HALCON Application Note

The Art of Image Acquisition

Provided Functionality

- ▷ Connecting to simple and complex configurations of frame grabbers and cameras

- ▷ Acquiring images in various timing modes

- ▷ Configuring frame grabbers and cameras online

Involved Operators

open_framegrabber
info_framegrabber
grab_image, grab_image_async, grab_image_start
set_framegrabber_param, get_framegrabber_param
close_framegrabber, close_all_framegrabbers

gen_image1, gen_image3, gen_image1_extern

Overview

Obviously, the acquisition of images is a task to be solved in all machine vision applications. Unfortunately, this task mainly consists of interacting with special, non-standardized hardware in form of the frame grabber board. To let you concentrate on the actual machine vision problem, HALCON already provides interfaces performing this interaction for a large number of frame grabbers (see section 1).

Within your HALCON application, the task of image acquisition is thus reduced to a few lines of code, i.e., a few operator calls, as can be seen in section 2. What's more, this simplicity is not achieved at the cost of limiting the available functionality: Using HALCON, you can acquire images from various configurations of frame grabbers and cameras (see section 3) in different timing modes (see section 5).

Unless specified otherwise, the example programs can be found in the subdirectory `image_acquisition` of the directory `%HALCONROOT%\examples\application_guide`. Note that most programs are preconfigured to work with a certain HALCON frame grabber interface; in this case, the name of the program contains the name of the interface. To use the program with another frame grabber, please adapt the parts which open the connection to the frame grabber. More example programs for the different HALCON frame grabber interfaces can be found in the subdirectory `hdevelop\Image\Framegrabber` of the directory `%HALCONROOT%\examples`.

Please refer to the HALCON/C User's Manual and the HALCON/C++ User's Manual for information about how to compile and link the C and C++ example programs; among other things, they describe how to use the example UNIX makefiles which can be found in the subdirectories `c` and `cpp` of the directory `%HALCONROOT%\examples`. Under Windows, you can use Visual Studio workspaces containing the examples, which can be found in the subdirectory `i586-nt4` parallel to the source files.

More information about HALCON can be found at:

```
http://www.mvtec.com/halcon/
```

Contents

1 The Philosophy Behind the HALCON Frame Grabber Interfaces

From the point of view of an user developing software for a machine vision application, the acquisition of images is only a prelude to the actual machine vision task. Of course it is important that images are acquired at the correct moment or rate, and that the camera and the frame grabber are configured suitably, but these tasks seem to be elementary, or at least independent of the used frame grabber.

The reality, however, looks different. Frame grabbers differ widely regarding the provided functionality, and even if their functionality is similar, the SDKs (*software development kit*) provided by the frame grabber manufacturers do not follow any standard. Therefore, if one decides to switch to a different frame grabber, this probably means to rewrite the image acquisition part of the application.

HALCON's answer to this problem are its *frame grabber interfaces* (HFGI) which are provided for currently more than 50 frame grabbers in form of *dynamically loadable libraries* (Windows NT/2000/XP: DLLs; UNIX: shared libraries). HALCON frame grabber interfaces bridge the gap between the individual frame grabbers and the HALCON library, which is independent of the used frame grabber, computer platform, and programming language (see figure 1). In other words, they

- provide a standardized interface to the HALCON user in form of 11 HALCON operators, and

- encapsulate details specific to the frame grabber, i.e., the interaction with the frame grabber SDK provided by the manufacturer.

Therefore, if you decide to switch to a different frame grabber, all you need to do is to install the corresponding driver and SDK provided by the manufacturer and to use different parameter values when calling the HALCON operators; the operators themselves stay the same.
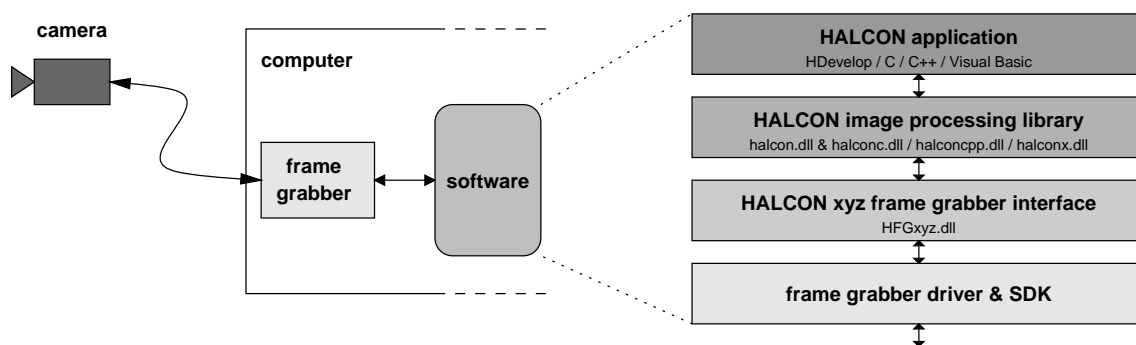

Figure 1: From the camera to a HALCON application.

In fact, the elementary tasks of image acquisition are covered by two HALCON operators:

- `open_framegrabber` connects to the frame grabber and sets general parameters, e.g., the type of the used camera or the port the camera is connected to, then

- `grab_image` (or `grab_image_async`, see section 5.1 for the difference) grabs images.

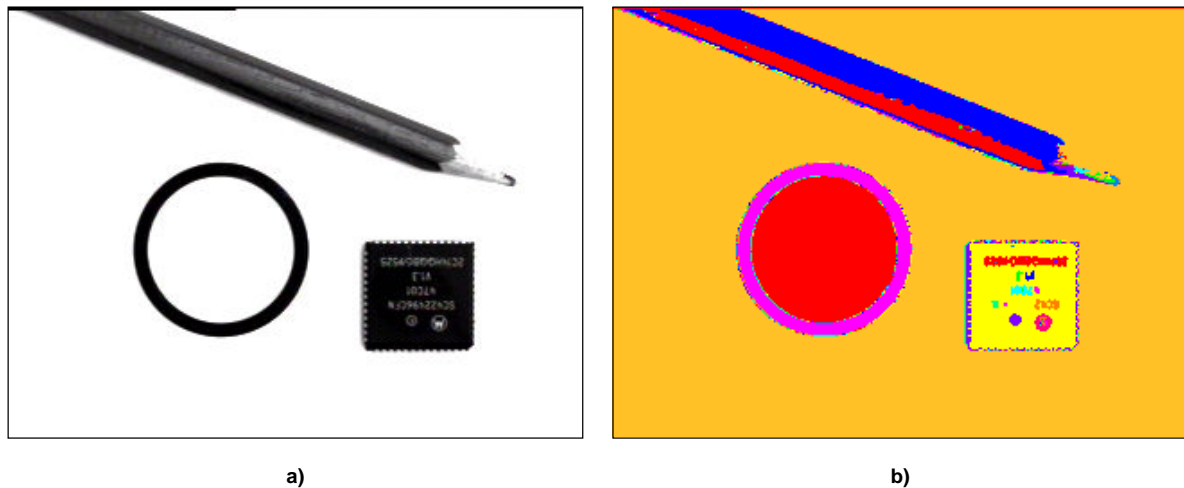<div align="center">a)   b)</div>

Figure 2: a) Acquired image; b) processed image (automatic segmentation).

