window and enables to execute the command file statements in step, trace or animate mode. While going through the command file in one of these modes the values of selected variables can be watched in the watch window.

Examples

`debug`	start debug mode
`if count == 44 debug`	start debug mode on a special occasion

Comment

With this command the debug mode is evoked from within the command file. The debug mode can also be entered by running the command file with the keyword **debug** added in the last position (e.g. after the parameters):

```
*example 1 debug
```

or pressing **ESC** during command file execution and choosing the **debug** option.

DEFAULT

See switch.

#DEFINE

Type

Compiler directive

Syntax

```
#define <name> <string>
```

Parameters

<*name*> - a string complying to the syntax of an identifier.

<*string*> - a string by which <*name*> is replaced.

Function

Macro substitution in every command file line of <*name*> by <*string*>.

Description . . .

Description

The macro substitution of <*name*> by <*string*> in every command file line is done literally. Therefore, any quotes in (or around!) <*string*> are substituted as well. <*name*> must comply to the syntax of an identifier, e.g. not contain blanks. The macro

```
#define first var          second
```

is not replacing "first var" with "second", but "first" with "var   second".

<string> can be anything. Macroname <name>, e.g. 'oper' in example 2, is not replaced if string 'oper' is a substring. That is, the macro of example 2 does *not* replace 'oper' in the string 'operation'.

Macro's operate on all lines that appear after their definition. This can be other macro's. If a macro is changed by another macro, the changed definition is used by the compiler. Macro's also operate on aliases.

Macro's can be used to replace TIM keywords. It almost goes without saying that excessive use of macro's can lead to very obscure and buggy programs. Use them wisely. A good practice is to write macro names in capitals.

Examples

Example 1. Define constants:

```
#define DIMENSION 512
print "Image dimension is now ", DIMENSION
```

result: **Image dimension is now   512**

Example 2. Define operations:

```
#define oper +
var1 = var2 oper var3
```

Example 3. Replace a TIM keyword by a macro:

```
#define unif perc

unif p
```

Comment

Defines can be collected in an include file for use in multiple command files.

The compiler directives for conditional compiling, **#ifdef**, **#elseif**, **#else** and **#endif**, control the compilation of **#define** statements also.

<*string*> can be empty.

See also

#include, #ifdef, #elseif, #else, #endif.

DOS

Type

TIM command

Syntax

```
dos "string"
```

Parameters

A string that is a valid DOS command.

Function

Executes a DOS command.

Description . . .

Description

dos can be used to create a directory, delete a file or to start a DOS program.

Example

```
dos "dir a:"
```

Comment

None.

ELSE

See if.

ELSEIF

See if.

ENDFOR

See for-endfor.

ENDIF

See if.

ENDPARMS

See parms-endparms.

ENDSW

See switch-endsw.

ENDW

See wile-endw.

EXIST

Type

TIM command

Syntax

```
exists <file>
```

Parameters

<*file*> a valid DOS file name

Function

Returns 0 if the file doesn't exist, 1 otherwise.

Description . . .

Description

To prevent error messages, or to make the program flow dependent on the presence of certain files, the existence of files can be determined before reading, copying or executing them.

Example

```
bExist = exist result.im
if bExist == 1
 dis result
else
 print "Image 'result.im' is not found"
endif
```

Comment

This function does not make use of any of the default paths.. The full path must be given if the files are not in the current directory.

EXP

Type

Mathematical function

Syntax

```
exp(<value>)
```

Parameters

<*value*> - a floating point value

Function

Calculates the exp of <*value*>.

Description . . .

Description

This function computes the natural exponent *e* to the power <*value*>. To get the power with base 10, compute exp (<*value*> * ln (10)).

Example

```
print exp(1.6)
```

result:

```
4.95303
```

Comment

None.

See also

ln, log10.

FILE

Type

declaration keyword

Syntax

```
file <name> [= "string"]
```

Parameters

<name> - a name string identifying a variable

"*string*" - an optional file name to be assigned; default: empty string

Function

Declares a file variable

Description . . .

Description

This command defines a variable for use within the command file. The type of the variable is a file. Variables must be declared before usage; recommended is to declare all variables at the beginning of the command file. One or more variables can be declared on a line, separated by comma's. A value can be assigned in the declaration statement (initialisation). If no initialisation value is given is has as default an empty string.

Examples

declare variable and assign value:

```
file fl1 = "debug.txt"
```

declare variable with empty string:

```
file fl2
```

multiple declaration:

```
file fl1, fl2 = "exp1.dat", fl3
```

Comment

Although it is possible to declare a **file** variable without immediate initialisation, the variable must be assigned a file name before using it. So this clearly gives a run-time error:

```
file test
float array [32]
.
.
.
wfile test array 32 ; Error; can't write to file.
```

and should be replaced by something like:

```
file test
float array [32]
.
.
.
test = "c:test.dat"
wfile test array 32     ; OK.
```

or the file variable can simply be initialised:

```
file test = "c:test.dat"
```

See also

char, float, variable, int, file, short.

FLOAT

Type

```
declaration keyword
```

Syntax

```
1. float <name> [ = # ]
2. float <array> [ = #1, #2, ... ]
```

Parameters

<name>	variable name
<array>	an array name followed by an dimension between brackets ('[]').
#	value to be assigned; default: 0.0

Function

Declares a floating point variable or a floating point array

Description . . .

Description

This command declares a variable for use within the command file. The type of the variable is 32bit floating point. Variables must be declared before usage; recommended is to declare all variables at the beginning of the command file. One or more variables can be declared on a line, separated by comma's. A value can be assigned in the declaration statement (initialisation). If no initialisation value is given is has a default value 0.0.

A floating point value must be specified in one of the following ways:

- ddd.fff digits with one decimal point, consisting of an integer and a fractional part;

- ddd.fffEeee scientific notation: mantissa and exponent (see also chapter 5) .

Examples

```
float fvar1 = 1.6
float fvar2
float fvar1, fvar2 = 23.0, fvar3
float farr[8] = 1, 2, 3, 4
float farr[] = 1, 2, 3, 4
```

(declare array with implicit dimension)

initialization with constant expression:

```
#define DIMENSION 256
float var1 = (12.0 * DIMENSION) / 16.0
```

Comment

Declaration types int and float can be preceded by the type modifier **short** to save memory, for example when declaring large arrays.

If the arrays is declared without dimension, the dimension is deduced from the number of initialisation elements (which must be present then).

See also

char, file, int, short, string.

FLOOR

Type

Mathematical function

Syntax

```
floor(<value>)
```

Parameters

<*value*> - a floating point value

Function

This function calculates the largest integer that is smaller than or equal to <*value*>

Description . . .

Description

This function calculates the largest integer that is smaller than or equal to <*value*>. For a negative <*value*> this leads to a result that is further from zero than <*value*>.

Example

```
print "Floor of 10.5 is ", floor(10.5)
print "Floor of -10.5 is ", floor(-10.5)
```

result:

```
Floor of 10.5 is 10
Floor of -10.5 is -11
```

Comment

None.

See also

ceil.

REST

Type

Mathematical function

Syntax

```
rest (<value1>, <value2>)
```

Parameters

<value1>, *<value2>* - floating point values

Function

rest computes the remainder of two floating point values.

Description . . .

Description

rest (<value1>, <value2>) computes the remainder R such that for some integer value I: <value1> = I * <value2> + R, where R has the same sign as <value2> and a smaller absolute value than <value2>.

Examples

```
print "rest(6.2, 2) = ", rest(6.2, 2)
print "rest(6.2, -2) = ", rest(6.2, -2)
print "rest(-6.2, 2) = ", rest(-6.2, 2)
print "rest(-6.2, -2) = ", rest(-6.2, -2)
```

result:

```
0.2
0.2
-0.2
-0.2
```

Comment

The remainder is not the same as the modulus function.

The integer version of this function is the remainder operator '**%**'.

FOR - ENDFOR,   FOR - NEXT

Type

Flow-control

Syntax

```
for <value> = #1 to #2 [ step #3 ]
  <statement>
  .
  .
  .
endfor [ <value> ]
```

Parameters

<*value*>	an integer variable as loop variable
#1	begin value (integer)
#2	end value (integer)
#3	optional step value (integer)

Description . . .

Description

The loop variable <value> starts with #1 and is then in every loop iteration increased with the step value #3 until the end value #2 is reached. If no explicit step value is given, it is assumed to be 1. Begin-, end- and step value can also be negative.

Example

```
for Index = 1 to 5
 print Index
endfor Index
```

result:

```
1
2
3
4
5
```

Comment

'next' is also accepted instead of 'endfor'.

See also

while - endw, repeat - until, goto.

FPRINT

Syntax

```
1. fprint <file> [0] "string", <variable>, <variable>, ...
```

Parameters

<*file*>	file (or device) to be written to
0	file attribute (0 = overwrite; default: append)
"*string*"	string to be written (also contains format specifiers)
<*variable*>, ...	variable list

Function

Writes formatted text and variables to file, the TTY window or DOS devices.

Description . . .

Description

This command writes formatted text and data to a file or device. Its format resembles the corresponding C-language statement (**printf**).

The <*file*> parameter specifies the file or device to be written to. Writing can be done in two modes:

1. append
2. overwrite.

If a `O` character is entered as the second parameter, the file is opened in the overwrite mode. If the file exists, the data will be written from the beginning of the file, whether or not it already contains data. If the `O` character is absent, the new data will be appended to the present content of the file. Overwrite/append plays a role only with 'real' files; with printer, console, etc. the parameter is ignored.

In a typical sequence the `O` option is used with the first **fprint** statement to reset an existing file. Further **fprint**'s to the file should not use the `O` option, allowing the data to be appended.

The "*string*", the third parameter, is the format string . If a variable list is specified, the format string contains format instructions to format the output of these variables.

If one or more variables are specified, there must be a strict correspondence between the ordering of the variables and the format specifiers in the string.

Examples

```
fprint c:name.dt "This is text; value is: %f", fvar
fprint c:name.dt "%ld %f", ivar, fvar
fprint c:file.log 0 "Discard file's old content\n"
fprint prn "Send text to the printer \n"
```

See also

print, wfile

FSCAN

Syntax

```
fscan <file>
```

Parameters

<*file*> ASCII file containing integers (256 at most)

Type

Miscellaneous operation.

Function

Reads an ASCII file containing integers; stores the values in ibuf

Return Parameter

Number of valid integers read.

Description . . .

Description

This command reads the content of an ASCII file, which is supposed to contain integer values. The values are read and stored into ibuf in the long integer format (32 bits). 256 values can be stored in ibuf.

The values in the file may be separated by any nonnumeric delimiter: space, comma, carriage return, etc.

The return value allows you to check the successful interpretation of the file's content.

Example

This example reads file **numbers** and stores the content into ibuf:

```
fscan numbers
```

Comment

This command differs from **ribuf** in that **ribuf** reads a binary file, which is an exact image of ibuf's content, whereas **fscan** reads a freeformat ASCII file, which may contain up to 256 items.

See also

fprint.

GOTO

Type

Flow-control

Syntax

```
goto <label>
```

Parameters

<label> a valid label

Function

Jumps to another place in the command file, indicated by <*label*>.

Description . . .

Description

This command breaks the sequential order in which the commands are normally executed. Execution resumes at the command following the label specified. Forward references as well as backward references are allowed.

Examples

```
begin:
<command>
. . .
goto begin ; jump back to label 'begin'.
```

Comment

<*label*> must satisfy the syntax for an identifier.

See also

call, return, label, stop.

IF

Type

Flow-control

Syntax

```
1.if <boolean expression> <statement>

2.if <boolean expression>
       <statement>
       . . . .
endif

3.if <boolean expression>
       <statement>
       . . . .
elseif <boolean expression>
       <statement>
       . . . .
else
       <statement>
       . . . .
endif
```

Description . . .

Description

The **if** command offers various possibilities for conditional execution of commands or groups of commands. The following three versions exist:

1. A single **TIM** or CFE command is executed if <*boolean expression*> evaluates TRUE.
2. The group of commands specified between the **if** statement and the **endif** statement are executed if <*boolean expression*> evaluates TRUE.
3. This version consists of an **if**, any number of **elseif**'s, an **else** and an **endif** directive. The group of commands situated between the **if** or **elseif** statement whose <*boolean expression*> evaluates TRUE and the next **elseif** or **else** statement, are executed.

All **TIM** or CFE commands are allowed in an **if**loop, so nesting of **if** and **while** loops is permitted.

