

ARTISAN®
TECHNOLOGY GROUP

Your **definitive** source for quality pre-owned equipment.

Artisan Technology Group
(217) 352-9330 | sales@artisantg.com | artisantg.com

Full-service, independent repair center
with experienced engineers and technicians on staff.

We buy your excess, underutilized, and idle equipment
along with credit for buybacks and trade-ins.

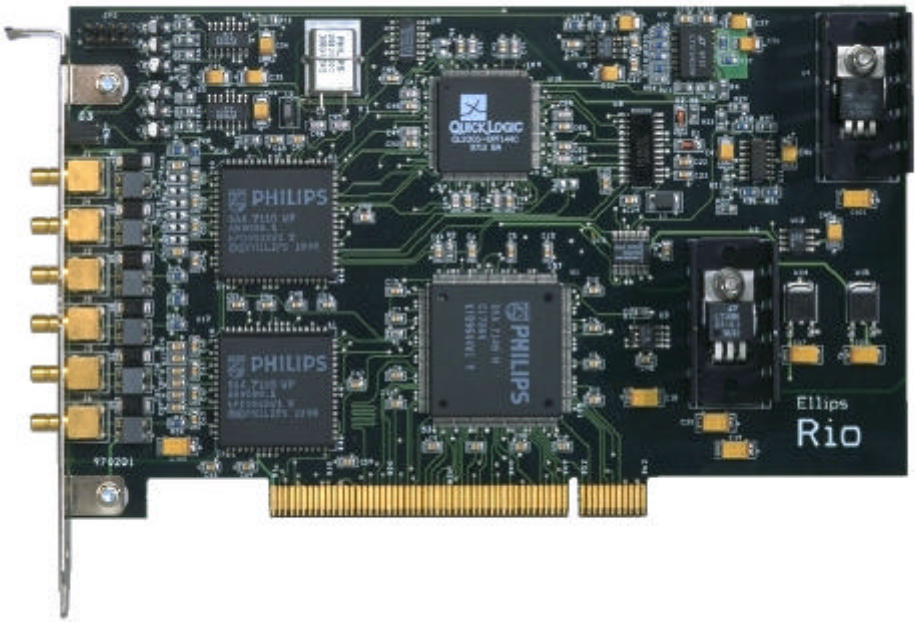
Custom engineering
so your equipment works exactly as you specify.

- Critical and expedited services
- In stock / Ready-to-ship
- Leasing / Rentals / Demos
- ITAR-certified secure asset solutions

Expert team | Trust guarantee | 100% satisfaction

All trademarks, brand names, and brands appearing herein are the property of their respective owners.

Find the *Ellips Full Rio* at our website: ***Click HERE***



Ellips Rio

MANUAL

High Performance Frame Grabber

Version 1.16
May 30, 2000

ELLIPS B.V.

Company address:
Urkhovenseweg 11
5641 KA Eindhoven
The Netherlands

Mail address:
P.O. Box 240
5600 AE Eindhoven
The Netherlands

Phone: +31-40-2456540
Fax: +31-40-2467183

Email: general: rio-info@ellips.nl
support: rio-support@ellips.nl
http://www.ellips.nl

Disclaimer

Ellips B.V. makes no representation or warranty with respect to the adequacy of this documentation or the hardware and software which it describes for any particular purpose. In no event shall Ellips B.V. be liable for any damages, losses, costs, charges, demands, or claims for lost profits, fees or expenses of any nature or kind.

TABLE OF CONTENTS

OVERVIEW 4

FEATURES 5

3RD GENERATION PCI BUS MASTER WITH FLEXIBLE TRANSFER CONTROLLER 5
6 (CVBS OR MONOCHROME) CAMERAS OR 3 Y/C CAMERAS CAN BE CONNECTED 5
BLACK & WHITE (GRAYSCALE) DIGITIZING 5
COLOR DIGITIZING, INPUTS Y/C OR CVBS 6
SIMULTANEOUS GRABBING FROM TWO BLACK AND WHITE CAMERAS 6
SUPPORTS PROGRESSIVE SCAN CAMERAS 6
INDEPENDENT PROCESSING OF TWO VIDEO SOURCES 6
FAST SWITCHING BETWEEN COLOR CAMERAS 6
SIMULTANEOUS GRABBING FROM FOUR BLACK AND WHITE CAMERAS 6
HIGH QUALITY BLACK AND WHITE INPUT WITH PROGRAMMABLE GAIN AND OFFSET 7
SUPPORTS RGB AND RGBA CAMERAS 7

GETTING STARTED 8

HARDWARE REQUIREMENTS 8
CONNECTING VIDEO SOURCES 8
INSTALLING YOUR RIO PCI FRAME GRABBER UNDER WINDOWS 95/ 2000 9
INSTALLING YOUR RIO PCI FRAME GRABBER UNDER WINDOWS NT 4.0 OR OLDER 9
INSTALLING YOUR RIO PCI FRAME GRABBER UNDER DOS 10

SOFTWARE REFERENCE 11

RIO MODULES 11
CAMERA OUTPUT, OFFSETS, SCALING AND PITCH 13
Camera Output 13
Offsets 13
How to scale 14
Choice of scaler 15
RIO_SCALER 15
Pitch 15
OUTPUT FORMATS 17
RIO_COLOR_OUTPUT_FORMAT 17
RIO_BW_OUTPUT_FORMAT 18
RIO_RGBA_OUTPUT_FORMAT 19
Special Output formats /Postprocessing 19
OTHER ENUMERATED TYPES 21
RIO_MODULE_MODE 21
RIO_ON_OFF_MODE 21
RIO_FIELD_OR_FRAME 21
RIO_INPUT_MODULE 22
RIO_CVBS_OR_YC 22
RIO_TV_OR_VTR 22
RIO_COLOR_VIDEO_STANDARD 22
RIO_BW_VIDEO_STANDARD 23
RIO_GPIO_MODE 23

RIO_GPIO_PIN *23*
STRUCTURES 24
PRIODL *24*
RIO_VIDEO_HDR *24*
RIO_FLASH_PARAMS *24*
RIO_TBUFFER_PARAMS *25*
RIO_MODULE_COLOR *27*
RIO_MODULE_BW *27*
RIO_MODULE_HQ *28*
RIO_MODULE_STEREO_LOCKED *28*
RIO_MODULE_S_FULL_FRAME *28*
RIO_MODULE_FULL_FRAME *29*
RIO_MODULE_RGBA *29*
RIO_MODULE *29*
ERROR CODES 30
FUNCTIONS FOR WINDOWS 33
RioCapture *34*
RioCaptureCancel *39*
RioCaptureStop *40*
RioCaptureTimeOut *41*
RioClose *42*
RioCreateRiodl *43*
RioDebiRead *44*
RioDebiWrite *45*
RioEepromRead *46*
RioEepromWrite *47*
RioExternalInt *48*
RioExternalIntCancel *51*
RioExternalIntTimeOut *52*
RioGetBoardType *53*
RioGetBrightness *54*
RioGetCamera *55*
RioGetChromaKey *56*
RioGetContrast *57*
RioGetHqBw *58*
RioGetHue *59*
RioGetInputGain *60*
RioGetInputModule *62*
RioGetOverlappedResult *64*
RioGetSaturation *65*
RioGetVideoStatus *66*
RioGpioControl *67*
RioGpioRead *69*
RioGpioWrite *70*
RioI2cRead *71*
RioI2cWrite *72*
RioIndexSetLed *73*
RioInitialize *74*
RioOpen *75*

RioPostProcess *76*
RioRegisterRead *77*
RioRegisterWrite *78*
RioScatterLock *79*
RioScatterUnlock *81*
RioSelectCamera *82*
RioSetBoardId *83*
RioSetBrightness *85*
RioSetChromaKey *87*
RioSetContrast *88*
RioSetHqBw *89*
RioSetHue *90*
RioSetInputGain *91*
RioSetInputModule *92*
RioSetLed *94*
RioSetSaturation *95*
RioSizeOfRiodl *96*
RioStreamAddEmptyBuffer *97*
RioStreamClose *98*
RioStreamGetFilledBuffer *99*
RioStreamGetStartTime *103*
RioStreamInit *104*
RioStreamStart *106*
RioStreamStop *107*
RioTriggerCapture *108*
RioTriggerCaptureCancel *114*
RioTriggerCaptureTimeOut *115*
USING THE SOFTWARE LIBRARY WITH OTHER COMPILERS 116
Watcom C/C++ compiler *116*
Borland C compiler *116*
FUNCTIONS FOR DOS 117
DOWNLOADING UPDATES 118

HARDWARE REFERENCE 119

ARCHITECTURAL OVERVIEW 119
VIDEO SOURCE CONNECTORS 120
GENERAL PURPUSE IO PINS 121

SPECIFICATIONS 122

HOST COMPUTER INTERFACE 122
ACQUISITION 122
SOFTWARE SUPPORT 123

KNOWN ISSUES 124

INDEX 126

OVERVIEW

Designed to get the most out of any S-Video/composite color or standard black and white camera, the Ellips Rio is a high performance PCI frame grabber, targeted at real-time industrial image processing applications. The Rio comes in two versions:

- Basic version with one input module
- Full version with two input modules

The Basic Rio features:

- 3rd generation PCI bus master with flexible transfer controller
- 6 (CVBS or monochrome) cameras or 3 Y/C cameras can be connected
- Black & white (grayscale) digitizing
- Color digitizing, inputs Y/C or CVBS
- Simultaneous grabbing from two black and white cameras
- Supports progressive scan cameras

The extra Full Rio features are:

- Independent processing of two video sources
- Fast switching between color cameras
- Simultaneous grabbing from four black and white cameras
- High quality black and white input with programmable gain and offset
- Supports RGB and RGBA cameras

All the features mentioned above will be explained in more detail in the next part ‘FEATURES’.

FEATURES

3rd generation PCI bus master with flexible transfer controller

Several PCI grabbers have been available on the market for some time, but, depending on their architecture, most of them are not suited for real-time imaging processing applications:

- Some grabbers based on a PCI SVGA controller have an image port that allows write access to the video memory from an external access bus (typical bandwidth 45 MB/s). Although this method offers live video on the desktop without hogging the PCI bus, reading the image back from video memory to main memory suffers from slow memory access due to the nature of these SVGA controllers: typical write bandwidth from PCI to SVGA is up to 90 MB/s, whereas read bandwidth from SVGA to PCI is limited to approximately 7 MB/s. Therefore SVGA-controllers based PCI grabbers are generally not suited for real-time imaging applications.

- Some PCI grabbers employ efficient PCI burst transfers, but lack flexible transfer controller capabilities. Therefore, more data is sent over the PCI bus than is actually needed, thus slowing the system down. For example, when tracking moving objects, the region of interest changes for every video frame. This calls for changing the region of interest at field rate.

The *Ellips Rio* offers PCI bus master DMA, allowing burst transfers of up to 132 MB/s (depending on the quality of the main board) without tying up the main CPU, combined with a very flexible transfer controller. Using a Register Program Sequencer, input window, output destination, scaling, and clipping can be reprogrammed on a line-by-line basis without CPU intervention. The sequencer program can be changed every field to incorporate complex changes of the image parts transferred over the PCI bus to main memory, thus tracking moving objects**.**
The PCI interface supports virtual memory addressing for operating systems running virtual demand paging (like Windows). The integrated Memory Management Unit (MMU) translates linear to physical addresses using a page table in system memory provided by the software driver. The MMU supports op to 4 Mbytes of virtual address space per DMA channel.

6 (CVBS or monochrome) cameras or 3 Y/C cameras can be connected

Up to 6 CVBS or monochrome cameras or 3 Y/C cameras can be connected to a Rio board. Also CVBS or monochrome connections in combination with Y/C connections are allowed. Of course every Y/C camera needs to be connected to Y and C connectors on the Rio that belong together.

Black & white (grayscale) digitizing

Black and white camera standards CCIR (50 fields per second) and EIA (60 fields per second) are supported. Different output formats are possible (grayscale, binary).

Color digitizing, inputs Y/C or CVBS

The NTSC (60 fields per second), PAL and SECAM (50 fields per second) color standards are supported. A wide variety of output formats is supported.

Simultaneous grabbing from two black and white cameras

Although the basic version of the Rio only has one input module, it still is capable of simultaneous grabbing at 50 (CCIR) or 60 (EIA) fields per second with two genlocked monochrome cameras (stereo locked mode). However to obtain conventional output formats, postprocessing is needed, which will have an impact on the capture rate.

Supports progressive scan cameras

The Rio can be used to digitize from a black and white progressive scan camera both non-interlaced at 25 (CCIR) or 30 (EIA) frames per second as well as interlaced at 50 (CCIR) or 60 (EIA) frames per second, e.g. JAI CV-M10). Full frame camera input is supported for the Full Rio as well as for the Basic Rio. For the Basic Rio however postprocessing is needed to obtain conventional output formats. Additionally the Full Rio can also adjust brightness and contrast in full frame mode, or the Full Rio can be used to digitize from two progressive scan cameras simultaneously.

Independent processing of two video sources

Because the full version of the Rio also has two DMA channels, two color cameras each can be handled totally separately, so it is possible for example for one input module to be connected to a color camera and one input module to a monochrome camera. Also simultaneous grabbing from two color cameras is possible with the Full Rio.

Fast switching between color cameras

For many image-processing applications, evaluating outputs from multiple cameras is desirable. When allowable with respect to speed constraints, hardware costs can be reduced by fast switching between multiple cameras on a field by field basis, using a single frame grabber. With traditional grabbers, fast switching between color cameras demands color carrier sub-locking in addition to a conventional genlock. This often involves modifying camera electronics, thus increasing costs.

The full version of the Ellips Rio frame grabber is capable of fast switching between two color cameras without color carrier sub-locking, because each of the two cameras can be connected to its own input module. In this mode, a field rate of 25 (PAL/SECAM) or 30 (NTSC) fields per second per camera can be achieved. When no RGB output is required, this mode can even be used at 50 (PAL/SECAM) or 60 (NTSC) fields per second per camera.

Simultaneous grabbing from four black and white cameras

With the Full Rio, because it has two input modules, both modules can be used in stereo locked mode, making simultaneous grabbing at 50 (CCIR) or 60 (EIA) fields per second with four

genlocked monochrome cameras possible. However to obtain conventional output formats, postprocessing is needed, which will have an impact on the capture rate.

High quality black and white input with programmable gain and offset

For high quality black and white measurements, the full version of the Rio has a separate converter stage with programmable gain and offset. Its input can be switched at field rate between genlocked cameras. In contrast to the other modules (which only have an output range of 192 values, which will result in missing values when mapped to a 256 value scale), the high quality black and white module has a full output range of 256 values.
Therefore for serious black and white measurements, the full version of the Rio is recommended.

Supports RGB and RGBA cameras

With the Full Rio, one RGB(A) camera can be connected. Postprocessing is needed to obtain one of the output formats that are supported (including planar output).

GETTING STARTED

Hardware Requirements

The Rio High Performance Frame Grabber is a PCI card that needs to be inserted into a PCI slot on the motherboard. For optimal performance, the Rio should be used on a Pentium motherboard with Triton chipset.

Connecting Video Sources

As shown in Figure 1, the Rio has six SMB connectors. Video sources must be connected to these SMB connectors. The grabber accepts almost all kinds of standard video signals, including those from video recorders. Supported input signals: CVBS (CCIR and EIA), Y/C (S-video, S-VHS or Hi8 for standards PAL-B/G, NTSC-M and SECAM) and RGBA (CCIR and EIA).

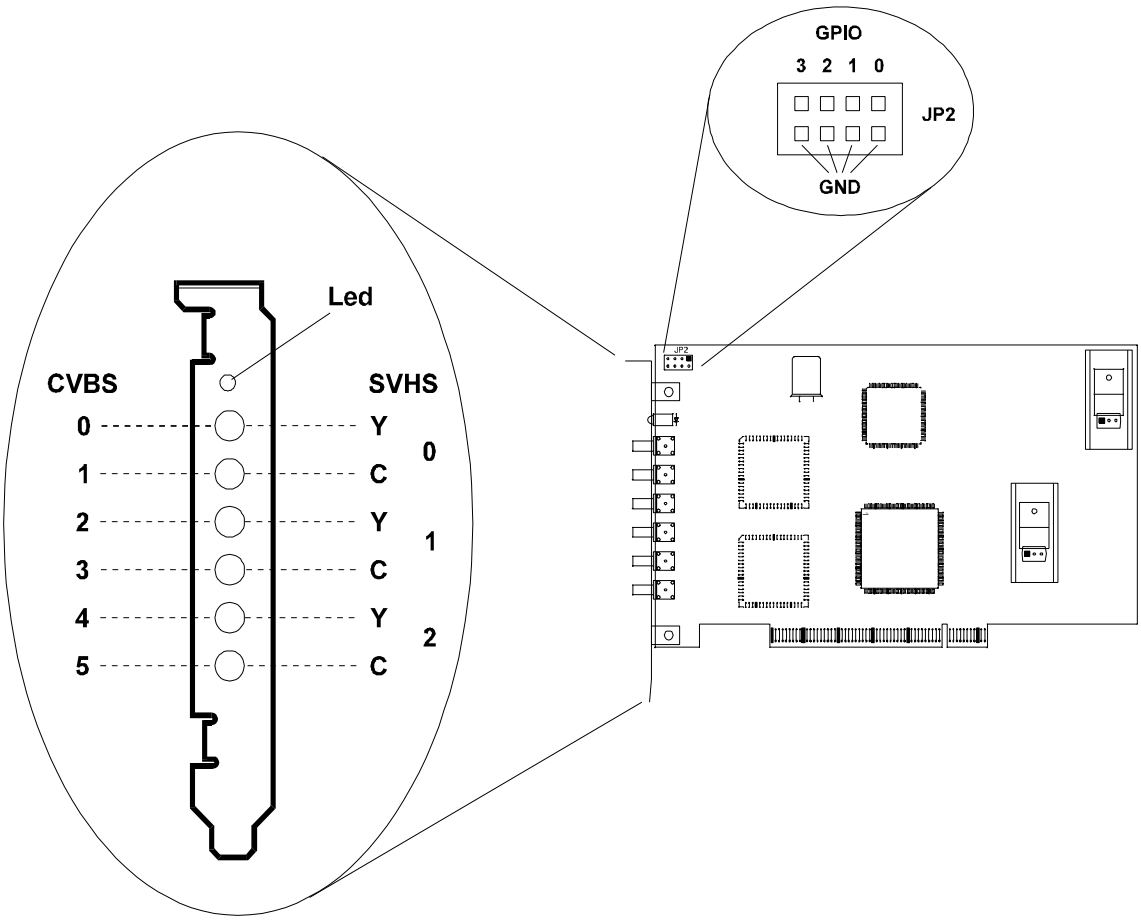


Figure 1. Video Source Connector configuration

Video sources that can be connected:

- *Composite video sources* (CVBS). Up to six cameras can be connected.
- *Y/C connectors, S-Video sources* (S-VHS or Hi8). Up to three Y/C cameras can be connected, occupying two connectors per camera, an Y connector and the corresponding C connector.
- *Progressive Scan.* Up to three progressive scan cameras can be connected, each camera should have video output 1 connected to an Y connector and video output 2 to the corresponding C connector.
- One *RGBA* camera can be connected, with connectors R, G, B and A to respectively CVBS inputs 0 through 3.
- Combinations of the above mentioned video sources also can be connected, for example 2 CVBS cameras and 2 Y/C cameras can be connected at the same time. Of course, every Y/C camera needs to be connected to Y and C connectors on the Rio that belong together.

Installing your Rio PCI Frame Grabber under Windows 95/ 2000

- Shut down your computer.
- Place the Rio PCI frame-grabber(s) into empty PCI slot(s).
- Turn on your computer.
- In case of Windows 2000, make sure you are logged on with administrative privileges.
- Windows 95/ 2000 automatically detects the change in hardware and the ‘New Hardware Found’ dialogbox appears.
- Click ‘Driver from disk provided by hardware manufacturer’.
- Click OK. The ‘Install From disk’ dialogbox appears.
- Insert the first Rio installation disk into the floppy disk drive or the Rio CD-ROM into the CD-ROM drive.
- Specify the correct path ('A:\Win95' for Windows 95 installation from floppy disk or CD-ROM drive letter +'\Win95' for Windows 95 CD-ROM install, 'A:\WinNT' for Windows 2000 installation from floppy disk or CD-ROM drive letter +'\WinNT' for Windows 2000 CD-ROM install) and click OK.
- Follow the instructions on the screen.
- After the installation has finished, run Setup.exe from the first Rio installation floppy disk or Rio CD-ROM.
- Follow the instructions on the screen.

Installing your Rio PCI Frame Grabber under Windows NT 4.0 or older

- Shut down your computer.
- Place the Rio PCI frame-grabber(s) into empty PCI slot(s).
- Turn on your computer.
- Make sure you are logged on to Windows NT with administrative privileges.
- Insert the first Rio installation disk into the floppy disk drive or the Rio CD-ROM into the CD-ROM drive.
- Run Setup.exe from the first Rio installation floppy disk or the Rio CD-ROM.
- Follow the instructions on the screen.

NOTE:
Installation under Windows will also include the DOS software library.

NOTE:
Iinstallation under Windows will also allow you to install an Image-Pro Plus driver when Image-Pro Plus is installed on your system. In case Image-Pro Plus is being installed AFTER the Ellips Rio installation, the user must install this driver (captRIO.dll) manually. It can be found on the first Rio installation disk or Rio CD-ROM in the sub-folder \IPWin3. The file needs to be copied to the main Image-Pro Plus directory.

Installing your Rio PCI Frame Grabber under DOS

- Shut down your computer.
- Place the Rio PCI frame-grabber(s) into empty PCI slot(s).
- Turn on your computer and boot DOS.
- Insert the Rio installation floppy disk labeled 'DOS installation' into the floppy disk drive or the Rio CD-ROM into the CD-ROM drive.
- Run Install.bat from the disk or the CD-ROM.
- Follow the instructions on the screen.

NOTE:
The DOS installation is packaged using Info-ZIP's compression utility. The installation program uses UnZip to read zip files from floppy. Info-ZIP's software (Zip, UnZip and related utilities) is free and can be obtained as source code or executables from various bulletin board services and anonymous-ftp sites, including CompuServe's IBMPRO forum and ftp.uu.net:/pub/archiving/zip/*.

SOFTWARE REFERENCE

Rio Modules

The Rio board can be used in different modes to capture an image. These different modes are indicated by their RIO_MODULE_MODE. Seven different Rio input module modes exist, of which three can only be used when a full version of the Rio is used. On the Basic Rio, only one input module is available, while with the Full Rio for some modes two input modules are available. In Figure 2 and Figure 3, the differences between the possible modes are explained. These differences are also reflected in the RIO_MODULE struct, which is used in the software library to pass module mode information.

Module Mode	RIO_COLOR	RIO_BW	RIO _STEREO _LOCKED	RIO_S _FULL _FRAME	RIO_HQ	RIO_FULL _FRAME	RIO_RGBA
Mode type	*Color*	*b&w*	*b&w*	*b&w*	*b&w*	*b&w*	*color*
Useable on Basic Rio or Only on Full Rio	Basic	Basic	Basic	Basic	Full	Full	Full
Number of input modules usable on Full Rio	2	2	2	2	1 (module 0) module 1 available for other use	1	1
Connectors	CVBS: one connector **or** Y/C: two connectors (one Y/C pair)	one Connector	two cameras: one camera on Y, one camera on C	full frame camera, two connectors (one Y/C pair)	one Y connector	full frame camera, two connectors (one Y/C pair)	RGB(A) camera four (or three) connectors, R,G,B (and A) connected to connectors 0,1,2 (and 3)
Scaler choice	yes	yes	yes	yes	yes	no	no
TV / VTR choice	yes	yes	yes	yes	yes	yes	yes
Field/ Frame choice	yes	yes	yes	no (Full Frame)	yes	no (Full Frame)	yes
Video Standards	RIO_PAL RIO_SECAM RIO_NTSC	RIO_CCIR RIO_EIA	RIO_CCIR RIO_EIA	RIO_CCIR RIO_EIA	RIO_CCIR RIO_EIA	RIO_CCIR RIO_EIA	RIO_CCIR RIO_EIA

Figure 2. The possible Rio modules and their settings

Module Mode	RIO_COLOR	RIO_BW	RIO_STEREO _LOCKED	RIO_S _FULL _FRAME	RIO_HQ	RIO _FULL _FRAME	RIO_RGBA
Output format	RIO_YUV16 RIO_RGB8 RIO_ARGB15 RIO_RGAB15 RIO_RGB16 RIO_RGB24 RIO_ARGB32	RIO_Y1 RIO_Y2 RIO_Y8	(see Special Output formats)	(see Special Output formats)	RIO_Y1 RIO_Y2 RIO_Y8	RIO_Y1 RIO_Y2 RIO_Y8	special format for RGB24 **or** ARGB32 postprocessing (see Special Output formats)
Output format with Compen-sation of Gamma Pre-Correc-tion	RIO_RGB8_GC RIO_ARGB15_GC RIO_RGAB15_GC RIO_RGB16_GC RIO_RGB24_GC RIO_ARGB32_GC						
Post-Proces-sing possible	no	no	yes	yes	no	no	yes
Output format after Post-Proces-sing			2 RIO_Y8 planes, first Y, then C, consecutive in memory	RIO_Y8			RIO_RGBA _RGB24 RIO_RGBA _ARGB32 RIO_RGBA _RGB24P RIO_RGBA _ARGB32P

Figure 3. The output format of the possible Rio modules

Camera Output, Offsets, Scaling and Pitch

Camera Output

The original camera output sizes depend on the camera type and on grabbing being done in field or frame mode. Possible sizes are shown in Figure 4. When grabbing is done in field mode, the programmer can indicate that square pixels should be maintained or not (see BOOL *SquarePixels* in RioCapture, RioStreamInit and RioTriggerCapture). In that case the horizontal camera output size will be downscaled by a factor 2, to compensate for the factor 2 downscale in vertical direction that is implicit to field grabbing.

Grabbing mode	PAL / SECAM / CCIR (50 Hz)		NTSC / EIA (60 Hz)	
	allowed coordinates	maximum rect size (camera output size)	allowed coordinates	maximum rect size (camera output size)
Frame grabbing	x: 0..768 y: 0..625	768 x 576 *	x: 0..640 y: 0..525	640 x 480 *
Field grabbing with *SquarePixels*==FALSE	x: 0..768 y: 0..312	768 x 288 *	x: 0..640 y: 0..262	640 x 240 *
Field grabbing with *SquarePixels*==TRUE	x: 0..384 y: 0..312	384 x 288 **	x: 0..320 y: 0..262	320 x 240 **
Grabbing mode	CCIR progressive scan (25 Hz)		EIA progressive scan (30 Hz)	
Non-interlaced progressive scan Field grabbing with *SquarePixels*==FALSE	x: 0..768 y: 0..625	768 x 576 *	x: 0..640 y: 0..525	640 x 480 *
Non-interlaced progressive scan Field grabbing with *SquarePixels*==TRUE	x: 0..384 y: 0..625	384 x 576 **	x: 0..320 y: 0..525	320 x 480 **
* In case of the binary ratio scaler (see paragraph 'Choice of scaler'), the width should be = 512 ** In case of the binary ratio scaler (see paragraph 'Choice of scaler'), the width should be = 256				

Figure 4. Coordinate Restrictions, Camera Output Size

Offsets

When capture parameters are set, a source rectangle (see RECT **SrcRect* in RioCapture, RioStreamInit and RioTriggerCapture) has to be given, which indicates the region of interest out of the available camera output. The restrictions, to which the source rectangle coordinates should comply, are indicated in Figure 4. Note that the source rectangle will be used up to, but not including the bottom and right coordinates.