If a frame grabber provides additional functionality, e.g., on-board modification of the image signal, special grabbing modes, or digital output lines, it is available via the operator `set_framegrabber_param` (see section 4).

Note, that for some frame grabbers not the full functionality is available within HALCON; please refer to the corresponding online documentation which can be found in the directory `%HALCONROOT%\doc\html\manuals` or via the HALCON folder in the Windows start menu (if you installed the documentation). The latest information can be found under `http://www.mvtec.com/halcon/framegrabber`.

If the frame grabber you want to use is not (yet) supported by HALCON, you can nevertheless use it together with HALCON. Please refer to section 6.1 for more details.

2 A First Example

In this section we start with a simple image acquisition task, which uses the frame grabber in its default configuration and the standard grabbing mode. The grabbed images are then segmented. To follow the example actively, start the HDevelop program `hdevelop\first_example_acquisition_ids.dev`; the steps described below start after the initialization of the application (press F5 once to reach this point). Note that the program is preconfigured for the HALCON frame grabber interface IDS; to use it with a different frame grabber, please adapt the parts which open the connection.

Step 1:   Connect to the frame grabber

```
open_framegrabber (FGName, 1, 1, 0, 0, 0, 0, 'default', -1, 'gray', -1,
                   'false', 'ntsc', 'default', -1, -1, FGHandle)
```

When opening the connection to your frame grabber using the operator `open_framegrabber`, the main parameter is the `Name` of the corresponding HALCON frame grabber interface. As a result, you obtain a so-called *handle* (`FGHandle`) which acts as your access to the frame grabber, e.g., in calls to the operator `grab_image`.

In the example, default values are used for most other parameters ('default' or -1); section 4.1 takes a closer look at this topic. How to connect to more complex frame grabber and camera configurations is described in section 3.

Step 2:   Grab an image

```
grab_image (Image, FGHandle)
```

After successfully connecting to your frame grabber you can grab images by calling the operator `grab_image` with the corresponding handle `FGHandle`. More advanced modes of grabbing images are described in section 5.

Step 3:   Grab and process images in a loop

```
while (Button # 1)
  grab_image (Image, FGHandle)
  auto_threshold (Image, Regions, 4)
  connection (Regions, ConnectedRegions)
  get_mposition (WindowHandleButton, Row, Column, Button)
endwhile
```

In the example, the grabbed images are then automatically segmented using the operator `auto_threshold` (see figure 2). This is done in a loop which can be exited by clicking into a window with the left mouse button.

3   Connecting to Your Frame Grabber

In this section, we show how to connect to different configurations of frame grabber(s) and camera(s), ranging from the simple case of one camera connected to one frame grabber board to more complex ones, e.g., multiple synchronized cameras connected to one or more boards.

3.1   Opening a Connection to a Specified Configuration

With the operator `open_framegrabber` you open a connection to a frame grabber, or to be more exact, via a frame grabber to a camera. This connection is described by four parameters (see figure 3): First, you select a frame grabber (family) with the parameter `Name`. If multiple boards are allowed, you can select one with the parameter `Device`; depending on the frame grabber interface, this parameter can contain a string describing the board or simply a number (in form of a string!).

Typically, the camera can be connected to the frame grabber at different ports, whose number can be selected via the parameter `Port` (in rare cases `LineIn`). The parameter `CameraType` describes the connected camera: For analog cameras, this parameter usually specifies the used signal norm, e.g., 'ntsc'; more complex frame grabber interfaces use this parameter to select a camera configuration file.

As a result, `open_framegrabber` returns a *handle* for the opened connection in the parameter `FGHandle`. Note that if you use HALCON's COM or C^{++} interface and call the operator via the classes `HFramegrabberX` or HFramegrabber, no handle is returned because the instance of the class itself acts as your handle.
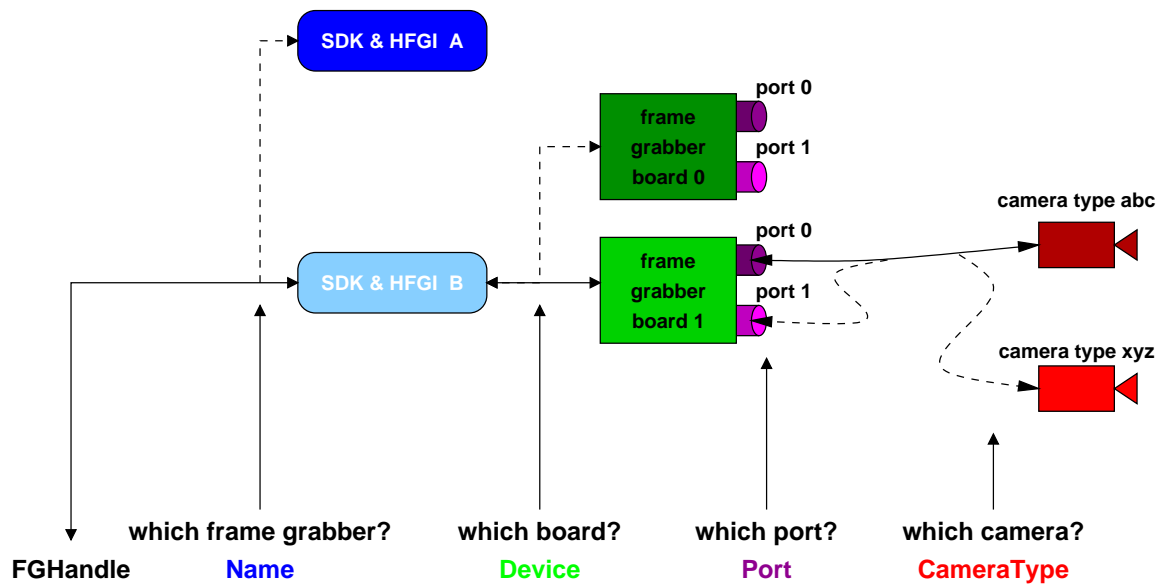
Figure 3: Describing a connection with the parameters of `open_framegrabber` .
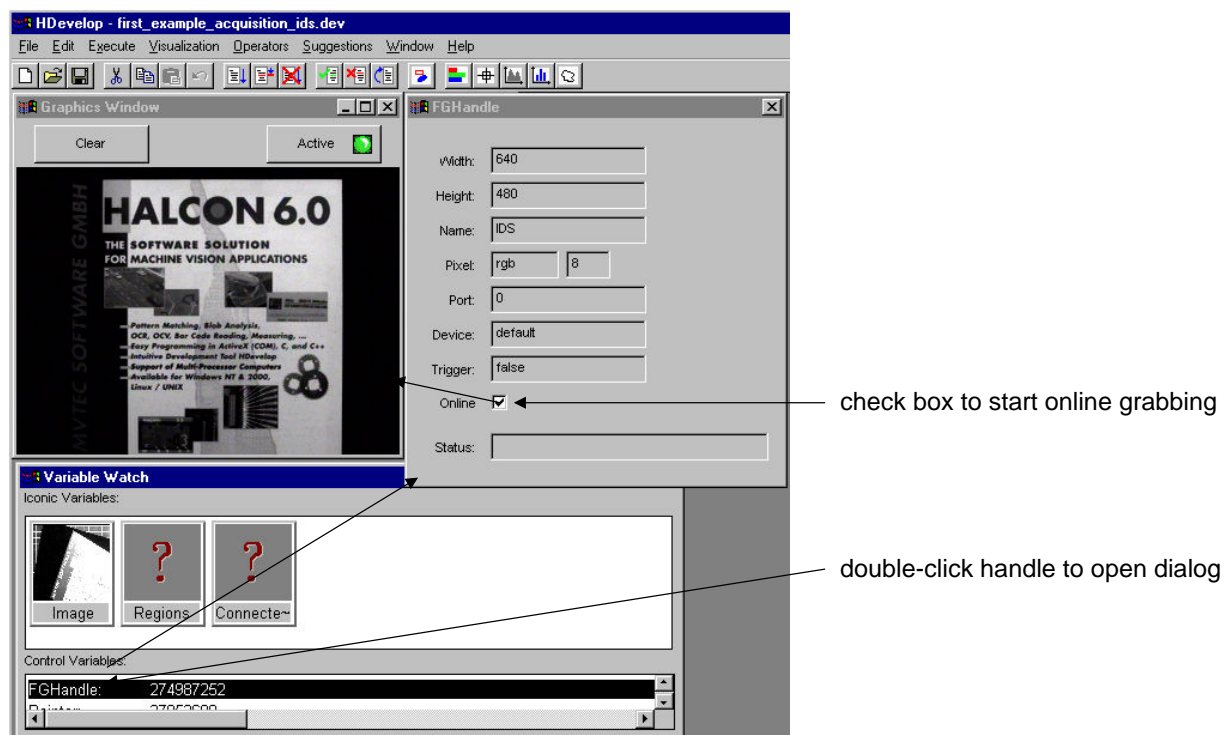
Figure 4: Online grabbing in HDevelop .