Examples

Simple **if** statement :

```
1. if var1 > var2 print "greater"
```

Conditional execution of series of commands:

```
2.if var1 < var2
      add a b
      copy p q
endif
```

More tests in one set of expressions:

```
3.if var1 == var2
       add a b
elseif var1 > var2
       sub a b
else
       sub b
endif
```

Comment

If a large number of **elseif** statements is used, the **switch** statement is probably more lucid.

See also

switch-endsw, #ifdef.

#IFDEF - #ELSEIF - #ENDIF

Type

Compiler directive

Syntax

```
#ifdef <name>
```

Parameters

<*name*> - a string complying to the syntax of an identifier.

Function

Directives for confitional compiling

Description . . .

Description

The statement between the **#ifdef** and the subsequent **#endif** (or **#elseif, #else**) are compiled if <*name*> has been defined, i.e. if <*name*> is an alias or a macro.

Examples

Assuming that FRAMEGRABBER has not been defined, the program in example 1 gets an image from disk to process:

Example 1.

```
#ifdef FRAMEGRABBER
  print "Digitizing image..."
  dig 0; digitize image till key press.
#else
  print "Getting image from disk..."
  dis scene1 ; get image 'scene1' from disk.
#endif
```

The program in example 2 chooses between different kinds of framegrabbers.

```
#ifdef PCVISION
  ....
#elseif FG100
  ....
#elseif VFG
  ....
#else
  ....
#endif
```

Comment

The **#ifdef**, **#elseif**, **#else** and **#endif** family has an appoximate run-time equivalence in **if**, **elseif**, **else** and **endif**. If it is certain that the condition, that leads to the choice of one set statements or another, will not change after compilation, the use of **#ifdef** will save a bit of space. Furthermore, **#ifdef** can switch on or off **#defines**, declaration of variables etc.

#INCLUDE

Type

Compiler directive

Syntax

```
#include <file>
```

Parameters

<file> - a valid DOS file name with or without quotes.

Function

The include file with name *<file>* is included in the command file containing the **#include** statement, i.e. *<file>* is scanned for **#define** statements (macro's).

Description . . .

Description

The file with name <*file*> is scanned for **#define** statements (macro's). These **#define** statements are added to local **#define**'s and the aliases. They are then applied to the source lines in the command file.

Include files cannot be nested. All other statements than **#define** statements are ignored.

Examples

```
#include a:extra.inc
#include "a:extra.inc"
```

Comment

Include files give the possibility the keep macro's, which are common to multiple command files, together in one file. Any changes to these macro's have to be done only once.

See also

#define.

INKEY

Type

TIM command

Syntax

```
1. inkey [0]
2. inkey  1
```

Parameters

1. If the argument is omitted, the default value (zero) is taken.
2. A non-zero value.

Function

Signals whether a key has been pressed.

1. No argument or an argument value of zero lets the function return immediately, whether a key has been pressed or not.
2. A non-zero argument lets the function wait until a key is pressed.

This function supplies information about keyboard activity since the last time the keyboard has been used. If a key has been pressed, it returns its ASCII code, if not it returns 0. If a function key, edit or cursor control key was pressed, its scan code + 128 is returned. Thus, normal (ASCII) keys can be recognised by their value, which is below 128.

The character signalled by **inkey** is removed from the keyboard buffer.

Return Vale

1. The ASCII value of the key which has been pressed, or zero if no key has been pressed.
2. The ASCII value of the key which has been pressed.

Examples

```
. . . . .
KeyPressed = inkey ; key contains key code or 0
if KeyPressed != 0
  . . . . . ; take action if key has been pressed
endif
```

Comment

The difference between the **inkey, pause, mouse** and the **wait** command is:

inkey checks whether a key has been pressed or not

pause prints user specified text

mouse reads a single key or mouse button

wait waits until a delay period expires.

See also

preadkb, variable

INT

Type

```
declaration keyword
```

Syntax

```
1. int <name> [ = # ]
2. int <array> [ = #1, #2, ... ]
```

Parameters

<*name*>	a name string identifying a variable
<*array*>	an array name followed by a dimension between brackets (**[ ]**)
#	an optional integer value to be assigned; default: 0

Function

Declares an integer variable or an integer array.

Description . . .

Description

This command declares a variable for use within the command file. The type of the variable is a 16 bits signed integer. Variables must be declared before usage; recommended is to declare all variables at the beginning of the command file. One or more variables can be declared on a line, separated by comma's. A value can be assigned in the declaration statement (initialisation). If no initialisation value is given is has a default value 0.

Examples

`int var1 = 256`	declare variable and assign value
`int counter`	declare variable with default value: 0
`int var1, var2 = 12, var3`	multiple declaration
`int arr [8] = 1, 2, 3, 4`	declare array and assign some values
`int arr [] = 1, 2, 3, 4`	declare array with implicit dimension

Initialisation with constant expression:

```
#define DIMENSION 256
int var1 = (12 * DIMENSION) / 16
```

Comment

Declaration types int and float can be preceded by the type modifier **short** to save memory, for example when declaring large arrays.

Arrays can be declared without dimension, the dimension is then deduced from the number of initialisation elements.

See also

char, file, float, short, string, variable.

LABEL

Type

Flow-control

Syntax

```
label:
```

Parameters

None

Function

Reference point for goto and call statements

Description . . .

Description

A label is a reference point in a command file. It is used by **goto** and **call** statements to continue execution.

A label consists of a label name, followed immediately by a colon (:). No other information, except comments, may be present on a label line. A label may contains the same characters as an identifier (variable).

Labels must be unique, no two or more labels with the same name are allowed in one command file.

Examples

```
begin:
. . .
goto begin    ; jump to label 'begin'.
```

Comment

It is illegal to jump into an if block or any kind of higher nesting level. So the following is an example of a useless label, because a jump to this label is not permitted:

```
if count >= 32
begin:
 total = total + count
endif

goto begin    ; here the compiler will complain about an illegal jump.
```

See also

call, goto,

LN

Type

Mathematical function

Syntax

```
ln(<value>)
```

Parameters

<*value*> - a floating point value

Function

Calculates the natural logarithm of <*value*>.

<*value*> must be greater than zero.

Description . . .

Description

This function produces the natural logarithm. Mathematical functions operate on floating point values. However, if you specify an integer, type conversion takes place automatically by rounding.

Example

```
print ln(5.85739)

result: 1.767704
```

See also

exp, log10

LOG10

Type

Mathematical function

Syntax

```
log10(<value>)
```

Parameters

<*value*> - a floating point value

Function

Calculates the logarithm of <*value*> with base 10.

<*value*> must be greater than zero.

Description . . .

Description

This function produces the logarithm with 10 as base. Mathematical functions operate on floating point values. However, if you specify an integer, type conversion takes place automatically by rounding.

Example

```
print log10(5.85739)

result: 0.767704
```

See also

exp, ln

MOUSE

Syntax

```
mouse
```

Parameters

None

Function

Waits for pressing of mouse button or keyboard key; returns value

Return Parameter

0 'return' pressed

1 left mouse button pressed

2 right mouse button pressed

Any other value: keyboard key pressed; ASCII value

Description . . .

Description

The **mouse** command is meant to program some user interaction into command files. It can be used whether a mouse is connected to the system or not.

When the **mouse** command is executed, the CFE waits for an action by the user: either pressing a mouse button or pressing a keyboard key. If a keyboard key is pressed, its ASCII value is returned immediately (no pressing of 'return' required). If a mouse button is pressed, the value 1 (left button) or 2 (right button) is returned. If 'escape' is pressed, 27 is returned.

Example

`button = mouse`	wait for action
`if button <= 2`	
`  . . . .`	mouse button pressed
`else`	
`  . . . .`	keyboard key hit
`endif`	

Comment

The difference between the **inkey, pause, mouse** and the **wait** command is:

inkey checks whether a key has been pressed or not

pause prints user specified text

mouse reads a single key or mouse button

wait waits until a delay period expires.

See also

inkey, preadkb, variable

NEXT

See for-endfor.

ON ERROR

Syntax

```
on error goto <label>
```

Parameters

<label> a valid label

Function

In case of a run_time error during execution of a command file, execution is continued with the routine with label *<label>*.

Description . . .

Description

The function **on error** tells TIM where to jump in case an error occured during command file execution. <*label*> is the label of an error-handling routine.

If the error label is not set with **on error**, TIM generates a messagebox with a run-time error message and aborts execution.

You can place more than one **on error** directives in a command file: one for each situation that requires separate error handling.

Example

```
int array [32]
file infile

on error goto err_handler
```

The next statement uses an undefined file variable. A run-time error is the result and TIM jumps to label 'err_handler':

```
rfile infile array 16

err_handler:
 cls
 print "Error occured in file reading"
 stop
```

Comment

After the error handling code, you can jump to another part of the program (using **goto**) or quit the program (using **stop**).

See also

goto.

PARMS - ENDPARMS

Syntax

```
parms
 <type> <variable> [= <value>]
 .
 .
 .
endparms
```

Parameters

<*variable*>	a parameter of the command file
<*type*>	the type of that parameter
<*value*>	an initial value is used when no parameter is specified

Function

Declares the parameters of the command file, i.e. the arguments following the command file name when calling it. An initial value is optional.

Description . . .

Description

The **parms** and **endparms** directives enclose the parameters of the command file. The parameters are the values that are passed to the command file when it is called.

The initial value is used if no parameter is given. You can use this feature only where the order of the parameters is not disturbed by the missing parameter.

It is the user's responsibility to ensure that argument and parameter list are indentical concerning the order, the types and the number of variables.

Example

In command file **comfile**:

```
parms
 int count
 float data []
 file InFile = "test.dat"
endparms
```

In a command file calling **comfile**:

```
float array []
file TestFile = "default.fil"

*comfile 34 array TestFile
```

The parameters in comfile get the values:

`count`	: 34
`array []`	: what was in **data**.
`InFile`	: "test.dat"

If the call hed been:

```
*comfile 34 array
```

Then **InFile** would have been: `default.fil`

Comment

Single variables like integers or floats are given to the command file by value, Arrays are passed as an address (by reference). Therefore array values can be modified in the called command file.

PAUSE

Syntax

```
1. pause
2. pause "string"
```

Parameters

"*string*" a string to be printed on console (TTY window).

Function

Temporarily halts execution of the command file and prints a message (if specified); then waits for a key to be pressed.

Description . . .

Description

1. The pause command writes a prompt ('>') and waits for a key to be pressed. After a key is pressed execution of the command file resumes, except if the **escape** key was pressed.
2. The specified string is written to console, and a prompt ('>') is added to it. Further behaviour is as specified under 1.

Examples

`1. pause`	wait for a key
`2. pause "Watch the screen"`	write text, wait for key

Comment

The difference between the **inkey, pause, mouse** and the **wait** command is:

inkey checks whether a key has been pressed or not

pause prints user specified text

mouse reads a single key or mouse button

wait waits until a delay period expires.

PFILE

Type

TIM command

Syntax

```
pfile <variable> <offset> <origin>
```

Parameters

<*variable*> - a variable of type file.

<*offset*> - the number of bytes from the origin for the new position in the file.

<*origin*> - A number that gives the origin:

-1: just return the current position.

0: origin is the beginning of the file.

1: origin is the current position.

2: origin is the end of the file.

Function

Positions the filepointer in a binary file

Return Value

The new position is returned.

Description . . .

Description

pfile positions the file <variable> on a new position that is <*offset*> number of bytes from the origin. This origin is determined by <*origin*>. It can be the current position, or beginning or end of the file. When writing integers or floats in a binary file, the offset has to be multiplied by the size of integer or float.

Examples

```
int array [32]
int index
file infile = "input"

; read items from file with 32 integers.
rfile infile array 4    ; read first four integers.
pfile infile 112 0      ; position file.
rfile infile array 4    ; read last four integers.

stop
```

Comment

Only binary files can be positioned.

If the origin is 2 (end of file), only negative offsets are meaningful. The positioning in the example could also have been done by

```
pfile infile -16 2
```

See also

fprint, fscan, rfile, wfile.

PREADKB

Type

    TIM command

Syntax

    preadkb <*char_array*>

Parameters

<*char_array*> - a character array with maximum length of 132.

Function

Receives a string from the user

Description . . .

Description

preadkb prints the string in the character array, then overwrites it with user input (string). The array may be empty: in this case nothing is printed.

Example

```
char InputArray [32] = "Give number of items: "
```

print string in character array and read user input into the same array:

```
preadkb InputArray
```

See also

readkb, variable.