The available horizontal coordinate range (pixel count) that can be used equals the maximum horizontal rectangle size that can be used. Figure 5 explains the available horizontal range relative to the first real video pixel (pixel offset), for different standards and grabbing modes. As can be seen from this table, the first real video pixel cannot be obtained for most situations. Grabbing usually starts a few pixels later.

The available vertical coordinate range (line count) that can be used is bigger than the maximum vertical rectangle size that can be used. To start the source rectangle with the first real video line, a

line offset should be used. Figure 6 explains the available vertical range relative to the first real video line, for different standards and grabbing modes. **For the current version of the Rio software library, the vertical offset should be higher than 4, preferable higher than 8.**

Grabbing mode	PAL / SECAM / CCIR (50 Hz)		NTSC / EIA (60 Hz)	
	x-coordinate	CCIR pixel count	x-coordinate	CCIR pixel count
Frame grabbing or Field grabbing with *SquarePixels*==FALSE	0 -3 *	158 155 (first video pixel)	0 -9..4 * (possible range for first video pixel)	122 113..126 (possible range for first video pixel)
Field grabbing with *SquarePixels*==TRUE	0 -1.5 *	158 155 (first video pixel)	0 -4.5..2 * (possible range for first video pixel)	122 113..126 (possible range for first video pixel)

* This is the x-coordinate that would result in the exact first video pixel (according to CCIR standard). However negative numbers or fractions can not be used, so in most cases the first video pixel cannot be obtained. In the case of RIO_S_FULL_FRAME, RIO_HQ or RIO_RGBA, the first video pixel will be situated a few pixels more to the left than indicated above, therefore in these modes the first pixel that can be obtained (x-coordinate 0) will be a few pixels later than CCIR 158 (PAL/SECAM/CCIR) or CCIR 122 (NTSC/EIA).

Figure 5. Pixel Offsets

Grabbing mode	PAL / SECAM / CCIR (50 Hz)		NTSC / EIA (60 Hz)	
	y-coordinate	CCIR line count (odd / even)	y-coordinate	CCIR line count (odd / even)
Frame grabbing	0, 1 30, 31	8, 321 23, 336 (first video lines)	0, 1 16,17..20,21 (possible range for first video lines)	11, 274 19, 282..21, 284 (possible range for first video lines)
Field grabbing	0 15	8 / 321 23/ 336 (first video line)	0 8..10 (possible range for first video line)	11 / 274 19..21 / 282..284 (possible range for first video line)

Figure 6. Line Offsets

How to scale

When capture parameters are set, besides the source rectangle, also a destination rectangle has to be given (see RECT **DestRect* in, RioCapture, RioStreamInit and RioTriggerCapture. As mentioned above, the source rectangle indicates the region of interest out of the available camera output. The destination rectangle indicates the part of the memory buffer that is to be used to store the data from the source rectangle. It must be equal or smaller in size than the source rectangle. **Note that prior to release 1.4, the offsets in the destination rectangle were not used.**

Together the source rectangle and destination rectangle determine the downscaling wanted by the programmer. Horizontal and vertical scaling can be done therefore with different downscale factors.

For the current version of the Rio software library, the destination rectangle should exactly have the same size as the source rectangle. Downscaling is not yet implemented.

Choice of scaler

As can be seen in Figure 13, two scalers are available on the Rio board: the high performance scaler and the binary ratio scaler. With the high performance scaler, any downscale ratio can be achieved. With the binary ratio scaler, downscaling can only be done in powers of 2. If the source rectangle sizes are not exactly powers of 2 times the destination rectangle sizes, scaling will be rounded off to a power of 2. Further more, when using the binary ratio scaler, RGB output formats are not supported and the horizontal size of the source rectangle should always be a multiple of 8

For Rio boards containing the SAA7146A chip, when using the binary ratio scaler, further restrictions need to be followed regarding the horizontal size of the source rectangle, which needs to be = 512 in case of frame grabbing or field grabbing with SquarePixels FALSE and = 256 in case of field grabbing with SquarePixels TRUE.

The programmer can choose which scaler should be used. Keep in mind, each scaler can only be used by one input module at a time. For example, during simultaneous grabbing from two color cameras in RGB output format, the programmer has to use the high performance scaler for both input modules on a full Rio board, thus the field rate will be twice as low: 25 (PAL/ SECAM) or 30 (NTSC) fields per second.

As is shown in Figure 2, with module modes RIO_FULL_FRAME and RIO_RGBA no choice of scaler is available to the programmer. In these modes both scalers are in use. Scaling can be done, provided the programmer implements binary ratio scaler restrictions.

Further more, when a Basic Rio board is used, the binary ratio scaler should not be used (without consulting Ellips). Only one scaler will be needed, so the binary ratio scaler with its limited possibilities compared to the high performance scaler, is not a good choice anyway.

RIO_SCALER

The programmer passes the choice in downscaler for an input via an enum that is used as part of the different module structs:

```
typedef enum _RIO_SCALER
{
    RIO_HPS,            high performance scaler
    RIO_BRS,            binary ratio scaler
    RIO_HPS_BRS,        used internally
    RIO_NR_SCALERS
} RIO_SCALER;
```

Pitch

When capture parameters are set, besides the source and destination rectangle, also a pitch has to be given (see int *Pitch* in RioCapture, RioStreamInit and RioTriggerCapture). The Pitch indicates the distance (in bytes) that two succeeding image lines are to be placed apart from one another in memory, in order to be correctly aligned.

The pitch depends on the output format that is selected. (See Output Formats).

For the Rio modules that use special output formats (RIO_MODULE_STEREO_LOCKED, RIO_MODULE_S_FULL_FRAME and RIO_MODULE_RGBA, see Figure 3), the pitch should be calculated according to the output format after postprocessing, even if no postprocessing will be performed.

For example for RIO_MODULE_STEREO_LOCKED, one byte per pixel should be used:

int Pitch= (XRes +3) & ~3 ;

Where XRes is assumed to be the picture number of pixels in horizontal direction (and the pitch was rounded up to multiple of 4 for Dword alignment).

In the case of RIO_MODULE_RGBA, the pitch depends on the possible output format:

One byte per pixel for RIO_RGBA_RGB24P and RIO_RGBA_ARGB32P:

int Pitch= (XRes +3) & ~3 ;

Three bytes per pixel for RIO_RGBA_RGB24:

*int Pitch= ((XRes *3) +3) & ~3 ;*

Four bytes per pixel for RIO_RGBA_ARGB32:

*int Pitch= (XRes * 4);*

Output Formats

As is shown in Figure 3, different output formats are possible depending on the module mode. These output formats (and the enumerated types that are used in the software library to limit the choice in output format to appropriate formats for each module mode) will be explained in this chapter.

RIO_COLOR_OUTPUT_FORMAT

The RIO_COLOR_OUTPUT_FORMAT enum is used to describe the choice in possible color output formats in RIO_COLOR mode.

```
typedef enum _RIO_COLOR_OUTPUT_FORMAT
{
     RIO_YUV16,
     RIO_RGB8,
     RIO_ARGB15,
     RIO_RGAB15,
     RIO_RGB16,
     RIO_RGB24,
     RIO_ARGB32,
     RIO_RGB8_GC,
     RIO_ARGB15_GC,
     RIO_RGAB15_GC,
     RIO_RGB16_GC,
     RIO_RGB24_GC,
     RIO_ARGB32_GC
} RIO_COLOR_OUTPUT_FORMAT;
```

These color output formats are explained below in Figure 7. Color Output formats:
(continued on next page)

RIO_COLOR _OUTPUT _FORMAT	*Format* *high … low*	*Explanation*
RIO_YUV16	2 pixels in one Dword: (YUV 4:2:2 format) 8 8 8 8 Y_1 V Y_0 U	U 8 bits, Y_0 =8 bits pixel 0, V 8 bits, Y_1 =8 bits pixel 1
RIO_RGB8	4 pixels in one Dword: 3 3 2 3 3 2 3 3 2 3 3 2 R_3 G_3 B_3 R_2 G_2 B_2 R_1 G_1 B_1 R_0 G_0 B_0	per pixel: Red 3 bits, Green 3 bits, Blue 2 bits
RIO_ARGB15	2 pixels in one Dword: 1 5 5 5 1 5 5 5 α_1 R_1 G_1 B_1 α_0 R_0 G_0 B_0	per pixel: α-bit, Red 5 bits, Green 5 bits, Blue 5 bits
RIO_RGAB15	2 pixels in one Dword: 5 5 1 5 5 5 1 5 R_1 G_1 α_1 B_1 R_0 G_0 α_0 B_0	per pixel: Red 5 bits, Green 5 bits, α-bit, Blue 5 bits
RIO_RGB16	2 pixels in one Dword: 5 6 5 5 6 5 R_1 G_1 B_1 R_0 G_0 B_0	per pixel: Red 5 bits, Green 6 bits, Blue 5 bits
RIO_RGB24	4 pixels in 3 Dwords: 8 8 8 8 8 8 8 8 8 8 8 8 R_3 G_3 B_3 R_2 G_2 B_2 R_1 G_1 B_1 R_0 G_0 B_0	per pixel: Red 8 bits, Green 8 bits, Blue 8 bits
RIO_ARGB32	one pixel per Dword: 8 8 8 8 α R G B	per pixel: α 8 bits, Red 8 bits, Green 8 bits,

RIO_COLOR _OUTPUT _FORMAT	*Format* *high … low*	*Explanation*
		Blue 8 bits
RIO_RGB8_GC	as RIO_RGB8	gamma pre correction compensated RIO_RGB8 values
RIO_ARGB15_GC	as RIO_ARGB15	gamma pre correction compensated RIO_ARGB15 values
RIO_RGAB15_GC	as RIO_RGAB15	gamma pre correction compensated RIO_RGAB15 values
RIO_RGB16_GC	as RIO_RGB16	gamma pre correction compensated RIO_RGB16 values
RIO_RGB24_GC	as RIO_RGB24	gamma pre correction compensated RIO_RGB24 values
RIO_ARGB32_GC	as RIO_ARGB32	gamma pre correction compensated RIO_ARGB32 values

Figure 7. Color Output formats

RIO_BW_OUTPUT_FORMAT

The RIO_BW_OUTPUT_FORMAT enum is used in RIO_BW, RIO_HQ or RIO_FULL_FRAME module mode to describe the choice in possible black & white output formats.

```
typedef enum _RIO_BW_OUTPUT_FORMAT
{
    RIO_Y1,
    RIO_Y2,
    RIO_Y8
} RIO_BW_OUTPUT_FORMAT;
```

These black & white output formats are explained below in Figure 8. Black & White Output formats:

RIO_BW _OUTPUT _FORMAT	*format* *high … low*	*Explanation*
RIO_Y1	32 pixels per Dword 1 1 1 1 1 1 1 1 …… 1 1 1 1 1 1 1 1 Y_{24} Y_{25} Y_{26} Y_{27} Y_{28} Y_{29} Y_{30} Y_{31} Y_0 Y_1 Y_2 Y_3 Y_4 Y_5 Y_6 Y_7	1 bit per pixel
RIO_Y2	16 pixels per Dword 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 Y_{12} Y_{13} Y_{14} Y_{15} Y_8 Y_9 Y_{10} Y_{11} Y_4 Y_5 Y_6 Y_7 Y_0 Y_1 Y_2 Y_3	2 bits per pixel
RIO_Y8	4 pixels per Dword 8 8 8 8 Y_3 Y_2 Y_1 Y_0	8 bits per pixel (256 gray values)

Figure 8. Black & White Output formats

RIO_RGBA_OUTPUT_FORMAT

The RIO_RGBA_OUTPUT_FORMAT enum is used in RIO_RGBA module mode to describe the choice in possible RGBA output formats that can be generated, when post-processing is set to RIO_ON. Otherwise, the special output-format will be generated, as explained in the next part.

```
typedef enum _RIO_RGBA_OUTPUT_FORMAT
{
    RIO_RGBA_RGB24,
    RIO_RGBA_ARGB32,
    RIO_RGBA_RGB24P,
    RIO_RGBA_ARGB32P
} RIO_RGBA_OUTPUT_FORMAT;
```

The RGBA output formats are explained below in Figure 9. RGBA Output formats:

RIO_RGBA_OUTPUT _FORMAT	*format* *high … low*	*Explanation*
RIO_RGBA_RGB24	as RIO_RGB24	
RIO_RGBA_ARGB32	as RIO_ARGB32 one pixel per Dword: 8 8 8 8 A R G B	per pixel: A value 8 bits, Red 8 bits, Green 8 bits, Blue 8 bits
RIO_RGBA_RGB24P	3 planes RIO_Y8, consecutive in memory	consecutively one plane for Red, one for Green and one for Blue
RIO_RGBA_ARGB32P	4 planes RIO_Y8, consecutive in memory	consecutively one plane for A value, one for Red, one for Green and one for Blue

Figure 9. RGBA Output formats

Special Output formats /Postprocessing

For some module modes, the output format is special. Setting PostProcess to RIO_ON can be used to convert this special format to a more general one. The driver will do this postprocessing in a separate thread, one level below system priority. The postprocessing will be performed at the latest possible moment. For example, when streaming capture is being performed, only just after a call to RioStreamGetFilledBuffer, postprocessing will be done for all the filled buffers.

For users that want to be able to do their own post-processing on the special format (by setting PostProcess to RIO_OFF), the special output format is explained here for each of the input module modes that use it.
For RIO_RGBA two special formats exist. The special RGBA output format that is generated, depends on the desired output format after post-processing, so also when PostProcess is set to RIO_OFF, the ultimate output format that is desired, still must be indicated by the programmer.

The special output formats are explained below in Figure 10:

RIO_MODULE_MODE	*format* *high … low*	*Explanation*
RIO_STEREO_LOCKED	2 pixels in one Dword: 8 8 8 8 * A_1 B_1 A_0 B_0	A_0 Intensity of pixel 0 Of camera on Y connector B_0 Intensity of pixel 0 Of camera on C connector A_1 Intensity of pixel 1 Of camera on Y connector B_1 Intensity of pixel 1 Of camera on C connector intensity values range: 192(black)…255, 0…32(gray)…127 (white)
RIO_S_FULL_FRAME	2 pixels in one Dword: 8 8 8 8 * A_1 B_1 A_0 B_0	A_0 Intensity of pixel 0 of even field B_0 Intensity of pixel 0 of odd field A_1 Intensity of pixel 1 of even field B_1 Intensity of pixel 1 of odd field Intensity values range: 192(black)…255, 0…32(gray)…127 (white)
RIO_RGBA ***with Output Format after PostProcessing:*** RIO_RGBA_RGB24 ***and*** RIO_RGBA_RGB24P	R and GB lines interleaved in memory: (GB values first, R values last, every line Dword aligned) GB Dword: 2 pixels in one Dword: 8 8 8 8 G_1 B_1 G_0 B_0 R Dword: 4 pixels in one Dword: 8 8 8 8 R_3 R_2 R_1 R_0	GB Dword: G_0 Green Intensity of pixel 0 B_0 Blue Intensity of pixel 0 G_1 Green Intensity of pixel 1 B_1 Blue Intensity of pixel 1 R Dword: R_0 Red Intensity of pixel 0 R_1 Red Intensity of pixel 1 R_2 Red Intensity of pixel 2 R_3 Red Intensity of pixel 3 Intensity values range: 192(black)…255, 0…32(gray)…127 (white)
RIO_RGBA ***with Output Format after PostProcessing:*** RIO_RGBA_ARGB32 ***and*** RIO_RGBA_ARGB32P	AR and GB lines interleaved in memory: (GB values first, AR values last, every line Dword aligned) GB Dword: 2 pixels in one Dword: 8 8 8 8 G_1 B_1 G_0 B_0 AR Dword: 2 pixels in one Dword: 8 8 8 8 A_1 R_1 A_0 R_0	GB Dword: G_0 Green Intensity of pixel 0 B_0 Blue Intensity of pixel 0 G_1 Green Intensity of pixel 1 B_1 Blue Intensity of pixel 1 AR Dword: A_0 A value of pixel 0 R_0 Red Intensity of pixel 0 A_1 A value of pixel 1 R_1 Red Intensity of pixel 1 Intensity values range: 192(black)…255, 0…32(gray)…127 (white)
* See chapter KNOWN ISSUES, issue ‘Different special output format’		

Figure 10. Special Output formats

Other Enumerated Types

The enumerated types used in the software library, will be explained in this part (except for those already described as part of the 'Camera Output, Offsets, Scaling' and Output Formats' sections). These types limit the programmers choice of variables to allowable combinations. Variables of the enumerated types occur in the software library structures, which are used as arguments in the software library functions.

RIO_MODULE_MODE

Indicates the mode of the input module. The different modes are explained in the 'Rio Modules' part.

```
typedef enum _RIO_MODULE_MODE
{
    RIO_COLOR,          color mode
    RIO_BW,             black and white mode
    RIO_HQ,             high quality black and white mode
    RIO_STEREO_LOCKED,  black and white stereo locked mode (for use with two cameras)
    RIO_S_FULL_FRAME,   black and white single full frame mode (for use with full frame camera)
    RIO_FULL_FRAME,     black and white full frame mode (for use with full frame camera)
    RIO_RGBA            color RGBA mode (for use with RGBA or RGB camera)
} RIO_MODULE_MODE;
```

RIO_ON_OFF_MODE

Used for variables that can either be set to 'on' or 'off', like PostProcess (postprocessing of special output format).

```
typedef enum _RIO_ON_OFF_MODE
{
    RIO_ON,             set parameter to 'on'
    RIO_OFF             set parameter to 'off'
} RIO_ON_OFF_MODE;
```

RIO_FIELD_OR_FRAME

Indicates field or frame grabbing.

```
typedef enum _RIO_FIELD_OR_FRAME
{
    RIO_FIELD,          field capture
    RIO_FRAME,          frame capture
    RIO_FIELD_ODD,      odd field capture
    RIO_FIELD_EVEN,     even field capture
    RIO_FRAME_ODD_EVEN  frame capture, guaranteed odd field first
} RIO_FIELD_OR_FRAME;
```

RIO_INPUT_MODULE

Used to select the input module. For the Basic Rio and for the special modes that can only be used on the full version of the Rio, RIO_IM_0 should be selected, because in those cases only one input module can be used.

```
typedef enum _RIO_INPUT_MODULE
{
    RIO_IM_0,           first input module
    RIO_IM_1,           second input module
    RIO_NR_IM
} RIO_INPUT_MODULE;
```

RIO_CVBS_OR_YC

Indicates the camera is connected with a composite video connector, or with Y/C connectors. Only used in RIO_COLOR mode.

```
typedef enum _RIO_CVBS_OR_YC
{
    RIO_CVBS,           composite video connection (one connector per camera)
    RIO_YC              YC (S-VHS, Hi8) connection (two connectors per camera)
} RIO_CVBS_OR_YC;
```

RIO_TV_OR_VTR

Choice between slow (small jitter) locking or fast (bigger jitter) locking.

```
typedef enum _RIO_TV_OR_VTR
{
    RIO_TV,             slow locking with small jitter, needs steady sync. as from television set
    RIO_VTR             fast locking with bigger jitter, select this when grabbing from videotape or
                        laserdisk
} RIO_TV_OR_VTR;
```

RIO_COLOR_VIDEO_STANDARD

Used for RIO_COLOR mode. Indicates the video standard that is used.

```
typedef enum _RIO_COLOR_VIDEO_STANDARD
{
    RIO_PAL,            50 Hz  PAL
    RIO_SECAM,          50 Hz  SECAM
    RIO_NTSC            60 Hz  NTSC
} RIO_COLOR_VIDEO_STANDARD;
```

RIO_BW_VIDEO_STANDARD

Used for all module modes except RIO_COLOR. Indicates the video standard that is used.

```
typedef enum _RIO_BW_VIDEO_STANDARD
{
    RIO_CCIR,                       50 Hz CCIR
    RIO_EIA,                        60 Hz  EIA
    RIO_CCIR_PROGRESSIVE_SCAN,      25 Hz CCIR progressive scan *
    RIO_EIA_PROGRESSIVE_SCAN        30 Hz  EIA progressive scan *
} RIO_BW_VIDEO_STANDARD;
```

* Select RIO_FIELD as the only sensible option for RIO_FIELD_OR_FRAME.

RIO_GPIO_MODE

Possible modes/use for a general-purpose-io pin.

```
typedef enum _RIO_GPIO_MODE
{
    RIO_GPIO_IGNORE,    general-purpose-io pin is not used
    RIO_GPIO_INPUT,     general-purpose-io pin is used as input
    RIO_GPIO_OUTPUT,    general-purpose-io pin is used as output
    RIO_GPIO_IRQ_RISE,  rise in signal on general-purpose-io pin will generate an interrupt
    RIO_GPIO_IRQ_FALL,  fall in signal on general-purpose-io pin will generate an interrupt
    RIO_GPIO_IRQ_BOTH   rise and fall in signal on general-purpose-io pin will generate an interrupt
} RIO_GPIO_MODE;
```

RIO_GPIO_PIN

Used to select a general-purpose-io pin (see Figure 14).

```
typedef enum _RIO_GPIO_PIN
{
    RIO_GPIO_0,         first general-purpose-io pin
    RIO_GPIO_1,         second general-purpose-io pin
    RIO_GPIO_2,         third general-purpose-io pin
    RIO_GPIO_3,         fourth general-purpose-io pin
    RIO_NR_GPIO
} RIO_GPIO_PIN;
```

Structures

The C-library structs that are used by the software library, will be explained in this chapter.

PRIODL

Contains information about the number of Rio boards and their IDs.

```
typedef struct _RIODL
{
    int NrBoards;              number of Rio boards on the PCI bus
                               (theoretical maximum = number of  busmaster PCI slots)
    int BoardId[1];            array of BoardIDs of the Rio boards
} *PRIODL;
```

RIO_VIDEO_HDR

Rio video header, indicates buffer used for streaming capture (see RioStreamAddEmptyBuffer and RioStreamGetFilledBuffer).

```
typedef struct _RIO_VIDEO_HDR
{
    HANDLE ImageBuffer;        HANDLE to the ImageBuffer
    DWORD User;                can be used to store a custom ID
    DWORD TimeCaptured;        time in milliseconds, indicating the moment of capture finish
                               (measured from StreamStart=0)
    DWORD Reserved[4];         RESERVED for internal use, NEVER to be used by user
} RIO_VIDEO_HDR, *PRIO_VIDEO_HDR;
```

RIO_FLASH_PARAMS

Flash parameters, indicates flash settings used during triggered capture (see RioTriggerCapture).

```
typedef struct _RIO_FLASH_PARAMS
{
   RIO_GPIO_PIN FlashPin;      general-purpose-io pin to be used for flash signal, must be set with
                               RioGpioControl to RIO_GPIO_OUTPUT
    uint16 StartLine;          image-line where flash may start
    uint16 EndLine;            image-line where flash will finish
    uint16 Length;             flash-length in image-lines, usually 1.
                               (restriction: Length <= (EndLine-StartLine)+1)
                               The flash will start at line (Endline-Length+1) and finish at line EndLine
    uint16 VideoOutDelay;      number of the field to be captured, counting from flash-field=0
                               (usually VideoOutDelay will be set to 1)
} RIO_FLASH_PARAMS, *PRIO_FLASH_PARAMS;
```

RIO_TBUFFER_PARAMS

Trigger buffer parameters, indicates an array of buffers that is used to store a capture-sequence during triggered capture (see RioTriggerCapture).

```
typedef struct _RIO_TBUFFER_PARAMS
{
    RIO_GPIO_PIN TriggerPin;     general-purpose-io pin to be used as trigger pin, must be set with
                                 RioGpioControl to RIO_GPIO_IRQ_RISE, RIO_GPIO_IRQ_FALL or
                                 RIO_GPIO_IRQ_BOTH
    uint32 NrBuffers;            number of buffers in the buffer-array
    HANDLE *ImageBuffer;         array of HANDLEs to imagebuffer
    uint32 TriggerPosition;      the position of the triggered capture in the capture sequence (zero based)
                                 (restriction: 0 <= TriggerPosition < NrBuffers)
    uint32 *TriggerBufferNr;     the number of the image-buffer that was used to store the triggered
                                 capture (zero based)
    uint16 CapturePeriod;        the period (in fields) between the individual captures that each fill an
                                 imagebuffer of the imagebuffer array.
                                 1= continuous capture, 2 = skip 1 field before next capture, and so on
} RIO_TBUFFER_PARAMS, *PRIO_TBUFFER_PARAMS;
```

A call to RioTriggerCapture will fill the total array of imagebuffers with a capture sequence. The array is used as a circular buffer (see Figure 12).

TriggerPosition indicates the zero-based position of the triggered capture in the capture sequence. It is used to indicate how many captures before and after the actual trigger moment should be stored:

Capture in the capture sequence	*Meaning*
0..(*TriggerPosition*-1)	the captures, preceding the trigger moment
TriggerPosition	the capture at trigger moment
(*TriggerPosition*+1)..(*NrBuffers*-1)	the captures following the capture at trigger moment

Figure 11. Explanation of variable TriggerPosition

TriggerBufferNr is the zero-based number of the imagebuffer where the actual capture at trigger moment was stored.

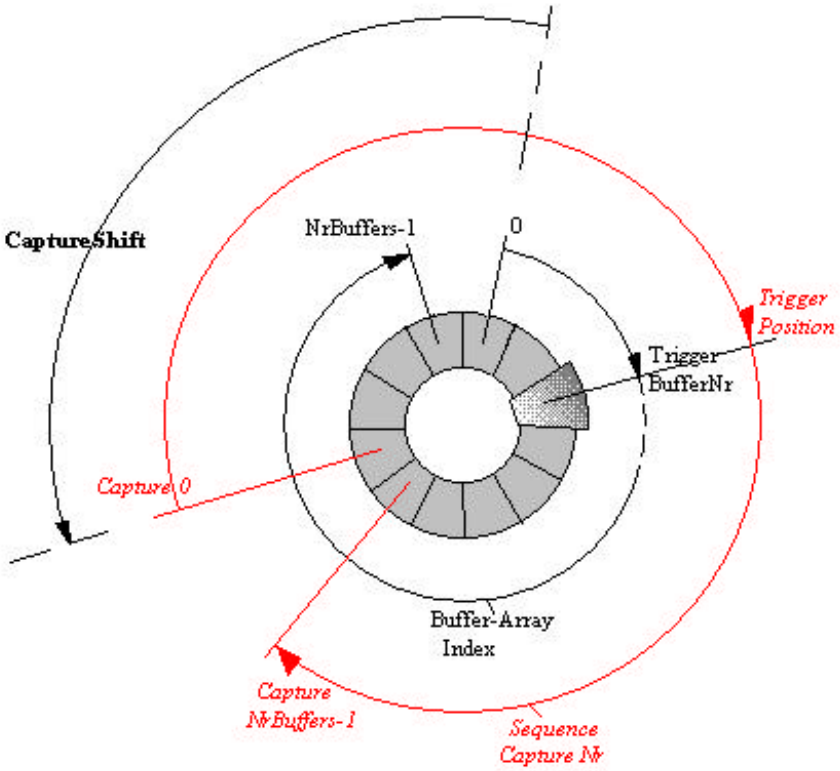


Figure 12. Circular use of array of ImageBuffers

Because the array of imagebuffers is used as a circular buffer by RioTriggerCapture as shown in Figure 12, the capture sequence will be stored into the array of imagebuffers, shifted by a number (*CaptureShift*):

CaptureShift = TriggerBufferNr - TriggerPosition

(*CaptureShift* can be a positive or negative number)

In general, to find the imagebuffer that was used to store any particular capture in the sequence, we calculate:

ImageBufferNr = (*Capture Nr+ CaptureShift+ NrBuffers*) mod *NrBuffers*

As explained in ‘Rio Modules’, several different Rio Module modes exist. Structs are used to set a specific input module on the Rio board via RioSetInputModule. RioGetInputModule can be used to obtain the information about the current input module setting.