In HDevelop, you can quickly check an opened connection by double-clicking FGHandle in the Variable Window as shown in figure 4. A dialog appears which describes the status of the connection. If you check the corresponding box, images are grabbed online and displayed in the Graphics Window. This mode is very useful to setup your vision system (illumination, focus, field of view).

3.2   Connecting to Multiple Boards and Cameras

Most HALCON frame grabbers interfaces allow to use multiple frame grabber boards and cameras. However, there is more than one way to connect cameras and boards and to access these configurations from within HALCON. Below, we describe the different configurations; please check the online documentation of the HALCON interface for your frame grabber (see `%HALCONROOT%\doc\html\manuals`, the HALCON folder in the Windows start menu, or `http://www.mvtec.com/halcon/framegrabber`) which configurations it supports.
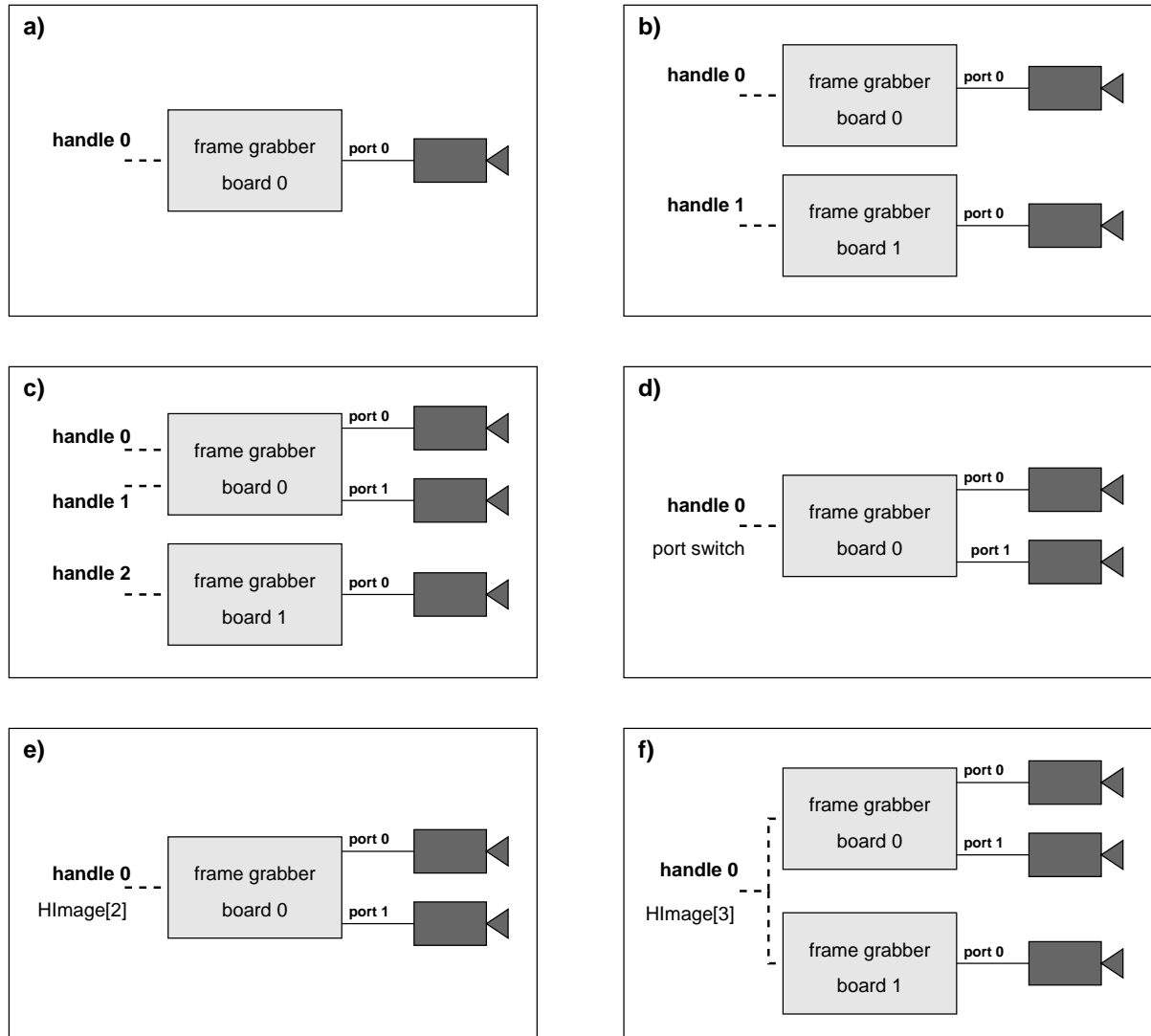
Figure 5: a) single board with single camera; b) multiple boards with one camera each; c) multiple boards with one or more cameras; d) single board with multiple cameras and port switching; e) single board with multiple cameras and simultaneous grabbing; f) simultaneous grabbing with multiple boards and cameras.