PRINT

Syntax

```
print [<print_item>] [[,] <print_item>] ....
```

Parameters

<*print_item*> can be one of the following:

<*attr*>	- screen attribute: the **'@'** character followed by a specification string
"*string*"	- string printed
<*variable*>	- variable whose value is printed

Function

Writes text and the value of variables to console.

Description . . .

Description

The **print** command writes the specified text and the value of the variables to the console (TTY window). Text and variables are written in the order they are encountered. Integer variables are printed with as many digits as necessary. Floating point variables are printed with 6 digits. If this format does not allow a correct representation of the variable (e.g. with very small or large numbers), printing is in scientific notation:

m.fffffEeee	where:
m.fffff	is the mantissa, consisting of 6 digits,
eee	is the exponent, consisting of 3 digits.

Print has various format options, that can be controlled using attributes.

Examples

```
1. print "Variable 1 is: " var1
2. print "Area is", area, ", perimeter is ", peri
3. print @20 "Positioning of text ", @40 var1
4. print @i "Inverse video Attribute", @n
5. print "Sequence:", @10, seq, @z ; No line feed
```

Comment

Print without arguments just generates a carriage return/linefeed.

The separation between the elements in a print statement is a SPACE as usual. However, you may find the print statement more legible if you use comma's (with or without SPACES) instead. See the examples.

See also

cls, fprint, scroll, scrs.

READKB

Type

`TIM command`

Syntax

`readkb <char_array>`

Parameters

<*char_array*> - a character array with maximum length of 132.

Function

Reads in a string, entered by the user.

Description . . .

Description

readkb puts a string from user input into <*char_array*>. The maximum length of the string to be read is 132 characters.

Example

```
char InputArray [32]

readkb InputArray ; get user string input in 'InputArray'.
```

See also

inkey, mouse, preadkb, variable.

REPEAT - UNTIL

Type

Flow-control

Syntax

```
repeat
<statements>
  .
  .
  .
until <boolean expression>
```

Parameters

<*boolean expression*> - loop condition

Function

conditional program loop

Description . . .

Description

The statements between **repeat** and **until** are executed. The **until** statement then evaluates the boolean expression.

If the result of the evaluation is TRUE, the loop is repeated. If the result is FALSE, execution continues at the command after the **until** statement.

Examples

```
repeat
  thre p var1
  var1 = mark p
until var1 == 0
```

Comment

Use the **while-endw** construction if the boolean expression must be tested before executing the statements in the loop.

See also

for - next, while - endw.

RETURN

Type

Flow-control

Syntax

return

Parameters

None

Function

Last statement of a subroutine

Description . . .

Description

The **return** command is the last statement of a subroutine. The **call** statement and **return** statement must occur in pairs; leaving a subroutine other than via a **return** statement will ruin stack administration.

Nesting of subroutine calls (one subroutine calling another) may occur up to 50 levels deep.

Example

```
call xyz            ;call of subroutine xyz
   .
   .
   .
stop

xyz:                ;start of subroutine xyz
<statement>
   .
   .
   .
return              ;end of subroutine xyz
```

Comment

None

See also

call, goto, label, stop.

RFILE

Type

TIM command

Syntax

```
rfile <variable> <name> <value>
```

Parameters

<*variable*> - a variable of type file.

<*name*> - the name of a byte, integer or float array.

<*value*> - the number of items (byte, integer or float) that must be written.

Function

Reads the content of a binary file into an array

Description . . .

Description

rfile reads the given number of items from a binary file into an array . The variable of type file may be initialised before using it in a file operation. Reading from the file starts at the beginning of the file, all subsequent calls of **rfile** start at the end of the previous read. To control the position in the file where the data is to be read, use pfile.

Example

```
int array [32]
int index
file infile = "input"

; read array from file
rfile infile array 32

stop
```

Return Value

The number of items written is returned. In case of error -1 is returned.

Comment

Binary files are not readable with a text editor like Notepad. To read data from files that are readable (ASCII files) use the function fscan.

See also

fprint, fscan, pfile, rfile.

RUN

Type

Flow-control

Syntax

run <*command file*>

Parameters

<*command file*> - the valid DOS name for a command file

Function

Runs the command file <*command file*>

Description . . .

Description

Keyword **run** is used to run a command file the same way the symbols `'*'` and `'/'` are used. **run** is used at places where `'*'` and `'/'`, being mathematical symbols, could cause ambiguities.

Example

```
run measure     ; run command file "measure.cmd"
```

```
if count == 12 *measure
```

Because blanks are insignificant to the compiler this could mean

```
if count == (12 * measure)
```

where **measure** is a variable we have not yet declared. To avoid this confusion we write:

```
if count == 12 run measure
```

See also

call, goto, label, stop.

SCROLL

Type

TIM command

Syntax

```
scroll [#]
```

Parameters

number of lines; default: 1

Function

Scrolls the content of the console screen up one or more lines.

Description . . .

Description

The **scroll** command scrolls up the content of the command file window (TTY) the specified number of lines.

Examples

`scrs 10 1` write on line 10, from position 1

Write variables in columns:

```
print @10 var1, @20 var2, @30 var3
```

`scroll` scroll console up 1 line

Comment

Only upward scrolling is possible

See also

cls, scrs

SCRS

Type

TIM command

Syntax

```
scrs #1 #2
```

Parameters

#1 line number (1   22; default: 1)

#2 column number (1   79; default: 1)

Function

Defines the place where computer output will be written by   positioning the console cursor.

Description . . .

Description

This command positions the console cursor at the specified position. Without additional specification subsequent output is written from there. This operation will not clear the console, so the information already there remains unaltered, until it is overwritten. The **print** command clears a line, prior to writing text, so after a print operation, the line is be updated completely, regardless of the length of the string written.

Examples

`scrs 10 20`	position cursor to line 10, column 20
`scrs`	position cursor to line 1, column 1

Comment

Column positioning while printing can more easily be done by the '@' operator in the **print** command.

See also

cls, scroll.

SHORT

Type

Declaration keyword

Syntax

```
short <type> <name>
```

Parameters

<*type*> - a declaration type **int** or **float**

<*name*> - an array to declare

Function

short modifies the declaration types **int** and **float** to short versions which use half the number of bytes.

Description . . .

Description

short modifies the declaration types **int** and **float**. It decrease the number of bytes needed to store an array of one of these types. This can save space if an array of int or float is used. As a consequence the precision will also decrease.

Example

```
short int IntArray [256]
short float FloatArray [256]
```

See also

char, file, float, int, string.

SIN

Type

Mathematical function

Syntax

```
sin(<value>)
```

Parameters

<*value*> - a floating point value

Function

Calculates the sine of <*value*> in the range -1 and +1.

<*value*> must be -pi/2 to pi/2.

Description . . .

Description

This function produces the sine result. Mathematical functions operate on floating point values. However, if you specify an integer, type conversion takes place automatically by rounding.

Example

```
print sin(1.0)

result:       0,41471
```

See also

acos, asin, atan, cos, tan.

SQRT

Type

Mathematical function

Syntax

```
sqrt (<value>)
```

Parameters

<*value*> - a floating point value

Function

Calculates the square root of <*value*>.

Description . . .

Description

This function produces the square root. If the argument is out of range, an error is generated. See on error for how to handle run-time errors.

Mathematical functions operate on floating point values. However, if you specify an integer, type conversion takes place automatically.

Example

```
print sqrt (2)

result:       1.41421
```

STOP

Type

Flow-control

Syntax

```
stop [<value>]
```

Parameters

<*value*> is an integer or float value.

Function

Ends execution of a command file and (optional) returns <*value*> as numerical result.

Description . . .

Description

The stop command returns control to ***TIMWIN***. It should be the last command in the main block of a command file, but can be followed by subroutines. It is not necessary for the **stop** command to be physically the last command in the file, as the example shows.

If endoffile is encountered during execution of a command file (caused by the absence of a **stop** statement), execution of the command file is terminated, and an error message is generated.

Examples

```
if var == 0 stop; stop if test evaluates true
```

In command file *foo*; if 'count' is not equal zero return its value:

```
if count != 0 stop count
```

In the command file calling *foo* use the returned value:

```
number += *foo
```

Comment

If the return value of a command file is used but the command file doesn't return any <*value*>, the result will be arbitrary.

See also

call, goto, label, return.

STRING

Type

declaration keyword

Syntax

```
string <name> [= "string"]
```

Parameters

<*name*>	- a name string identifying a variable
"*string*"	- an optional string value to be assigned; default: empty

Function

Declares an string variable

Description . . .

Description

This command defines a variable for use within the command file. The type of the variable is a string. Variables must be declared before usage; recommended is to declare all variables at the beginning of the command file. One or more variables can be declared on a line, separated by comma's. A value can be assigned in the declaration statement (initialisation). If no initialisation value is given the default is an empty string. The initialisation of a string array can be extended over more than one line.

Examples

declare variable and assign value:

```
string str1 = "amsterdam"
```

declare variable with empty string:

```
string str2
```

multiple declaration:

```
string str1, str2 = "rotterdam", str3
```

See also

char, file, float, int, short, variable.

SWITCH

Type

Flow control

Syntax

```
switch <expression>
 case <number list>
   <statements>
   .
 case <number list>
   <statements>
   .
 case ...
   .
 default
   <statements>
   .
 endsw
```

Parameters

#1, #2, etc.   possible values for <expression>

Description . . .

Description

The switch statement performs multiple branches, depending on the result of an expression in the **switch** part.

Each **case** statement must contain a unique value or a range of values. If (one of) these values corresponds with the value of the **switch** expression, the instructions after that **case** are executed, up to the next **case**, **default** or **endsw** statement.

- A value list consist of one or more of the following elements:
- A list of numbers, separated by comma's: 1, 2, 3, 9
- A range: 10 to 20
- A logical expression: <10

These may be combined:

```
1, 3, 4 to 10, >=11
```

specifies everything except 2 and 0

If the **default** statement is present, the statements after it are executed if the value of the **switch** expression does not correspond to any **case** value.

Example

```
value = "Enter 1 to 10 or 20: "
switch value
case 1
  *comfile1   ;run a command file
case 2, 3, 4
  *comfile2   ;run another
case 5 to 10, 20
  *comfile3
default
  print "You entered a wrong value"
endsw
```

See also

if

TAN

Type

Mathematical function

Syntax

```
tan(<value>)
```

Parameters

<*value*> - a floating point value

Function

Calculates the tangent of <value> in the range -∞ and +∞.

<*value*> must be -pi/2 to pi/2.

Description . . .

Description

This function produces the tangent result. Mathematical functions operate on floating point values. However, if you specify an integer, type conversion takes place automatically by rounding.

Example

```
print tan(1.0)
```

result:

```
1.557408
```

See also

asin, acos, atan, cos, sin.

TIMER

Syntax

```
timer [#]
```

Parameters

preset value for timer

Type

Control operation

Function

The timer is set to the given # number of seconds and then ticks off to zero.

Return Parameter

Number of seconds to go

<u>**Description . . .**</u>

Description

This function measures time in seconds. When the internal counter is loaded with a value, this value is decremented every second, until 0 is reached. Its current state can be read by executing **timer** without a parameter.

Also **timer** stores a more detailed time stamp in lbuf (long integer format):

0 Seconds (0 59)

1 - Minutes (0 59)

2 Hours (0 23)

3 Day of the Month (1 31)

4 Month (0 11)

5 Year (1900)

6 Day of the week (Sunday = 0)

7 Day of the year (January 1 = 0)

8 Daylight Saving Time flag (0 or 1)

Examples

```
timer 100    ; preset timer to 100 sec
```

After 47 seconds, for instance, we give the command without parameter:

```
timer       ; observe elapsed time
```

and get the remaining time as output:

```
53
```

Comment

If the timer function is used to wait for a certain number of seconds, the wait function can also be used. The difference is that the timer function gives the opportunity to perform actions during waiting. Example:

```
timer 40          ; we want to wait 40 seconds before <action>

while timer != 0 ; stay in the loop until time has expired.
  <statement>     ; meanwhile, do something else.
     .
     .
     .
endw

<action>          ; time to do this now.
```

See also

inkey, pause, wait.

TODEGR

Type

Mathematical function

Syntax

```
tograd(<value>)
```

Parameters

<*value*> - a floating point value

Function

Converts a value in radians to degrees.

Description . . .

Description

This function produces conversion of radians to degrees. Mathematical functions operate on floating point values. However, if you specify an integer, type conversion takes place automatically by rounding.

Example

```
print tograd(1.0)
```

result:

```
57.2958
```

See also

torad.