The structs that are used to indicate the different Module modes are described below. Each of them contains the parameter set specifically possible for that mode. The different enumerated types that are used as part of the struct, already were explained in ‘Camera Output, Offsets, Scaling’, ‘Output Formats’ and ‘Other Enumerated Types’.

RIO_MODULE_COLOR

Used to set an input module on the Rio board in RIO_COLOR mode, as described in ‘Rio Modules’.

```
typedef struct _RIO_MODULE_COLOR
{
    RIO_INPUT_MODULE InputModule;          which Input module to use (use RIO_IM_0 for
                                           Basic Rio)
    RIO_SCALER Scaler;                     scaler selection
    RIO_COLOR_VIDEO_STANDARD VideoStandard; color video standard that is used
    RIO_TV_OR_VTR TvOrVtr;                 TV or VTR pixel locking
    RIO_CVBS_OR_YC CvbsOrYc;               composite video or YC connection
    RIO_FIELD_OR_FRAME FieldOrFrame;       field or frame grabbing
    RIO_COLOR_OUTPUT_FORMAT OutputFormat;  choice of color output format
} RIO_MODULE_COLOR, *PRIO_MODULE_COLOR;
```

RIO_MODULE_BW

Used to set an input module on the Rio board in RIO_BW mode, as described in ‘Rio Modules’.

```
typedef struct _RIO_MODULE_BW
{
    RIO_INPUT_MODULE InputModule;          which Input module to use (use RIO_IM_0 for
                                           Basic Rio board)
    RIO_SCALER Scaler;                     scaler selection
    RIO_BW_VIDEO_STANDARD VideoStandard;   black & white video standard that is used
    RIO_TV_OR_VTR TvOrVtr;                 TV or VTR pixel locking
    RIO_FIELD_OR_FRAME FieldOrFrame;       field or frame grabbing
    RIO_BW_OUTPUT_FORMAT OutputFormat;     choice of black & white output format
} RIO_MODULE_BW, *PRIO_MODULE_BW;
```

RIO_MODULE_HQ

Used to set RIO_IM_0 on the Rio board in RIO_HQ mode (which can only be used for the full version of the Rio), as described in 'Rio Modules'.

```
typedef struct _RIO_MODULE_HQ
{
    RIO_SCALER Scaler;                      scaler selection
    RIO_BW_VIDEO_STANDARD VideoStandard;    black & white video standard that is used
    RIO_TV_OR_VTR TvOrVtr;                  TV or VTR pixel locking
    RIO_FIELD_OR_FRAME FieldOrFrame;        field or frame grabbing
    RIO_BW_OUTPUT_FORMAT OutputFormat;      choice of black & white output format
} RIO_MODULE_HQ, *PRIO_MODULE_HQ;
```

RIO_MODULE_STEREO_LOCKED

Used to set an input module on the Rio board in RIO_STEREO_LOCKED mode, as described in 'Rio Modules'.

```
typedef struct _RIO_MODULE_STEREO_LOCKED
{
    RIO_INPUT_MODULE InputModule;           which Input module to use (use RIO_IM_0 for Basic Rio
                                            board)
    RIO_SCALER Scaler;                      scaler selection
    RIO_BW_VIDEO_STANDARD VideoStandard;    black & white video standard that is used
    RIO_TV_OR_VTR TvOrVtr;                  TV or VTR pixel locking
    RIO_FIELD_OR_FRAME FieldOrFrame;        field or frame grabbing
    RIO_ON_OFF_MODE PostProcess;            RIO_ON to postprocess the special output format to 2
                                            RIO_Y8 planes or RIO_OFF to do no postprocessing
} RIO_MODULE_STEREO_LOCKED, *PRIO_MODULE_STEREO_LOCKED;
```

RIO_MODULE_S_FULL_FRAME

Used to set an input module on the Rio board in RIO_S_FULL_FRAME mode, as described in 'Rio Modules'.

```
typedef struct _RIO_MODULE_S_FULL_FRAME
{
    RIO_INPUT_MODULE InputModule;           which Input module to use (use RIO_IM_0 for Basic Rio
                                            board)
    RIO_SCALER Scaler;                      scaler selection
    RIO_BW_VIDEO_STANDARD VideoStandard;    black & white video standard that is used
    RIO_TV_OR_VTR TvOrVtr;                  TV or VTR pixel locking
    RIO_ON_OFF_MODE PostProcess;            RIO_ON to postprocess the special output format to
                                            RIO_Y8 or RIO_OFF to do no postprocessing
} RIO_MODULE_S_FULL_FRAME, *PRIO_MODULE_S_FULL_FRAME;
```

RIO_MODULE_FULL_FRAME

Used to set the Rio board in RIO_FULL_FRAME mode (which can only be used for the full version of the Rio), as described in 'Rio Modules'.

```
typedef struct _RIO_MODULE_FULL_FRAME
{
    RIO_BW_VIDEO_STANDARD VideoStandard;   black & white video standard that is used
    RIO_TV_OR_VTR TvOrVtr;                 TV or VTR choice
    RIO_BW_OUTPUT_FORMAT OutputFormat;     choice of black & white output format
} RIO_MODULE_FULL_FRAME, *PRIO_MODULE_FULL_FRAME;
```

RIO_MODULE_RGBA

Used to set the Rio board in RIO_RGBA mode (only available on the full version of the Rio), as described in 'Rio Modules'.

```
typedef struct _RIO_MODULE_RGBA
{
    RIO_BW_VIDEO_STANDARD VideoStandard;   black & white video standard that is used
    RIO_TV_OR_VTR TvOrVtr;                 TV or VTR choice
    RIO_FIELD_OR_FRAME FieldOrFrame;       field or frame grabbing
    RIO_ON_OFF_MODE PostProcess;           RIO_ON to postprocess the special output format to one
                                           of the possible RGBA output formats, or RIO_OFF to do
                                           no postprocessing
    RIO_RGBA_OUTPUT_FORMAT OutputFormat;   choice of RGBA output format in case of postprocessing
} RIO_MODULE_RGBA, *PRIO_MODULE_RGBA;
```

RIO_MODULE

The RIO_MODULE struct is the structure that encapsulates the possible Rio module structs into one struct, and that is passed to RioSetInputModule to set a specific input module on the Rio board. RioGetInputModule uses this struct to give back the information about the current input module setting.

```
typedef struct _RIO_MODULE
{
    RIO_MODULE_MODE Mode;                  the module mode that is presented by the
                                           RIO_MODULE struct
    union                                  the member to use indicated by the Mode
    {
        RIO_MODULE_COLOR Color;            member to use when Mode==RIO_COLOR
        RIO_MODULE_BW Bw;                  member to use when Mode==RIO_BW
        RIO_MODULE_HQ Hq;                  member to use when Mode==RIO_HQ
        RIO_MODULE_STEREO_LOCKED Sl;       member to use when Mode==RIO_STEREO_LOCKED
        RIO_MODULE_S_FULL_FRAME Sff;       member to use when Mode==RIO_S_FULL_FRAME
        RIO_MODULE_FULL_FRAME Ff;          member to use when Mode==RIO_FULL_FRAME
        RIO_MODULE_RGBA Rgba;              member to use when Mode==RIO_RGBA
    } Def;
} RIO_MODULE, *PRIO_MODULE;
```

Error Codes

- The next error can occur at system start-up. In this case it can be detected by using the Control Panel applet:

  RIO_ERR_INIT initialization failed.

- A listing of the possible low-level error return codes is shown below. These return codes can, generally speaking, be returned by any function.

  RIO_ERR_DEBI_READ
  RIO_ERR_DEBI_SWTIMEOUT
  RIO_ERR_DEBI_WRITE
  RIO_ERR_I2C
  RIO_ERR_I2C_AL
  RIO_ERR_I2C_APERR
  RIO_ERR_I2C_BUSYTIMEOUT
  RIO_ERR_I2C_DRERR
  RIO_ERR_I2C_DTERR
  RIO_ERR_I2C_RING_ADD
  RIO_ERR_I2C_RING_REMOVE
  RIO_ERR_I2C_SHORTTIMEOUT
  RIO_ERR_I2C_SPERR
  RIO_ERR_I2C_SPUR_IRQ
  RIO_ERR_PARAM
  RIO_ERR_PARAM_COUNT
  RIO_ERR_PARAM_CHANNEL
  RIO_ERR_PARAM_WRITE
  RIO_ERR_PARAM_MASK
  RIO_ERR_PARAM_BURST
  RIO_ERR_PARAM_THRES
  RIO_ERR_PARAM_IN_WIDTH_HEIGHT
  RIO_ERR_PARAM_IN_OFFSET
  RIO_ERR_PARAM_OUT_WIDTH_HEIGHT
  RIO_ERR_PARAM_FORMAT
  RIO_ERR_PARAM_GAMMA
  RIO_ERR_PARAM_XYSCALE
  RIO_ERR_PARAM_YSCALE
  RIO_ERR_PARAM_XSCALE
  RIO_ERR_PARAM_VIDPORT
  RIO_ERR_PARAM_IN_WIDTH_HEIGHT_BRS
  RIO_ERR_PARAM_IN_OFFSET_BRS
  RIO_ERR_PARAM_OUT_WIDTH_HEIGHT_BRS
  RIO_ERR_PARAM_ALIGN
  RIO_ERR_PARAM_FORMAT_BRS
  RIO_ERR_PARAM_XSCALE_BRS

RIO_ERR_PARAM_YSCALE_BRS
RIO_ERR_PARAMS
RIO_ERR_PARAMS_VALUE
RIO_ERR_PARAMS_INPUT
RIO_ERR_PARAMS_GAIN
RIO_ERR_RING_ADJUST
RIO_ERR_SEM_CREATE
RIO_ERROR
RIO_OCF_REG_OUT_OF_RANGE
RIO_SCALER_BUSY capture error

Contact Ellips to get more information about the specific errors.

- Apart from those documented in the chapter 'Functions for Windows', the next error return values can be returned by any function under Windows NT/2000:

  RIO_DEVICE_IO_CONTROL
  RIO_TIMEOUT
  RIO_WAIT_FAILED

  Contact Ellips to get more information about these errors.

- The other errors that can occur, are explained at the appropriate places in the chapter 'Functions for Windows':

  RIO_ALREADY_GETTING_BUFFER
  RIO_ALREADY_STARTED_CAPTURE Windows NT/2000 only
  RIO_ALREADY_WAITING_FOR_EXT_INT
  RIO_BOARD_ID_WRITE_ERROR
  RIO_CAPTURE_ERROR
  RIO_DEBI_READ_ERROR
  RIO_DEBI_WRITE_ERROR
  RIO_DRIVER_NOT_LOADED
  RIO_INSUFFICIENT_MEMORY
  RIO_INVALID_AUTO_GAIN_MODE
  RIO_INVALID_BOARD_ID
  RIO_INVALID_CAMERA_FOR_INPUT_MODULE
  RIO_INVALID_CAPTURE_PERIOD
  RIO_INVALID_CVBS_OR_YC
  RIO_INVALID_DEST_RECT
  RIO_INVALID_EEPROM_ADDRESS
  RIO_INVALID_EEPROM_DATASIZE_FOR_ADDRESS
  RIO_INVALID_EVENT
  RIO_INVALID_FIELD_OR_FRAME
  RIO_INVALID_FLASH_LINE

RIO_INVALID_FLASH_PIN
RIO_INVALID_GPIO_MODE
RIO_INVALID_GPIO_PIN
RIO_INVALID_HANDLE
RIO_INVALID_HQ_GAIN
RIO_INVALID_HQ_OFFSET
RIO_INVALID_I2C_REG
RIO_INVALID_IMAGE_BUFFER
RIO_INVALID_INPUT_FOR_INPUT_MODULE
RIO_INVALID_INPUT_GAIN
RIO_INVALID_INPUT_MODULE
RIO_INVALID_LED_STATE
RIO_INVALID_MICROSEC_PER_CAPTURE
RIO_INVALID_MODULE_MODE
RIO_INVALID_NR_BUFFERS
RIO_INVALID_OUTPUT_FORMAT
RIO_INVALID_OVERLAPPED
RIO_INVALID_POINTER
RIO_INVALID_POST_PROCESS
RIO_INVALID_SCALER
RIO_INVALID_SRC_RECT
RIO_INVALID_TBUFFER_PARAMS
RIO_INVALID_TRIGGER_BUFFER_NR
RIO_INVALID_TRIGGER_PIN
RIO_INVALID_TRIGGER_POSITION
RIO_INVALID_TV_OR_VTR
RIO_INVALID_VIDEO_OUT_DELAY
RIO_INVALID_VIDEO_STANDARD
RIO_LOCK_FAILED
RIO_MEM_ALLOC
RIO_NO_FILLED_BUFFERS
RIO_OK
RIO_PENDING
RIO_UNABLE_TO_CREATE_FILE
RIO_UNABLE_TO_MAP_FILE
RIO_UNABLE_TO_POST_PROCESS
RIO_UNLOCK_FAILED
RIO_VIDEO_OVERFLOW
RIO_VIDEO_OVERFLOW_NO_POST_PROCESS

Functions for Windows

To facilitate developing 32-bit applications for the Rio Frame grabber, a C library is included that provides support functions that are easy to use. For Windows 95 and Windows NT/2000 the driver and DLL are different, but the programmer interface to access the functions of the grabber are the same. On the next pages the functions will be explained. The structures that are passed as arguments to these functions are explained in the 'Structures' part of this reference.

RioCapture

#include <Rio.h>

Function int RioCapture(int *BoardId*, RIO_INPUT_MODULE *InputModule*,
BOOL *Continuous*, BOOL *SquarePixels*, BOOL *TopDown*,
RECT **SrcRect*, RECT **DestRect*, int *Pitch*,
HANDLE *ImageBufferHandle*, OVERLAPPED **pOverlapped*);

Return Value

RIO_OK	successful (in case *pOverlapped*==NULL)
RIO_PENDING	capture started successfully (in case *pOverlapped*!=NULL). WaitForSingleObject(*pOverlapped*->hEvent,..) must be used to wait for the capture to be completed.
RIO_ALREADY_STARTED_CAPTURE	**only Windows NT/2000** RioCapture already was called previously, WaitForSingleObject(*pOverlapped*->hEvent,..) should first be used to wait for that capture, before calling RioCapture again.
RIO_CAPTURE_ERROR	internal error, user has probably forgotten to call RioScatterLock before starting capture.
RIO_DEVICE_IO_CONTROL	internal error, contact Ellips.
RIO_VIDEO_OVERFLOW	some pixels in the image were not obtained correctly
RIO_UNABLE_TO_POST_PROCESS	postprocessing failed
RIO_VIDEO_OVERFLOW_NO_POST_PROCESS	some pixels in the image were not obtained correctly and postprocessing failed
RIO_TIMEOUT	the time-out has expired (in case *pOverlapped*==NULL)
RIO_WAIT_FAILED	the wait failed (in case *pOverlapped*==NULL)

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_INPUT_MODULE	an invalid *InputModule* was given
RIO_INVALID_SRC_RECT	*SrcRect* was NULL or **SrcRect* was not normalized
RIO_INVALID_DEST_RECT	*DestRect* was NULL or **DestRect* was not normalized
RIO_INVALID_HANDLE	*ImageBufferHandle* was NULL
RIO_INVALID_OVERLAPPED	The invalid combination of *Continuous* capture and *pOverlapped*==NULL was used

Parameters

BoardId	BoardID of the Rio board to use for capture.
InputModule	Input module to be used for the capture. Can be either RIO_IM_0 or RIO_IM_1.
Continuous	BOOL, indicating continuous capture on (TRUE) or off (FALSE).
SquarePixels	BOOL, Only used when RIO_FIELD_OR_FRAME is set to RIO_FIELD, RIO_FIELD_ODD or RIO_FIELD_EVEN. If *SquarePixels*==TRUE, the captured source will be in square pixels, although in field mode.
Topdown	BOOL, indicating topdown line image-building (TRUE) or not.
**SrcRect*	The source rectangle (bottom and right will not be included).
**DestRect*	The destination rectangle (bottom and right will not be included). Must have the same size as the source rectangle. In future versions, the destination rectangle may be smaller than source rectangle (downscale). Both rectangles need to be normalized (top<bottom, left<right) and must represent a minimum image size of one Dword in memory.
Pitch	The line-pitch in bytes, needed to correctly align the lines in memory (depending on resolution).
ImageBufferHandle	The HANDLE passed to indicate the memory buffer
pOverlapped	Pointer to an OVERLAPPED struct. *pOverlapped*->hEvent can be used to signal the completed capture.

Remarks

Capture can be achieved in three different ways:

- If NULL is passed for *pOverlapped*, the function will only return after the capture has finished (RIO_OK or RIO_VIDEO_OVERFLOW returned) or the time-out that was set in RioCaptureTimeOut will have expired (RIO_TIMEOUT returned) (see Example1). *Continuous* MUST BE set to FALSE if *pOverlapped*==NULL!! The combination of continuous capture and *pOverlapped*==NULL is NOT ALLOWED!!
- If *pOverlapped* is a valid pointer to an OVERLAPPED struct with *Continuous* set to TRUE, continuous capture is done with the same imagebuffer. RIO_PENDING is returned if RioCapture started continuous capture successfully. To obtain an image, RioCaptureStop first must be called. After WaitForSingleObject with the *pOverlapped*->hEvent, the last image can be obtained. *pOverlapped* can (then) be passed to RioGetOverlappedResult to get the return code of the overlapped operation, that indicates if capture was completed successfully (see Example 2).
- If *pOverlapped* is a valid pointer to an OVERLAPPED struct with *Continuous* set to FALSE, one image will be captured. RIO_PENDING is returned if RioCapture started capture successfully. The programmer must use WaitForSingleObject with the *pOverlapped*->hEvent member, to wait for the capture to be completed. *pOverlapped* can (then) be passed to RioGetOverlappedResult to get the return code of the overlapped operation, that indicates if capture was completed successfully (see Example 3).

If the mode is RIO_HQ, RIO_FULL_FRAME or RIO_RGBA, only RIO_IM_0 should be used as value for *InputModule.*

Example 1. In the first example RioCapture is called for the first input module, with *pOverlapped* == NULL, in a non-multithreaded way. pRiodl is assumed to be a correctly created PRIODL Rio device list of the available Rio boards (see RioCreateRiodl example). First input module and a camera are assumed to be set correctly on the first Rio board (see RioSetInputModule example). Pitch (line-pitch in bytes of imagebuffer), XRes (horizontal number of pixels in imagebuffer), YRes (vertical number of pixels in imagebuffer) and ImageBufferHandle are assumed to be correctly created (see RioScatterLock example).

```
char s[80];
int Result;

RECT SourceRect, DestRect;

// setting time-out to 3 seconds
Result = RioCaptureTimeOut(pRiodl->BoardId[0], RIO_IM_0, 3000);
if ( Result != RIO_OK )
{
   sprintf(s,"Changing time-out to other then default, failed. Result: 0x%X", Result);
   MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
}


// in the example the total imagebuffer will be filled
int LineOffset = 15;
SetRect(&SourceRect, 0, LineOffset, XRes, LineOffset+YRes);
SetRect(&DestRect, 0, 0, XRes, YRes);

Result = RioCapture(pRiodl->BoardId[0], RIO_IM_0, FALSE, FALSE, TRUE,
                      &SourceRect, &DestRect, Pitch, ImageBufferHandle, NULL);

if ( Result != RIO_OK )                  // will be RIO_TIMEOUT or RIO_VIDEO_OVERFLOW
{
    sprintf(s, "Capture failed. Result: 0x%X", Result);
    MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
}
else
{
    // user can now visualize or save imagebuffer to file
}
```

Example 2. In the second example RioCapture is called with *pOverlapped* != NULL, in a multithreaded way with continuous capture. pRiodl is assumed to be a correctly created PRIODL Rio device list of the available Rio boards (see RioCreateRiodl example). First input module and a camera are assumed to be set correctly on the first Rio board (see RioSetInputModule example). Pitch (line-pitch in bytes of imagebuffer), XRes (horizontal number of pixels in imagebuffer), YRes (vertical number of pixels in imagebuffer) and ImageBufferHandle are assumed to be correctly created (see RioScatterLock example).

```
char s[80];
int Result, OverlapResult;
BOOL bOverlappedProcStarted = FALSE;
OVERLAPPED Overlapped;
DWORD WFSO_Result;
RECT SourceRect, DestRect;

Overlapped.hEvent  = CreateEvent(NULL,              // pointer to security attributes
                                 TRUE,              // flag for manual-reset event
                                 FALSE,             // flag for initial state
```

```
                                    NULL);              // pointer to event-object name

// in the example the total imagebuffer will be filled
int LineOffset = 15;
SetRect(&SourceRect, 0, LineOffset, XRes, LineOffset+YRes);
SetRect(&DestRect, 0, 0, XRes, YRes);

// start continuous capture
Result = RioCapture(pRiodl->BoardId[0], RIO_IM_0, TRUE, FALSE, TRUE,
                        &SourceRect, &DestRect, Pitch, ImageBufferHandle, &Overlapped);
if ( Result != RIO_PENDING )
{
    sprintf(s, "Starting capture failed. Result: 0x%X", Result);
    MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
}
else bOverlappedProcStarted = TRUE;

.. .. .. ..


// if (bOverlappedProcStarted), continuous capture has started. It can be stopped and the
last capture can be waited for in a separate thread:
Result = RioCaptureStop(pRiodl->BoardId[0], RIO_IM_0);
if ( Result != RIO_OK )
{
    sprintf(s, "Continuous capture could not be stopped. Result: 0x%X", Result);
    MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
}
else
{
    WFSO_Result= WaitForSingleObject(Overlapped.hEvent, 3000);
    if ( WFSO_Result == WAIT_FAILED )
    {
        sprintf(s, "WaitForSingleObject(Overlapped.hEvent, 3000) failed");
        MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION | MB_SETFOREGROUND);
    }
    else
    {
        if ( WFSO_Result == WAIT_TIMEOUT )
        {
            sprintf(s, "The time-out has expired");
            MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION | MB_SETFOREGROUND);
        }
        else
        {
            Result = RioGetOverlappedResult(&Overlapped, &OverlapResult, TRUE);
            if ( (Result != RIO_OK) || (OverlapResult !=  RIO_OK) )
            {
                sprintf(s, "Error, Res: 0x%X, OverlRes: 0x%X", Result, OverlapResult);
                MessageBox(NULL,s,NULL, MB_OK | MB_ICONEXCLAMATION | MB_SETFOREGROUND);
            }
            else
            {
                // user can now visualize or save imagebuffer to file
            }
        }
    }
}

.. .. .. ..

// continuous capture or waiting for the last capture after RioCaptureStop can be
// canceled:
Result = RioCaptureCancel(pRiodl->BoardId[0], RIO_IM_0);
if ( Result != RIO_OK )
{
    sprintf(s, "RioCaptureCancel failed. Result: 0x%X", Result);
    MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION | MB_SETFOREGROUND);
}
else
{
    // in case we were waiting for the last capture in a separate thread, this
    // thread where WaitForSingleObject is being done, can now be closed
}
```

Example 3. In the third example RioCapture is called with *pOverlapped* != NULL, in a multithreaded way. A single capture is done (non-continuous capture). pRiodl is assumed to be a correctly created PRIODL Rio device list of the available Rio boards (see RioCreateRiodl example). First input module and a camera are assumed to be set correctly on the first Rio board (see RioSetInputModule example). Pitch (line-pitch in bytes of imagebuffer), XRes (horizontal number of pixels in imagebuffer), YRes (vertical number of pixels in imagebuffer) and ImageBufferHandle are assumed to be correctly created (see RioScatterLock example).

```
char s[80];
int Result, OverlapResult;
BOOL bOverlappedProcStarted = FALSE;
OVERLAPPED Overlapped;
DWORD WFSO_Result;
RECT SourceRect, DestRect;
Overlapped.hEvent  = CreateEvent(NULL,              // pointer to security attributes
                                 TRUE,              // flag for manual-reset event
                                 FALSE,             // flag for initial state
                                 NULL);             // pointer to event-object name
// in the example the total imagebuffer will be filled
int LineOffset = 15;
SetRect(&SourceRect, 0, LineOffset, XRes, LineOffset+YRes);
SetRect(&DestRect, 0, 0, XRes, YRes);

Result = RioCapture(pRiodl->BoardId[0], RIO_IM_0, FALSE, FALSE, TRUE,
                    &SourceRect, &DestRect, Pitch, ImageBufferHandle, &Overlapped );
if ( Result != RIO_PENDING )
{
    sprintf(s, "Starting capture failed. Result: 0x%X", Result);
    MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
}
else bOverlappedProcStarted = TRUE;

.. .. .. ..

// if (bOverlappedProcStarted), waiting for capture can be done in separate thread:
WFSO_Result= WaitForSingleObject(Overlapped.hEvent, 3000);
if ( WFSO_Result == WAIT_FAILED )
{
    sprintf(s, "WaitForSingleObject(Overlapped.hEvent, 3000) failed");
    MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION | MB_SETFOREGROUND);
}
else
{
    if ( WFSO_Result == WAIT_TIMEOUT )
    {
        sprintf(s, "The time-out has expired");
        MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION | MB_SETFOREGROUND);
    }
    else
    {
        Result = RioGetOverlappedResult(&Overlapped, &OverlapResult, TRUE);
        if ( (Result != RIO_OK) || (OverlapResult !=  RIO_OK) )
        {
           sprintf(s, "Error, Res: 0x%X, OverlapRes: 0x%X", Result, OverlapResult);
           MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION | MB_SETFOREGROUND);
        }
        else  { // user can now visualize or save imagebuffer to file }
    }
}
.. .. .. ..

// waiting for the capture can be canceled
Result = RioCaptureCancel(pRiodl->BoardId[0], RIO_IM_0);
if ( Result != RIO_OK )
{
    sprintf(s, "RioCaptureCancel failed. Result: 0x%X", Result);
    MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION | MB_SETFOREGROUND);
}
else  {  // now the thread where WaitForSingleObject is being done, can now be closed }
```

RioCaptureCancel

#include <Rio.h>

Function int RioCaptureCancel(int *BoardId*, RIO_INPUT_MODULE *InputModule*);

Return Value RIO_OK successful

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_INPUT_MODULE	an invalid *InputModule* was given

Parameters

BoardId	BoardID of the Rio board on which to cancel the capture.
InputModule	Input module for which to cancel the capture. Can be either RIO_IM_0 or RIO_IM_1.

Remarks Cancels the capture. No event will be returned after the capture is canceled. If the mode is RIO_HQ, RIO_FULL_FRAME or RIO_RGBA, only RIO_IM_0 should be used as value for *InputModule*.

Example See RioCapture Example 2 and Example 3.