3.2.1   Single Camera

Figure 5a shows the simplest configuration: a single camera connected to a single board, accessible via a single handle. Some frame grabbers, especially digital ones, only support this

configuration; as described in the following section, you can nevertheless use multiple cameras with such frame grabbers by connecting each one to an individual board.

3.2.2 Multiple Boards

Figure 5b shows a configuration with multiple cameras, each connected to a separate board. In this case you call the operator `open_framegrabber` once for each connection as in the HDevelop example program hdevelop\multiple_boards_px.dev. Note that the program is preconfigured for the HALCON Px interface; to use it with a different frame grabber, please adapt the parts which open the connection.

```
open_framegrabber (FGName, 1, 1, 0, 0, 0, 0, 'default', -1, 'default', -1,
                   'default', 'default', Board0, -1, -1, FGHandle0)
open_framegrabber (FGName, 1, 1, 0, 0, 0, 0, 'default', -1, 'default', -1,
                   'default', 'default', Board1, -1, -1, FGHandle1)
```

In this example, the two calls differ only in the value for the parameter `Device` ('0' and '1'); of course, you can use different values for other parameters as well, and even connect to different frame grabber interfaces.

To grab images from the two cameras, you simply call the operator `grab_image` once with the two handles returned by the two calls to `open_framegrabber`:

```
grab_image (Image0, FGHandle0)
grab_image (Image1, FGHandle1)
```

3.2.3 Multiple Handles Per Board

Many frame grabbers provide multiple input ports and thus allow to connect more than one camera to the board. Depending on the HALCON frame grabber interface, this configuration is accessed in different ways which are described in this and the following sections.

The standard HALCON method to connect to the cameras is depicted in figure 5c: Each connection gets its own handle, i.e., `open_framegrabber` is called once for each camera with different values for the parameter `Port`, like in the HDevelop example program hdevelop\multiple_ports_px.dev (preconfigured for the HALCON Px interface, please adapt the parts which open the connection for your own frame grabber):

```
open_framegrabber (FGName, 1, 1, 0, 0, 0, 0, 'default', -1, 'default', -1,
                   'default', 'default', 'default', Port0, -1, FGHandle0)
open_framegrabber (FGName, 1, 1, 0, 0, 0, 0, 'default', -1, 'default', -1,
                   'default', 'default', 'default', Port1, -1, FGHandle1)
grab_image (Image0, FGHandle0)
grab_image (Image1, FGHandle1)
```

As figure 5c shows, you can also use multiple boards with multiple connected cameras.

3.2.4 Port Switching

Some frame grabber interfaces access the cameras not via multiple handles, but by switching the input port dynamically (see figure 5d). Therefore, `open_framegrabber` is called only once,

like in the HDevelop example program `hdevelop\port_switching_inspecta.dev` (precon-
figured for the HALCON `Inspecta` interface, please adapt the parts which open the connection
for your own frame grabber):

```
Port1 := 4
open_framegrabber (FGName, 1, 1, 0, 0, 0, 0, 'default', -1, 'default', -1,
```

Between grabbing images you switch ports using the operator `set_framegrabber_param` (see
section 4.2 for more information about this operator):

```
while (1)
set_framegrabber_param (FGHandle, 'port', Port0)
disp_image (Image0, WindowHandle0)
set_framegrabber_param (FGHandle, 'port', Port1)
```

Note that port switching only works for compatible (similar) cameras because
`open_framegrabber` is only called once, i.e., the same set of parameters values is used
for all cameras. In contrast, when using multiple handles as described above, you can specify
different parameter values for the individual cameras (with some board-specific limitations).

3.2.5   Simultaneous Grabbing

In the configurations described above, images were grabbed from the individual cameras by
multiple calls to the operator `grab_image`. In contrast, some frame grabber interfaces allow
to grab images from multiple cameras with a single call to `grab_image`, which then returns
a multi-channel image (see figure 5e; appendix A.1 contains more information about multi-
channel images). This mode is called *simultaneous grabbing* (or *parallel grabbing*); like port
switching, it only works for compatible (similar) cameras. For example, you can use this mode
to grab synchronized images from a stereo camera system.

In this mode, `open_framegrabber` is called only once, as can be seen in the HDevelop ex-
ample program `hdevelop\simultaneous_grabbing_inspecta.dev` (preconfigured for the
HALCON `Inspecta` interface, please adapt the parts which open the connection for your own
frame grabber):

```
               TM-6701/6705 1-plane, HD out'
open_framegrabber (FGName, 1, 1, 0, 0, 0, 0, 'default', -1, 'default', -1,
```

You can check the number of returned images (channels) using the operator `count_channels`

```
* step 2: open correctly sized windows
get_image_pointer1 (SimulImages, Pointer, Type, Width, Height)
```

and extract the individual images, e.g., using `decompose2`, `decompose3` etc., depending on the
number of images:

```
grab_image (SimulImages, FGHandle)
if (num_channels = 2)
```

Alternatively, you can convert the multi-channel image into an image array using
`image_to_channels` and then select the individual images via `select_obj`.

Note that some frame grabber interfaces allow simultaneous grabbing also for multiple boards
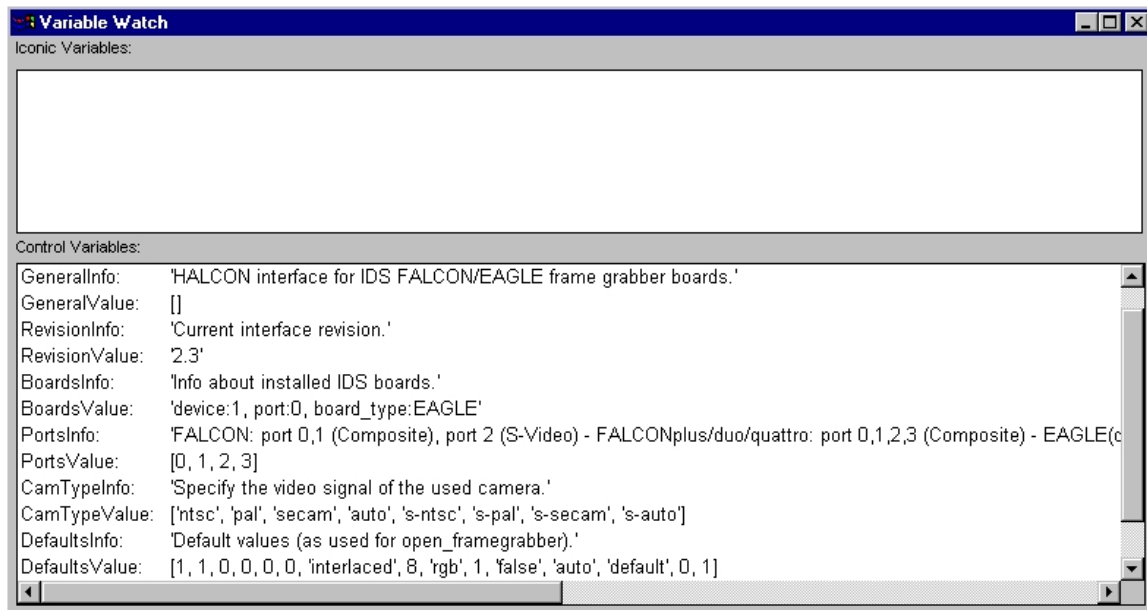(see figure 5f). Please refer to section 5.3.2 for additional information.