TORAD

Type

Mathematical function

Syntax

```
torad(<value>)
```

Parameters

<*value*> - a floating point value

Function

Converts a value in degrees to radians.

Description . . .

Description

This function produces conversion of degrees to radians. Mathematical functions operate on floating point values. However, if you specify an integer, type conversion takes place automatically by rounding.

Example

```
print torad(45)
```

result:

```
0.7854
```

See also

todegr.

VARIABLE

Syntax

```
1. <variable> = #
2. <variable> = <arithmetic expression>
3. <variable> = "string"
4. <variable> = <timcommand>
```

Parameters

Parameter	Description
<*variable*>	a predefined variable
"*string*"	a text string
<*tiimcommand*>	a valid **TIM** command
#	a constant of the correct type (integer or floating point)
<*arithmetic expression*>	a valid arithmetic expression

Function

Assigning a value to a variable

Description . . .

Description

There are four ways in which a value can be assigned to a variable. They have in common a predefined variable followed by an '**=**' character. After this, one of the following can be specified:

1. a value. This value is assigned to the variable.
2. an arithmetic expression. This expression is evaluated and the result is assigned to the variable.
3. a string. Case (a): if the variable is of a numerical type (**integer** or **float**) the string is printed and the CFE waits for the user to enter a value. This value is assigned to the variable .
4. Case (b): if the variable is of a non-numerical type (**file, string**) the string is assigned to the variable.a valid **TIM** command. The command is executed and the return value is assigned to the variable.

Type conversion (integer to floating point and viceversa) takes place automatically.

Examples

```
int var1, number, val, overfl
file OutFile


1. var1 = 10
2. number = var2  10 + 20 * 30
3a. val = "Enter a value for this variable: "
3b. OutFile = "a:\\data.1"
4. overfl = add a 10
```

See also

inkey, mouse, preadkb, readkb.

WAIT

Syntax

```
wait [#]
```

Parameters

number of seconds to wait

Function

Halts the execution of a command file. Execution resumes when a key is pressed or if a defined period of time has expired.

Description . . .

Description

The **wait** command halts the execution of the command file, and lets the user resume execution by pressing a key. The text

```
Press a key:
```

is written to the console. If [Esc] is pressed, the execution of the command file is aborted and control is returned to **TIM**. If a number is specified with the **wait** command, a delay period, consisting of the specified number of seconds, is applied. This delay can be interrupted by pressing a key.

Examples

`wait`	wait until a key is pressed
`wait 10`	wait 10 seconds, or until a key is pressed, whichever is first.

Comment

The difference between the **inkey, pause, mouse** and the **wait** command is:

- inkey checks whether a key has been pressed or not
- pause prints user specified text
- mouse reads a single key or mouse button
- wait waits until a delay period expires.

See also

inkey, timer.

WFILE

Type

TIM command

Syntax

```
wfile <variable> <name> <value>
```

Parameters

<*variable*>	- a variable of type file.
<*name*>	- the name of a byte, integer or float array.
<*value*>	- the number of items (byte, integer or float) that must be written.

Function

Writes the content of an array to a binary file

Description . . .

Description

wfile writes a given number of items from an array in a binary file. The variable of type file may be initialised before using it in a file operation. Writing to the file starts at the beginning of the file, all subsequent calls of **wfile** append the data to the file. To control the position in the file where the data is to be written, use pfile.

Return Value

The number of items written is returned. In case of error -1 is returned.

Examples

```
int array [32]
int index
file outfile = "output"

; fill array with something
for index = 0 to 31
 array [index] = index
next index

; write it to file
wfile outfile array 32

stop
```

Comment

Binary files are not readable with a text editor like Notepad. To write data to files that must be readable (ASCII files) use the function fprint..

See also

rfile, pfile, fprint, fscan.

WHILE - ENDW

Type

Flow-control

Syntax

```
while <conditional expression>
  <statement>
  .
  .
  .
endw
```

Parameters

<conditional expression>

Function

conditional program loop

Description . . .

Description

The **while** statement evaluates the conditional expression.

If the result of the evaluation is TRUE, the statements between **while** and **endw** are executed. When the **endw** statement is encountered, the conditional expression is evaluated again, and the loop is repeated if the result is TRUE.

If the result is FALSE, execution continues at the command after the **endw** statement.

Example

```
while var1 > 128
  thre p var1
  var1 = label p
endw
```

Comment

If the statement within the loop has to be executed at least once, the repeat-until construction should be used.

See also

for - next, repeat - until.

ASCII file

The date in an ASCII file is readable with a text editor because it is stored as characters. For example the number 234 is stored as the characters '2', '3' and '4' in stead of the bit pattern 11101010. Of course the characters themselves are stored as bit patterns. In the example these patters are 00110010, 00110011 and 00110100. Storing your data in an ASCII file takes therefore more space than a binary file.

Attributes

Print has various format options, that can be controlled using the attribute character (**@**). When a string or a variable is preceded by a specifier like **@#** (where **#** is a number between 1 and 79, see example 1) printing occurs from the specified column number.

When a string is preceded by a specifier such as **@x** (where **x** is a character from the following list) a special printing option is selected. This option remains in effect during printing the rest of the string, unless it is overwritten by another one. Printing options are:

b	blinking colours
c	Colour option, followed by further specification (see below)
h	high intensity
i	inverse video
n	normal (terminates all options)
r	a carriage return & line feed is inserted
s	the screen is cleared
u	underline (monochrome display only)
v	invisible
z	no newline (only carriage return)

The **c** option needs further specification: **@cf#** specifies foreground colour, **@cb#** specifies background colour, where **#** is a number from the following list:

0	black
1	red
2	green
3	yellow
4	blue
5	magenta
6	cyan
7	white

The **@r** directive does not change an attribute, but issues a carriage return and line feed, thereby forcing the string to be printed on two (or more) successive lines.

The **@z** directive puts the cursor at the beginning of the current line, so the line just written will be overwritten by whatever comes next. This is useful to print running variables (see the last example) .

Examples

Start printing string at column 20, variable at column 40:

```
1. print @20 "Positioning of text ", @40 var1

2. print @i "Inverse video Attribute", @n
```

```
3. print "Sequence:", @10, seq, @z ; No line feed
```

Background colour black, foreground colour light red:

```
4. print @cb0 @cf1 @h "Red on black", @n
```

Binary file

The date in a binary file is not readable with a text editor because it is stored as bit patterns. For example the number 234 is stored as the bit pattern 11101010 in stead of the characters '2', '3' and '4'. In an ASCII file these characters would be stored as bit patterns: 00110010, 00110011 and 00110100. Storing your data in a binary file therefore saves space in comparison with an ASCII file.

Boolean expression

An expression that evaluates to a BOOLEAN value, i.e. the result of the expression is TRUE or FALSE. Example:

`(4 < 6)` evaluates to TRUE.

`(4 > 6)` evaluates to FALSE.

CFE - Command File Executer

The part of TIM that takes care of the execution of command file statements.

Character set

A character set is a collection of letters, digits and other symbols used by TIM to write text into an image. Another name for a character set is font. TIM font files have the extension '**fnt**'. They are not interchangeable with other font systems.

Debug Window

If a compiled command file is executing in debug mode, the debug window shows the statements in the source command file.

Devices

DOS knowns several standard devices:

DOS device	Explanation
`con`	console (the computer screen)
`aux`	serial output port #
`com1`	serial output port #1
`com2`	serial output port #2
`prn`	parallel printer port #1
`lpt1`	parallel printer port #1
`lpt2`	parallel printer port #2
`nul`	the bit bucket

File path

A file path is a DOS path, used to find the file.

Example

In:	`c:\timwin\im\cermet.im`
the path is:	`c:\timwin\im\`

Note: in command files specify a path using double backslashes:

```
c:\\timwin\\im\\cermet.im
```

File name

A string that is a valid DOS file name, with or without the path.

Format string

The string specifier contains text which is written literally, format specifiers to define the format of variables, and special characters for control of the string.

A general format specifier can be written as: **%**[**flag**] [**width**][**.precision**] **type**, where:

`%` Format indicator

This character indicates the start of a format specifier. To print a **%**, just enter %%.

flag (optional)

This position specifies format attributes. See the following table (items may be combined):

left adjust the variable in its field (default: right adjust)

+ prefix the output with a '+' if positive (default: only a '-' if negative)

`' '` (space): prefix the output with a 'space' if positive

add the following attributes, depending on type:

with **x** type: prefix nonzero output with **0x**

with **o** type: prefix nonzero output with **0**

with **e**, **f** and **g** type: force decimal point

with **g** type: prevent truncation of trailing zeros

width (optional)

the minimum number of characters output

.precision (optional)

the number of characters after the decimal point (except for the **g** type specifier: maximum number of significant digits)

type (required)

The data type is specified by one of the following characters:

`d` decimal signed integer (to be used for *short integers*)

`ld` decimal signed long integer (to be used for the standard TIMWIN integer type)

`u` decimal unsigned integer

`x` hexadecimal integer

`o` octal integer

`s` character string

`c` single character

`f` fixed decimal floating point

`e` exponential floating point

`g` **`f`** or **`e`**, whichever is shorter

Special characters may be included in the text:

`\n`	new line (don't use with console; see **scroll**)
`\t`	horizontal tab
`\b`	backspace
`\r`	carriage return
`\f`	form feed (for printers (prn) only)
`\a`	bell (alert)
`\xddd`	ASCII character in hexadecimal notation (ddd = 3 digits)

Examples . . .

Formatting Examples

In these examples you see the formatstring, the value to be formatted and the formatted result.

decimal:

```
%d:             23          23
%6d             23              23
%06d            23          000023
%#6d            23              23
%+6d            23             +23
%#06d           23          000023
```

hexadecimal:

```
%x              23          17
%6x             23              17
%06x            23          000017
%#6x            23            0x17
%+6x            23              17
%#06x           23          0x0017
```

octal:

```
%o              23          27
%6o             23              27
%06o            23          000027
%#6o            23             027
%+6o            23              27
%#06o           23          000027
```

string:

```
%s            "timwin"      timwin
%8s           "timwin"        timwin
%08s          "timwin"      00timwin
%#8s          "timwin"        timwin
%+8s          "timwin"        timwin
%#08s         "timwin"      00timwin
```

single character:

```
%c              68          D
%6c             68               D
%06c            68          00000D
%#6c            68               D
%+6c            68               D
%#06c           68          00000D
```

fixed decimal floating point:

```
%f             45.67        45.670000
%8.4f          45.67         45.6700
%08.4f         45.67        045.6700
%#8.4f         45.67         45.6700
%+8.4f         45.67        +45.6700
```

```
%#08.4f         45.67         045.6700
```

***exponential floating point*:**

```
%e              45.67         4.567000e+001
%14.4e          45.67            4.5670e+001
%014.4e         45.67         0004.5670e+001
%#14.4e         45.67            4.5670e+001
%+014.4e        45.67         +004.5670e+001
%#014.4e        45.67         0004.5670e+001
```

***fixed decimal or exponential floating point*:**

```
%g              45.67         45.67
%10.4g          45.67              45.67
%010.4g         45.67         0000045.67
%#10.4g         45.67              45.67
%+010.4g        45.67         +000045.67
%#010.4g        45.67         0000045.67
```

Global variables

Variables that are 'visible' in all parts of a command file, i.e. main program and subroutines. Such a variable can be used or modified everwhere in the command file. .

Identifier

A string of symbols used to denote a variable name or a label. The identifier must start with a letter (a, b, c, ...), thereafter all symbols are allowed (including numbers) except the reserved symbols that TIM uses, like **+**, *, **/ , <, >,** etc.

Example:

`jr7564!` is a valid identifier, `77465` and `*+ff` are not.

IF block

All statements between an **if** and an **elseif**, **else** or **endif** keyword; or between an **elseif** and an **else** or **endif** keyword; or between an **else** and an **endif** keyword.

Include File

A file with preprocessor directives that is included during compilation of a command file. The content of the file is treated as if it was included in the file directly.

Nesting of include files is allowed up to five levels.

Example of an include file specification:

```
#include constant.inc
```

or

```
#include "constant.inc".
```

Installation File

The installation file contains information that is used by TIM during a session, like the type of framegrabber or the file paths. The installation file can be manipulated with the Install menu item.

Macro

A simple macro (no arguments) is a definition for a replacement of a string by another string. In TIM macro's are defined by the keyword #define.

Example: the macro

```
#define IMAGE s
```

has as a result that every occurrence of `IMAGE` in the command file is replaced by `s`, but `IMAGE_DIMENSION` is not replaced.