RioCaptureStop

#include <Rio.h>

Function int RioCaptureStop(int *BoardId*, RIO_INPUT_MODULE *InputModule*);

Return Value RIO_OK successful

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_INPUT_MODULE	an invalid *InputModule* was given

Parameters

BoardId	BoardID of the Rio board on which to stop continuous capture.
InputModule	Input module for which to stop the continuous capture. Can be either RIO_IM_0 or RIO_IM_1.

Remarks Only does something when the module is in continuous capture. Stops the capture. An event will be returned after the capture is stopped, in the *pOverlapped*->hEvent that was passed in RioCapture(), which must be waited for with WaitForSingleObject().

If the mode is RIO_HQ, RIO_FULL_FRAME or RIO_RGBA, only RIO_IM_0 should be used as value for *InputModule.*

Example See RioCapture Example 2.

RioCaptureTimeOut

#include <Rio.h>

Function int RioCaptureTimeOut(int *BoardId*, RIO_INPUT_MODULE *InputModule*, long *TimeOut*);

Return Value

RIO_OK	successful

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_INPUT_MODULE	an invalid *InputModule* was given

Parameters

BoardId	BoardID of the Rio board on which to set the capture time-out.
InputModule	Input module for which to set the capture time-out. Can be either RIO_IM_0 or RIO_IM_1.
TimeOut	The time-out to be set (in msec).

Remarks This function sets the time-out for the capture. Default after initialization before this function is called for the first time is 1000 msec. This time-out will be used in RioCapture when called with *pOverlapped* = NULL.

If the mode is RIO_HQ, RIO_FULL_FRAME or RIO_RGBA, only RIO_IM_0 should be used as value for *InputModule.*

Example See RioCreateRiodl and RioCapture Example 1.

RioClose

#include <Rio.h>

Function int RioClose(void);

Return Value RIO_OK successful

Remarks Should be called after all other operations are finished. Is necessary to clean up neatly and avoid memory leaks.

RioCreateRiodl

#include <Rio.h>

Function int RioCreateRiodl(PRIODL *pRiodl*);

Return Value RIO_OK successful

parameter checking:

RIO_INVALID_POINTER *pRiodl* was NULL

Parameters *pRiodl* Pointer to a struct (Rio device list), containing information about the number of Rio boards and BoardIDs.

Remarks Returns information about the number of Rio boards and BoardIDs. The information is stored in *pRiodl (see PRIODL struct at page 24).

Example In this example, a Rio device list is created, and for all Rio boards out of the device list, the capture time-out is set. Another example can be found at 'RioSetBoardId'.

```
int Result;
char s[80];

// allocating memory for Rio device list
PRIODL pRiodl = (PRIODL) new BYTE[RioSizeOfRiodl()];

Result = RioCreateRiodl(pRiodl);            // device list is filled with information
if ( Result != RIO_OK )
{
    sprintf(s, "Unable to create device list. Result: 0x%X", Result);
    MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
}
else
{
    BOOL IsBasicRioVersion;

    // going through all Rio-boards in the list
    for (int  i = 0; i < pRiodl->NrBoards; i++ )
    {
        // setting capture time-out of first input module
        RioCaptureTimeOut(pRiodl->BoardId[i], RIO_IM_0, 1000);

        RioGetBoardType(pRiodl->BoardId[i], &IsBasicRioVersion);

        // basic version only can be used with first input module
        if ( !IsBasicRioVersion )
        {
            // setting capture time-out of second input module
            RioCaptureTimeOut(pRiodl->BoardId[i], RIO_IM_1, 1000);
        }
    }
}

delete[] ((BYTE*) pRiodl);    // cleaning up, after device list is not needed anymore
```

----------DIRECT HARDWARE ACCESS FUNCTIONS----------
----------NEVER USE WITHOUT CONSULTING ELLIPS----------

RioDebiRead

#include <Rio.h>

Function int RioDebiRead(int *BoardId*, int *Address*, int **Value*);

Return Value

RIO_OK	successful
RIO_DEBI_READ_ERROR	RioDebiRead failed

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given

Parameters

BoardId	BoardID of the Rio board on which to read data from the Debi bus.
Address	Address on the Debi bus to read data from.
**Value*	One byte value to be read from the Debi bus.

Remarks Reads data from the Debi bus on a Rio board. For more information on the Debi bus, see Philips SAA7146 Data Sheet.

----------DIRECT HARDWARE ACCESS FUNCTIONS----------
----------NEVER USE WITHOUT CONSULTING ELLIPS----------

----------DIRECT HARDWARE ACCESS FUNCTIONS----------
----------NEVER USE WITHOUT CONSULTING ELLIPS----------

RioDebiWrite

#include <Rio.h>

Function int RioDebiWrite(int *BoardId*, int *Address*, int *Value*);

Return Value

RIO_OK	successful
RIO_DEBI_WRITE_ERROR	RioDebiWrite failed

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given

Parameters

BoardId	BoardID of the Rio board on which to write data to the Debi bus.
Address	Address on the Debi bus to write data to.
Value	One byte value to be written to the Debi bus.

Remarks Writes data to the Debi bus a Rio board, **NEVER USE WITHOUT CONSULTING ELLIPS**. For more information on the Debi bus, see Philips SAA7146 Data Sheet.

----------DIRECT HARDWARE ACCESS FUNCTIONS----------
----------NEVER USE WITHOUT CONSULTING ELLIPS----------

----------DIRECT HARDWARE ACCESS FUNCTIONS----------
----------NEVER USE WITHOUT CONSULTING ELLIPS----------

RioEepromRead

#include <Rio.h>

Function int RioEepromRead(int *BoardId*, int *Address*, void **Data*, int *Size*);

Return Value

RIO_OK	successful

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_EEPROM_ADDRESS	an invalid *Address was given*
RIO_INVALID_EEPROM_DATASIZE_FOR_ADDRESS	*Address*+*Size* -1 resulted in invalid address

Parameters

BoardId	BoardID of the Rio board on which to read data from eeprom.
Address	Address on the Eeprom to read data from.
Data	Pointer to the data-array, to be used to store the data.
Size	Size in bytes of the data-array to be read.

Remarks Reads data from eeprom.

----------DIRECT HARDWARE ACCESS FUNCTIONS----------
----------NEVER USE WITHOUT CONSULTING ELLIPS----------

----------DIRECT HARDWARE ACCESS FUNCTIONS----------
----------NEVER USE WITHOUT CONSULTING ELLIPS----------

RioEepromWrite

#include <Rio.h>

Function int RioEepromWrite(int *BoardId*, int *Address*, void **Data*, int *Size*);

Return Value RIO_OK successful

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_EEPROM_ADDRESS	an invalid *Address* was given
RIO_INVALID_EEPROM_DATASIZE_FOR_ADDRESS	*Address*+*Size* -1 resulted in invalid address

Parameters

BoardId	BoardID of the Rio board on which to write data to eeprom.
Address	Address on the Eeprom to write data to.
Data	Pointer to the data-array to be written to Eeprom.
Size	Size in bytes of the data-array to be written (max. 128 byte).

Remarks Writes data to eeprom on a Rio board, **NEVER USE WITHOUT CONSULTING ELLIPS.**

----------DIRECT HARDWARE ACCESS FUNCTIONS----------
----------NEVER USE WITHOUT CONSULTING ELLIPS----------

RioExternalInt

#include <Rio.h>

Function int RioExternalInt(int *BoardId*, RIO_GPIO_PIN *GpioPin*,
OVERLAPPED **pOverlapped)*;

Return Value

RIO_OK	successful (in case *pOverlapped*==NULL)
RIO_PENDING	the external interrupt handler was set successfully (in case *pOverlapped*!=NULL) WaitForSingleObject(*pOverlapped*->hEvent,..) must be used to wait for the external interrupt.
RIO_ALREADY_WAITING_FOR_EXT_INT	RioExternalInt already was called previously, WaitForSingleObject(*pOverlapped*->hEvent,..) should be used first to wait for that external interrupt, before calling RioExternalInt again.
RIO_DEVICE_IO_CONTROL	internal error, contact Ellips.
RIO_TIMEOUT	the time-out has expired (in case *pOverlapped*==NULL)
RIO_WAIT_FAILED	the wait failed (in case *pOverlapped*==NULL)

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_GPIO_PIN	an invalid *GpioPin* was given

Parameters

BoardId	BoardID of the Rio board to set the external interrupt handler for.
GpioPin	General-purpose-io pin to set the external interrupt handler for, can be RIO_GPIO_0, RIO_GPIO_1, RIO_GPIO_2 or RIO_GPIO_3. The pin must be set to RIO_GPIO_IRQ_RISE, RIO_GPIO_IRQ_FALL or RIO_GPIO_IRQ_BOTH with RioGpioControl.
pOverlapped	Pointer to an OVERLAPPED struct. The *pOverlapped*->hEvent member is used to signal the interrupt event.

Remarks Sets an external interrupt handler.

- If *pOverlapped* is a valid pointer to an OVERLAPPED struct, RIO_PENDING is returned if the handler was started successfully. The programmer must use WaitForSingleObject with the *pOverlapped*->hEvent member, to wait for the interrupt. *pOverlapped* can (then) be passed to RioGetOverlappedResult to get the return code of the overlapped operation, that indicates if successfully was waited for the external interrupt.
- If NULL is passed for *pOverlapped*, the function will only return after the interrupt has occurred (RIO_OK returned), or the time-out that was set in RioExternalIntTimeOut will have expired (RIO_TIMEOUT returned).

Example 1. In the first example RioExternalInt is called with *pOverlapped* == NULL, in a non multithreaded way. pRiodl is assumed to be a correctly created PRIODL Rio device list of the available Rio boards (see RioCreateRiodl example).

```
char s[80];
int Result;

// setting time-out to 3 seconds
Result = RioExternalIntTimeOut(pRiodl->BoardId[0], RIO_GPIO_0, 3000);
if ( Result != RIO_OK )
{
   sprintf(s,"Changing time-out to other than default, failed. Result: 0x%X", Result);
   MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
}

// in this example, the first general purpose io pin is set to interrupt on rise
Result = RioGpioControl(pRiodl->BoardId[0], RIO_GPIO_IRQ_RISE,
                                RIO_GPIO_IGNORE, RIO_GPIO_IGNORE, RIO_GPIO_IGNORE);
if ( Result != RIO_OK )
{
    sprintf(s, "General purpose pin could not be set.  Result: 0x%X", Result);
    MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
}
else
{
    Result = RioExternalInt(pRiodl->BoardId[0], RIO_GPIO_0, NULL);
    if ( Result != RIO_OK )                                    // will be RIO_TIMEOUT
    {
      sprintf(s, "Waiting for the external interrupt failed. Result: 0x%X", Result);
      MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
    }
    else   { // user can take actions here }
}
```

Example 2. In the second example RioExternalInt is called with *pOverlapped* != NULL, in a multithreaded way. pRiodl is assumed to be a correctly created PRIODL Rio device list of the available Rio boards (see RioCreateRiodl example).

```
char s[80];
int Result, OverlapResult;
BOOL bOverlappedProcStarted = FALSE;
OVERLAPPED Overlapped;
DWORD WFSO_Result;

Overlapped.hEvent  = CreateEvent(NULL,              // pointer to security attributes
                                 TRUE,              // flag for manual-reset event
                                 FALSE,             // flag for initial state
                                 NULL);             // pointer to event-object name

// in this example, the first general purpose io pin is set to interrupt on rise
Result = RioGpioControl(pRiodl->BoardId[0], RIO_GPIO_IRQ_RISE,
                                 RIO_GPIO_IGNORE, RIO_GPIO_IGNORE, RIO_GPIO_IGNORE);
if ( Result != RIO_OK )
{
    sprintf(s, "General purpose pin could not be set.  Result: 0x%X", Result);
    MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
}
else
{
    Result = RioExternalInt(pRiodl->BoardId[0], RIO_GPIO_0, &Overlapped);
    if ( Result != RIO_PENDING )
    {
        sprintf(s, "Setting external interrupt handler failed. Result: 0x%X", Result);
        MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
    }
    else bOverlappedProcStarted = TRUE;
}

.. .. .. ..

// if (bOverlappedProcStarted),waiting for interrupt can be done in a separate thread:
WFSO_Result= WaitForSingleObject(Overlapped.hEvent, 3000);
```

```
if ( WFSO_Result == WAIT_FAILED )
{
    sprintf(s, "WaitForSingleObject(Overlapped.hEvent, 3000) failed");
    MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION | MB_SETFOREGROUND);
}
else
{
    if ( WFSO_Result == WAIT_TIMEOUT )
    {
        sprintf(s, "The time-out has expired");
        MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION | MB_SETFOREGROUND);
    }
    else
    {
        Result = RioGetOverlappedResult(&Overlapped, &OverlapResult, TRUE);
        if ( (Result != RIO_OK) || (OverlapResult !=  RIO_OK) )
        {
           sprintf(s, "Error, Res: 0x%X, OverlapRes: 0x%X", Result, OverlapResult);
           MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION | MB_SETFOREGROUND);
        }
        else  { // user can take actions here }
    }
}

.. .. .. ..


// waiting for the interrupt can be canceled
Result = RioExternalIntCancel(pRiodl->BoardId[0], RIO_GPIO_0);
if ( Result != RIO_OK )
{
    sprintf(s, "RioExternalIntCancel failed. Result: 0x%X", Result);
    MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION | MB_SETFOREGROUND);
}
else
{
       // now the thread where WaitForSingleObject is being done, can now be closed
}
```

RioExternalIntCancel

#include <Rio.h>

Function int RioExternalIntCancel(int *BoardId*, RIO_GPIO_PIN *GpioPin*);

Return Value RIO_OK successful

parameter checking:

RIO_INVALID_BOARD_ID an invalid *BoardId* was given
RIO_INVALID_GPIO_PIN an invalid *GpioPin* was given

Parameters

BoardId BoardID of the Rio board to cancel the external interrupt handler for.
GpioPin General-purpose-io pin to cancel the external interrupt handler for, can be RIO_GPIO_0, RIO_GPIO_1, RIO_GPIO_2 or RIO_GPIO_3.

Remarks Cancels an external interrupt handler, set by RioExternalInt. No event will be returned after the handler is canceled.

Example See RioExternalInt Example 2.

RioExternalIntTimeOut

#include <Rio.h>

Function int RioExternalIntTimeOut(int *BoardId*, RIO_GPIO_PIN *GpioPin*, long *TimeOut*);

Return Value RIO_OK successful

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_GPIO_PIN	an invalid *GpioPin* was given

Parameters

BoardId	BoardID of the Rio board on which to set the external interrupt time-out.
GpioPin	General-purpose-io pin to set the time-out for, can be RIO_GPIO_0, RIO_GPIO_1, RIO_GPIO_2 or RIO_GPIO_3.
TimeOut	The time-out to be set (in msec).

Remarks This function sets the time-out for the external interrupt. Default after initialization before this function is called for the first time is 5000 msec. This time-out will be used in RioExternalInt when called with *pOverlapped* = NULL.

Example See RioExternalInt Example 1.

RioGetBoardType

#include <Rio.h>

Function int RioGetBoardType(int *BoardId*, BOOL **IsBasicVersion*);

Return Value RIO_OK successful

parameter checking:

RIO_INVALID_BOARD_ID an invalid *BoardId* was given

Parameters *BoardId* BoardID of the Rio board to acquire the information from.
**IsBasicVersion* BOOL returning if the board is a Basic Rio or a Full Rio.

Remarks Returns FALSE in **IsBasicVersion* if the Rio Board is a full version of the Rio with two input modules or TRUE if it is a basic version of the Rio with one input module (see 'Hardware Reference' and 'Rio Modules').

Example See RioCreateRiodl and RioSetLed.

RioGetBrightness

#include <Rio.h>

Function int RioGetBrightness(int *BoardId*, RIO_INPUT_MODULE *InputModule*, uchar* *Value*);

Return Value RIO_OK successful

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_INPUT_MODULE	an invalid *InputModule* was given

Parameters

BoardId	BoardID of the Rio board from which to obtain the brightness setting.
InputModule	Input module from which to obtain the brightness setting. Can be either RIO_IM_0 or RIO_IM_1.
**Value*	Brightness value, see RioSetBrightness for range details.

Remarks Gets the brightness value of a module on a Rio board. This function only has meaning when the mode is RIO_COLOR, RIO_BW or RIO_FULL_FRAME. If the mode is RIO_FULL_FRAME, only RIO_IM_0 should be used as value for *InputModule.*

Example The brightness of the first input module on the first Rio board is obtained. pRiodl is assumed to be a correctly created PRIODL Rio device list of the available Rio boards (see RioCreateRiodl example). pModule is assumed to be the input module PRIO_MODULE list (see RioGetInputModule example), reflecting the current input module state of the first Rio board.

```
int Result;
char s[80];
uchar Bright;

if ( (pModule[0].Mode == RIO_COLOR) || ( pModule[0].Mode == RIO_BW) ||
        (pModule[0].Mode == RIO_FULL_FRAME) )      // only then brightness has meaning
{
    Result = RioGetBrightness(pRiodl->BoardId[0], RIO_IM_0, &Bright);
    if ( Result != RIO_OK )
        sprintf(s, "Brightness could not be read.  Result: 0x%X", Result);
    else
        sprintf(s, "Brightness is set to %i. ", Bright);
    MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
}
else
{
    MessageBox(NULL, "Brightness not relevant", NULL, MB_OK | MB_ICONEXCLAMATION );
}
```

RioGetCamera

#include <Rio.h>

Function int RioGetCamera(int *BoardId*, RIO_INPUT_MODULE *InputModule*,
int **Camera*);

Return Value RIO_OK successful

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_INPUT_MODULE	an invalid *InputModule* was given

Parameters

BoardId	BoardID of the Rio board from which to get the current camera number.
InputModule	Input module from which to get the current camera number. Can be either RIO_IM_0 or RIO_IM_1.
**Camera*	Camera connection number currently selected (numbering depending on module mode, see RioSetInputModule and Figure 14).

Remarks Gets the current camera selection for a module on a Rio board.
If the mode is RIO_HQ, RIO_FULL_FRAME or RIO_RGBA, only RIO_IM_0 should be used as value for *InputModule.*

Example The current camera-number for the first input module is obtained. pRiodl is assumed to be a correctly created PRIODL Rio device list of the available Rio boards (see RioCreateRiodl example).

See RioSetBrightness for another example.

```
int CameraNr;
int Result;
char s[80];

// get the currently selected camera of the first input module of the Rio board
// (see 'Video Source Connectors')

Result = RioGetCamera(pRiodl->BoardId[0], RIO_IM_0, &CameraNr);
if ( Result != RIO_OK )
{
    sprintf(s, "Could not get selected camera. Result: 0x%X", Result);
    MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
}
else
{
    sprintf(s, "Camera nr. %i is selected for input module 0.", CameraNr);
    MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
}
```

RioGetChromaKey

#include <Rio.h>

Function int RioGetChromaKey(int *BoardId*,
char **VLowerLimit*, char **VUpperLimit*,
char **ULowerLimit*, char **UUpperLimit*,
BOOL **Enabled*);

Return Value

RIO_OK	successful

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_POINTER	*VLowerLimit*, *VUpperLimit*, *ULowerLimit*, *UUpperLimit* or *Enabled* was NULL

Parameters

BoardId	BoardID of the Rio board from which to obtain the chroma key settings.
**VLowerLimit*	Lower V-value limit of V-range to be used for chroma key, see RioSetChromaKey for range details
**VUpperLimit*	Upper V-value limit of V-range to be used for chroma key, see RioSetChromaKey for range details
**ULowerLimit*	Lower U-value limit of U-range to be used for chroma key, see RioSetChromaKey for range details
**UUpperLimit*	Upper U-value limit of U-range to be used for chroma key, see RioSetChromaKey for range details
**Enabled*	BOOL, specifying whether chroma key functionality is currently enabled or not

Remarks Gets the chroma key settings for the high performance scaler on a Rio board (see section 'Choice of scaler' on page 15). These settings will be used when a module is using the high performance scalar and only have meaning when the mode is RIO_COLOR.

RioGetContrast

#include <Rio.h>

Function int RioGetContrast(int *BoardId*, RIO_INPUT_MODULE *InputModule*, char* *Value*);

Return Value RIO_OK successful

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_INPUT_MODULE	an invalid *InputModule* was given

Parameters

BoardId	BoardID of the Rio board from which to obtain the contrast setting.
InputModule	Input module from which to obtain the contrast setting. Can be either RIO_IM_0 or RIO_IM_1.
**Value*	Contrast value, see RioSetContrast for range details.

Remarks Gets the contrast value of a module on a Rio board. This function only has meaning when the mode is RIO_COLOR, RIO_BW or RIO_FULL_FRAME. If the mode is RIO_FULL_FRAME, only RIO_IM_0 should be used as value for *InputModule*.

Example The contrast of the first input module on the first Rio board is obtained. pRiodl is assumed to be a correctly created PRIODL Rio device list of the available Rio boards (see RioCreateRiodl example). pModule is assumed to be the input module PRIO_MODULE list (see RioGetInputModule example), reflecting the current input module state of the first Rio board.

```
int Result;
char s[80];
char Contrast;

if ( (pModule[0].Mode == RIO_COLOR) || ( pModule[0].Mode == RIO_BW) ||
        (pModule[0].Mode == RIO_FULL_FRAME) )         // only then contrast has meaning
{
    Result = RioGetContrast(pRiodl->BoardId[0], RIO_IM_0, &Contrast);
    if ( Result != RIO_OK )
        sprintf(s, "Contrast could not be read.  Result: 0x%X", Result);
    else
        sprintf(s, "Contrast is set to %i. ", Contrast);
    MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
}
else
{
    MessageBox(NULL, "Contrast not relevant", NULL, MB_OK | MB_ICONEXCLAMATION );
}
```

RioGetHqBw

#include <Rio.h>

Function int RioGetHqBw(int *BoardId*, float **Gain*, float **Offset*);

Return Value RIO_OK successful

parameter checking:

RIO_INVALID_BOARD_ID an invalid *BoardId* was given

Parameters

BoardId	BoardID of the Rio board of which to get the information.
**Gain*	Current gain value (Range: see RioSetHqBw).
**Offset*	Current offset value (Range: see RioSetHqBw).

Remarks Gets the current Black and White High Quality Gain and Offset of a Rio board. It is only applicable for module mode RIO_HQ. The function is used to get these Black and White High Quality properties, even if not currently in RIO_HQ mode.

Example The Black and White High Quality settings are read from the first Rio board. pRiodl is assumed to be a correctly created PRIODL Rio device list of the available Rio boards (see RioCreateRiodl example).

```
int Result;
char s[80];

float gain, offset;

Result = RioGetHqBw(pRiodl->BoardId[0], &gain, &offset);
if ( Result != RIO_OK )
    sprintf(s, "High quality B&W settings could not be read.  Result: 0x%X", Result);
else
    sprintf(s, "HQ B&W Gain : %.4f \nHQ B&W Offset : %.4f", gain, offset );
MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
```

RioGetHue

#include <Rio.h>

Function int RioGetHue(int *BoardId*, RIO_INPUT_MODULE *InputModule*, char* *Value*);

Return Value RIO_OK successful

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_INPUT_MODULE	an invalid *InputModule* was given

Parameters

BoardId	BoardID of the Rio board from which to obtain the hue setting.
InputModule	Input module from which to obtain the hue setting. Can be either RIO_IM_0 or RIO_IM_1.
**Value*	Hue value, see RioSetHue for range details.

Remarks Gets the hue value of an input-module on a Rio board.
This function only has meaning when the module is in RIO_COLOR_MODE with the RIO_COLOR_VIDEO_STANDARD set to RIO_NTSC.

Example The hue of the first input module on the first Rio board is obtained. pRiodl is assumed to be a correctly created PRIODL Rio device list of the available Rio boards (see RioCreateRiodl example). pModule is assumed to be the input module PRIO_MODULE list (see RioGetInputModule example), reflecting the current input module state of the first Rio board.

```
int Result;
char s[80];
char Hue;

if ( (pModule[0].Mode == RIO_COLOR) &&
     (pModule[0].Def.Color.VideoStandard == RIO_NTSC ) )  // only then hue has meaning
{
    Result = RioGetHue(pRiodl->BoardId[0], RIO_IM_0, &Hue);
    if ( Result != RIO_OK )
        sprintf(s, "Hue could not be read.  Result: 0x%X", Result);
    else
        sprintf(s, "Hue is set to %i. ", Hue);
    MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
}
else
{
    MessageBox(NULL, "Hue not relevant", NULL, MB_OK | MB_ICONEXCLAMATION );
}
```

RioGetInputGain

#include <Rio.h>

Function int RioGetInputGain(int *BoardId*, RIO_INPUT_MODULE *InputModule*, int *Input*, uchar **Value*, RIO_ON_OFF_MODE **AutoGainMode*);

Return Value

RIO_OK	successful

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_INPUT_MODULE	an invalid *InputModule* was given
RIO_INVALID_INPUT_FOR_INPUT_MODULE	*Input* was an invalid number for input module

Parameters

BoardId	BoardID of the Rio board from which to obtain the input gain setting.
InputModule	Input module from which to obtain the input gain setting. Can be either RIO_IM_0 or RIO_IM_1.
Input	Input number of input of which the input gain has to be obtained. Range 0..5 when module is in CVBS mode, 0..2 when in Y/C mode (see Figure 14).
**Value*	Input gain value, see RioSetInputGain for range details.
**AutoGainMode*	Indicating input gain is set in Auto Gain mode, either RIO_ON or RIO_OFF.

Remarks Gets the input gain settings from the input of a module on a Rio board.
This function has no meaning when the mode is RIO_HQ. If the mode is RIO_RGBA or RIO_FULL_FRAME, only RIO_IM_0 should be used as value for *InputModule* .When *AutoGainMode* is set to be RIO_ON, *Value* will not have any meaning.

Example The input gain of the current first input module on the first Rio board is obtained. pRiodl is assumed to be a correctly created PRIODL Rio device list of the available Rio (see RioCreateRiodl example). pModule is assumed to be the input module PRIO_MODULE list (see RioGetInputModule example), reflecting the current input module state of the first Rio board.

```
int CurCam;
char s[80];
char gain;
BOOL bOK= TRUE;
RIO_ON_OFF_MODE AGain;


if ( pModule[0].Mode == RIO_HQ )                        // input gain has no meaning
    MessageBox(NULL, "Input Gain not relevant", NULL, MB_OK | MB_ICONEXCLAMATION );
else
{
    // get current camera info
    bOK = (bOK && (RioGetCamera(pRiodl->BoardId[0], RIO_IM_0, &CurCam) == RIO_OK));
    // get input gain info
    bOK = ( bOK &&
      (RioGetInputGain(pRiodl->BoardId[0], RIO_IM_0, CurCam, gain, &AGain)== RIO_OK));

    if ( !bOK )
        MessageBox(NULL, "Gain could not be read", NULL, MB_OK | MB_ICONEXCLAMATION );
    else
    {
        if ( AGain == RIO_ON )
            MessageBox(NULL,"Autogain set to on", NULL, MB_OK | MB_ICONEXCLAMATION );
        else
        {
            sprintf(s, "Input %i has gain set to %i", CurCam, gain);
            MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
        }
    }
}
```

RioGetInputModule

#include <Rio.h>

Function int RioGetInputModule(int *BoardId*, uchar **NrModules*, PRIO_MODULE *Module*);

Return Value RIO_OK successful

parameter checking:

RIO_INVALID_BOARD_ID an invalid *BoardId* was given

Parameters

BoardId	BoardID of the Rio board to get the information from.
**NrModules*	The number of modules present on the Rio board.
Module	Pointer to an array of RIO_MODULE structs, containing the information about the input modules on the Rio board. The RIO_MODULE struct is explained in more detail in the 'Structures' section of this manual. NULL may also be used as entry (see remarks).