Figure 6: An example result of the operator `info_framegrabber` .

3.3   Requesting Information About the Frame Grabber Interface

As mentioned already, the individual HALCON frame grabber interfaces are described in detail on HTML pages which can be found in the directory `%HALCONROOT%\doc\html\manuals` or in the HALCON folder in the Windows start menu (if you installed the documentation). Another way to access information about a frame grabber interface is to use the operator `info_framegrabber`.

In the HDevelop example program `hdevelop\info_framegrabber_ids.dev` (preconfigured for the HALCON IDS interface, please adapt the interface name for your own frame grabber) this operator is called multiple times to query the version number of the interface, the available boards, port numbers, camera types, and the default values for all parameters of `open_framegrabber`; the result, i.e., the values displayed in the HDevelop Variable Windows, is depicted in figure 6.

```
info_framegrabber (FGName, 'general', GeneralInfo, GeneralValue)
info_framegrabber (FGName, 'revision', RevisionInfo, RevisionValue)
info_framegrabber (FGName, 'info_boards', BoardsInfo, BoardsValue)
info_framegrabber (FGName, 'ports', PortsInfo, PortsValue)
info_framegrabber (FGName, 'camera_types', CamTypeInfo, CamTypeValue)
info_framegrabber (FGName, 'defaults', DefaultsInfo, DefaultsValue)
```

The operator `info_framegrabber` can be called before actually connecting to a frame grabber with `open_framegrabber`. The only condition is that the HALCON frame grabber interface and the frame grabber SDK and driver have been installed.

4   Configuring the Acquisition

As explained in section 1, the intention of HALCON's frame grabber interfaces is to provide the user with a common interface for many different frame grabbers. This interface was kept as simple as possible; as shown, you can connect to your frame grabber and grab a first image using only two operators.

However, HALCON's second goal is to make the full functionality of a frame grabber available to the user. As frame grabbers differ widely regarding the provided functionality, this is a difficult task to realize within a simple, common interface. HALCON solves this problem by dividing the task of configuring a frame grabber connection into two parts: Those parameters which are common to most frame grabber interfaces (therefore called *general parameters*) are set when calling the operator `open_framegrabber`. In contrast, the functionality which is not generally available can be configured by setting so-called *special parameters* using the operator `set_framegrabber_param`.

4.1   General Parameters

When opening a connection via `open_framegrabber`, you can specify the following general parameters:

`HorizontalResolution`, `VerticalResolution`	spatial resolution of the transferred image in relation to the original size (see appendix B.1)
`ImageWidth`, `ImageHeight`, `StartRow`, `StartColumn`	size and upper left corner of the transferred image in relation to the original size (see appendix B.1)
`Field`	grabbing mode for analog cameras, e.g., interlaced-scan, progressive-scan, field grabbing (see appendix B.2)
`BitsPerChannel`, `ColorSpace`	data contained in a pixel (number of bits, number of channels, color encoding, see appendix B.3)
`Gain`	amplification factor for the video amplifier on the frame grabber board (if available)
`ExternalTrigger`	hooking the acquisition of images to an external trigger signal (see also section 5.2)
`CameraType`, `Device`, `Port`, `LineIn`	Configuration of frame grabber(s) and camera(s) from which images are to be acquired (see section 3.1).

In section 3.1, we already encountered the parameters describing the frame grabber / camera configuration. Most of the other parameters of `open_framegrabber` specify the image format; they are described in more detail in appendix B. The parameter `ExternalTrigger` activates a special grabbing mode which is described in detail in section 5.2. Finally, the parameter `Gain` can be used to manipulate the acquired images on the frame grabber board by configuring the video amplifier.

Note that when calling `open_framegrabber` you must specify values for all parameters, even if your frame grabber interface does not support some of them or uses values specified in a camera

configuration file instead. To alleviate this task, the HALCON frame grabber interfaces provide
suitable default values which are used if you specify 'default' or -1 for string or numeric
parameters, respectively. The actually used default values can be queried using the operator
`info_framegrabber` as shown in section 3.3.

After connecting to a frame grabber, you can query the current value of general parameters
using the operator `set_framegrabber_param`; some interface even allow to modify general
parameters dynamically. Please refer to section 4.3 for more information about these topics.

4.2 Special Parameters

Even the functionality which is not generally available for all frame grabber can be accessed
and configured with a general mechanism: by setting corresponding special parameters via the
operator `set_framegrabber_param`. Typical parameters are, for example:

`'grab_timeout'`	timeout after which the operators `grab_image` and `grab_image_async` stop waiting for an image and return an error (see also sections 5.2.1 and 6.2)
`'volatile'`	enable volating grabbing (see also section 5.1.3)
`'continuous_grabbing'`	switch on a special acquisition mode which is necessary for some frame grabbers to achieve real-time performance (see also section 5.1.5)
`'trigger_signal'`	signal type used for external triggering, e.g., rising or falling edge
`'image_width'`, `'image_height'`, `'start_row'`, `'start_column'`, `'gain'`, `'external_trigger'`, `'port'`	"doubles" of the some of the general parameters described in section 4.1, allowing to modify them dynamically, i.e., after opening the connection (see also section 4.3)

Depending on the frame grabber, various other parameters may be available, which allow, e.g.,
to add an offset to the digitized video signal or modify the brightness or contrast, to specify
the exposure time or to trigger a flash. Some frame grabbers also offer special parameters for
the use of line scan cameras (see also section 6.3), or parameters controlling digital output and
input lines.