Macro's are local to the file in which they are defined. Aliases have the same function, but their scope is global.

Passing arguments by value

A command file passes an argument by value to a second command file if only the value of the argument is passed. It is then impossible for the second command file to alter the original value of the argument.

Variables like integers and floats are passed by value. Arrays and strings are passed by reference.

Passing arguments by reference

A command file passes an argument by reference to a second command file if the address of the argument is passed. It is then possible for the second command file to alter the original value(s) of the argument.

Array arguments are passed by reference. Variables like integers and float are passed by value.

Run-time error

Error that occurs during execution of the command file. In contrast to a syntax error, a run-time error cannot be detected by the compiler.

Typical run-time errors are:

- a file that is not found
- an array index out of bounds
- a division by zero.

Size of types

The size of each variable type is:

char:	1 byte
integer	4 bytes
float:	8 bytes

TTY Window

The output of a command file that is executed is written to the TTY window. This window is compatible with the old-time DOS screen (colour possibilities etc.).

Watch Window

This window belongs to the debug window. If a command file is executed in debug mode, the value of selected variables can be observed in the watch window.

File

The File menu offers you several file related functions, and facilities for communication with other programs

- Get image	find a image file in the default image directory
- Get font	load a character font
- IO operations	printing, plotting, managing Ibuf files, DDE

Get Image (File menu)

This dialog box allows you to delect an image file from the default image directory (as defined in Install). When selected, presing OK copies the image to the default destination image.

You can also specify a directory manually.

Get Font (File Menu)

This dialog box lets you select a character font for writing text into images. After selecting a file, pressing OK loads the selected font into memory.

IO Operations (File menu)

Print/Plot	Not available
Postscript	Creating PostScript files from images
Read Ibuf	Reading Ibuf files
Write Ibuf	Writing Ibuf files
Excel Link	Establish a DDE link for sending Ibuf data to MS-Excel

Print/Plot (File menu)

Currently not available

Postscript (File menu)

PostScript is a graphical description language. Images in PostScript format can be printed by many devices and imported in many text processors.

Image

Specify the source image here (may be a sub-image)

File

Specify a filename. Default is: TIMWIN in the TIMWIN directory

Bits

Specify the number of bits in the image: 1, 2, 4 or 8. The higher the number, the more detailed the image will be, and the bigger the file.
In each case the top bits will be used. For example, if you specify 2, bits 8 and 7 will be used.

Width

The width of the image in cm. The heigth of the image will be calculated accordingly, keeping the frame grabber's aspect ratio in mind. If this correction is not desired, first set the Aspect ratio to 1.0. (See Aspect Ratio)

Type

Select one of the radio buttons:

- PostScript - to get a file that can be immediately printed. File extension will be: .ps
- Encapsulated PostScript - to get a file that can be imported in a document. File extension will be: .eps

Graphic Invert

Check this box if your image has a graphic character. On screen, lines generally are white on black. The printed result will look more natural if printed black on white.

Command equivalent: ps

PostScript is a trade mark of Adobe Systems Inc.

Read Ibuf (File menu)

To read an Ibuf file:

1. select a file
2. press OK

The data will be loaded from the file into Ibuf, as well as the housekeeping data (data type, amount of data). To write an Ibuf file, use the Write Ibuf menu

Command equivalent: ribuf

Write Ibuf (File menu)

To write an Ibuf file:

1. select a file
2. press OK

The data in Ibuf will be written to the file, as well as its housekeeping data (data type, amount of data). The data can be read back into Ibuf using the Read Ibuf menu.

Command equivalent: wibuf

Excel link (File menu)

This function establishes a two-way link with Microsoft Excel using Window's DDE mechanism.
With a data link you can copy the content of Ibuf to an Excel spreadsheet.
With a command link, you can enter a regular TIMWIN command (including TIMWIN programs) in a selected spreadsheet cell, which is then sent to TIMWIN and executed, as if it were entered in TIMWIN's edit field. This is useful if you must control TIMWIN from an Excel macro.

To set up the link:

1. invoke Microsoft Excel
2. in the Sheet name edit control, enter a sheet name. Or, use the default sheet (Sheet1)
3. in the File menu, select IO-operations, and then Excel Link
4. in the dialog box, select the link you want to establish by clicking one of the following radio buttons in the Link field (you may establish both links, one after another):
   - Ibuf if you want to establish a data link
   - command if you want to establish a command link

Ibuf Link (Excel link - File menu)

To set up a data link for bringing Ibuf data to an Excel spreadsheet, fill in the following fields:

Link:
check **Ibuf**

Sheet name:
the name of the spreadsheet where you want your data to appear. This must be an active sheet. Only one sheet can be open at a time, so if a Command link is already present, you must use the current sheet.

Start cell:
enter the horizontal and vertical values of the starting cell

Orientation:
Select colums if you want the data to be written columnwise

Auto Update Ibuf:
Check this if you want the data to be copied automatically whenever Ibuf's content changes

- To make the link active, click the **Link** button. The link status is indicated.
- To send data, click the **Send Ibuf** button.
- To end the link, click the **End Link** button

See also: Command link

Command equivalent: excelo. See also: excels and excelc

Command Link (Excel link - File menu)

To set up a command link for issuing TIMWIN commands from an Excel spreadsheet, fill in the following fields:

Link:
check **Command**

Sheet name:
the name of the spreadsheet in which you want to enter TIMWIN commands. This must be an active sheet. Only one sheet can be open at a time, so if a Data link is already present, you must use the current sheet.
You don't need to specify this entry if you select **Excel copy** in **Cell info**.

Cell info
If you want to select specific cell for entering commands, check **row, column.**
You can also select a cell in Excel, and copy it to the Clipboars using the (Excel) Edit - Copy menu.
After that, you can check **Excel copy** to import the cell's data.

Command cell:
Enter the horizontal and vertical values of the cell in which you want to enter TIMWIN commands.
You don't need to specify this entry if you selected **Excel copy** in **Cell info**.

- To make the link active, click the Link button. The link status is indicated.
- To end the link, click the End Link button

See also: Data link

Edit Menu

This menu offers you Clipboard and various Window Edit functions

Copy	Copy selected data to the clipboard
Paste	Paste the clipboard content into the window
Clear	Clear the edit window
Filter	Bring up the filter window
Ibuf	Bring up the Ibuf edit window
Image	Bring up the image edit window

Copy (Edit menu)

Use this command to copy selected text onto the Clipboard. This command is unavailable if you have no selected text in the edit window.

Copying text or graphics to the Clipboard replaces the contents previously stored there.

Paste (Edit menu)

Use this command to insert a copy of the Clipboard contents at the insertion point. This command is unavailable if the Clipboard is empty.

Clear (Edit menu)

Use this command to clear the Edit window

Filter (Edit menu)

This command brings up the Filter window. The Filter window allows you to enter a convolution kernel, that can be used by the filt command.

File	Manipulate filter files
Edit	Clipboard usage
Size	Choose a kernel size
Symm	Specify symmetry

File (Filter menu)

This menu allows you to control files, associated with the Filter function. In addition you can exit this function.

New Start a new worksheet
Open Open an existing filter file
Save Save the worksheet in the current file
Save As Save the worksheet in a new file
Exit Exit the Filter menu

Edit (Filter menu)

Use this command to copy the content of the Filter edit window onto the Clipboard.

Size (Filter menu)

Use this command to specify the size of the convolution kernel. The following sizes are supported:

3x3
5x5
7x7
9x9

Symm (Filter menu)

Use this command to specify the presence of symmetry in the convolution kernel. This feature helps you to enter a kernel faster and more accurate.

Horizontal	If you enable horizontal symmetry, a number entered in the left part of the window is mirrored at the right, and vv.
Vertical	If you enable vertical symmetry, a number entered in the upper part of the window is mirrored in the lower part, and vv.
Both	If both are enabled, a number entered in one quadrant is mirrored in the other three.

Ibuf (Edit menu)

The Ibuf window allows you to view and modify the content of TIMWIN's Cut & Paste buffer Ibuf.

File	Manipulate Ibuf files
Edit	Clipboard usage
Type	Control the type of Ibuf numbers
Options	Control updating of Ibuf edit window

File (Ibuf menu)

This menu allows you to control files, associated with the Ibuf function. In addition you can exit this function.

New	Start a new buffer
Open	Open an existing Ibuf file
Save	Save the Ibuf worksheet in the current file
Save As	Save the Ibuf worksheet in a new file
Exit	Close the Ibuf window

Edit (Ibuf menu)

Use this command to copy the content of the Ibuf edit window onto Window's Clipboard. This allows you to paste the data into another application.
The data is copied in ASCII (text) format, so that you can import it easily in a spreadsheet or word processor.

Type (Ibuf menu)

Use this command to change the current Ibuf data type
Ibuf can contain numbers of the following types, depending on use:

Data type	size	application
byte	8 bits	look up tables
word	16 bits	
long word	32 bits	histograms

If you change data type the bytes currently present in Ibuf will be rearranged in the new type, so that their original meaning generally will be lost.

Options (Ibuf menu)

If the Update menu is checked, the Ibuf edit window is refreshed atomatically whenever Ibuf's content changes. If it is not checked, the window remains the same. This does not mean that the data in Ibuf will not change!

Image (Edit menu)

The image edit window allows you to view and modify the content of an image in numerical mode.

If you select this function, a dialog box appears. that allows you to choose an image. You can select an image and then click OK. This will bring up the image edit window.

You can position the image edit window over any part of the image, using either of the following methods:

- using the scroll bars
- using the arrow keys
- moving the cursor in the image

You can edit the pixel values by entering a value in the Value box or the central box

Edit	Clipboard usage
Options	Control updating of Ibuf edit window

Edit (Image Edit menu)

Use this command to copy the content of the Image edit window onto the Clipboard. The data is copied in ASCII format, so that you can import it easily in a spreadsheet or word processor.

Options (Image Edit menu)

If the Update menu is checked, the Image edit window is refreshed atomatically whenever the image's content changes. If it is not checked, the window remains the same. This does not mean that the data in the image will not change!

Contr Menu

The Control menu offers facilities for setting up your system.

Alias	Alias definitions
Destination image	Choose a destination image
Images	Define available images
Installation	Installation properties
LUT	Look Up Table functions
Set	Various system settings

Alias (Contr. menu)

Use this menu if you want to change the alias definitions.

This menu brings up an editor, that allows you to enter or modify alias definitions, which are stored in a file **ALIAS.TIM**. The left column contains the alias terms, and the right column contains the substitutions. You can also enter comments: everything between a semicolon and the end of the line is ignored.

Changes in the alias definitions have effect from the next time you run TIMWIN.

Destination image (Contr. menu)

Use this menu if you want to change the destination image.

The list box allows you to select from the currently selected image type. You can change the image type in either of the following ways:

- enter another image name in the **Image** field manually
- select another group using the status bar
- use the dest command

Images (Contr. menu)

Use this menu if you want to add, delete or change image definitions.

This menu brings up an editor, that allows you to enter or modify image definitions, which are stored in a file **IMAGES.TIM**. The file format is demonstrated in the following fragment.

Name	type	pixel	size		upper left pt.	
fg	dis	12	1024	1024		
m	-	-	512	768	0	0
n	-	-	512	768	512	0

See also:

- groups
- combining pixel types
- size groups

Name (Control Images menu)

The Name column contains the name, that must be used to address the image. Any name is permitted, but be careful to avoid reserved keywords, etc.

The use of standard names (as can be found in the Images files that come with your package) is recommended, since this guarantees compatibility with programs developed elsewhere.

Type (Control Images menu)

The following image types can be selected:

`dis` (display) an image located in a frame grabber
`mem` (memory) an image located in computer memory
`win` (windows) an memory image with a (Windows) display window attached

See also: Image groups

Pixel (Control Images menu)

The following pixel types are available (the numbers indicate pixel size in bits)

8 standard pixel size for most applications
12 pixel size in 12-bits frame grabbers
16 16 bits memory images
32 32 bits memory images (reserved)
64 complex floating point images for FFT applications

12 bits pixels are actually 16 bits pixels. Since 12 bits frame grabbers lack the upper 4 bits, specifying 12 bits informs TIMWIN that these bits contain garbage so that they can be erased when necessary.

Size (Control Images menu)

The maximum image size TIMWIN can handle is 1024x1024, the minimum 9x9
Horizontal size in memory images must be a multiple of 16. Any size below the maximum is allowed.

The first number is the vertical size, the second number is the horizontal size.