Remarks Reads the input module state(s) of a Rio board. When the function is called with NULL as argument for *Module*, the only thing returned will be **NrModules*. Therefore the function can initially be used to simply obtain the number of modules, so that the programmer can allocate the right size for the RIO_MODULE array to be passed to Module to obtain module information in a second call to RioGetInputModule.

See RIO_MODULE_MODE enum at page 21, RIO_MODULE struct at page 29 and section 'Rio Modules' for more information about Rio Modules.

Example In the next example, the set of Rio modules is detected, and for every module in RIO_COLOR mode, a messagebox is generated. pRiodl is assumed to be a correctly created PRIODL Rio device list of the available Rio boards (see RioCreateRiodl example).

```
char s[80];
int Result;
uchar NumofModules;
PRIO_MODULE  pModule;

// obtain the number of input modules currently set on the Rio board
Result =  RioGetInputModule(pRiodl->BoardId[0], &NumofModules, NULL);
if (( Result != RIO_OK ) || ( NumofModules==0 ) || ( NumofModules>2 ))
{
    sprintf(s, "Could not get correct number of modules.  Result: 0x%X", Result);
    MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
}
else
{
    pModule = new RIO_MODULE[NumofModules];  // allocate space for list of Rio modules
    // fill list with information about the modules
    Result =  RioGetInputModule(pRiodl->BoardId[0],  &NumofModules, pModule);
    if (Result != RIO_OK)
    {
        sprintf(s, "Input module(s) info could not be read.  Result: 0x%X", Result);
        MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
    }
    else
    {
        for (int  i =0; i<Numofmodules; i++)
        {
            if (pModule[i].Mode == RIO_COLOR)
            {
                // a messagebox appears for every module that exists in RIO_COLOR mode
                sprintf(s, "Input module %i is a Rio module in RIO_COLOR  mode.", i);
                MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
            }
        }
    }
    delete[] pModule;
}
```

RioGetOverlappedResult

#include <Rio.h>

Function int RioGetOverlappedResult(OVERLAPPED **pOverlapped*,
int **ReturnCode*, BOOL *bWait*);

Return Value

RIO_OK	successful
RIO_PENDING	function failed, because overlapped operation is still pending
RIO_WAIT_FAILED	internal error, call GetLastError to get more information

parameter checking:

RIO_INVALID_OVERLAPPED	*pOverlapped*==NULL is NOT ALLOWED for this function.
RIO_INVALID_POINTER	*ReturnCode* was NULL

Parameters

pOverlapped	Pointer to an OVERLAPPED struct, that was specified when the overlapped operation was started.
**ReturnCode*	The return code returned by the overlapped operation, can be (among others)

RIO_OK	success
RIO_VIDEO_OVERFLOW	some pixels in the image were not obtained correctly

bWait	Specifies whether the function should wait for the pending overlapped operation to be completed.

Remarks The function returns the result of an overlapped operation, that was started by RioExternalInt, RioCapture, RioStreamGetFilledBuffer and RioTriggerCapture.
The result of the overlapped operation is returned in **ReturnCode*.
If *bWait*==TRUE, the function does not return until the pending overlapped operation has been completed. If *bWait*==FALSE and the operation is still pending, the function returns RIO_PENDING.

Example See RioExternalInt Example 2, RioCapture Example 2 and Example 3, RioStreamGetFilledBuffer Example and RioTriggerCapture Example 2.

RioGetSaturation

#include <Rio.h>

Function int RioGetSaturation(int *BoardId*, RIO_INPUT_MODULE *InputModule*, char* *Value*);

Return Value RIO_OK successful

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_INPUT_MODULE	an invalid *InputModule* was given

Parameters

BoardId	BoardID of the Rio board from which to obtain the saturation setting.
InputModule	Input module from which to obtain the saturation setting. Can be either RIO_IM_0 or RIO_IM_1.
**Value*	Saturation value, see RioSetSaturation for range details.

Remarks Gets the saturation value of a module on a Rio board. This function only has meaning when the mode is RIO_COLOR.

Example The saturation of the first input module on the first Rio board is obtained. pRiodl is assumed to be a correctly created PRIODL Rio device list of the available Rio boards (see RioCreateRiodl example). pModule is assumed to be the input module PRIO_MODULE list (see RioGetInputModule example), reflecting the current input module state of the first Rio board.

```
int Result;
char s[80];
char Sat;

if (pModule[0].Mode == RIO_COLOR)                  // only then saturation has meaning
{
    Result = RioGetSaturation(pRiodl->BoardId[0], RIO_IM_0, &Sat);
    if ( Result != RIO_OK )
        sprintf(s, "Saturation could not be read.  Result: 0x%X", Result);
    else
        sprintf(s, "Saturation is set to %i. ", Sat);
    MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
}
else
{
    MessageBox(NULL, "Saturation not relevant", NULL, MB_OK | MB_ICONEXCLAMATION );
}
```

RioGetVideoStatus

#include <Rio.h>

Function int RioGetVideoStatus(int *BoardId*, RIO_INPUT_MODULE *InputModule*, uchar* *Value*);

Return Value RIO_OK successful

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_INPUT_MODULE	an invalid *InputModule* was given

Parameters

BoardId	BoardID of the Rio board from which to obtain the contrast setting.
InputModule	Input module from which to obtain the contrast setting. Can be either RIO_IM_0 or RIO_IM_1.
**Value*	Video status value, see riodef.h for bit definitions.

Remarks Gets the video status of a module on a Rio board. If the RIO_VS_HLCK bit is set, a video signal is not present on the input selected for the input module. After selecting a new input for the input module, some time (several fields) should pass before video status is valid.

Example The status of the first input module on the first Rio board is obtained. pRiodl is assumed to be a correctly created PRIODL Rio device list of the available Rio boards (see RioCreateRiodl example). pModule is assumed to be the input module PRIO_MODULE list (see RioGetInputModule example), reflecting the current input module state of the first Rio board.

```
int Result;
char s[80];
uchar VidStat;

Result = RioGetVideoStatus(pRiodl->BoardId[0], RIO_IM_0, &VidStat);
if ( Result != RIO_OK )
    sprintf(s, "Videostatus could not be read.  Result: 0x%X", Result);
else
    sprintf(s, "Videostatus : 0x%x", VidStat);
MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
```

RioGpioControl

#include <Rio.h>

Function int RioGpioControl(int *BoardId*,
RIO_GPIO_MODE *Gpio0*,
RIO_GPIO_MODE *Gpio1*,
RIO_GPIO_MODE *Gpio2*,
RIO_GPIO_MODE *Gpio3*);

Return Value RIO_OK successful

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_GPIO_MODE	an invalid value was given for *Gpio0, Gpio1, Gpio2* or *Gpio3*

Parameters *BoardId* BoardID of the Rio board of which the general-purpose-io pins are to be set.

Gpio0, Gpio1, Gpio2 and *Gpio3*

Modes for the general-purpose-io pins 0, 1, 2 and 3 (see 'General Purpuse IO pins').
Each one can be one of the following modes:

RIO_GPIO_IGNORE	pin mode will not be changed
RIO_GPIO_INPUT	pin is used as input
RIO_GPIO_OUTPUT	pin is used as output
RIO_GPIO_IRQ_RISE	rise in signal on pin will generate an interrupt
RIO_GPIO_IRQ_FALL	fall in signal on pin will generate an interrupt
RIO_GPIO_IRQ_BOTH	rise and fall in signal on pin will generate an interrupt

Remarks Sets each one of the general-purpose-io pins into the desired mode. Before the function is used for the first time after initialization, the pins will be in RIO_GPIO_IGNORE mode.

Example In the example, all four pins of the first Rio board are set to be ignored (which also is the default situation). pRiodl is assumed to be a correctly created PRIODL Rio device list of the available Rio boards (see RioCreateRiodl example).

See RioGpioWrite for another example.

```
int Result;
char s[80];

// setting all pins to ignore
Result = RioGpioControl(pRiodl->BoardId[0],
               RIO_GPIO_IGNORE, RIO_GPIO_IGNORE, RIO_GPIO_IGNORE, RIO_GPIO_IGNORE);

if ( Result != RIO_OK )
{
    sprintf(s, "Pins could not be set. Result: 0x%X", Result);
    MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
}
```

RioGpioRead

#include <Rio.h>

Function int RioGpioRead(int *BoardId*, uchar **Data*);

Return Value RIO_OK successful

parameter checking:

RIO_INVALID_BOARD_ID an invalid *BoardId* was given

Parameters *BoardId* BoardID of the Rio board to read the general-purpose-io pins from.
**Data* The total general-purpose-io pin state as a 4bit number (0x00 .. 0x0F).

Remarks Reads the total general-purpose-io pin state as a 4bit number.

Example Reads the state of the pins of the first Rio board, and stores the result (high or low) in a boolean array. pRiodl is assumed to be a correctly created PRIODL Rio device list of the available Rio boards (see RioCreateRiodl example).

See RioGpioWrite for another example.

```
BOOL PinHigh[4];
uchar PinState;
int Result;
char s[80];

// reading pin state from first Rio board:
Result = RioGpioRead(pRiodl->BoardId[0],&PinState);
if ( Result != RIO_OK )
{
    sprintf(s, "Could not read pin state. Result: 0x%X", Result);
    MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
}
else
{
       PinHigh[0] = ( (PinState & 0x01) != 0 );          // first bit for pin 0
       PinHigh[1] = ( (PinState & 0x02) != 0 );          // second bit for pin 1
       PinHigh[2] = ( (PinState & 0x04) != 0 );          // third bit for pin 2
       PinHigh[3] = ( (PinState & 0x08) != 0 );          // fourth bit for pin 3
}
```

RioGpioWrite

#include <Rio.h>

Function int RioGpioWrite(int *BoardId*, uchar *Data*);

Return Value RIO_OK successful

parameter checking:

RIO_INVALID_BOARD_ID an invalid *BoardId* was given

Parameters *BoardId* BoardID of the Rio board on which to write to the general purpose io pins.

Data The total general-purpose-io pin state to be set, a 4bit number (0x00.. 0x0F).

Remarks Writes the total general-purpose-io pin state as a 4bit number. Only those pins that are set to RIO_GPIO_OUTPUT with RioGpioControl will be set.

Example Writes the state of all pins of the first Rio board to high.
pRiodl is assumed to be a correctly created PRIODL Rio device list of the available Rio boards (see RioCreateRiodl example).

```
uchar PinState, NewPinState;
int Result;
char s[80];

// first set all pins to output, otherwise the pinstate will not be set.
Result = RioGpioControl(pRiodl->BoardId[0],
                RIO_GPIO_OUTPUT, RIO_GPIO_OUTPUT, RIO_GPIO_OUTPUT, RIO_GPIO_OUTPUT);
if ( Result != RIO_OK )
{
    sprintf(s, "Pins not set for output. Result: 0x%X", Result);
    MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
}
else
{
    PinState = 0x0F;                                                    // all pins high
    Result = RioGpioWrite(pRiodl->BoardId[0], PinState);
    if ( Result != RIO_OK )
    {
        sprintf(s, "Could not write pin state. Result: 0x%X", Result);
        MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
    }
    else
    {
        Result = RioGpioRead(pRiodl->BoardId[0], &NewPinState);
        if ( Result != RIO_OK )
        {
            sprintf(s, "Could not read pin state. Result: 0x%X", Result);
            MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
        }
        else
        {
            if (NewPinState!=PinState)
            {
                sprintf(s, "Pin state was not set correctly. Result: 0x%X", Result);
                MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
            }
        }
    }
}
```

----------DIRECT HARDWARE ACCESS FUNCTIONS----------
----------NEVER USE WITHOUT CONSULTING ELLIPS----------

RioI2cRead

#include <Rio.h>

Function int RioI2cRead(int *BoardId*, RIO_INPUT_MODULE *InputModule*,
uchar *Reg*, uchar **Value*);

Return Value

RIO_OK	successful

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_INPUT_MODULE	an invalid *InputModule* was given
RIO_INVALID_I2C_REG	an invalid address was used for *Reg*

Parameters

BoardId	BoardID of the Rio board from which to read data.
InputModule	Input module from which to read data.
Reg	Register on input module to read data from.
**Value*	One byte value to be read from the register.

Remarks Reads data from an input module on a Rio board.

----------DIRECT HARDWARE ACCESS FUNCTIONS----------
----------NEVER USE WITHOUT CONSULTING ELLIPS----------

----------DIRECT HARDWARE ACCESS FUNCTIONS----------
----------NEVER USE WITHOUT CONSULTING ELLIPS----------

RioI2cWrite

#include <Rio.h>

Function int RioI2cWrite(int *BoardId*, RIO_INPUT_MODULE *InputModule*,
uchar *Reg*, uchar *Value*);

Return Value RIO_OK successful

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_INPUT_MODULE	an invalid *InputModule* was given
RIO_INVALID_I2C_REG	an invalid address was used for *Reg*

Parameters

BoardId	BoardID of the Rio board to which to write data.
InputModule	Input module for which to write the data.
Reg	Register on input module to write data to.
Value	One byte value to be written to the register.

Remarks Writes data to an input module on a Rio board, **NEVER USE WITHOUT CONSULTING ELLIPS.**

----------DIRECT HARDWARE ACCESS FUNCTIONS----------
----------NEVER USE WITHOUT CONSULTING ELLIPS----------

RioIndexSetLed

#include <Rio.h>

Function int RioIndexSetLed(int *BoardIndex*, RIO_ON_OFF_MODE *LedState*);

Return Value RIO_OK successful

parameter checking:

RIO_INVALID_BOARD_ID	(*BoardIndex*>= number of boards) or (*BoardIndex*< 0)
RIO_INVALID_LED_STATE	an invalid *LedState* was given

Parameters

BoardIndex	Zero based index of the Rio board on which the led is to be set. Except for RioIndexSetLed, all other functions use the BoardID to identify a Rio board.
LedState	State to set the board led in, can be either RIO_ON or RIO_OFF.

Remarks Sets the led of a Rio board to on or off. RioSetLed does the same, except the *BoardID* is used to identify a Rio board, instead of *BoardIndex.*

Example Functions that set all leds of all Rio boards to on or off.
pRiodl is assumed to be a correctly created global PRIODL Rio device list of the available Rio boards (see RioCreateRiodl example).

```
void OnSetAllLedsOn()
{
    int Result;
    char s[80];

    for (int  i = 0; i < pRiodl->NrBoards; i++ )
    {
        // setting led of Rio with index i from list to on
        Result = RioIndexSetLed(i, RIO_ON);
        if ( Result != RIO_OK )
        {
          sprintf(s, "Unable to set led of Rio nr. %i to on. Result: 0x%X", i, Result);
          MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
        }
    }
}

void OnSetAllLedsOff()
{
    int Result;
    char s[80];

    for (int  i = 0; i < pRiodl->NrBoards; i++ )
    {
        // setting led of Rio with index i from list to off
        Result = RioIndexSetLed(i, RIO_OFF);
        if ( Result != RIO_OK )
        {
          sprintf(s, "Unable to set led of Rio nr. %i to off.Result: 0x%X", i, Result);
          MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
        }
    }
}
```

RioInitialize

#include <Rio.h>

Function int RioInitialize(int *BoardId*);

Return Value

RIO_OK	successful
RIO_ERR_INIT	initialization failed

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given

Parameters *BoardId* BoardID of the Rio board to be re-initialized.

Remarks Re-initializes a Rio board to its default state, in the same way this was done for all Rio boards during the startup of the driver. A call to this function will take about 80 milliseconds.

Example The first Rio board out of the Rio device list is re-initialized.
pRiodl is assumed to be a correctly created PRIODL Rio device list of the available Rio boards (see RioCreateRiodl example).

```
int Result;
char s[80];

Result = RioInitialize(pRiodl->BoardId[0]);
if ( Result != RIO_OK )
{
    sprintf(s, "Unable to re-initialize first Rio board. Result: 0x%X", Result);
    MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
}
```

RioOpen

#include <Rio.h>

Function int RioOpen(void);

Return Value

RIO_OK	successful
RIO_DRIVER_NOT_LOADED	cannot find Rio.vxd (Windows 95) or Rio.sys (Windows NT/2000)
RIO_MEM_ALLOC	not enough memory
RIO_UNABLE_TO_CREATE_FILE	internal error, contact Ellips
RIO_UNABLE_TO_MAP_FILE	internal error, contact Ellips
RIO_INVALID_EVENT	internal error, CreateEvent failed, contact Ellips
RIO_INSUFFICIENT_MEMORY	not enough memory

Remarks Should always be the first function to be called. Initializes the available Rio boards.

Example

```
int Result;
char s[80];

Result = RioOpen();           // initializing the available Rio boards
if ( Result != RIO_OK )
{
    sprintf(s, "Unable to open Rio. Result : 0x%X", Result);
    MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
}
```

RioPostProcess

#include <Rio.h>

Function int RioPostProcess(int *BoardId*, PRIO_MODULE *Module*,
BOOL *TopDown*, RECT **DestRect*,
int *Pitch*, HANDLE *ImageBufferHandle*);

Return Value

RIO_OK	successful
RIO_UNABLE_TO_POST_PROCESS	postprocessing failed

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_POINTER	*Module* was NULL
RIO_INVALID_MODULE_MODE	an invalid *Module->Mode* was given
RIO_INVALID_OUTPUT_FORMAT	for *Module->Def.* relevant modules, an invalid *OutputFormat* was given
RIO_INVALID_DEST_RECT	*DestRect* was NULL or **DestRect* was not normalized
RIO_INVALID_HANDLE	*ImageBufferHandle* was NULL

Parameters

BoardId	BoardID of the Rio board.
Topdown	BOOL, indicating topdown line image-building was set (TRUE) or not during capture of the image.
**DestRect*	The destination rectangle (bottom and right not included), that was used during capture of the image.
Pitch	The line-pitch in bytes that was used during capture of the image.
ImageBufferHandle	The handle indicating the allocated imagebuffer memory, containing the image we want to postprocess.

Remarks This function can be used by the programmer to postprocess images, that were initially captured without postprocessing, but require postprocessing on the special output format. (Useful for RIO_STEREO_LOCKED, RIO_S_FULL_FRAME or RIO_RGBA mode). Apart from a handle to the imagebuffer containing the image, the programmer has to give the parameters, that were used during capture of the image.

Normally, it will make more sense to immediately use postprocessing during capture. But in some cases this function may come in handy. For instance when the capture rate needs to be as high as possible for some period of time, streaming captured buffers could be stored to disk without the performance loss due to postprocessing. At a more convenient time, postprocessing could then be performed using this function.

----------DIRECT HARDWARE ACCESS FUNCTIONS----------
----------NEVER USE WITHOUT CONSULTING ELLIPS----------

RioRegisterRead

#include <Rio.h>

Function int RioRegisterRead(int *BoardId*, int *Address*, int **Value*);

Return Value RIO_OK successful

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_ERR_PARAMS	an invalid address was used for *Address*

Parameters

BoardId	BoardID of the Rio board from which to read data.
Address	Address to read data from.
**Value*	32 bits value to be read from the register.

Remarks Reads data from the SAA7146 registers on the Rio board.

----------DIRECT HARDWARE ACCESS FUNCTIONS----------
----------NEVER USE WITHOUT CONSULTING ELLIPS----------

----------DIRECT HARDWARE ACCESS FUNCTIONS----------
----------NEVER USE WITHOUT CONSULTING ELLIPS----------

RioRegisterWrite

#include <Rio.h>

Function int RioRegisterWrite(int *BoardId*, int *Address*, int *Value*);

Return Value RIO_OK successful

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_ ERR_PARAMS	an invalid address was used for *Address*

Parameters

BoardId	BoardID of the Rio board to which to write data.
Address	Address to write data to.
Value	32 bits value to be written to the register.

Remarks Writes data to the SAA7146 registers on the Rio board, **NEVER USE WITHOUT CONSULTING ELLIPS.**

----------DIRECT HARDWARE ACCESS FUNCTIONS----------
----------NEVER USE WITHOUT CONSULTING ELLIPS----------

RioScatterLock

#include <Rio.h>

Function int RioScatterLock(void **ImageDstPtr*, long *ImageSize*,
HANDLE **ImageBufferHandle*);

Return Value

RIO_OK	successful
RIO_LOCK_FAILED	RioScatterLock failed

Parameters

ImageDstPtr	Pointer to the memory allocated by the programmer, to be used as buffer to store the acquired image.
ImageSize	Memory-size in bytes of the memory allocated by the programmer to be used as buffer to store the acquired image.
**ImageBufferHandle*	The handle indicating the allocated imagebuffer memory. Instead of the memory pointer, this handle has to be used in all capture functions to store the image.

Remarks Uses the image memory pointer given by the programmer (after malloc) to return an imagebuffer handle, that is to be used in all capture functions to store the resulting image. The memory will be locked for DMA transfer, making sure Windows cannot swap the memory to disk.

Example In the next example, memory is allocated for the case RIO_RGB24 output format is desired and the memory needs be used to create a device independent bitmap (DIB), so Dword alignment is needed. After allocating the memory, RioScatterLock is used to lock the memory for DMA transfer.

```
int Result;
char s[80];

int XRes= ..;                 // the horizontal number of pixels of the memory map
int YRes= ..;                 // the vertical number of pixels of the memory map

// calculate the pitch (and round up to nearest multiple of 4 for Dword alignment
int Pitch = ((XRes * 3) + 3) & ~3;           // 3 byte per pixel for RIO_RGB24

int DataSize = Pitch * YRes;                 // total datasize required for storage

// allocate bitmap memory (3 bytes more, keeping in mind the next step):
BYTE* pOrgDibData = new BYTE[DataSize + 3];

// use the bitmap memory, starting from the first Dword aligned address:
BYTE* pDibData = (BYTE *) (((UINT) pOrgDibData + 3) & ~3);

// lock the imagebuffer for DMA
HANDLE  ImageBufferHandle;
Result = RioScatterLock(pDibData, DataSize, &ImageBufferHandle);
if ( Result != RIO_OK )
{
    sprintf(s, "Scatter Lock failed. Result: 0x%X", Result);
    MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
}

// if RioScatterLock did not fail, the programmer can now use ImageBufferHandle to
// manipulate the memory with the software library capture functions and visualize.
.
.
.
// after all grabbing and visualization is done, clean up
// first unlock the imagebuffer
```

```
Result = RioScatterUnlock(ImageBufferHandle);
if ( Result != RIO_OK )
{
    sprintf(s, "Scatter Unlock failed. Result: 0x%X", Result);
    MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
}
delete[] pOrgDibData;                        // free the bitmap memory buffer
```

RioScatterUnlock

#include <Rio.h>

Function int RioScatterUnlock(HANDLE *ImageBufferHandle*);

Return Value

RIO_OK	successful
RIO_UNLOCK_FAILED	RioScatterUnlock failed

Parameters *ImageBufferHandle* The handle indicating the allocated imagebuffer memory.

Remarks Unlocks the previously locked image memory buffer. After this function has been used, the programmer can free the memory, using the original pointer address that was used previously in RioScatterLock.

Example See RioScatterLock.

RioSelectCamera

#include <Rio.h>

Function int RioSelectCamera(int *BoardId*, RIO_INPUT_MODULE *InputModule*, int *Camera*);

Return Value

RIO_OK	successful

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_INPUT_MODULE	an invalid *InputModule* was given
RIO_INVALID_CAMERA_FOR_INPUT_MODULE	*Camera* was an invalid number for input module

Parameters

BoardId	BoardID of the Rio board on which to select a camera.
InputModule	Input module for which to select a camera. Can be either RIO_IM_0 or RIO_IM_1.
Camera	Camera connection number to be selected (numbering depending on module mode, see RioSetInputModule and Figure 14).

Remarks Selects a camera for a module on a Rio board. If the mode is RIO_HQ, RIO_FULL_FRAME or RIO_RGBA, only RIO_IM_0 should be used as value for *InputModule.*

Example See RioSetInputModule.

RioSetBoardId

#include <Rio.h>

Function int RioSetBoardId(int *BoardIndex*, int *BoardId*);

Return Value

RIO_OK	successful
RIO_BOARD_ID_WRITE_ERROR	internal error, contact Ellips
parameter checking:	
RIO_INVALID_BOARD_ID	(*BoardIndex*>= number of boards) or (*BoardIndex*< 0)

Parameters

BoardIndex	Zero based index of the board on which the BoardID is to be set.
BoardId	BoardID to be given to the selected Rio board.

Remarks Maps a BoardID to the board with index *BoardIndex* Be careful not to give the same BoardID to more than one Rio board. This is not checked by the function. The BoardID is used to identify the Rio board to which functions have to be applied to. The Ellips Rio Control Panel Applet basically uses this function to set BoardIDs.

Example In this example, the BoardIDs of the available Rio boards are set to the same number as their place (index) in the device list.

```
void OnSetRioBoardIDs()
{
    int Result;
    char s[80];
    // allocating memory for Rio device list
    PRIODL pRiodl = (PRIODL) new BYTE[RioSizeOfRiodl()];

    Result = RioCreateRiodl(pRiodl);         // device list is filled with information
    if ( Result != RIO_OK )
    {
        sprintf(s, "Unable to create device list. Result: 0x%X", Result);
        MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
    }
    else
    {
        for (int  i = 0; i < pRiodl->NrBoards; i++ )
        {
            RioSetBoardId(i, i);                    // setting boardid equal to index
        }

        delete[] ((BYTE*)pRiodl);
        // re-allocating memory for Rio device list
        pRiodl = (PRIODL) new BYTE[RioSizeOfRiodl()];

        Result = RioCreateRiodl(pRiodl);   // device list is re-filled with information
```

```
            if ( Result != RIO_OK )
            {
                sprintf(s, "Unable to create new device list. Result: 0x%X", Result);
                MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
            }
            else
            {
                for (int i = 0; i < pRiodl->NrBoards; i++ )
                {
                    if ( i != pRiodl->BoardId[i] )
                    {
                        sprintf(s, "ID of Rio board nr %i was not correctly set", i);
                        MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
                    }
               }
            }
            delete[] ((BYTE*)pRiodl);
        }
}
```

RioSetBrightness

#include <Rio.h>

Function int RioSetBrightness(int *BoardId*, RIO_INPUT_MODULE *InputModule*, uchar *Value*);

Return Value RIO_OK successful

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_INPUT_MODULE	an invalid *InputModule* was given

Parameters

BoardId	BoardID of the Rio board on which to set the brightness.
InputModule	Input module for which to set the brightness. Can be either RIO_IM_0 or RIO_IM_1.
Value	Brightness value to set, any number ranging from 0 .. 255. Scale: 0 = dark, 139 = CCIR level (default) , 255 = bright.

Remarks Sets the brightness for a module on a Rio board. This function only has effect when the mode is RIO_COLOR, RIO_BW or RIO_FULL_FRAME. If the mode is RIO_FULL_FRAME, only RIO_IM_0 should be used as value for *InputModule*.