Which special parameters are provided by a frame grabber interface is described in the al-
ready mentioned online documentation. You can also query this information by calling the
operator `info_framegrabber` as shown below; figure 7 depicts the result of double-clicking
`ParametersValue` in the Variable Window after executing the line:

```
info_framegrabber (FGName, 'parameters', ParametersInfo, ParametersValue)
```

To set a parameter, you call the operator `set_framegrabber_param`, specifying the name of
the parameter to set in the parameter `Param` and the desired value in the parameter `Value`. For
example, in section 3.2.4 the following line was used to switch to port 0:

```
while (1)
```

You can also set multiple parameters at once by specifying tuples for `Param` and `Value` as in
the following line:
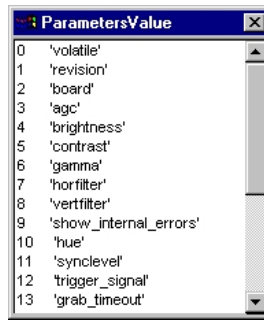
Figure 7: Querying available special parameters via `info_framegrabber` .

```
set_framegrabber_param (FGHandle, ['image_width','image_height'], [256,
                        256])
```

For all parameters which can be set with `set_framegrabber_param`, you can query the current value using the operator `get_framegrabber_param`. Some interfaces also allow to query additional information like minimum and maximum values for the parameters. For example, the HALCON `Fire-i` interface allows to query the minimum and maximum values for the brightness:

```
get_framegrabber_param (FGHandle, 'brightness_min_value', MinBrightness)
get_framegrabber_param (FGHandle, 'brightness_max_value', MaxBrightness)
```

Thus, you can check a new brightness value against those boundaries before setting it:

```
get_framegrabber_param (FGHandle, 'brightness', CurrentBrightness)
NewBrightness := CurrentBrightness + 10
if (NewBrightness > MaxBrightness)
  NewBrightness := MaxBrightness
endif
set_framegrabber_param (FGHandle, 'brightness', NewBrightness)
```

4.3   Fixed vs. Dynamic Parameters

The distinction between fixed and dynamic parameters is made relating to the lifetime of a frame grabber connection. *Fixed parameters*, e.g., the `CameraType`, are set once when opening the connection with `open_framegrabber`. In contrast, those parameters which can be modified via `set_framegrabber_param` during the use of the connection are called *dynamic parameters*.

As already noted in section 4.2, some frame grabber interfaces allow to modify general parameters like `ImageWidth` or `ExternalTrigger` dynamically via `set_framegrabber_param`, by providing a corresponding special parameter with the same name but written with small letters and underscores, e.g., 'image_width' or 'external_trigger'.

Independent of whether a general parameter can be modified dynamically, you can query its current value by calling the operator `get_framegrabber_param` with its "translated" name, i.e., capitals replaced by small letters and underscores as described above.

5 The Various Modes of Grabbing Images

Section 2 showed that grabbing images is very easy in HALCON– you just call `grab_image`!
But of course there's more to image grabbing than just to get an image, e.g., how to assure an
exact timing. This section therefore describes more complex grabbing modes.

5.1 Real-Time Image Acquisition

As a technical term, the attribute *real-time* means that a process guarantees that it meets given
deadlines. Please keep in mind that **none of the standard operating systems, i.e., neither**
Windows NT/2000/XP nor Linux, are real-time operating systems. This means that the op- !
erating system itself does not guarantee that your application will get the necessary processing
time before its deadline expires. From the point of view of a machine vision application running
under a non-real-time operating system, the most you can do is assure that real-time behavior is
not already prevented by the application itself.

In a machine vision application, real-time behavior may be required at multiple points:

Image delay: The camera must "grab" the image, i.e., expose the chip, at the correct moment,
i.e., while the part to be inspected is completely visible.

Frame rate: The most common real-time requirement for a machine vision application is to
"reach frame rate", i.e., acquire and process all images the camera produces.

Processing delay: The image processing itself must complete in time to allow a reaction to its
results, e.g., to remove a faulty part from the conveyor belt. As this point relates only
indirectly to the image acquisition it is ignored in the following.

5.1.1 Non-Real-Time Grabbing Using *grab_image*

Figure 8 shows the timing diagram for the standard grabbing mode, i.e., if you use the operator
`grab_image` from within your application. This operator call is "translated" by the HALCON
frame grabber interface and the SDK into the corresponding signal to the frame grabber board
(marked with *'Grab'*).

The frame grabber now waits for the next image. In the example, a free-running analog
progressive-scan camera is used, which produces images continuously at a fixed frame rate;
the start of a new image is indicated by a so-called *vertical sync signal*. The frame grabber then
digitizes the incoming analog image signal and transforms it into an image matrix. If a digital
camera is used, the camera itself performs the digitizing and transfers a digital signal which is
then transformed into an image matrix by the frame grabber. Please refer to appendix B.2 for
more information about interlaced grabbing.

The image is then transferred from the frame grabber into computer memory via the PCI bus
using *DMA* (direct memory access). This transfer can either be *incremental* as depicted in
figure 8, if the frame grabber has only a FIFO buffer, or in a single burst as depicted in figure 9,
if the frame grabber has a frame buffer on board. The advantage of the incremental transfer is
that the transfer is concluded earlier. In contrast, the burst mode is more efficient; furthermore,
if the incremental transfer via the PCI bus cannot proceed for some reason, a FIFO overflow
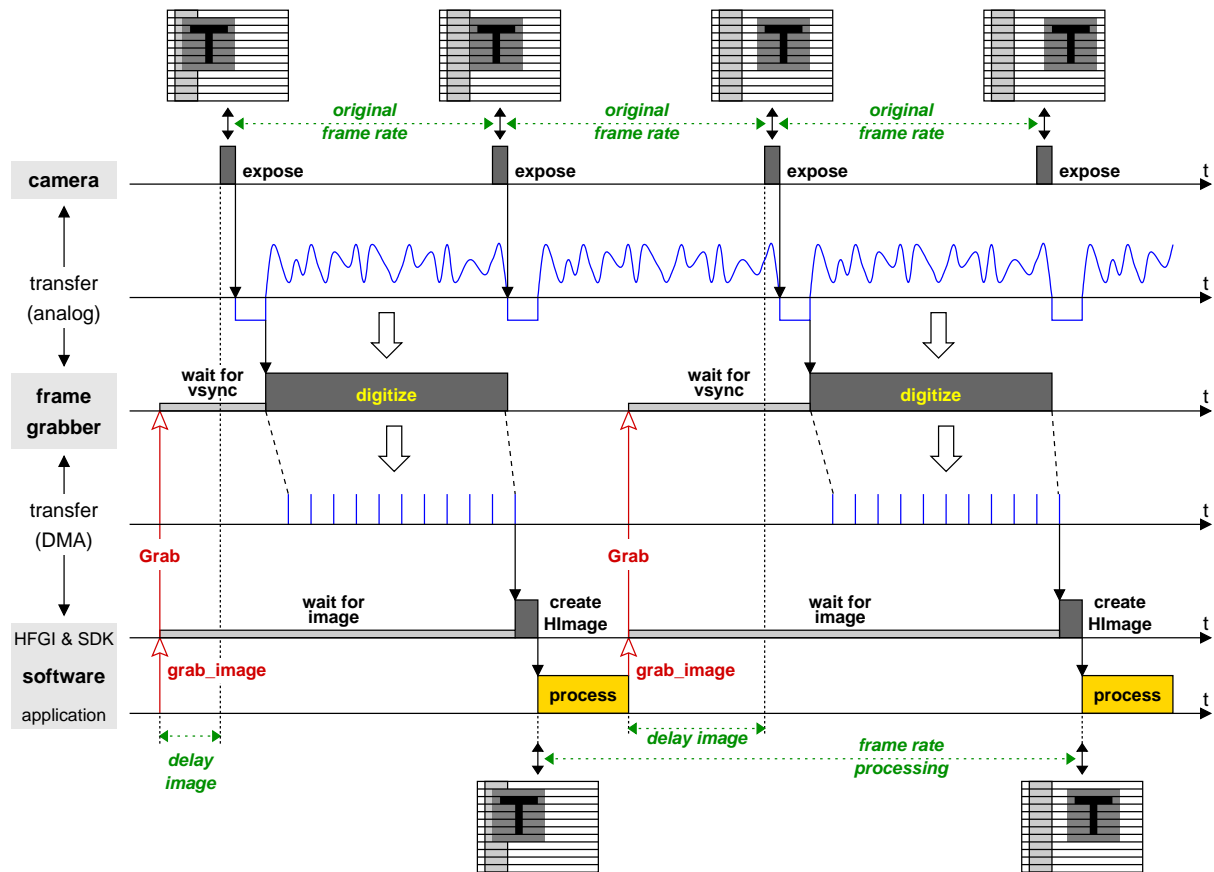
Figure 8: Standard timing using `grab_image` (configuration: free-running progressive-scan camera, frame grabber with incremental image transfer).