Upper Left Point (Control Images menu)

Child images are located somewhere in a master image (see also: image groups). The location is determined by specifying the coordinates of the upper left corner of the child image.

Be sure to specify the upper left point so, that the entire image is located inside the master image.

Groups

Images of different sizes can be grouped together, so that they occuppy the same physical memory. Since, in a typical application, images of a common size are used, this makes memory usage more economical.

Images belonging to group are specified in one block in the file, without empty lines between them.

Master image	The first line specifies the master image. This image must be large enough to contain every other image of the group.
Child images	The following lines specify the child images. They have an additional Upper Left Point field, that determines their position in the master image.
Pixel type	Child images don't need to be of the same pixel type as the master image: a 'lower' type may be specified. If the pixel type is equal, a hyphen (-) may be specified.
Image type	The image type of a child image must be equal to the masters'. This is simply indicated by using a hyphen (-) in this field.

Important:	Windows images must be specified individually. They must not be part of a group

Combining pixel types

Some important notes on combining pixel types:

- In frame grabbers the pixel type is determined by the hardware. Don't specify a deviating pixel type there.
- Windows (type win) images must be 8 bits
- To calculate the mapping of different pixel types in memory, multiply the horizontal image size with the pixel size

Size groups

Working with images, you combine images from several sources (memory, display and windows), but usually of the same size. These size groups are in contrast to the definition groups (master and child images), which are important in the definition phase.

Installation (Contr. menu)

Use this menu if you want to record a certain definition. The definitions in this menu are stored in a file timwin.ini, and remembered until they are changed.

The following fields can be chosen:

File Paths	default directories for command files and images
Camera type	video camera options
Frame grabber	frame grabber options
Clock	video synchronisation
Screen output	return parameter options
Compiler options	compiler options
Editor	editor to be used with TIMWIN
Bit patterns	graphic options
Aspect ratio	correction for non-square pixels

File Paths (Contr. menu - Installation)

Initialization file	The command file that must be run automatically when TIMWIN is started
History file	The file to contain history information
Path command files	The path where command files (sources) must be found
Path compiled files	The path where the compiler has to store compiled command files and where TIMWIN must look for executables
Path image files	The path where TIMWIN looks for image files.

Note: If a file to be read is not found in the specified directory, TIMWIN also looks in directories specified by the following entries in the file TIMWIN.INI (located in the Windows directory):

[Default Settings]	header
CmcPath1=	alternative command file path
ImagePath1=	alternative image path
CmcPath2=	2nd alternative
ImagePath2=	etc.

Images are always stored in the directories, specified in the Install menu; for writing files the environment variables play no role.

Camera type (Contr. menu - Installation)

This dialog box allows you to select standard and special camera's that TIMWIN supports.

- If you have a camera, that is listed in the list box, select that one.
- If you are working in variable scan mode, and your camera is not mentioned, select **Variable**
- If you have another camera or a standard video camera, select **Standard**
- If you don't have a camera, select **None**

Frame grabber (Contr. menu - Installation)

This dialog box allows you to select frame grabbers that TIMWIN supports. In addition, you must specify some values that TIMWIN needs to know to access the frame grabber.

- If your frame grabber is listed in the list box, select it.
- If there is no frame graber in your system, or if you choose not to use it, select **None**

If you selected a frame grabber, you must specify values in the Segment and I/O address boxes.

Segment specify the segment value
IO address specify an IO address

Segment value (Contr. menu - Installation)

In order to be able to access its memory, the frame grabber must be incorporated in the computer's memory map. This is done by specifying the Segment value.

Some important notices:

- The segment value you enter must correspond to the value that the frame grabber is configured to using its jumpers or dip switches. Usually `D000` is a good value.
- Be sure that no other device is mapped to the selected area.
- The Windows System file SYSTEM.INI must have a line in the `[386Enh]` part, like:

  ```
  EMMExclude=D000-DFFF
  ```

  The value must correspond to the value the frame grabber is configured to.

Consult the frame grabber, Windows and MS-DOS manuals for details.

IO address (Contr. menu - Installation)

In order to be able to control the frame grabber's registers, TIMWIN must know its IO-address. This is done by specifying the IO-address value.

Some important notices.

- The IO-address value you enter must correspond to the value that the frame grabber is configured to using its jumpers or dip switches. See the frame grabber's manual for details.
- Be sure to specify a value, that is not used by the computer for any other installed device (e.g. a network card). Usually 300 is a good value (Cortex-I: 230)

Note: these values are considered hexadecimal

Clock (Contr. menu - Installation)

The frame grabber clock is the base of the video timing signals. If the frame grabber runs in a stand alone configuration, whatever the frame grabber generates is OK. But if a camera is connected to the frame grabber, there must be some synchronizing between them in order to grab a correct image. The clock setting allows you to make the frame grabber a master or a slave.

Use this dialog box to specify whether

- the frame grabber generates its own video timing signals: select Xtal
- the frame grabber is a slave of a video camera: select PLL.

If you select PLL and no video camera is connected, the frame grabber automatically switches to XTAL mode.

If the dig operation results in a "running" image, check the clock setting.

Screen output (Contr. menu - Installation)

Use this dialog box if you want to control the output messages, that TIMWIN writes after executing a command. These messages consist of a result (usually numerical), embedded in an informative text.

- if you want to see the message, check the **Print return parameter** box.
- if you want the return values to appear in the decimal format, select **decimal**
- if you want the return values to appear in the hexadecimal format, select **hex**

Compiler options (Contr. menu - Installation)

Use this dialog box if you want to control the following compiler options

Update old cmc file
If a compiled command file is out of date with respect to the source file, you can select from the following options:

- automatic — the compiler compiles automatically whenever necessary
- prompt — the compiler prompts you for confirmation if it encounters an out-of-date source file
- no update — the compiler is never invoked

Aliases
Check this box if you want to use the definitions in the file ALIAS.TIM in the compiling process. If checked, you can use aliases in command files, which will be correctly substituted by the compiler.

Debug
Check this box if you want the compiler to include source- and symbol information in the compiled command file. This is required if you want to use the debugger.

Editor (Contr. menu - Installation)

Use this dialog box if you want to specify an editor to be used with TIMWIN. The editor you select in this dialog box will be invoked by TIMWIN whenever a text editor is necessary.

Default is TIMWIN's own editor EditCF

Bit patterns (Contr. menu - Installation)

Use this dialog box if you want to change any of the following bit patterns for immediate and future use:

- Graphics value — Fill in: the bit pattern to use. E.g. 128: only the most significant bit will be modified
- Drawing value — Fill in: the pixel value to use
- Cursor value — This value is used to display the (frame grabber) image cursor. Fill in: the bit pattern to use

---

Note: if you only want to change this setting temporarily, you'd better use the corresponding **Set** menu. This setting has an effect that lasts as long as the session.

Aspect ratio (Contr. menu - Installation)

Use this dialog box to modify the value, that TIMWIN uses to compensate for non-square pixels.

A good starting value is:

1.00 for frame grabbers having square pixels (VS100, VFG)
1.40 for other frame grabbers (PCVision, PCVisionPlus, Cortex-I)

Image extension

The image extension makes TIMWIN distinguish between two image file types:

.im	standard TIWIN type; no header - just pixel data
.tif	TIFF images

The selected image type will be visible in the image file list box.

For handling standard images, use copy, dis and save
For handling TIFF images, use tcopy, tdis and tsave

LUT (Contr. menu)

Use this menu if you want to change look up tables for the frame grabber or Windows images.

LUT	Select any of the targets for the LUT operation
Table no.	For FG-Luts: the number to fill
Lut to fill	Select a single colour or all colours
Function	Select the LUT pattern to create

Command equivalent: lut

LUT target (Contr. menu - LUT)

You can select any of the following targets for the LUT operation:

Frame Grabber Input	The frame grabber input LUTs allow you to transform an image while it is aquired.
Frame Grabber Output	The frame grabber output LUTs allow you to transform the appearance of the image, displayed on the frame grabber.
Windows image	The Windows LUT allow you to transform the appearance of the image, displayed in a Windows image

Table no. (Contr. menu - LUT)

The table number allows you to specify one of the Look Up Tables, that is available in the frame grabber. This field is not available when a Windows image is selected as a target.

The following table shows the number of available LUTs in various frame grabbers:

Frame Grabber	Input tables	Output tables
PCVision	4	4
PCVisionPlus	8	8
Series 100	16	16
VFG	16	16

Lut to fill (Contr. menu - LUT)

In the output circuit are three channels: one for red, green and blue. You can select any of these or all of them as a target for any non-colour table pattern.

If you select a single colour, only that channel will be written. The other channels keep their original pattern.

If you select all, all channels will be written with the same pattern. This will result in a black and white display.

This field is not available for Input LUTs.

Function (Contr. menu - LUT)

This field specifies the pattern that will be loaded in the selected LUT(s). You can select from :

Black and White patterns

- linear — 0 - black, 255 - white
- inverse — 255 - black, 0 - white
- logarithmic — compresses dark parts, expands bright parts
- load Ibuf — load any user-defined pattern

Colours ... (Standard Colour Patterns)

- function 5 — hard colours in 3 bitplanes
- function 6 — soft colours in 3 bitplanes
- function 7 — make one bitplane red
- function 8 — make one bitplane green
- function 9 — make one bitplane blue
- function 10 — real colour display
- function 14 — spectrum table
- function 15 — sine table
- function 105 — hard colours in 3 bitplanes (12-bits tables)
- function 106 — soft colours in 3 bitplanes (12-bits tables)
- function 110 — real colour display (12-bits tables)

Manual...

create a pattern graphically

Colour function 5 (Contr. menu - LUT - colour)

This function assigns the red, green and blue output channels to a group of 3 bitplanes. If a bit contains a 1, the colour is on, otherwise it is off.

The bitplanes 'below' are still used for display of grey values. The grey value step size is adjusted according to the available range.

The bitplanes 'above' don't play a role.

Colour function 6 (Contr. menu - LUT - colour)

This function assigns the red, green and blue output channels to a group of 3 bitplanes. The intensity of a displayed colour depends upon the underlaying grey value.

The grey value step size (determined by the bitplanes 'under' the colour bits) is adjusted according to the available range.

The bitplanes 'above' don't play a role.

Colour function 7 (Contr. menu - LUT - colour)

This function modifies an existing table so, that a pixel will be red when the specified bit is one.

Note: Since not all table entries will be set by this function, be sure that a base table (e.g. a linear table) is loaded, before starting this function

Colour function 8 (Contr. menu - LUT - colour)

This function modifies an existing table so, that a pixel will be green when the specified bit is one.

Note: Since not all table entries will be set by this function, be sure that a base table (e.g. a linear table) is loaded, before starting this function

Colour function 9 (Contr. menu - LUT - colour)

This function modifies an existing table so, that a pixel will be blue when the specified bit is one.

Note: Since not all table entries will be set by this function, be sure that a base table (e.g. a linear table) is loaded, before starting this function

Colour function 10 (Contr. menu - LUT - colour)

This function is used for display of real colour images. You must specify the number of colour levels for each colour. Since the available number of colours is 256, the product of these numbers must be 256 or less. A good compromise is 6 (red), 7 (green) and 6 blue).

To display a colour image you must have a red, green and blue image, whose pixel values must correspond to the selected colour levels. See the various 'colour' command files for examples.

Colour function 14 (Contr. menu - LUT - colour)

The spectrum function maps the pixel values to a continuous colour spectrum. This is done as follows:

colour	pixval 0	pixval 128	pixval 255
red	100%	50%	0
green	0	100%	0
blue	0	50%	100%

Colour function 15 (Contr. menu - LUT - colour)

The sine function distributes the grey values as a sine. For each colour the phase can individually be specified

Colour function 110 (Contr. menu - LUT - colour)

This function is used for display of real colour images in a 12-bits display. You must specify the number of colour levels for each colour. Since the available number of colours is 4096, the product of these numbers must be 4096 or less. A good compromise is 16 (red), 16 (green) and 16 blue).

To display a colour image you must have a red, green and blue image, whose pixel values must correspond to the selected colour levels. See the various 'colour' command files for examples.

Colour function 105 (Contr. menu - LUT - colour)

This function assigns the red, green and blue output channels to the bitplanes 9, 10 and 11. In addition, bitplane 12 sets all colours which results in white. If a bit contains a 1, the colour is on, otherwise it is off.

The bitplanes 1 to 8 are used for display of grey values and filled with a linear table.