Example In this example, default values are set for all input modules out of an input module list, which is assumed to reflect the current input module state of a Rio board.
pRiodl is assumed to be a correctly created global PRIODL Rio device list of the available Rio boards (see RioCreateRiodl example). pModule is assumed to be the input module PRIO_MODULE list (see RioGetInputModule example), reflecting the current input module state of the first Rio board, NumofModules (see RioGetInputModule example) is assumed to be the current number of available input modules on the first Rio board.

```
BOOL SetInputModulesToDefault(PRIO_MODULE pModule, uchar NumofModules)
{
    RIO_INPUT_MODULE I_Mod;
    int CurCamera;
    BOOL bOK = TRUE;

    for (int i = 0; i<NumofModules, i++)
    {
        // determine which input module
        // (will be RIO_IM_0 if (i==0) or RIO_IM_1 if (i==1), but this is more robust)
        switch(pModule[i].Mode)
        {
            case RIO_COLOR :
                I_Mod= pModule[i].Def.Color.InputModule;
                break;
            case RIO_BW :
                I_Mod= pModule[i].Def.Bw.InputModule;
                break;
            case RIO_HQ :
                I_Mod= RIO_IM_0;
                break;
            case RIO_STEREO_LOCKED :
                I_Mod= pModule[i].Def.Sl.InputModule;
```

```
                break;
            case RIO_S_FULL_FRAME :
                I_Mod= pModule[i].Def.Sff.InputModule;
                break;
            case RIO_FULL_FRAME :
                I_Mod= RIO_IM_0;
                break;
            case RIO_RGBA :
                I_Mod= RIO_IM_0;
                break;
            default :                                   // will never be so
                I_Mod= RIO_IM_0;
                break;
        }

        // determine current camera for input module
        bOK= ( bOK && (RioGetCamera(pRiodl->BoardId[0], I_Mod, &CurCamera) == RIO_OK));

        // now set the defaults for the module
        switch(pModule[i].Mode)
        {
            case RIO_COLOR :
              // only in case RIO_COLOR
              bOK= (bOK && (RioSetSaturation(pRiodl->BoardId[0], I_Mod, 64) ==RIO_OK));
              if ( pModule[i].Def.Color.VideoStandard == RIO_NTSC )
                    bOk = (bOK && (RioSetHue(pRiodl->BoardId[0], I_Mod, 0) ==RIO_OK));

            case RIO_BW :
            case RIO_FULL_FRAME :
              // only in case RIO_COLOR, RIO_BW or RIO_FULL_FRAME
              bOK= (bOK && (RioSetBrightness(pRiodl->BoardId[0],I_Mod,139) == RIO_OK));
              bOK= (bOK && (RioSetContrast(pRiodl->BoardId[0], I_Mod, 70) == RIO_OK));

            case RIO_STEREO_LOCKED :
            case RIO_S_FULL_FRAME :
            case RIO_RGBA :
              // in all modes except RIO_HQ
              bOK = (bOK &&
                (RioSetInputGain(pRiodl->BoardId[0],I_Mod,Camera,0,IO_OFF) == RIO_OK));
              break;

            case RIO_HQ :
              // only in case RIO_HQ
              bOK= (bOK && (RioSetHqBw(pRiodl->BoardId[0], 1.0f, 0.0f) == RIO_OK));
              break;

            default :
              break;
        } // end switch
    } // end for

    if (!bOK)
    {
        MessageBox(NULL, "Defaults not set correctly.",
                                    NULL, MB_OK | MB_ICONEXCLAMATION );
    }
    return bOK;
}
```

RioSetChromaKey

#include <Rio.h>

Function int RioSetChromaKey(int *BoardId*,
char *VLowerLimit*, char *VUpperLimit*,
char *ULowerLimit*, char *UUpperLimit*,
BOOL *Enable*);

Return Value RIO_OK successful

parameter checking:

RIO_INVALID_BOARD_ID an invalid *BoardId* was given

Parameters

BoardId	BoardID of the Rio board from which to obtain the chroma key settings.
VLowerLimit	Lower V-value limit of V-range to be used for chroma key, any number ranging from -128 .. 127, but lower than *VUpperLimit.*
VUpperLimit	Upper V-value limit of V-range to be used for chroma key, any number ranging from -128 .. 127, but higher than *VLowerLimit.*
ULowerLimit	Lower U-value limit of U-range to be used for chroma key, any number ranging from -128 .. 127, but lower than *UUpperLimit.*
UUpperLimit	Upper U-value limit of U-range to be used for chroma key, any number ranging from -128 .. 127, but higher than *ULowerLimit.*
Enable	BOOL, specifying whether chroma key functionality is to be enabled or not

Remarks Sets the chroma key settings for the high performance scaler on a Rio board (see section "Choice of scaler" on page 15). These settings will be used when a module is using the high performance scaler and only have meaning when the mode is RIO_COLOR. When a new mode is set which uses the high performance scaler, the chroma key settings will be disabled.

RioSetContrast

#include <Rio.h>

Function int RioSetContrast(int *BoardId*, RIO_INPUT_MODULE *InputModule*, char *Value*);

Return Value RIO_OK successful

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_INPUT_MODULE	an invalid *InputModule* was given

Parameters

BoardId	BoardID of the Rio board on which to set the contrast.
InputModule	Input module for which to set the contrast. Can be either RIO_IM_0 or RIO_IM_1.
Value	Contrast value to set, any number ranging from -128 .. 127. Scale : -128 = -2 inverse, -64 = -1 inverse, 0 = contrast off, 64= contrast 1, 70= CCIR level (default), 127 =1.99 max. contrast.

Remarks Sets the contrast for a module on a Rio board. This function only has effect when the mode is RIO_COLOR, RIO_BW or RIO_FULL_FRAME. If the mode is RIO_FULL_FRAME, only RIO_IM_0 should be used as value for *InputModule*.

Example See RioSetBrightness.

RioSetHqBw

#include <Rio.h>

Function int RioSetHqBw(int *BoardId*, float *Gain*, float *Offset*);

Return Value

RIO_OK	successful

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_HQ_GAIN	invalid *Gain* value was given
RIO_INVALID_HQ_OFFSET	invalid *Offset* value was given

Parameters

BoardId	BoardID of the Rio board for which to set the Gain and Offset.
Gain	Gain value to be set, (Range: 0.325x …1x (default) … 8x multiplication).
Offset	Offset value to be set (Range: -1.0 Volts … 0 Volts (default)… 1.0 Volts).

Remarks Sets the Black and White High Quality Gain and Offset for a Rio board. It is only used for module mode RIO_HQ. The function is used to set these Black and White High Quality properties, even if not currently in RIO_HQ mode. Therefore this function can also be used preceding a switch to RIO_HQ mode.

Example See RioSetBrightness.

RioSetHue

#include <Rio.h>

Function int RioSetHue(int *BoardId*, RIO_INPUT_MODULE *InputModule*, char *Value*);

Return Value

RIO_OK	successful

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_INPUT_MODULE	an invalid *InputModule* was given

Parameters

BoardId	BoardID of the Rio board on which to set the hue.
InputModule	Input module for which to set the hue. Can be either RIO_IM_0 or RIO_IM_1.
Value	Hue value to set, any number ranging from -128 .. 127. Scale: -128 = -180 º , 0 = 0 º (default), 127 = 178.6 º.

Remarks Sets the hue for an input-module on a Rio board. This function only has meaning when the module is in RIO_COLOR_MODE with the RIO_COLOR_VIDEO_STANDARD set to RIO_NTSC.

Example See RioSetBrightness.

RioSetInputGain

#include <Rio.h>

Function int RioSetInputGain(int *BoardId*, RIO_INPUT_MODULE *InputModule*, int *Input*, uchar *Value*, RIO_ON_OFF_MODE *AutoGainMode*);

Return Value

RIO_OK	successful

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_INPUT_MODULE	an invalid *InputModule* was given
RIO_INVALID_INPUT_FOR_INPUT_MODULE	*Input* was invalid number for input module
RIO_INVALID_AUTO_GAIN_MODE	an invalid *AutoGainMode* was given
RIO_INVALID_INPUT_GAIN	invalid *Value* was given

Parameters

BoardId	BoardID of the Rio board on which to set the input gain.
InputModule	Input module for which to set the input gain. Can be either RIO_IM_0 or RIO_IM_1
Input	Input (connector) number (numbering depending on module mode, see RioSetInputModule and Figure 14).
Value	Input Gain value to set, any number ranging from 0 .. 63. Scale: 0 = -2.82 dB, 15= 0 dB (default), 31= 3 dB, 47 = 6 dB, 63 = 9 dB (every +16 = +3 dB).
AutoGainMode	Indicating input gain is to be set in Auto Gain mode, can be either RIO_ON or RIO_OFF (default).

Remarks Sets the gain settings for an input of a module on a Rio board. This function has no effect when the mode is RIO_HQ. If the mode is RIO_RGBA or RIO_FULL_FRAME, only RIO_IM_0 should be used as value for *InputModule*. When *AutoGainMode* is set to be RIO_ON, *Value* will not be used.

Example See RioSetBrightness.

RioSetInputModule

#include <Rio.h>

Function int RioSetInputModule(int *BoardId*, PRIO_MODULE *Module*);

Return Value

RIO_OK	successful

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_POINTER	*Module* was NULL
RIO_INVALID_MODULE_MODE	an invalid *Module->Mode* was given
RIO_INVALID_INPUT_MODULE	for *Module->Def.* relevant modules, an invalid *InputModule* was given
RIO_INVALID_SCALER	for *Module->Def.* relevant modules, an invalid *Scaler* was given
RIO_INVALID_VIDEO_STANDARD	for *Module->Def.* relevant modules, an invalid *VideoStandard* was given
RIO_INVALID_TV_OR_VTR	for *Module->Def.* relevant modules, an invalid *TvOrVtr* was given
RIO_INVALID_OUTPUT_FORMAT	for *Module->Def.* relevant modules, an invalid *OutputFormat* was given
RIO_INVALID_FIELD_OR_FRAME	for *Module->Def.* relevant modules, an invalid *FieldOrFrame* was given
RIO_INVALID_CVBS_OR_YC	for *Module->Def.Color*, an invalid *CvbsOrYc* was given
RIO_INVALID_POST_PROCESS	for *Module->Def.* relevant modules, an invalid *PostProcess* was given

Parameters

BoardId	BoardID of the Rio board to set the module state for.
Module	Pointer to one RIO_MODULE struct, containing information about a Rio module to be set. It contains information about the module mode to be set and information about the state to set the needed input module(s) in.

Remarks Sets a desired input module state. The RIO_MODULE to be set will be in one of the modes listed below. The connector numbering that is appropriate during this mode is listed below for every mode:

	connector numbering
• RIO_COLOR	0, 1, 2 (per Y/C)
• RIO_BW	0, 1, 2, 3, 4, 5
• RIO_HQ,	0, 1, 2 (only Y connectors are used)
• RIO_STEREO_LOCKED	0, 1, 2 (per Y/C). 2 cameras are needed
• RIO_S_FULL_FRAME	0, 1, 2 (per Y/C)
• RIO_FULL_FRAME	0, 1, 2 (per Y/C)
• RIO_RGBA	fixed: R, G, B, A to first 4 connectors, indicated by 0

The modes RIO_HQ, RIO_FULL_FRAME or RIO_RGBA, only can be used on a Full Rio. See RIO_MODULE_MODE enum at page 21, RIO_MODULE struct at page 29 and section 'Rio Modules' for more information about Rio Modules.
RioSetInputModule sets the selected camera number to 0, so after calling RioSetInputModule the programmer has to call RioSelectCamera with the connector number that is needed.
Also when the high performance scaler is used, the chroma key setting will be disabled. When the programmer wants to use chroma keying (only useful for RIO_COLOR mode), he has to call RioSetChromaKey with the settings he wants to use.

Example In the example an input module is set on the first Rio board. pRiodl is assumed to be a correctly created PRIODL Rio device list of the available Rio boards (see RioCreateRiodl example).

```
int Result;
RIO_MODULE m_Module;

// setting the rio module to rio color
m_Module.Mode = RIO_COLOR;
m_Module.Def.Color.InputModule     = RIO_IM_0;       // selected is first input module
m_Module.Def.Color.Scaler          = RIO_HPS;        // high performance scaler
m_Module.Def.Color.VideoStandard   = RIO_PAL ;       // 50 Hz PAL video standard
m_Module.Def.Color.TvOrVtr         = RIO_TV;         // slow locking, small jitter
m_Module.Def.Color.CvbsOrYc        = RIO_YC;         // camera with YC connectors
m_Module.Def.Color.FieldOrFrame    = RIO_FIELD;      // field grabbing
m_Module.Def.Color.OutputFormat    = RIO_RGB24;      // RIO_RGB24 output format

// now set the first Rio board to this input module
Result= RioSetInputModule(pRiodl->BoardId[0], &m_Module);
if ( Result != RIO_OK )
{
    sprintf(s, "Module on first Rio board could not be set. Result: 0x%X", Result);
    MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
}
else
{
    // select a camera, in this example the second Y/C connectors were used
    // (see 'Video Source Connectors')

    Result = RioSelectCamera(pRiodl->BoardId[0], RIO_IM_0, 1);
    if ( Result != RIO_OK )
    {
        sprintf(s, "Could not select camera. Result: 0x%X", Result);
        MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
    }
}
```

RioSetLed

#include <Rio.h>

Function int RioSetLed(int *BoardId*, RIO_ON_OFF_MODE *LedState*);

Return Value RIO_OK successful

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_LED_STATE	an invalid *LedState* was given

Parameters

BoardId	BoardID of the Rio board on which the led is to be set.
LedState	State to set the board led in, can be either RIO_ON or RIO_OFF.

Remarks Sets the led of a Rio board to on or off. RioIndexSetLed does the same, except the *BoardIndex* is used to identify a Rio board, instead of *BoardID*.

Example Function that sets the leds of all basic Rio boards to off and the leds of all full Rio boards to on. pRiodl is assumed to be a correctly created global PRIODL Rio device list of the available Rio boards (see RioCreateRiodl example).

```
void OnSetAllFullRioBoardsLedsOn()
{
    BOOL IsBasicRioVersion;
    int Result;
    char s[80];

    for (int  i = 0; i < pRiodl->NrBoards; i++ )
    {
        RioGetBoardType(pRiodl->BoardId[i], &IsBasicRioVersion);
        if ( IsBasicRioVersion )
        {
           // set led of Rio with BoardId[i] to off
           Result = RioSetLed(pRiodl->BoardId[i], RIO_OFF);

           if ( Result != RIO_OK )
           {
              sprintf(s, "Unable to set led of Rio ID: %i to off", pRiodl->BoardId [i]);
              MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
           }
        }
        else
        {
           // set led of Rio with BoardId[i] to on
           Result = RioSetLed(pRiodl->BoardId[i], RIO_ON);

           if ( Result != RIO_OK )
           {
              sprintf(s, "Unable to set led of Rio ID: %i to on", pRiodl->BoardId [i]);
              MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
           }
        }
    }
}
```

RioSetSaturation

#include <Rio.h>

Function int RioSetSaturation(int *BoardId*, RIO_INPUT_MODULE *InputModule*,
char *Value*);

Return Value

RIO_OK	successful

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_INPUT_MODULE	an invalid *InputModule* was given

Parameters

BoardId	BoardID of the Rio board on which to set the saturation.
InputModule	Input module for which to set the saturation. Can be either RIO_IM_0 or RIO_IM_1.
Value	Saturation value to set, any number ranging from -128 .. 127. Scale: -128 = -2 inverse, -64 = -1 inverse, 0 = color off, 64 = CCIR level (default), 127 = 1.99 max. saturation.

Remarks Sets the saturation for a module on a Rio board. This function only has effect when the mode is RIO_COLOR.

Example See RioSetBrightness.

RioSizeOfRiodl

#include <Rio.h>

Function int RioSizeOfRiodl(void);

Return Value Size of RIODL struct to be allocated by the user, before used in RioCreateRiodl.

Remarks Indicates the amount of memory to be allocated (malloc) by the programmer to a RIODL struct, to be used to obtain information about the amount of Rio boards and BoardIDs by way of RioCreateRiodl (see next function).

Example See RioCreateRiodl and RioSetBoardId.

RioStreamAddEmptyBuffer

#include <Rio.h>

Function int RioStreamAddEmptyBuffer(int *BoardId*, RIO_INPUT_MODULE *InputModule*, PRIO_VIDEO_HDR *VideoHdr*);

Return Value RIO_OK successful

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_INPUT_MODULE	an invalid *InputModule* was given
RIO_INVALID_POINTER	*VideoHdr* was NULL

Parameters

BoardId	BoardID of the Rio board on which to add a stream buffer.
InputModule	Input module for which to add the stream buffer. Can be either RIO_IM_0 or RIO_IM_1.
VideoHdr	Pointer to RIO_VIDEO_HDR struct, containing the information about the buffer to be added in the array of stream buffers (1 buffer per image).

Remarks Adds a empty buffer into the array of stream buffers to be used for streaming capture. If a buffer has been read using RioStreamGetFilledBuffer, this function also has to be called to re-use this buffer for a new streaming capture.

For more information about the video header, see RIO_VIDEO_HDR at page 24.

If the mode is RIO_HQ, RIO_FULL_FRAME or RIO_RGBA, only RIO_IM_0 should be used as value for *InputModule*.

Example See RioStreamGetFilledBuffer

RioStreamClose

#include <Rio.h>

Function int RioStreamClose(int *BoardId*, RIO_INPUT_MODULE *InputModule*);

Return Value

RIO_OK	successful

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_INPUT_MODULE	an invalid *InputModule* was given

Parameters

BoardId	BoardID of the Rio board on which to close the streaming capture.
InputModule	Input module for which to close the streaming capture. Can be either RIO_IM_0 or RIO_IM_1.

Remarks Closes the streaming capture setting. It should be called after all stream operations are finished. If the mode is RIO_HQ, RIO_FULL_FRAME or RIO_RGBA, only RIO_IM_0 should be used as value for *InputModule.*

Example See RioStreamGetFilledBuffer

RioStreamGetFilledBuffer

#include <Rio.h>

Function int RioStreamGetFilledBuffer(int *BoardId*, RIO_INPUT_MODULE *InputModule*, PRIO_VIDEO_HDR **VideoHdr*, OVERLAPPED **pOverlapped*);

Return Value

RIO_OK	successful, WaitForSingleObject(*pOverlapped*->hEvent,..) is not needed any more strictly speaking, because the buffer is already correctly filled and available.
RIO_PENDING	WaitForSingleObject(*pOverlapped*->hEvent,..) must be used to wait for the filled stream-buffer to be ready to be used.
RIO_ALREADY_GETTING_BUFFER	RioStreamGetFilledBuffer already was called previously, WaitForSingleObject(*pOverlapped*->hEvent,..) should first be used to wait for that filled stream-buffer, before calling RioStreamGetFilledBuffer again.
RIO_CAPTURE_ERROR	internal error, user has probably forgotten to call RioScatterLock before starting capture.
RIO_DEVICE_IO_CONTROL	internal error, contact Ellips.
RIO_NO_FILLED_BUFFERS	RioStreamStop was called, and the last available buffer already was obtained with a previous call to RioStreamGetFilledBuffer.
RIO_VIDEO_OVERFLOW	some pixels in the image were not obtained correctly. WaitForSingleObject(*pOverlapped*->hEvent,..) is not needed any more strictly speaking, because the buffer is already filled and available.
RIO_UNABLE_TO_POST_PROCESS	postprocessing failed. WaitForSingleObject(*pOverlapped*->hEvent,..) is not needed any more strictly speaking, because the buffer is already filled and available.
RIO_VIDEO_OVERFLOW_NO_POST_PROCESS	some pixels in the image were not obtained correctly and postprocessing failed. WaitForSingleObject(*pOverlapped*->hEvent,..) is not needed any more strictly speaking, because the buffer is already filled and available.

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_INPUT_MODULE	an invalid *InputModule* was given.

RIO_INVALID_OVERLAPPED	*pOverlapped* was not passed correctly. *pOverlapped*==NULL is NOT ALLOWED for this function. Function cannot handle waiting for event itself.

Parameters

BoardId	BoardID of the Rio board from which to get a stream buffer.
InputModule	Input module from which to get the stream buffer. Can be either RIO_IM_0 or RIO_IM_1.
**VideoHdr*	Pointer to RIO_VIDEO_HDR struct, containing the information about the buffer to be used.
pOverlapped	Pointer to OVERLAPPED struct. *pOverlapped*->hEvent can be used to signal the stream buffer is filled and can be used. *pOverlapped*==NULL is NOT ALLOWED for this function. Function cannot handle waiting for event itself.

Remarks

Gets a filled stream buffer from a module on a Rio board. If the function returns RIO_OK or RIO_VIDEO_OVERFLOW, the imagebuffer is already filled and available, therefore WaitForSingleObject(*pOverlapped*->hEvent,..) is not needed any more strictly speaking.
If RIO_PENDING is returned, the programmer must use WaitForSingleObject with the *pOverlapped*->hEvent member, to wait for the capture to be completed. *pOverlapped* can (then) be passed to RioGetOverlappedResult to get the return code of the overlapped operation, that indicates if capture was completed successfully.

For more information about the video header, see RIO_VIDEO_HDR at page 24.

If the mode is RIO_HQ, RIO_FULL_FRAME or RIO_RGBA, only RIO_IM_0 should be used as value for *InputModule*.

Example.

In this example streaming capture is demonstrated. pRiodl is assumed to be a correctly created PRIODL Rio device list of the available Rio boards (see RioCreateRiodl example). First input module and a camera are assumed to be set correctly on the first Rio board (see RioSetInputModule example). Pitch (line-pitch in bytes for one memory buffer), XRes (horizontal number of pixels for one memory buffer), YRes (vertical number of pixels for one memory buffer) are assumed to be correctly created imagebuffer parameters, pImageBufferHandle is assumed to be a pointer to an array of imagebuffer HANDLEs, each correctly created with RioScatterLock (in the same way as ImageBufferHandle in RioScatterLock example). iNumofBuffers is assumed to be the number of imagebuffers and therefore also the number of imagebuffer handles in the imagebuffer handle array.

```
char s[80];
int Result, OverlapResult;
OVERLAPPED Overlapped;
DWORD WFSO_Result;
RECT SourceRect, DestRect;
PRIO_VIDEO_HDR pVideoHdr;
BOOL StreamStarted = FALSE;

// in the example the total imagebuffer size will be filled for all buffers
int LineOffset = 15;
SetRect(&SourceRect, 0, LineOffset, XRes, LineOffset+YRes);
SetRect(&DestRect, 0, 0, XRes, YRes);
```

```
// setting stream parameters
Result = RioStreamInit(pRiodl->BoardId[0], RIO_IM_0, 40000, FALSE, TRUE,
                       &SourceRect, &DestRect, Pitch);
if ( Result != RIO_OK )
{
    sprintf(s, "Initializing streaming capture failed. Result: 0x%X", Result);
    MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
}
else
{
    // allocate array of videoheaders (one for every imagebuffer)
    pVideoHdr = new RIO_VIDEO_HDR[iNumofBuffers];
    BOOL AllHeadersAddedOK = TRUE;
    for (int i=0; i<iNumofBuffers; i++)
    {
        pVideoHdr[i].ImageBuffer = pImageBufferHandle[i];
        pVideoHdr[i].User = i;              // setting a custom video header id
        Result = RioStreamAddEmptyBuffer(pRiodl->BoardId[0], RIO_IM_0, &pVideoHdr[i]);
        if ( Result != RIO_OK ) AllHeadersAddedOK = FALSE;
    }

    if (!AllHeadersAddedOK)
    {
        sprintf(s, "Not all headers could be added. Result: 0x%X", Result);
        MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
    }
    else
    {
        Overlapped.hEvent  = CreateEvent(NULL,       // pointer to security attributes
                                         TRUE,       // flag for manual-reset event
                                        FALSE,       // flag for initial state
                                         NULL);      // pointer to event-object name


        // starting stream capture
        Result = RioStreamStart(pRiodl->BoardId[0], RIO_IM_0);
        if ( Result != RIO_OK )
        {
            sprintf(s, "Starting actual stream failed. Result: 0x%X", Result);
            MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
        }
        else StreamStarted = TRUE;
    }
}

.. .. .. ..


// if (StreamStarted), stream capture can be done from a separate thread
// (for instance from within a while loop in the thread :)

// in this example, checking for a user request for stream stop, is also done from
// within this loop:
if .. .. ..    // the situation, test or event that indicates stream stop is requested
               // by the user
{
    Result = RioStreamStop(pRiodl->BoardId[0], RIO_IM_0);   // stopping actual stream
    if ( Result != RIO_OK )
    {
        sprintf(s, "Could not stop streaming capture. Result: 0x%X", Result);
        MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
    }
}

ResetEvent(Overlapped.hEvent);             // first making sure event can be (re)used


// now a streamed capture can be obtained
PRIO_VIDEO_HDR pReturnedVideoHdr;        // pointer used to get returned PRIO_VIDEO_HDR
Result = RioStreamGetFilledBuffer(pRiodl->BoardId[0], RIO_IM_0,
                                  &pReturnedVideoHdr, &Overlapped);
```

```
if (( Result != RIO_PENDING ) && (Result != RIO_OK))
{
    if ( Result == RIO_NO_FILLED_BUFFERS )
    {
        // obviously stream capture has been stopped with a call to RioStreamStop The
        // user can stop this thread now (close up the while loop and end the stream
        // capture (clean up), because the last capture has already been obtained
    }
    else
    {
        sprintf(s, "Start waiting for streaming capture failed. Result: 0x%X", Result);
        MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION | MB_SETFOREGROUND);
    }
}
else
{
    // overlapped procedure started
    WFSO_Result= WaitForSingleObject(Overlapped.hEvent, 3000);
    if ( WFSO_Result == WAIT_FAILED )
    {
        sprintf(s, "WaitForSingleObject(Overlapped.hEvent, 3000) failed");
        MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION | MB_SETFOREGROUND);
    }
    else
    {
        if ( WFSO_Result == WAIT_TIMEOUT )
        {
            sprintf(s, "The time-out has expired");
            MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION | MB_SETFOREGROUND);
        }
        else
        {
            Result = RioGetOverlappedResult(&Overlapped, &OverlapResult, TRUE);
            if ( (Result != RIO_OK) || (OverlapResult !=  RIO_OK) )
            {
                sprintf(s, "Error, Res: 0x%X, OverlRes: 0x%X", Result, OverlapResult);
                MessageBox(NULL, s, NULL, MB_OK| MB_ICONEXCLAMATION| MB_SETFOREGROUND);
            }
            else
            {
                // user can now visualize or save to file imagebuffer returned via
                // pReturnedVideoHdr

                // after finished with buffer, give buffer back to stream process to
                // be re-used:
                Result = RioStreamAddEmptyBuffer(pRiodl->BoardId[0], RIO_IM_0,
                                                 pReturnedVideoHdr);
                if ( Result != RIO_OK )
                {
                    sprintf(s, "Re-use of buffer failed. Result: 0x%X", Result);
                    MessageBox(NULL, s, NULL,
                                    MB_OK | MB_ICONEXCLAMATION | MB_SETFOREGROUND);
                }
            }
        }
    }
}

.. .. .. ..

// after the last buffer has been obtained, after RioStreamStop was done, cleaning up:
Result = RioStreamClose(pRiodl->BoardId[0], RIO_IM_0);
if ( Result != RIO_OK )
{
    sprintf(s, "Could not close streaming capture mode. Result: 0x%X", Result);
    MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
}
else
{
    // now we can clean up video headers
    delete[] pVideoHdr;
}
```

RioStreamGetStartTime

#include <Rio.h>

Function int RioStreamGetStartTime(int *BoardId*, RIO_INPUT_MODULE *InputModule*, DWORD **StartTime*);

Return Value

RIO_OK	successful

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_INPUT_MODULE	an invalid *InputModule* was given
RIO_INVALID_POINTER	*StartTime* was NULL

Parameters

BoardId	BoardID of the Rio board from which to get the time streaming capture started.
InputModule	Input module for which to get the time the streaming capture started. Can be either RIO_IM_0 or RIO_IM_1.
**StartTime*	The time (in milliseconds) at which time the streaming capture was started.