results, i.e., image data is lost. Note that in both modes the transfer performance depends on whether the PCI bus is used by other devices as well!

When the image is completely stored in the computer memory, the HALCON frame grabber interface transforms it into a HALCON image and returns the control to the application which processes the image and then calls `grab_image` again. However, even if the processing time is short in relation to the frame rate, the camera has already begun to transfer the next image which is therefore "lost"; the application can therefore only process every second image.

You can check this behavior using the HDevelop example program `hdevelop\real_time_grabbing_ids.dev` (preconfigured for the HALCON IDS interface, please adapt the parts which open the connection for your own frame grabber), which determines achievable frame rates for grabbing and processing (here: calculating a difference image) first separately and then together as follows:

```
grab_image (BackgroundImage, FGHandle)
count_seconds (Seconds1)
for i := 1 to 20 by 1
  grab_image (Image, FGHandle)
  sub_image (BackgroundImage, Image, DifferenceImage, 1, 128)
endfor
count_seconds (Seconds2)
TimeGrabImage := (Seconds2-Seconds1)/20
FrameRateGrabImage := 1 / TimeGrabImage
```
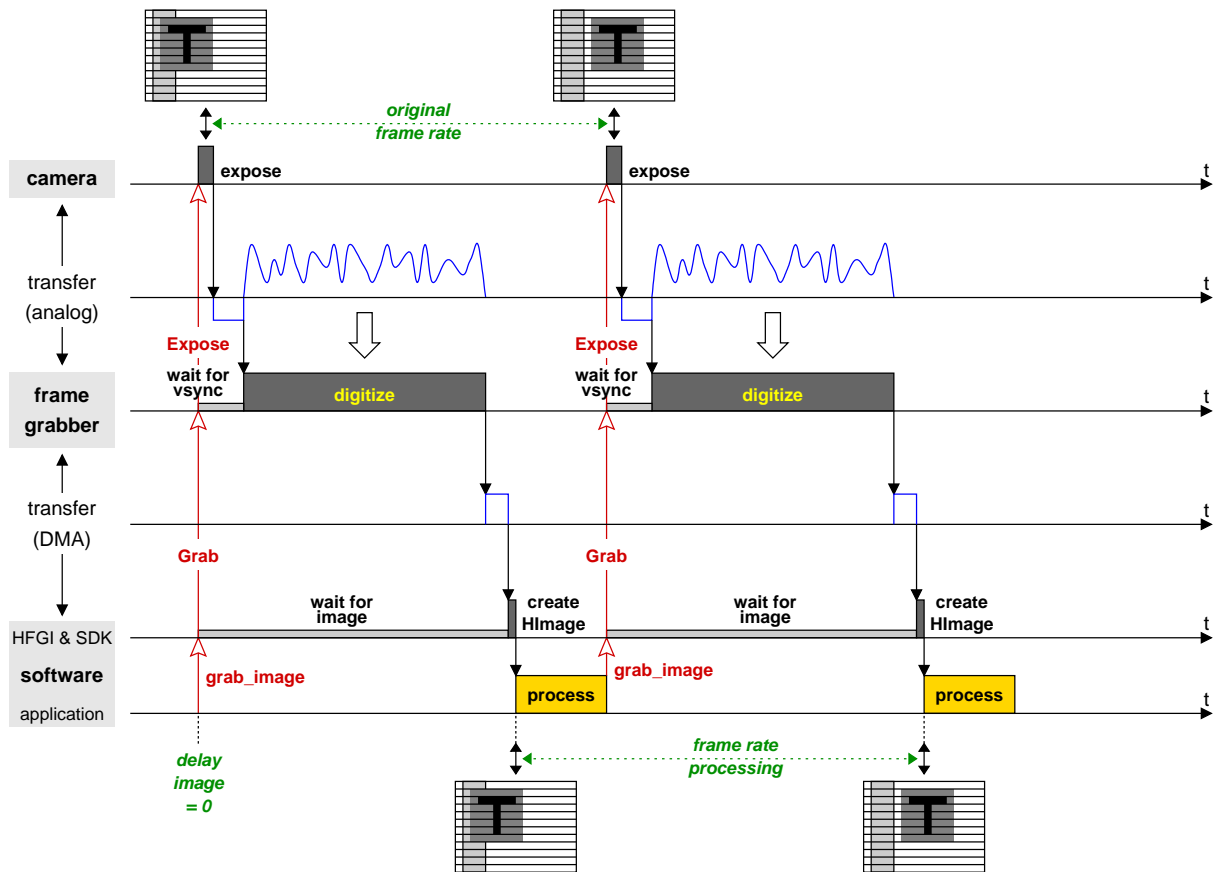
Figure 9: Using a asynchronously resettable camera together with `grab_image` (configuration: progressive-scan camera, frame grabber with burst transfer, volatile grabbing).

To see the non-deterministic image delay, execute the operator `grab_image` in the step mode by pressing F6 ; the execution time displayed in HDevelop's status bar will range between once and twice the original frame period. Please note that on UNIX system, the time measurements are performed with a lower resolution than on Windows systems.

5.1.2 Grabbing Without Delay Using Asynchronously Resettable Cameras

If you use a free-running camera, the camera itself determines the exact moment an image is acquired (exposed). This leads to a delay between the moment you call `grab_image` and the actual image acquisition (see figure 8). The delay is not deterministic, but at least it is limited by the frame rate; for example, if you use an NTSC camera with a frame rate of 30 Hz, the maximum delay can be 33 milliseconds.

Of course, such a delay is not acceptable in an application that is to inspect parts at a high rate. The solution is to use cameras that allow a so-called *asynchronous reset*. This means that upon a signal from the frame grabber, the camera resets the image chip and (almost) immediately starts to expose it. Typically, such a camera does not grab images continuously but only on demand.

An example timing diagram is shown in figure 9. In contrast to figure 8, the image delay is (almost) zero. Furthermore, because the application now specifies when images are to be grabbed, all images can be processed successfully; however, the achieved frame rate still includes the processing time and therefore may be too low for some machine vision applications.