This table allows you to concurrently display a full scale black and white image in the bitplane 1 to 8, while performing bitplane operations or graphics in the overlay bitplanes 9 to 12.

Colour function 106 (Contr. menu - LUT - colour)

This function assigns the red, green and blue output channels to the bitplanes 9, 10 and 11. In addition, bitplane 12 sets all colours which results in white. If a bit contains a 1, the colour is on, otherwise it is off.

The bitplanes 1 to 8 are used for display of grey values and filled with a linear table. This table differs from table 105 in that the colour intensity depends on the underlaying grey value.

This table allows you to concurrently display a full scale black and white image in the bitplane 1 to 8, while performing bitplane operations or graphics in the overlay bitplanes 9 to 12.

Manual colour function (Contr. menu - LUT - manual)

This function allows you to create a look up table pattern graphically by dragging a curve. On a frame grabber image, you will see the colours change immediately.

Red	Select the red LUT to fill
Green	Select the green LUT to fill
Blue	Select the blue LUT to fill
As one Show all	Select all LUTs to fill at once
Copy LUT . . .	Copy a created LUT pattern into another colour

Procedure

1. Select a colour by clicking one of the colour option buttons
2. Position the cursor in the graphic field
3. While keeping the left mouse button pressed, move the cursor pointer along the path that you want your curve to follow.
4. When done, select another colour and repeat the procedure

Set (Contr. menu)

Use this menu if you want to change one of the following settings during a session.

Access bit mask	Masks for protecting frame grabber bitplanes
Bit pattern	Several patterns used with graphics.
Calibration factor	Constant for modifying return value (scaling)
Clock	Set the frame grabber's internal timing
Frame Grabber no.	Select one of three frame grabbers
Gain/Offset	Adjustment of frame grabbers's video input circuitry
Video Input	Select one of the frame grabber's video inputs
Window Update	Control flags for updating windows

Command equivalent: set

Note: Modifying a value in the Set dialog box has effect only during the session. After quitting TIMWIN the settings are lost. To make changes permanent, use the **Install** menu

Access bit mask (Set)

Most frame grabbers allow you to set up a hardware mask, that prevents selected bitplanes from being written to. Usually you can choose between two modes:

- host access - the software can't write to the bitplanes
- video access - while grabbing the bitplanes remain unmodified

Example:
Keeping the lease significant bitplane from being overwritten during grabbing allows you to write graphics into that bitplane (for example: a histogram), or write an image processing result (for example: the location of a detected object).

Bit patterns (Set)

Use this dialog box if you want to change any of the following bit patterns for immediate and future use:

- Graphics value — Fill in: the bit pattern to use. E.g. 128: only the most significant bit will be modified
- Drawing value — Fill in: the pixel value to use
- Cursor value — This value is used to display the (frame grabber) image cursor. Fill in: the bit pattern to use

---

Note: if you only want to change this setting for future use, you'd better use the corresponding **Install** menu.

Calibration factor (Set)

Use this function to specify the Calibration value.

Clock (Set)

Use this function to specify the source of the frame grabber's clock circuitry:

- Xtall (Frame grabber is master)
- PLL (Frame grabber is slave)

To make the change permanent, use the corresponding Install function

Frame Grabber no. (Set)

If you have more than one frame grabber in your system, you can use this function to select another frame grabber (make another frame grabber active).

Only one frame grabber can be active at a time.

Frame Grabber Gain/Offset (Set)

This dialog box allows you to adjust the settings of the video input circuits of your frame grabber. The procedure to do so is as follows:

1. Connect a video source that produces a standard video signal
2. Select a linear input LUT (`lut 1 1 1`)
3. Select a contrast enhanced output LUT (pseudo colour: `lut 2 2 6 6`)
4. Start grabbing (`dig`)
5. Adjust the offset so, that the darkest part of the image is not entirely black (some structure must remain visible. If necessary, adjust the monitor)
6. Adjust the gain so, that the brightest part of the image is between the colours magenta and white
7. Repeat the adjustments 5 and 6, until no changes are necessary anymore.

Frame Grabber Video Input (Set)

Most frame grabbers have more than one video inputs. This setting allows you to select one of them.

Window Update Flags (Set)

The following windows can be instructed to update their content each time the corresponding data changes. These settings allow you to enable or disable automatic updating of the windows.
The flags are continuously modified by the corresponding windows functions as well, so that the status that it's reporting only represents a momentary one.

Graph	to update the graphic display of Ibuf's content in the graphic window
Histogram	to display the histogram (graphically) of every processed image
Ibuf	to update the content of Ibuf in the Ibuf Edit window
Image	to update the image content in the Image Edit window
Statistics	to update statistical data in the Statistical window with each processed image

ImageProc Menu

This menu gives you access to the image processing functions. The operations are grouped into related groups: families. An alternative grouping corresponds to an ~~application sequence~~.

FFT (Image Proc Menu)

The FFT dialog box makes setting up an FFT image processing sequence easy.

1. Prepare an image to be processed in a standard 8-bits image.
2. Select this image in the Source list box.
3. Select an image in the Complex list box. This list box shows the available complex floating point images. If the box is empty, then there is no image of this type and size. Select another image size and try again.
4. Click the arrow between the Source and Complex box.
5. Click the Perform button. Now the image is converted from 8-bits integer to 64-bits complex floating point.
6. Click the arrow between the Complex and Fourier Domain boxes
7. Click the Perform button. Now the image is transformed from space domain to frequency (Fourier) domain. This may take some time.
8. Now you can:
   - visually check the FFT image
   - perform filtering in the FFT domain
   - transform back the FFT image into space domain

Visualize the FFT image

To visually check the FFT image:

1. In the Display box, select an image where the result can be written to.
2. Check the Converted Modulus option button.
3. In the edit field near this button, enter a multiplication factor. Generally, 128 will do.
4. Click the arrow between the Complex and Display box.
5. Click the Perform button. The Fourier image will appear

Filtering in the FFT domain

To filter in the FFT domain:

1. Prepare an image that contains an appropriate frequency mask. A good start can be the visualized FFT image itself, which shows the main frequencies.
2. In the Filter box, select this image.
3. Check the arrow between the Filter and Fourier domain bex
4. Click the Perform button. Now the FFT image will be multiplied with the mask image, which will result in enhancement and suppression of frequencies in the FFT image

Transform back the FFT image

To transform the FFT image back into space domain:

1. Click the (left pointing) arrow between the Fourier domain and Complex box
2. Click the Perform button. Now the FFT image will be transformd from frequency domain to space domain
3. In the Display box, select an image where the result can be written to.
4. Check the Modulus option button.
5. Click the arrow between the Complex and Display box.
6. Click the Perform button. Now the complex floating point image will be transformed to 8-bits integer pixels, that can be displayed.

Applic. Menu

This menu gives you access to several image processing functions, grouped according to project stage.
To see the operations grouped functionally, see ~~families~~.

Graph Menu

This menu contains several graphic functions

Drawing	Drawing lines and other figures
Plotting	Plotting several 2- and 3-D graphs in an image
Text	Writing text into an image
Increment	Increment pixel values along a line

Sources of related information:

- Graphics Command family
- Concepts of graphics

Graphics Concepts

Graphics in TIMWIN is like a toolbox. There are several elements, which can be combined. The command names represent the elements, as can be seen in the following table (the references will bring you to a representative command):

shape elements	name	
lines	ln	
vectors	vec	
Freeman patterns	pat	See also: fcont

drawing elements	name	uses default
drawing pixels	dr	drawing value
incrementing pixels	inc	
reading pixels	rd	
setting bitplanes	or	graphics value
changing bitplanes	xor	graphics value
scanning to bitplane	sb	
scanning to greyvalue	sg	

Examples:

`drln`	drawing a line
`xorvec`	writing a vector by changing pixels in specified bitplanes
`rdpat`	reading pixels along a Freeman path

Notes:

- Not all combinations exist. For an overview of existing operations, see the family of Graphic operations and the description of the individual commands.

Pattern Recognition Menu

Reserved for future use. See also: image processing pattern recognition .

FrGrabber Menu

This menu contains functions for aquisition of images using a frame grabber and a video camera.

Digitizing	Normal aquisition
Real time	Real time processing of grabbed images using special hardware

CommFile Menu

Use this dialog box to manage a TIMWIN command file program.

Managing files	Select an existing file or initiate a new one
Compile	Compile the selected command file
Run	Run the selected command file
Edit	Invoke the editor on the selected file
Alias	Enable/disable the alias function
Debug	Enable/disable the debug facility

To learn more about creating command file programs, consult the Command File Description.

Managing files (CommFile menu)

The file and directory list box point to the Command file source- and executable directories, selected in the Install menu.

To change the source directory

- Select another directory using the Directories list box

To change the executable directory

- Edit the directory string in the ExePath edit field. This directory is where the compiler writes its results. It is not the directory, where TIMWIN reads its executable command files!

To select a file:

- double click a file name in the Files list box
- Or, fill in a file name in the Filename edit field (this may be a non-existing name)

Compile (CommFile menu)

To start compiling a selected source command file,click the Compile button. To select a command file, see Managing files

Run (CommFile menu)

To run a selected command file , click the Run button. If a valid compiled version exists, it will be started. If not, the compiler will first compile it.

Edit (CommFile menu)

To edit a selected command file, click the Edit button. The editor, selected in the Install emnu, will be invoked with the selected command file name as target.

Alias (CommFile menu)

To compile with the Alias function enabled, check the Aliases checkbox. The file ALIAS.TIM must be located in TIMWIN's home directory.

Debug (CommFile menu)

To compile with the Debug function enabled, check the Debug checkbox. This option is required if you want to debug the command file program.

View Menu

Use this menu if you want to control the following windows

Command file window	input/output for command files
Graph window	shows graphics
History window	keeps the command history
Statistics window	shows image statistics
Status window	shows image and frame grabber status
Image windows	controls windows images

Command file window (View menu)

This window mimics a 80 character x 24 line TTY computer terminal. Command files write their output there, and user input can be asked via this window.

The Command file window comes up automatically whenever a command file writes to it, or you can bring it up using this menu function.

A log file is available, that keeps everything that is written to the window. The file name is: TIMWIN.LOG in the default TIMWIN directory.

File menu

- Open Log file — This function opens a Log file,
- Save Log file — This function saves an opened Log file
- Exit — To kill this window, select exit.

Edit menu

- Copy — To copy the content of the window to the clipboard
- Clear — To clear the window

Graph (View menu)

The graph window shows the content of Ibuf graphically. It is also used to draw image histograms, manually (using the hist command), or automatically (see below).

Options

- Ibuf update — check this menu item to repaint the graphics window everytime a function writes new data into Ibuf
- Histogram update — check this menu item to plot a histogram after each image processing function

History (View menu)

The history window keeps the executed commands. You can select one or more commands for editing or immediate execution, write them to a file.

You can also toggle the History window on and off by pressing the ALT-H key combination.

Selecting lines

- You can select one or more lines by clicking on them

Copy

- Click the Copy button to copy the selected lines to the Clipboard

Delete

- Click the Delete button to delete the selected lines from the history

Clear

- Click the Clear button to clear the entire history buffer

Execute

- Click the Execute button to execute the selected lines

Exec/Edit

- Click the Exec/Edit button to execute the selected lines but the last. The last line will appear in the edit field in the main window, allowing you to make changes.

Statistics (View menu)

The Statistics window shows several statistic values, calculated from an image's histogram. The window comes up after executing the stat command. It can also be used to show image statistics after each image processing operation (see below).

File

- Exit Press exit to kill this window

Edit

- Copy Press copy to copy the window's content onto the Clipboard.

Options

- Update Press Update to calculate statistic values after each image processing operation

Status (View menu)

The status bar shows several image- and frame grabber properties. It can also be used to control these properties.

Shape	Select a horizontal or vertical status bar
Destination	Select the active (destination) image
Cursor	Show and control the active image's cursor value
LUT	Show and control the frame grabber's LUTs
Format	Show and control the active image's sub image format
Zoom	Show and control the frame grabber's zoom setting

Shape (Status bar)

You can select a horizontal or a vartical status bar, depending on the arrangement of your window.

To select a horizontal status bar:

- Check Horizontal in the View menu

To select a vertical status bar:

- Check Vertical in the View menu

Destination (Status bar)

To find the available images and their properties:

- In the Status Bar, press the Change button, This will bring up the Image Groups dialog box, which shows you the available images in the current active group. Pressing Next will show the images of the next group, etc.