Remarks Gets the time the streaming capture started on an input module. This function can be used to synchronize streams.

RioStreamInit

#include <Rio.h>

Function int RioStreamInit(int *BoardId*, RIO_INPUT_MODULE *InputModule*, int *MicroSecPerCapture*, BOOL *SquarePixels*, BOOL *TopDown*, RECT **SrcRect*, RECT **DestRect*, int *Pitch*);

Return Value

RIO_OK	successful
RIO_ALREADY_STARTED_CAPTURE	**only Windows NT/2000** Capture already was started (with a call to RioStreamStart), streaming capture parameters cannot be set during streaming capture.

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_INPUT_MODULE	an invalid *InputModule* was given
RIO_INVALID_MICROSEC_PER_CAPTURE	*MicroSecPerCapture* must be >=0
RIO_INVALID_SRC_RECT	*SrcRect* was NULL or **SrcRect* was not normalized
RIO_INVALID_DEST_RECT	*DestRect* was NULL or **DestRect* was not normalized

Parameters

BoardId	BoardID of the Rio board to be used for streaming capture.
InputModule	Input module to be used for the streaming capture. Can be either RIO_IM_0 or RIO_IM_1.
MicroSecPerCapture	Desired streaming capture rate (in microseconds per capture). Streaming capture will be performed as close to the desired capture rate as possible. The user should be aware of the effects. For instance at 50 Hz for PAL/ SECAM/ CCIR, *MicroSecPerCapture* =20000 microseconds is the highest rate that can be achieved, and choosing *MicroSecPerCapture*= 50000, 70000… microseconds in frame mode, will result in alternating even/odd and odd/even captures.
SquarePixels	BOOL, Only used when RIO_FIELD_OR_FRAME is set to RIO_FIELD, RIO_FIELD_ODD or RIO_FIELD_EVEN. If *SquarePixels*==TRUE, the captured source will be in square pixels, although in field mode.
Topdown	BOOL, indicating topdown line image-building (TRUE) or not.
**SrcRect*	The source rectangle (bottom and right will not be included).
**DestRect*	The destination rectangle (bottom and right will not be included). Must have the same size as the source rectangle. In future versions, the destination rectangle may be smaller than source rectangle (downscale).

	Both rectangles need to be normalized (top<bottom, left<right) and must represent a minimum image size of one Dword in memory.
Pitch	The line-pitch in bytes, needed to correctly align the lines in memory (depending on resolution).

Remarks Initializes the streaming capture parameters. Its counter-part RioStreamClose() should be called after all stream operations are finished.
If the mode is RIO_HQ, RIO_FULL_FRAME or RIO_RGBA, only RIO_IM_0 should be used as value for *InputModule*.

Example See RioStreamGetFilledBuffer.

RioStreamStart

#include <Rio.h>

Function int RioStreamStart(int *BoardId*, RIO_INPUT_MODULE *InputModule*);

Return Value

RIO_OK	successful
RIO_ALREADY_STARTED_CAPTURE	**only Windows NT/2000** Capture already was started with a previous call to RioStreamStart

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_INPUT_MODULE	an invalid *InputModule* was given

Parameters

BoardId	BoardID of the Rio board on which to start streaming capture.
InputModule	Input module for which to start the streaming capture. Can be either RIO_IM_0 or RIO_IM_1.

Remarks Actually starts streaming capture on a module on a Rio board. Start stream only after you have correctly initialized the streaming capture with RioStreamInit.
If the mode is RIO_HQ, RIO_FULL_FRAME or RIO_RGBA, only RIO_IM_0 should be used as value for *InputModule*.

Example See RioStreamGetFilledBuffer

RioStreamStop

#include <Rio.h>

Function int RioStreamStop(int *BoardId*, RIO_INPUT_MODULE *InputModule*);

Return Value

RIO_OK	successful

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_INPUT_MODULE	an invalid *InputModule* was given

Parameters

BoardId	BoardID of the Rio board on which to stop streaming capture.
InputModule	Input module for which to stop the streaming capture. Can be either RIO_IM_0 or RIO_IM_1.

Remarks Stops streaming capture on a module on a Rio board.
If the mode is RIO_HQ, RIO_FULL_FRAME or RIO_RGBA, only RIO_IM_0 should be used as value for *InputModule*.

Example See RioStreamGetFilledBuffer

RioTriggerCapture

#include <Rio.h>

Function int RioTriggerCapture(int *BoardId*, RIO_INPUT_MODULE *InputModule*,
BOOL *SquarePixels*, BOOL *TopDown*,
PRIO_FLASH_PARAMS *FlashParams*,
PRIO_TBUFFER_PARAMS *TBufferParams*,
RECT **SrcRect*, RECT **DestRect*, int *Pitch*,
OVERLAPPED **pOverlapped*);

Return Value

RIO_OK	successful (in case *pOverlapped*==NULL)
RIO_PENDING	capture sequence started successfully (in case *pOverlapped*!=NULL). WaitForSingleObject(*pOverlapped*->hEvent,..) must be used to wait for the capture to be completed.
RIO_ALREADY_STARTED_CAPTURE	**only Windows NT/2000** RioTriggerCapture already was called previously, WaitForSingleObject(*pOverlapped*->hEvent,..) should first be used to wait for that capture, before calling RioTriggerCapture again.
RIO_CAPTURE_ERROR	internal error, user has probably forgotten to call RioScatterLock before starting capture.
RIO_DEVICE_IO_CONTROL	internal error, contact Ellips.
RIO_VIDEO_OVERFLOW	some pixels in the images were not obtained correctly
RIO_UNABLE_TO_POST_PROCESS	postprocessing failed
RIO_VIDEO_OVERFLOW_NO_POST_PROCESS	some pixels in the image were not obtained correctly and postprocessing failed
RIO_TIMEOUT	the time-out has expired (in case *pOverlapped*==NULL)
RIO_WAIT_FAILED	the wait failed (in case *pOverlapped*==NULL)
parameter checking:	
RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_INPUT_MODULE	an invalid *InputModule* was given
RIO_INVALID_FLASH_PIN	an invalid *FlashParams->FlashPin* was given
RIO_INVALID_FLASH_LINE	the following check failed on *FlashParams*: (*Endline*>=*StartLine*) &&

	(*Length* <= (*Endline- StartLine* +1))
RIO_INVALID_VIDEO_OUT_DELAY	*FlashParams->VideoOutDelay* was 0
RIO_INVALID_TBUFFER_PARAMS	*TBufferParams* was NULL
RIO_INVALID_TRIGGER_PIN	an invalid *TBufferParams->TriggerPin* was given
RIO_INVALID_NR_BUFFERS	*TBufferParams->NrBuffers* was 0
RIO_INVALID_IMAGE_BUFFER	*TBufferParams->ImageBuffer* was NULL
RIO_INVALID_TRIGGER_POSITION	the following check failed on *TBufferParams:* (0<=*TriggerPosition* < *NrBuffers*)
RIO_INVALID_TRIGGER_BUFFER_NR	*TBufferParams->TriggerBufferNr* was NULL
RIO_INVALID_CAPTURE_PERIOD	*TBufferParams->CapturePeriod* was 0
RIO_INVALID_SRC_RECT	*SrcRect* was NULL or **SrcRect* was not normalized
RIO_INVALID_DEST_RECT	*DestRect* was NULL or **DestRect* was not normalized

Parameters

BoardId	BoardID of the Rio board from which to get a triggered capture.
InputModule	Input module from which to get the triggered capture. Can be either RIO_IM_0 or RIO_IM_1.
SquarePixels	BOOL, Only used when RIO_FIELD_OR_FRAME is set to RIO_FIELD, RIO_FIELD_ODD or RIO_FIELD_EVEN. If *SquarePixels*==TRUE, the captured source will be in square pixels, although in field mode.
Topdown	BOOL, indicating topdown line image-building (TRUE) or not.
FlashParams	Pointer to a RIO_FLASH_PARAMS struct, containing flash information (such as flash pin to be used). In order to be able to use flash during triggered capture, the flash pin should be set with RioGpioControl to RIO_GPIO_OUTPUT, before RioTriggerCapture is called. When NULL is passed for *FlashParams*, no flash signal will be generated.
TBufferParams	Pointer to a RIO_TBUFFER_PARAMS struct, containing the buffer information, including the general-purpose-io pin to be monitored as trigger pin and an array of imagebuffers, indicating where to store captured images. The pin used as trigger pin should be set to RIO_GPIO_IRQ_RISE, RIO_GPIO_IRQ_FALL or RIO_GPIO_IRQ_BOTH with RioGpioControl before RioTriggerCapture is called.
**SrcRect*	The source rectangle (bottom and right will not be included).
**DestRect*	The destination rectangle (bottom and right will not be included). Must have the same size as the source rectangle. In future versions, the destination rectangle may be smaller than source rectangle (downscale).

	Both rectangles need to be normalized (top<bottom, left<right) and must represent a minimum image size of one Dword in memory.
Pitch	The line-pitch in bytes, needed to correctly align the lines in memory (depending on resolution).
pOverlapped	Pointer to OVERLAPPED struct. *pOverlapped*->hEvent can be used to signal the capture sequence is done and can be used.

Remarks

Gets a triggered capture as part of a sequence of captures from a module on a Rio board. A sequence of captures can be obtained, including the capture at trigger moment. The programmer can define the number of captures to be done in the sequence. To know more about the way in which the capture sequence is returned via *TBufferParams*, see RIO_TBUFFER_PARAMS at page 25.

Capture can be achieved in two different ways:

- If NULL is passed for *pOverlapped*, the function will only return after the captures have finished (RIO_OK or RIO_VIDEO_OVERFLOW returned) or the time-out that was set in RioTriggerCaptureTimeOut will have expired (RIO_TIMEOUT returned) (Example 1).
- If *pOverlapped* is a valid pointer to an OVERLAPPED struct, the function will return RIO_PENDING if capture started successfully. The programmer must use WaitForSingleObject with *pOverlapped*->hEvent, to wait for the sequence of captures to be completed. *pOverlapped* can (then) be passed to RioGetOverlappedResult to get the return code of the overlapped operation, that indicates if captures were completed successfully (Example 2).

If NULL is passed for *FlashParams*, no flash signal will be generated. To know more about the flash parameters that can be passed, see RIO_FLASH_PARAMS at page 24.

If the mode is RIO_HQ, RIO_FULL_FRAME or RIO_RGBA, only RIO_IM_0 should be used as value for *InputModule*.

Example 1. In the first example RioTriggerCapture is called for the first input module, with *pOverlapped*==NULL, in a non multithreaded way. pRiodl is assumed to be a correctly created PRIODL Rio device list of the available Rio (see RioCreateRiodl example). First input module and a camera are assumed to be set correctly on the first Rio board (see RioSetInputModule example). Pitch (line-pitch in bytes for one memory buffer), XRes (horizontal number of pixels for one memory buffer), YRes (vertical number of pixels for one memory buffer) are assumed to be correctly created imagebuffer parameters, pImageBufferHandle is assumed to be a pointer to an array of imagebuffer HANDLEs, each correctly created with RioScatterLock (in the same way as ImageBufferHandle in RioScatterLock example). iNumofBuffers is assumed to be the number of imagebuffers and therefore also the number of imagebuffer handles in the imagebuffer handle array.

```
char s[80];
int Result;

RECT SourceRect, DestRect;
RIO_FLASH_PARAMS       FParams;
RIO_TBUFFER_PARAMS     TBufferParams;

// in the example the total imagebuffer size will be filled for all buffers
int LineOffset = 15;
SetRect(&SourceRect, 0, LineOffset, XRes, LineOffset+YRes);
SetRect(&DestRect, 0, 0, XRes, YRes);

// setting up buffer information
TBufferParams.TriggerPin= RIO_GPIO_0;                    // first pin used for trigger
TBufferParams.NrBuffers= iNumofBuffers;                    // as stated above
TBufferParams.ImageBuffer= pImageBufferHandle;             // as stated above
TBufferParams.TriggerPosition= 0;  // capture 0 in sequence will be capture at trigger
TBufferParams.CapturePeriod= 1;                          // continuous capture
uint32 TBufferNr;
TBufferParams.TriggerBufferNr= &TBufferNr;        // will return trigger buffer number

// setting up flash information
FParams.FlashPin= RIO_GPIO_1;
// available flash period set to maximum, (taking occasional black images for granted)
FParams.Startline= 0;
FParams.EndLine= YRes-1;
FParams.Length= 1;
FParams.VideoOutDelay= 1;

// setting time-out to 3 seconds
Result = RioTriggerCaptureTimeOut(pRiodl->BoardId[0], RIO_IM_0, 3000);
if ( Result != RIO_OK )
{
   sprintf(s,"Changing time-out to other then default, failed. Result: 0x%X", Result);
   MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
}
// in this example, the trigger pin is set to interrupt on rise
Result = RioGpioControl(pRiodl->BoardId[0], RIO_GPIO_IRQ_RISE, RIO_GPIO_OUTPUT,
                                          RIO_GPIO_IGNORE, RIO_GPIO_IGNORE);
if ( Result != RIO_OK )
{
    sprintf(s, "General purpose pin could not be set.  Result: 0x%X", Result);
    MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
}
else
{
    Result = RioTriggerCapture(pRiodl->BoardId[0], RIO_IM_0, FALSE, TRUE,
                                    &FParams, &TBufferParams,
                                    &SourceRect, &DestRect, Pitch, NULL);
    if ( Result != RIO_OK )   // will be RIO_TIMEOUT or RIO_VIDEO_OVERFLOW
    {
        sprintf(s, "Capture failed. Result: 0x%X", Result);
        MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
    }
    else
    {  // user can now visualize or save imagebuffer array to file }
}
```

Example 2. In the second example RioTriggerCapture is called with *pOverlapped* != NULL, in a multithreaded way. pRiodl is assumed to be a correctly created PRIODL Rio device list of the available Rio boards (see RioCreateRiodl example). First input module and a camera are assumed to be set correctly on the first Rio board (see RioSetInputModule example). Pitch (line-pitch in bytes for one memory buffer), XRes (horizontal number of pixels for one memory buffer), YRes (vertical number of pixels for one memory buffer) are assumed to be correctly created imagebuffer parameters, pImageBufferHandle is assumed to be a pointer to an array of imagebuffer HANDLEs, each correctly created with RioScatterLock (in the same way as ImageBufferHandle in RioScatterLock example). iNumofBuffers is assumed to be the number of imagebuffers and therefore also the number of imagebuffer handles in the imagebuffer handle array.

```
char s[80];
int Result, OverlapResult;
BOOL bOverlappedProcStarted = FALSE;
OVERLAPPED Overlapped;
DWORD WFSO_Result;
RECT SourceRect, DestRect;
RIO_FLASH_PARAMS       FParams;
RIO_TBUFFER_PARAMS     TBufferParams;

Overlapped.hEvent  = CreateEvent(NULL,              // pointer to security attributes
                                 TRUE,              // flag for manual-reset event
                                 FALSE,             // flag for initial state
                                 NULL);             // pointer to event-object name


// in the example the total imagebuffer size will be filled for all buffers
int LineOffset = 15;
SetRect(&SourceRect, 0, LineOffset, XRes, LineOffset+YRes);
SetRect(&DestRect, 0, 0, XRes, YRes);

// setting up buffer information
TBufferParams.TriggerPin= RIO_GPIO_0;                    // first pin used for trigger
TBufferParams.NrBuffers= iNumofBuffers;                    // as stated above
TBufferParams.ImageBuffer= pImageBufferHandle;             // as stated above
TBufferParams.TriggerPosition= 0;  // capture 0 in sequence will be capture at trigger
TBufferParams.CapturePeriod= 1;                          // continuous capture
uint32 TBufferNr;
TBufferParams.TriggerBufferNr= &TBufferNr;        // will return trigger buffer number

// setting up flash information
FParams.FlashPin= RIO_GPIO_1;
// available flash period set to maximum, (taking occasional black images for granted)
FParams.Startline= 0;
FParams.EndLine= YRes-1;
FParams.Length= 1;
FParams.VideoOutDelay= 1;

// setting time-out to 3 seconds
Result = RioTriggerCaptureTimeOut(pRiodl->BoardId[0], RIO_IM_0, 3000);
if ( Result != RIO_OK )
{
   sprintf(s,"Changing time-out to other then default, failed. Result: 0x%X", Result);
   MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
}


// in this example, the trigger pin is set to interrupt on rise
Result = RioGpioControl(pRiodl->BoardId[0], RIO_GPIO_IRQ_RISE, RIO_GPIO_OUTPUT,
                                            RIO_GPIO_IGNORE, RIO_GPIO_IGNORE);
if ( Result != RIO_OK )
{
    sprintf(s, "General purpose pin could not be set.  Result: 0x%X", Result);
    MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
}
else
{
    Result = RioTriggerCapture(pRiodl->BoardId[0], RIO_IM_0, FALSE, TRUE,
                                    &FParams, &TBufferParams,
```

```
                                        &SourceRect, &DestRect, Pitch, &Overlapped);
    if ( Result != RIO_PENDING )
    {
        sprintf(s, "Starting capture failed. Result: 0x%X", Result);
        MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION );
    }
    else bOverlappedProcStarted = TRUE;
}

.. .. .. ..

// if (bOverlappedProcStarted), waiting for capture sequence to be finished can be
// done in a separate thread:
WFSO_Result= WaitForSingleObject(Overlapped.hEvent, 3000);
if ( WFSO_Result == WAIT_FAILED )
{
    sprintf(s, "WaitForSingleObject(Overlapped.hEvent, 3000) failed");
    MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION | MB_SETFOREGROUND);
}
else
{
    if ( WFSO_Result == WAIT_TIMEOUT )
    {
        sprintf(s, "The time-out has expired");
        MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION | MB_SETFOREGROUND);
    }
    else
    {
        Result = RioGetOverlappedResult(&Overlapped, &OverlapResult, TRUE);
        if ( (Result != RIO_OK) || (OverlapResult !=  RIO_OK) )
        {
           sprintf(s, "Error, Res: 0x%X, OverlapRes: 0x%X", Result, OverlapResult);
           MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION | MB_SETFOREGROUND);
        }
        else
        {
           // user can now visualize or save imagebuffer array to file
        }
    }
}

.. .. .. ..

// waiting for the capture sequence can be canceled
Result = RioTriggerCaptureCancel(pRiodl->BoardId[0], RIO_IM_0);
if ( Result != RIO_OK )
{
    sprintf(s, "RioTriggerCaptureCancel  failed. Result: 0x%X", Result);
    MessageBox(NULL, s, NULL, MB_OK | MB_ICONEXCLAMATION | MB_SETFOREGROUND);
}
else
{
    // now the thread where WaitForSingleObject is being done, can now be closed
}
```

RioTriggerCaptureCancel

#include <Rio.h>

Function int RioTriggerCaptureCancel(int *BoardId*, RIO_INPUT_MODULE *InputModule*);

Return Value RIO_OK successful

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_INPUT_MODULE	an invalid *InputModule* was given

Parameters

BoardId	BoardID of the Rio board on which to cancel the triggered capture.
InputModule	Input module for which to cancel the triggered capture. Can be either RIO_IM_0 or RIO_IM_1.

Remarks Cancels the triggered capture. No event will be returned after the capture is canceled. If the mode is RIO_HQ, RIO_FULL_FRAME or RIO_RGBA, only RIO_IM_0 should be used as value for *InputModule*.

Example See RioTriggerCapture Example 2.

RioTriggerCaptureTimeOut

#include <Rio.h>

Function int RioTriggerCaptureTimeOut(int *BoardId*, RIO_INPUT_MODULE *InputModule*, long *TimeOut*);

Return Value

RIO_OK	successful

parameter checking:

RIO_INVALID_BOARD_ID	an invalid *BoardId* was given
RIO_INVALID_INPUT_MODULE	an invalid *InputModule* was given

Parameters

BoardId	BoardID of the Rio board on which to set the trigger-capture time out.
InputModule	Input module for which to set the triggered capture time-out. Can be either RIO_IM_0 or RIO_IM_1.
TimeOut	The time-out to be set (in msec).

Remarks This function sets the time-out for the triggered capture. Default after initialization before this function is called for the first time is 5000 msec. This time-out will be used in RioTriggerCapture when called with *pOverlapped* =NULL.
If the mode is RIO_HQ, RIO_FULL_FRAME or RIO_RGBA, only RIO_IM_0 should be used as value for *InputModule*.

Example See RioTriggerCapture Example 1.

Using the software library with other compilers

Support for other compilers than the Visual C++ compiler is unofficial. A number of issues need to be handled when using the software library with other compilers such as Watcom C/C++ or Borland C compiler.

The following compiler options need to be set for all compilers:

- *char* must be set *signed* by default: (Watcom C/C++ compiler option: -j)
- enums must be 32 bit values: (Watcom C/C++ compiler option: -ei)

Watcom C/C++ compiler

When using the Watcom C/C++ compiler, the following change needs to be made in the *Rio.h* header file:

the line:

```
#define DllImport      __declspec( dllimport )
```

must be replaced with:

```
#define DllImport      __declspec( dllimport __cdecl)
```

which ensures standard C calling convention for the dll functions. A *Rio.h* header file where this change already has been made, can be found in the "Soft/Windows/Unsupported/Watcom" directory of the software library.

Borland C compiler

When using the Borland C compiler, linking must be done using the *Riod.def* file, which can be found in the "Soft/Windows/Unsupported/Borland" directory of the software library, instead of linking with *Riod.lib*.

Also, the following line needs to be added to the *Rio.h* header file:

```
#pragma option -pc
```

which ensures standard C calling convention for the dll functions. A *Rio.h* header file where this change already has been made, can be found in the "Soft/Windows/Unsupported/Borland" directory of the software library.

Functions for DOS

The 32-bit DOS library is setup the same way as the Windows library. It uses the exact same syntax.

However the following features are not currently supported under DOS:

- Streaming capture (the RioStream… functions).
- Triggered capture (the RioTrigger… functions).
- Postprocessing.
- Chroma keying

Downloading updates

Updates can be downloaded from our internet-site (www.ellips.nl).

HARDWARE REFERENCE

Architectural overview

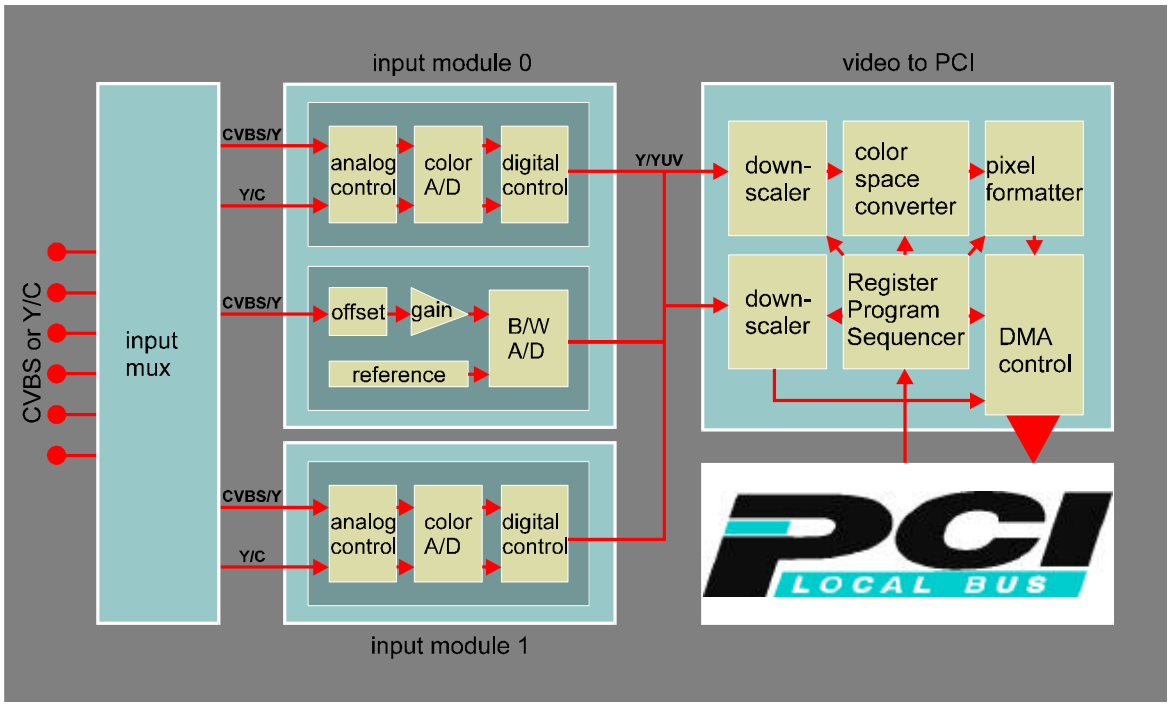


Figure 13. Rio hardware architecture

Figure 13 shows the architecture of the Rio board.

On a Basic Rio, only input module 0 is available (see 'Rio Modules'). Also the High Quality Black & White functionality of input module 0 is only available on the Full Rio.

Connected to the input modules are the two scalers; the High Performance Scaler (HPS) that consists of three parts (downscaler, color space converter and pixel formatter) and the more basic Binary Ratio Scaler. As mentioned in 'Camera Output, Offsets, Scaling', each input module can be connected to either scaler.

The scalers are connected to the DMA control (two DMA channels), which on its turn is connected to the PCI local bus.

Video Source Connectors

As shown in Figure 14, the Rio has six SMB connectors. The grabber accepts almost all kinds of standard video signals, including those from video recorders. Supported input signals: CVBS (CCIR and EIA), Y/C (S-video, S-VHS or Hi8 for standards PAL-B/G, NTSC-M and SECAM) and RGBA (CCIR and EIA).

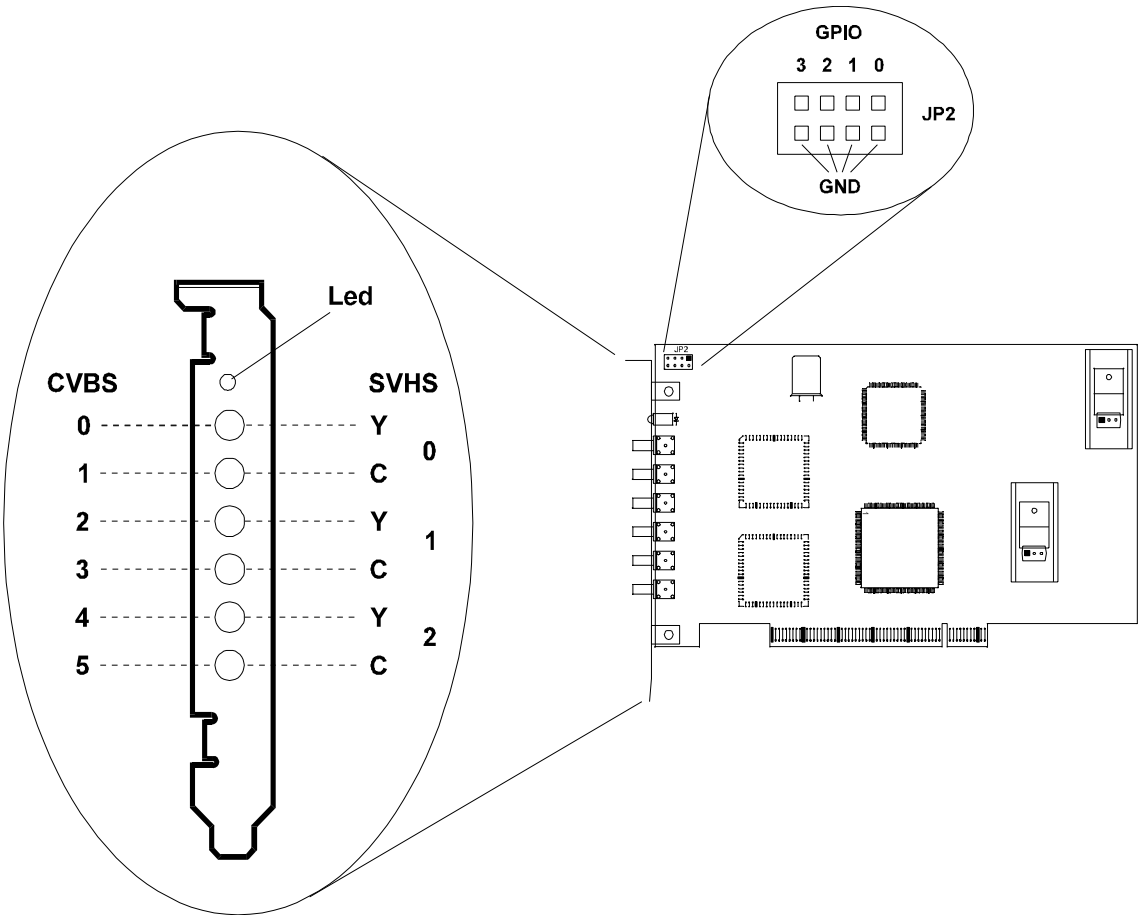


Figure 14. Video Source Connector and General Purpuse IO pin configuration

Video sources that can be connected:

- *Composite video sources* (CVBS). Up to six cameras can be connected.
- *Y/C connectors, S-Video sources* (S-VHS or Hi8). Up to three Y/C cameras can be connected, occupying two connectors per camera, an Y connector and the corresponding C connector.
- *Progressive Scan.* Up to three progressive scan cameras can be connected, each camera should have video output 1 connected to an Y connector and video output 2 to the corresponding C connector.
- One *RGBA* camera can be connected, with connectors R, G, B and A to respectively CVBS inputs 0 through 3.
- Combinations of the above mentioned video sources also can be connected, for example 2 CVBS cameras and 2 Y/C cameras can be connected at the same time. Of course, every Y/C camera needs to be connected to Y and C connectors on the Rio that belong together.

General Purpuse IO pins

As shown in Figure 14, the Rio contains 4 general purpose io pins. The pins can be used to trigger (on) external events. The pin numbering and ground are also indicated in Figure 14. The software library functions that can be used to control the pin functionality, are RioGpioControl, RioGpioRead and RioGpioWrite.

SPECIFICATIONS

Host Computer Interface

- High speed DMA burst transfer (up to 132MB/s)
- Programmable transfer controller
- Support for multiple boards in one system

Acquisition

- RS-170, CCIR, NTSC, PAL, SECAM, Y/C or monochrome inputs
- Resolutions: any size up to:
  - PAL, SECAM, CCIR native: 768x576
  - NTSC, EIA native: 640x480
- Output formats:
  - RGB32 (αRGB)
  - RGB24 (RGBR...)
  - RGB16 (5:6:5)
  - RGB15 (α:5:5:5 or 5:5:α:5)
  - RGB8 (3:3:2)
  - YUV16 (CCIR)
  - Y8 (256 grayscale)
  - Y2 (2 bit grayscale)
  - Y1 (B/W)

  Compensation of Gamma Pre-Correction is supported on RGB modes.

- Sample rate: 14.75 / 7.38 MHz for PAL / SECAM; 12.27 / 6.13 MHz for NTSC
- Digital PLL provides stable synchronization even when using still video cameras and VCRs
- Support for any combination of up to three Y/C video sources and six composite or monochrome video sources
- 6 SMB connectors for six monochrome, CVBS, three Y/C pairs (S-Video) or one RGB(A) camera

Software support

- Portable operating system independent C library
  Supported platforms:
  - DOS (32-bit): (including full source code)
    Watcom C/C++ (11.0) + Pharlap TNT
  - Windows 95/NT/2000 (32-bit):
    Microsoft Visual C 4.x / 5.x / 6.x
    VxD/device driver and DLL for C library
- Support for real-time operating systems: VxWorks
- Support for Image-Pro Plus (development tool)
- Capture and sample applications for DOS and Windows (RioWin demo program for Windows 95/NT/2000)

Specifications subject to change without notice.
For more information, feel free to contact Ellips.

KNOWN ISSUES

- **Architecture** Compaq Prosignia 300
  Platform(s) All
  Issue During capture at regular intervals a RIO_VIDEO_OVERFLOW error occurs.
  Cause The Compaq Prosignia 300 disrupts the PCI bus performance regularly (every few minutes).
  Workaround None

- **Architecture** Compaq Deskpro 590 in combination with Rio board with SAA7146 V1c chip
  Platform(s) All
  Issue Single capture and streaming capture work OK, but continuous and triggered capture are not reliable.
  Cause Sometimes a RPS jump instruction is not executed.
  Workaround None

- **Architecture** Pentium Pro with Intel Natoma 440 FX chipset
  Platform(s) All
  Issue During capture, RIO_VIDEO_OVERFLOW continuously occurs (some lines contain a lot of bad pixel values).
  Cause Unknown.
  Workaround None

- **Architecture** All systems with the Full Rio
  Platform(s) All
  Issue In RIO_RGBA mode noise occurs in the Red (and A-) values, when grabbing is done in field mode, or grabbing is done in frame mode with scaling.
  Cause Undetermined.
  Workaround Use RIO_RGBA in frame mode and do not scale. For the current version of the Rio software library, downscaling is not yet implemented anyway.

- **Architecture** All
  Platform(s) All
  Issue Except for the high quality black and white module on the Full Rio, the other modules generate an output range of 192 values, which will result in missing values when mapped to a 256 value scale.
  Cause The SAA 7110 chip only has an output range of 192 values.
  Workaround In case of black and white measurements, use the Full Rio in High Quality black and white mode, which covers the full 256 value range. For all other cases, develop your own post-processing algorithms to smoothen out the value gaps. The extra postprocessing will have an impact on the capture rate however.

- **Architecture** All
  Platform(s) All
  Issue Different special output format: In the module modes RIO_S_FULL_FRAME and RIO_STEREO_LOCKED the special output format is not as indicated in Figure 10 when capture is done bottom up (*TopDown*==FALSE):

  high … low in stead of high … low
  8 8 8 8 8 8 8 8
  B_1 A_1 B_0 A_0 A_1 B_1 A_0 B_0

  Cause Unknown.
  Workaround The post-processing by the driver compensates for this behaviour. In case the user wants to write his own post-processing algorithms, he has to take care of this issue himself.

- **Architecture** Rio board with SAA7146 chip
  Platform(s) DOS
  Issue Computers with more than 16 MB memory may experience capture problems. This behaviour does not occur for Rio boards with the SAA7146A chip.
  Cause RPS problems in the SAA7146 chip.
  Workaround The driver tries to fix this problem in release 1.5 and higher, but does not always succeed.

- **Architecture** All
  Platform(s) Windows 95
  Issue Sometimes RioScatterLock fails, (due to a _LinePageLock fail in Windows 95), although the memory provided to the function, was properly allocated.
  Cause Unknown.
  Workaround Do not use this memory anymore, but try again with a different piece of allocated memory.

- **Architecture** Rio board with SAA7146A chip
  Platform(s) All
  Issue In 30 % of all cases, after initialazation of the Rio, usage of the binary ratio scaler results in a horizontally shifted image. This behaviour does not occur for Rio boards with the SAA7146 chip.
  Cause A ‘feature’ in the SAA7146A chip.
  Workaround The problem is fixed in release 1.5 and higher. (But captures with source width < 512 will always exhibit the problem).

- **Architecture** Motherboard chipsets which demand full PCI 2.1 compliance (such as the Intel 820 chipset) in combination with Rio board with SAA7146 chip
  Platform(s) All
  Issue Problems range from black lines in captured images to systemhangs. Note that the Intel BX chipset works without problems. Also this behaviour does not occur for Rio boards with the SAA7146A chip.
  Cause The SAA7146 chip is not fully PCI 2.1 compliant.
  Workaround None.

INDEX

A

Applet 83

B

BoardId 24, 34, 35, 36, 37, 38, 39, 40, 41, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 65, 66, 67, 68, 69, 70, 71, 72, 74, 76, 77, 78, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 97, 98, 99, 100, 101, 102, 103, 104, 106, 107, 108, 109, 111, 112, 113, 114, 115
BoardIndex....73, 83, 94

C

CCIR....5, 6, 8, 11, 13, 14, 23, 85, 88, 95, 104, 120, 122
chipset....8, 124, 125
CVBS....4, 5, 6, 8, 9, 11, 22, 27, 31, 60, 92, 120, 122

D

Debi....44, 45
DestRect 14, 15, 34, 35, 36, 37, 38, 76, 100, 101, 104, 108, 109, 111, 112, 113
dialogbox 9
downscale 13, 14, 15, 35, 104, 109

E

eeprom....46, 47
EIA 5, 6, 8, 11, 13, 14, 23, 120, 122
enum....15, 17, 18, 19, 21, 22, 23, 62, 93

G

genlock....6
grayscale 4, 5, 122

H

Hi8....8, 9, 22, 120

I

ImageBuffer 24, 25, 101, 109, 111, 112
Image-Pro Plus 10, 123

J

jitter....22, 93

L

laserdisk....22

M

MessageBox....36, 37, 38, 43, 49, 50, 54, 55, 57, 58, 59, 61, 63, 65, 66, 68, 69, 70, 73, 74, 75, 79, 80, 83, 84, 86, 93, 94, 101, 102, 111, 112, 113
Module....11, 12, 27, 54, 57, 59, 61, 62, 63, 65, 66, 76, 85, 92, 93

N

NTSC....6, 8, 11, 13, 14, 15, 22, 59, 86, 90, 120, 122

O

offset....4, 7, 13, 58, 89
OutputFormat....27, 28, 29, 76, 92, 93
Overlapped....36, 37, 38, 49, 50, 100, 101, 102, 112, 113

P

PAL....6, 8, 11, 13, 14, 15, 22, 93, 104, 120, 122
PCI....4, 5, 8, 9, 10, 24, 119, 124, 125
PostProcess 19, 21, 28, 29, 92
PRIODL 24, 36, 38, 43, 49, 54, 55, 57, 58, 59, 60, 62, 65, 66, 67, 69, 70, 73, 74, 83, 85, 93, 94, 100, 111, 112

R

ResetEvent 101
RGB....4, 6, 7, 11, 15, 21, 122
RGBA 4, 7, 8, 9, 11, 12, 14, 15, 16, 19, 20, 21, 29, 36, 39, 40, 41, 55, 60, 76, 82, 86, 91, 92, 93, 97, 98, 100, 105, 106, 107, 110, 114, 115, 120, 124
Rio.h 34, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 62, 64, 65, 66, 67, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 81, 82, 83, 85, 87, 88, 89, 90, 91, 92, 94, 95, 96, 97, 98, 99, 103, 104, 106, 107, 108, 114, 115, 116
RIO_ALREADY_GETTING_BUFFER....31, 99
RIO_ALREADY_STARTED_CAPTURE 31, 34, 104, 106, 108
RIO_ALREADY_WATING_FOR_EXT_INT 31
RIO_BOARD_ID_WRITE_ERROR....31, 83
RIO_BW 11, 12, 18, 21, 23, 27, 28, 29, 54, 56, 57, 85, 86, 87, 88, 92
RIO_BW_OUTPUT_FORMAT **18**, 27, 28, 29
RIO_BW_VIDEO_STANDARD....**23**, 27, 28, 29
RIO_CAPTURE_ERROR....31, 34, 99, 108
RIO_COLOR....11, 12, 17, 21, 22, 23, 27, 29, 54, 56, 57, 59, 62, 63, 65, 85, 86, 87, 88, 90, 92, 93, 95
RIO_COLOR_OUTPUT_FORMAT **17**, 27
RIO_COLOR_VIDEO_STANDARD **22**, 27, 59, 90
RIO_CVBS_OR_YC **22**, 27
RIO_DEBI_READ_ERROR 31, 44
RIO_DEBI_WRITE_ERROR....31, 45
RIO_DEVICE_IO_CONTROL 31, 34, 48, 99, 108
RIO_DRIVER_NOT_LOADED....31, 75
RIO_ERR_DEBI_READ 30
RIO_ERR_DEBI_SWTIMEOUT 30
RIO_ERR_DEBI_WRITE....30
RIO_ERR_I2C....30
RIO_ERR_I2C_AL....30
RIO_ERR_I2C_APERR....30
RIO_ERR_I2C_BUSYTIMEOUT....30
RIO_ERR_I2C_DRERR 30

RIO_ERR_I2C_DTERR 30
RIO_ERR_I2C_RING_ADD....30
RIO_ERR_I2C_RING_REMOVE....30
RIO_ERR_I2C_SHORTTIMEOUT 30
RIO_ERR_I2C_SPERR....30
RIO_ERR_I2C_SPUR_IRQ....30
RIO_ERR_INIT....30, 74
RIO_ERR_PARAM....30, 77
RIO_ERR_PARAM_ALIGN....30
RIO_ERR_PARAM_BURST....30
RIO_ERR_PARAM_CHANNEL 30
RIO_ERR_PARAM_COUNT....30
RIO_ERR_PARAM_FORMAT 30
RIO_ERR_PARAM_FORMAT_BRS 30
RIO_ERR_PARAM_GAMMA 30
RIO_ERR_PARAM_IN_OFFSET....30
RIO_ERR_PARAM_IN_OFFSET_BRS....30
RIO_ERR_PARAM_IN_WIDTH_HEIGHT 30
RIO_ERR_PARAM_IN_WIDTH_HEIGHT_BRS 30
RIO_ERR_PARAM_MASK 30
RIO_ERR_PARAM_OUT_WIDTH_HEIGHT 30
RIO_ERR_PARAM_OUT_WIDTH_HEIGHT_BRS 30
RIO_ERR_PARAM_THRES....30
RIO_ERR_PARAM_VIDPORT....30
RIO_ERR_PARAM_WRITE 30
RIO_ERR_PARAM_XSCALE 30
RIO_ERR_PARAM_XSCALE_BRS 30
RIO_ERR_PARAM_XYSCALE....30
RIO_ERR_PARAM_YSCALE 30
RIO_ERR_PARAM_YSCALE_BRS 31
RIO_ERR_PARAMS....31, 77
RIO_ERR_PARAMS_GAIN....31
RIO_ERR_PARAMS_INPUT....31
RIO_ERR_PARAMS_VALUE 31
RIO_ERR_RING_ADJUST 31
RIO_ERR_SEM_CREATE....31
RIO_ERROR....31
RIO_FIELD_OR_FRAME... **21**, 23, 27, 28, 29, 35, 104, 109
RIO_FLASH_PARAMS **24**, 109, 110, 111, 112
RIO_FULL_FRAME ..... 15, 18, 21, 29, 36, 39, 40, 41, 54, ...55, 56, 57, 60, 82, 85, 86, 87, 88, 91, 92, ... 93, 97, 98, 100, 105, 106, 107, 110, 114, 115
RIO_GPIO_MODE....**23**, 67
RIO_GPIO_PIN....**23**, 24, 25, 48, 51, 52
RIO_HQ..... 11, 12, 14, 18, 21, 28, 29, 36, 39, 40, ... 41, 55, 58, 60, 61, 82, 85, 86, 89, 91, 92, 93, ... 97, 98, 100, 105, 106, 107, 110, 114, 115
RIO_INPUT_MODULE ....**22**, 27, 28, 34, 39, 40, 41, ... 54, 55, 57, 59, 60, 65, 66, 71, 72, 82, 85, 88, 90, ...91, 95, 97, 98, 99, 103, 104, 106, 107, 108, 114, 115
RIO_INSUFFICIENT_MEMORY 31, 75
RIO_INVALID_AUTO_GAIN_MODE....31, 91
RIO_INVALID_BOARD_ID ....31, 34, 39, 40, 41, 44, ... 45, 46, 47, 48, 51, 52, 53, 54, 55, 56, 57, 58, 59, ...60, 62, 65, 66, 67, 69, 70, 71, 72, 73, 74, 76, 77, ... 78, 82, 83, 85, 87, 88, 89, 90, 91, 92, 94, 95, 97, ...98, 99, 103, 104, 106, 107, 108, 114, 115
RIO_INVALID_CAMERA_FOR_INPUT_MODULE 31, 82
RIO_INVALID_CAPTURE_PERIOD....31, 109
RIO_INVALID_CVBS_OR_YC 31, 92
RIO_INVALID_DEST_RECT 31, 34, 76, 104, 109
RIO_INVALID_EEPROM_ADDRESS....31, 46, 47
RIO_INVALID_EEPROM_DATASIZE_FOR_ADDRESS 31, 46, 47
RIO_INVALID_EVENT....31, 75
RIO_INVALID_FIELD_OR_FRAME....31, 92
RIO_INVALID_FLASH_LINE 31, 108
RIO_INVALID_FLASH_PIN....32, 108
RIO_INVALID_GPIO_MODE....32, 67
RIO_INVALID_GPIO_PIN.... 32, 48, 51, 52
RIO_INVALID_HANDLE 32, 34, 76
RIO_INVALID_HQ_GAIN....32, 89
RIO_INVALID_HQ_OFFSET 32, 89
RIO_INVALID_I2C_REG 32, 71, 72, 77, 78
RIO_INVALID_IMAGE_BUFFER....32, 109
RIO_INVALID_INPUT_FOR_INPUT_MODULE 32, 60, 91
RIO_INVALID_INPUT_GAIN 32, 91
RIO_INVALID_INPUT_MODULE ..... 32, 34, 39, 40, ... 41, 54, 55, 57, 59, 60, 65, 66, 71, 72, 82, 85, ... 88, 90, 91, 92, 95, 97, 98, 99, 103, ... 104, 106, 107, 108, 114, 115
RIO_INVALID_LED_STATE....32, 73, 94
RIO_INVALID_MICROSEC_PER_CAPTURE 32, 104
RIO_INVALID_MODULE_MODE 32, 76, 92
RIO_INVALID_NR_BUFFERS....32, 109
RIO_INVALID_OUTPUT_FORMAT....32, 76, 92
RIO_INVALID_OVERLAPPED 32, 34, 64, 100
RIO_INVALID_POINTER.... 32, 43, 56, 64, 76, 92, 97, 103
RIO_INVALID_POST_PROCESS....32, 92
RIO_INVALID_SCALER....32, 92
RIO_INVALID_SRC_RECT..... 32, 34, 104, 109
RIO_INVALID_TBUFFER_PARAMS....32, 109
RIO_INVALID_TRIGGER_BUFFER_NR....32, 109
RIO_INVALID_TRIGGER_PIN 32, 109
RIO_INVALID_TRIGGER_POSITION 32, 109
RIO_INVALID_TV_OR_VTR....32, 92
RIO_INVALID_VIDEO_OUT_DELAY....32, 109
RIO_INVALID_VIDEO_STANDARD....32, 92
RIO_LOCK_FAILED 32, 79
RIO_MEM_ALLOC....32, 75
RIO_MODULE ....11, 16, 20, 21, 27, 28, ... **29**, 62, 63, 92, 93
RIO_MODULE_BW **27**, 29
RIO_MODULE_COLOR....**27**, 29
RIO_MODULE_FULL_FRAME **29**
RIO_MODULE_HQ....**28**, 29
RIO_MODULE_MODE 11, 20, **21**, 29, 62, 93
RIO_MODULE_RGBA....16, **29**
RIO_MODULE_S_FULL_FRAME 16, **28**, 29
RIO_MODULE_STEREO_LOCKED 16, **28**, 29
RIO_NO_FILLED_BUFFERS 32, 99, 102
RIO_OCF_REG_OUT_OF_RANGE....31
RIO_OK..... 32, 34, 35, 36, 37, 38, 39, 40, 41, 42, ...43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, ...56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, ... 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, ... 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, ... 95, 97, 98, 99, 100, 101, 102, 103, 104, 106, 107, ... 108, 110, 111, 112, 113, 114, 115
RIO_ON_OFF_MODE **21**, 28, 29, 60, 61, 73, 91, 94
RIO_PENDING....32, 34, 35, 37, 38, 48, 49, 64, ... 99, 100, 102, 108, 110, 113
RIO_RGBA....11, 12, 14, 15, 16, 19, 20, 21, 29, 36, 39, ... 40, 41, 55, 60, 76, 82, 86, 91, 92, 93, 97, 98, ... 100, 105, 106, 107, 110, 114, 115, 124

RIO_RGBA_OUTPUT_FORMAT....**19**, 29
RIO_S_FULL_FRAME ... 14, 20, 21, 28, 29, 76, 86, 92, 125
RIO_SCALER.... 15, 27, 28, 31
RIO_SCALER_BUSY 31
RIO_STEREO_LOCKED 20, 21, 28, 29, 76, 85, 86, 92, 125
RIO_TBUFFER_PARAMS **25**, 109, 110, 111, 112
RIO_TIMEOUT 31, 34, 35, 36, 48, 49, 108, 110, 111
RIO_TV_OR_VTR.... **22**, 27, 28, 29
RIO_UNABLE_TO_CREATE_FILE 32, 75
RIO_UNABLE_TO_MAP_FILE....32, 75
RIO_UNABLE_TO_POST_PROCESS....32, 34, 76, 99, 108
RIO_UNLOCK_FAILED....32, 81
RIO_VIDEO_HDR.... **24**, 97, 100, 101
RIO_VIDEO_OVERFLOW.... 32, 34, 35, 36, 64, ... 99, 100, 108, 110, 111, 124
RIO_VIDEO_OVERFLOW_NO_POST_PROCESS 32, 34, 99, 108
RIO_WAIT_FAILED....31, 34, 48, 64, 108
RioCapture.....13, 14, 15, **34**, 35, 36, 37, 38, 39, 40, 41, 64
RioCaptureCancel 37, 38, **39**
RioCaptureStop.... 35, 37, **40**
RioCaptureTimeOut 35, 36, 41, 43
RioClose **42**
RioCreateRiodl..... 36, 38, 41, **43**, 49, 53, 54, 55, 57, 58, ... 59, 60, 62, 65, 66, 67, 69, 70, 73, 74, 83, 85, ... 93, 94, 96, 100, 111, 112
RioDebiRead.... **44**
RioDebiWrite.... 44, **45**
RioEepromRead **46**
RioEepromWrite **47**
RioExternalInt.... **48**, 49, 51, 52, 64
RioExternalIntCancel 50, **51**
RioExternalIntTimeOut 48, 49, **52**
RioGetBoardType.... 43, **53**, 94
RioGetBrightness **54**
RioGetCamera.... **55**, 61, 86
RioGetChromaKey **56**
RioGetContrast.... **57**
RioGetHqBw.... **58**
RioGetHue **59**
RioGetInputGain **60**, 61
RioGetInputModule.... 27, 29, 54, 57, 59, ... 60, **62**, 63, 65, 66, 85
RioGetOverlappedResult.... 35, 37, 38, 48, 50, **64**, ... 100, 102, 110, 113
RioGetSaturation.... **65**
RioGetVideoStatus.... **66**
RioGpioControl.... 24, 25, 48, 49, **67**, 68, ... 70, 109, 111, 112, 121
RioGpioRead.... **69**, 70, 121
RioGpioWrite.... 67, 69, **70**, 121
RioI2cRead.... **71**
RioI2cWrite.... **72**
RioIndexSetLed.... **73**, 94
RioInitialize **74**
RioOpen.... **75**
RioPostProcess.... **76**
RioRegisterRead.... **77**
RioRegisterWrite.... **78**
RioScatterLock.... 34, 36, 38, **79**, 81, 99, ... 100, 108, 111, 112, 125
RioScatterUnlock 80, **81**
RioSelectCamera.... **82**, 93
RioSetBoardId.... 43, **83**, 96
RioSetBrightness..... 54, 55, **85**, 86, 88, 89, 90, 91, 95, 117
RioSetChromaKey.... 56, **87**, 93
RioSetContrast 56, 57, 86, 87, **88**
RioSetHqBw 58, 86, **89**
RioSetHue.... 59, 86, **90**
RioSetInputGain.... 60, 86, **91**
RioSetInputModule .... 27, 29, 36, 38, 55, 82, ... 91, **92**, 93, 100, 111, 112
RioSetLed 53, 73, **94**
RioSetSaturation 65, 86, **95**
RioSizeOfRiodl 43, 83, **96**
RioStreamAddEmptyBuffer 24, **97**, 101, 102
RioStreamClose.... **98**, 102, 105
RioStreamGetFilledBuffer .... 19, 24, 64, 97, 98, ... **99**, 101, 103, 105, 106, 107
RioStreamGetStartTime **103**
RioStreamInit.... 13, 14, 15, 101, **104**, 106
RioStreamStart.... 101, 104, **106**
RioStreamStop 99, 101, 102, **107**
RioTriggerCapture.... 13, 14, 15, 24, 25, 26, 64, **108**, ... 109, 111, 112, 114, 115
RioTriggerCaptureCancel 113, **114**
RioTriggerCaptureTimeOut.... 110, 111, 112, **115**

S

SAA7146.... 15, 44, 45, 77, 78, 124, 125
scaler.... 13, 15, 27, 28, 56, 87, 93, 119, 125
SECAM.... 6, 8, 11, 13, 14, 15, 22, 104, 120, 122
SourceRect/ SrcRect .... 13, 14, 35, 36, 37, 38, 100, ... 101, 104, 108, 109, 111, 112, 113
SquarePixels 13, 14, 15, 34, 35, 104, 108, 109
struct.... 11, 24, 25, 27, 28, 29, 35, 43, 48, 62, ... 64, 92, 93, 96, 97, 100, 109, 110
SVGA 5
S-VHS.... 8, 9, 22, 120

T

TimeOut.... 41, 52, 115
Topdown 35, 76, 104, 109
typedef 15, 17, 18, 19, 21, 22, 23, 24, 25, 27, 28, 29

W

WaitForSingleObject .... 34, 35, 37, 38, 40, 48, 49, 50, ... 99, 100, 102, 108, 110, 113

Y

Y/C 4, 5, 6, 8, 9, 11, 22, 60, 92, 93, 120, 122

References to definitions of enumerated types, structures and functions are marked in bold.

Artisan Technology Group is an independent supplier of quality pre-owned equipment

Gold-standard solutions

Extend the life of your critical industrial, commercial, and military systems with our superior service and support.

We buy equipment

Planning to upgrade your current equipment? Have surplus equipment taking up shelf space? We'll give it a new home.

Learn more!

Visit us at **artisantg.com** for more info on price quotes, drivers, technical specifications, manuals, and documentation.

Artisan Scientific Corporation dba Artisan Technology Group is not an affiliate, representative, or authorized distributor for any manufacturer listed herein.

We're here to make your life easier. How can we help you today?

(217) 352-9330 | sales@artisantg.com | artisantg.com

ARTISAN® TECHNOLOGY GROUP