To select an active image:

If the requested image is on one of the buttons:

- click the button

If the requested image is not on one of the buttons, but it has the same size as the current active image

1. press the [<<] and/or [>>] buttons to bring the requested image on one of the buttons
2. click this button.

If the requested image does not have the same size as the current active image:

1. press the Change button, This will bring up the Image Groups dialog box
2. Press Next repeatedly until the list box shows the requested image
3. Select the image by clicking it
4. Click OK

Cursor (Status bar)

The Cursor field shows the active image's cursor value. To change this value:

- Press the arrow buttons
- Or, fill in a value in the edit field

To let the mouse control the image cursor instead of Windows, press the Cursor button

To let the mouse control Windows again, press the right mouse button.

Command equivalent: curs

LUT (Status bar)

The LUT field shows the frame grabber's Look Up Table status. If no number is visible, no LUT command has been issued since starting TIMWIN, so that the LUT status is still indeterminate.

To select a LUT:
Press the appropriate arrow buttons until the desired LUT is selected.

Command equivalent: lut

Format (Status bar)

The Format field shows the active image's sub image format. To change this value:

- Press the arrow buttons
- Or, fill in a value in the edit field

Command equivalent: frmt

Zoom (Status bar)

The Zoom field shows and controls the frame grabber's zoom status. If no number is visible, no zoom command has been issued since starting TIMWIN, so that the zoom status is still indeterminate.

To change a zoom value:

- Press the appropriate arrow buttons until the desired zoom value is reached.

Command equivalent: zoom

Images (View menu)

The images field controls the presence of the Windows images.

To make an image visible:

- check the desired image
- Or, make it the target of an image processing operation.

To make an image invisible:

- remove the check on the image

Other options to hide, iconize or control the size of Windows images are in the image window itself.

PROCEDURES

This section gives information about procedures. The following topics are covered:

1. Introducing TIMWIN Demo
2. Installation & Setting Up
3. Managing Windows
4. Entering commands
5. Displaying images
6. Command file programs

TIMWIN Demo

TIMWIN Demo Release is a full version of TIMWIN, with a few limitations and enhancements.

Enhancements

- The Demo version is not protected from illegal use by a dongle.
- The Demo version may be copied and distributed freely.

Limitations

- The Demo version can only deal with 256x256 images. You may define other sizes, but they cannot act as a destionation of operations (with a few exceptions)
- The Demo version cannot write images to disk
- The Demo version has no compiler. However, it can run command files, that are compiled on a full system.

This program can be used in either of two ways:

- you can use it interactively, entering commands using the menu or the command line (in the bottom of the main window). To become familiar with the system, consult the main help index (Press the Contents button)
- you can run the demo command file program This program can be invoked by entering
  `/demo1`
  on the command line.

Installation

After installation from the distribution floppy disks, TIMWIN is configured in a standard way. You can change this using the Installation function under the Contr menu.

The standard set up includes:

Images

In accordance with the content of the image definition file IMAGES.TIM.This definition is usually fine for most environments. You can change this by editing IMAGE.TIM (see in the Contr menu Images).

Be careful of changing too much in the Images environment, since this can make your system incompatible with the TIMWIN Demo programs and other useful enhancements. Of course you can maintain different image definition files for different situations.

Frame grabber

During the installation procedure you specified the framegrabber that's installed in your system, or None if you don't have a frame grabber. If the Memory base and IO base settings differ from TIMWIN's default assumptions, you must change these. See the Installation function in the Contr menu.

Managing windows

Many of TIMWIN's functions interact with their own (sub) windows. Dealing with so many windows (see the list below) may prove to be a little complicated, especially if you don't have a high resolution screen. See for some hints & tips to keep your screen conveniently arranged.

The following list gives all of TIMWIN's windows

- Main window — contains the main menu bar, the message area and the command line
- Status window — contains information and controls for images
- Ibuf edit window — shows Ibuf's contents
- Image edit window — shows images in numerical format
- Graphic window — shows Ibuf data graphically
- Statistic window — shows image statistics
- Debug window — shows a running command file's source
- TTY window — shows text output of a command file
- Filter window — shows a convolution kernel

Main window

TIMWIN's main window is resposible for the user interaction. Itconsists of the following parts (from top to bottom)

- **the menu bar**
  the top part of the window. The menu bar allows yo to control most of TIMWIN's fucntions using self explaining menu's and dialog boxes.
- **the system area**
  the middle section of the window. Here system messages appear, like error messages, command responses, etc.
- **the command edit line**
  this is the command line, that allows you to enter commands using the keyboard

The TTY window

The TTY window is the interface between command file programs and the user.
It is character based (22 lines of 80 characters), and it is compatible with the TIM for DOS output screen.

See also: using the TTYWindow

The Status window

The status bar contains several controls that allows you to select images, look up tables, observe and change cursor positions, sub image size, etc.

You can use also the status bar to switch the mouse from Windows to the image cursor.
Press the Cursor button to switch to the windows cursor. Press the mouse's right button to switch back to Windows.

See also: using the Status Window

Note: controlling an image's cursor can only be done with frame grabber images.

The Statistics window

The statistics window shows several statistic values, derived from an image's histogram. You can choose to update the statistic window's content automatically with each image processing operation by setting the Update flag in the Update menu.

See also: using the StatisticsWindow

The Graphics window

The graphic window shows the following data graphically:

- the content of Ibuf
- the histogram of an image

You can select either of these, or both.
You can choose to update the graphic window automatically with each image processing operation that changes the involved data by setting the Update flag in the Update menu.

See also: using the Graphics Window

The Image edit window

The image edit window shows a selected part of an image numerically. You can edit the values and and change the selected area.

You can choose to update the image edit window automatically with each image processing operation that changes the involved data by setting the Update flag in the Update menu.

See also: using the Image Edit Window

The Ibuf edit window

The image edit window shows a selected part of the Ibuf buffer. You can edit the values and change the selected area.

You can choose to update the Ibuf edit window automatically with each image processing operation that changes the involved data by setting the Update flag in the Update menu.

See also: using the Ibuf Edit Window

The Filter window

The filter window allows you to enter values for convolution kernels. You can indicate horizontal and/or vertical symmetry to reduce the number of items to add.

See also: using the Filter Window

The Debug Window

The debug window is used when debugging command files. It contains functions to single step the program, watch variables, set breakpoints, etc.

The Watch window

When in debug mode, this window shows the value of selected variables.

Hints for managing windows

- Keep the Main window small. Usually it is sufficient to have only a few lines in the system area. If you want to see more, you can always scroll old lines back.
- Use the ALT-S and ALT-H accelerator keys to toggle the status and history windows, respectively.
- Reduce windows to icons by clicking the **[v]** button in the window caption bar, if you don't need them for some time. Realize, however, that they may not automatically come back if they become active again.
- Remove windows, that you don't need anymore, by double clicking the **[--]** button.

Image use

In TIMWIN you start image processing by selecting an image (commands: dis, dest; status window).

From then on this image will be the destination for image processing operations, as well as he default source for image processing operations. Thus, in commands you specify the source image, but never the destination image.

In the example below we'll use the image processing operation inv, that inverts the pixels of an image. Assume image **h** is selected. Then:

`inv`	The pixels of image **h** are inverted, the result is written back into image **h**
`inv a`	The pixels of image **a** are inverted, the result is written into image **h**
`inv a >b`	As above, but the result in **h** is copied to **b** afterward. This construction is called: post transport

There are a few exceptions to this rule. The following function groups:

- graphic functions,
- bitplane functions,
- CLP functions,
- test image commands (wig, dump)

operate on the specified image. Thus:

`wig`	A wedge is drawn in the default image (**h**)
`wig a`	A wedge is drawn in image **a**

Entering commands

TIMWIN needs a command to start doing something. Commands must be entered from the main window. You can enter commands in either of the following ways:

- Using the command line
- Using menus

Using the command line

The command line is at the bottom of the main TIM Window, after the `TIM>` prompt. This is a fast method of entering commands, if you know the command's name and syntax. If not, you'd better use the menus

To enable help, press the `TIM>` prompt button. This opens a Command list box, which shows all available commands. Once you enter a command or part of a command, pressing the F1 key or clicking the Help button brings up the Help window for that command.

Using menus

The menu system helps you to assemble an image processing command by directing you to the correct function, and giving suggestions for the options. To use the image processing menus you can use one of two main menus:

ImageProc	shows the functions sorted into function groups
Applic	shows the functions sorted into project groups (only a subset is available)

After you executed the specified function, the corresponding command shows up in the edit window, and in the history window.

Displaying images

To be able to see the result of an image processing operation, you need a visible image. The following determines the visibility of an image:

the location

the image must be located in a frame grabber, or it must be a window image. To find out the properties of the available images, see the concerning paragraphs in the description of the Status bar.

the display look up table (FG images)

a proper look up table (LUT) must be loaded, in accordance with the content of the image. In an initialized system, a few standard LUTs have been defined, and can be selected using the **Out** buttons in the LUT field of the Status bar.
To initialize the frame grabber LUTs, run command file `*init`.

Initialized LUT content:

1 standard black and white
2 pseudo colours in the most significant bitplanes
3 binary image in red in the least significant bitplane, the rest black and white
4 as 3., but also green in the 2nd bitplane and blue in the 3th.

the display look up table (Window images)

To use the standard LUTs (as described above) in a Windows image, you can:

- use the **Win** buttons in the Status bar
- use the lut command

See also: ~~Windows images~~, lut command

Windows images

The ability to display Windows images correctly depends upon the capabilities of your graphics adapter. The following situations exist:

Standard VGA
Standard VGA has 640x480 pixels, with 16 colours. Such a configuration is not recommended for displaying images. However, for running TIMWIN on a frame grabber system it is adequate.

Super VGA
Super VGA exists in several flavours. Resolutions are 800x600 or 1024x768, and the number of colours can be 16 or 256. The ability to show 256 colours (or grey values) is a must for displaying images.

Super VGA with an active processor.
These VGA adapters have the same properties as standard Super VGA adapters, but the are much faster due to the presence of graphic accelerators. These adapters are recommended for use in the highest resolution modes on all but the fastest computers.

Note. To simulate grey values on a system with only 16 colours, use the dot command

Command file programs

The ability of running command file programs is very important in TIMWIN. To run a command file, you must do one of the following:

On the command line:

- enter the command file's name, preceded by a `*` or a `/` .

In the menu system:

1. On the menu bar, press CommFile
2. In the File list box select a file
3. Press the Run button

To create a command file yourself, see Command Files. Notice, that in the Demo version of TIMWIN the compiler (necessary to compile command file sources) is not available.

Using Ibuf

Ibuf is TIM's internal cut & paste buffer. Several operations put data into Ibuf, while others read data from it. By executing commands in a clever order, you can move data in your processing sequence, and thus take advantage of these properties.

Ibuf properties
Showing the content of Ibuf
Manually changing Ibuf
Displaying Ibuf graphically
Commands that use Ibuf
Ibuf in Command files

Ibuf properties

Ibuf can store 1024 bytes, or 512 words, or 256 long words. This range is appropriate for the purpose, that Ibuf serves. The actual data type depends on the operation, that generated the data.

The current data properties can be seen in the top of the window. They are:

Data type:

- Byte
- Word
- Long word

Number of data items:

- Ibuf is not always entirely filled. The **Items** field shows the number of valid data items. If it is 0, there is no data in Ibuf.

Showing the content of Ibuf

To view the data in Ibuf numerically, open the Ibuf window:

- In the Edit menu, click Ibuf. See also: the Edit Ibuf Menu
- Or, on the command line, enter the command `editi`

The Ibuf window shows only those values in Ibuf, that were written during a previous operation.Therefore, the window may be empty.

Displaying Ibuf graphically

The Graph window gives a graphic representation of Ibuf's content.

To open the graphic window:

- In the View menu, click the Graph Window

See also: the View Graph menu

Commands that use Ibuf

Commands that interact with IBUF are, for example:

- All table operations (they create their table in IBUF)
- The rdln command, which reads an image line and stores it into IBUF. Also see the other `rd..` commands
- The ~~ribuf~~ and wibuf file read/write commands, that load/store Ibuf data from/to files
- The graphic functions, that read plot data from IBUF.

Ibuf in command files

In command file programs you can use Ibuf as a standard array, using the common array notation. Of course you can use the command ibuf as well, but the array notation is much more efficient.

Manually changing Ibuf

You can modify Ibuf values interactively in the following ways:

- by command (ibuf). This option gives you control over a single value at a time.
- by using the Ibuf Edit window. This option shows an adjustable part of Ibuf and allows you to modify individual values.