5.1.3   Volatile Grabbing

As shown in figure 8, after the image has been transferred into the computer memory, the HALCON frame grabber interface needs some time to create a corresponding HALCON image which is then returned in the output parameter `Image` of `grab_image`. Most of this time (about 3 milliseconds on a 500 MHz Athlon K6 processor for a gray value NTSC image) is needed to copy the image data from the buffer which is the destination of the DMA into a newly allocated area.

You can switch off the copying by using the so-called *volatile grabbing*, which can be enabled via the operator `set_framegrabber_param` (parameter `'volatile'`):

```
set_framegrabber_param (FGHandle, 'volatile', 'enable')
```

Then, the time needed by the frame grabber interface to create the HALCON image is significantly reduced as visualized in figure 9. Note that usually volatile grabbing is only supported for gray value images!

The drawback of volatile grabbing is that grabbed images are overwritten by subsequent grabs. To be more exact, the overwriting depends on the number of image buffers allocated by the frame grabber interface or SDK. Typically, at least two buffers exist; therefore, you can safely process an image even if the next image is already being grabbed as in figure 11. Some frame grabber interfaces allow to use more than two buffers, and even to select their number dynamically via `set_framegrabber_param` (parameter `'num_buffers'`).

You can check this behavior using the HDevelop example program `hdevelop\volatile_grabbing_ids.dev` (preconfigured for the HALCON IDS interface, please adapt the parts which open the connection for your own frame grabber). After grabbing a first image and displaying it via

```
grab_image (FirstImage, FGHandle)
dev_open_window (0, 0, Width/2, Height/2, 'black', FirstWindow)
dev_display (FirstImage)
```

change the scene and grab a second image which is displayed in an individual window:

```
grab_image (SecondImage, FGHandle)
dev_open_window (0, Width/2 + 8, Width/2, Height/2, 'black', SecondWindow)
dev_display (SecondImage)
```

Now, images are grabbed in a loop and displayed in a third window. The two other images are also displayed each time. If you change the scene before each grab you can see how the first two images are overwritten in turn, depending on the number of buffers.

```
dev_open_window (Height/2 + 66, Width/4 + 4, Width/2, Height/2, 'black',
                 ThirdWindow)
for i := 1 to 10 by 1
  grab_image (CurrentImage, FGHandle)
  dev_set_window (ThirdWindow)
  dev_display (CurrentImage)
  dev_set_window (FirstWindow)
  dev_display (FirstImage)
  dev_set_window (SecondWindow)
  dev_display (SecondImage)
endfor
```
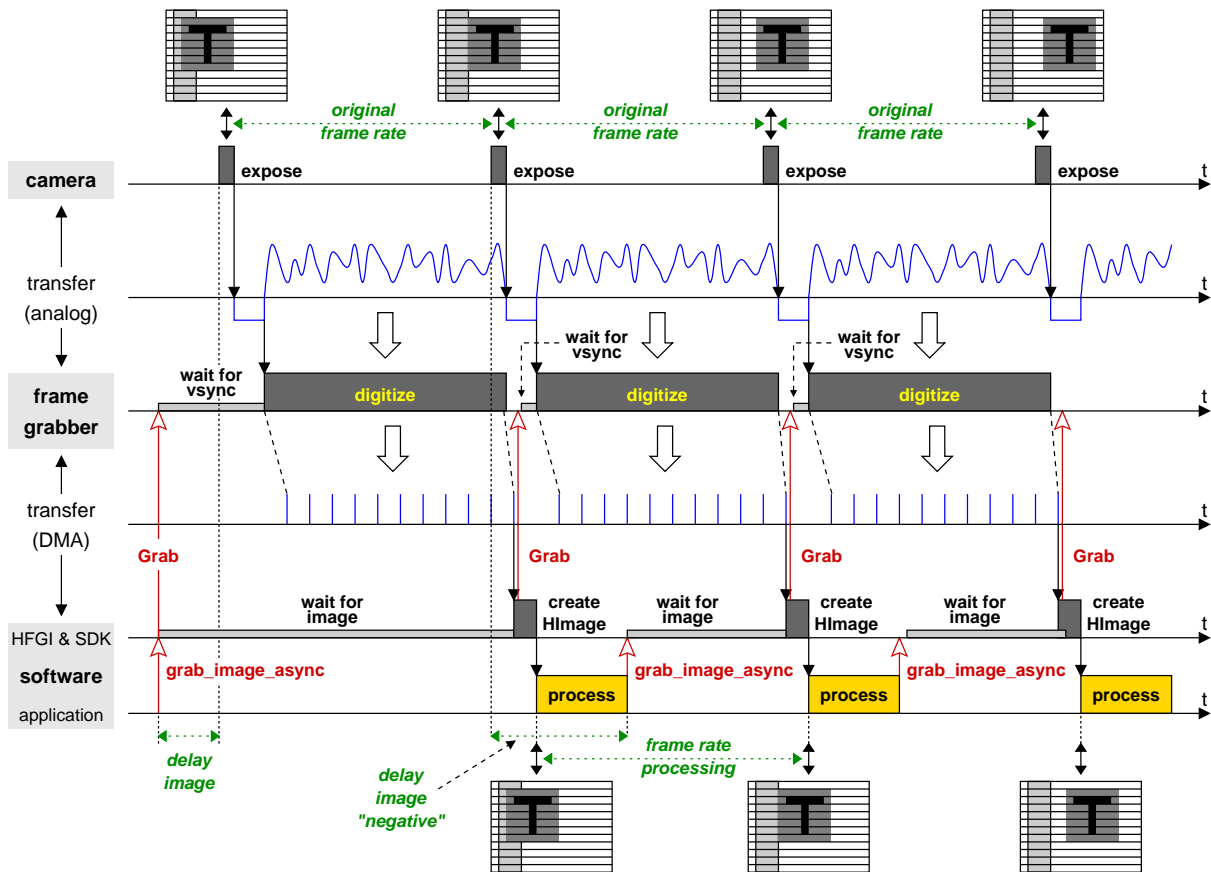
Figure 10: Grabbing and processing in parallel using `grab_image_async` .

5.1.4   **Real-Time Grabbing Using *grab_image_async***

The main problem with the standard timing using `grab_image` is that the two processes of image grabbing and image processing run sequentially, i.e., one after the other. This means that the time needed for processing the image is included in the resulting frame rate, with the effect that the frame rate provided by the camera cannot be reached by definition.

This problem can be solved by using the operator `grab_image_async`. Here, the two processes are decoupled and can run asynchronously, i.e., **an image can be processed while the next image is already being grabbed**. Figure 10 shows a corresponding timing diagram: The first call to `grab_image_async` is processed similar to `grab_image` (compare figure 8). The difference becomes apparent after the transfer of the image into computer memory: Almost immediately after receiving the image, the frame grabber interface automatically commands the frame grabber to acquire a new image. Thus, the next image is grabbed while the application processes the previous image. After the processing, the application calls `grab_image_async` again, which waits until the already running image acquisition is finished. Thus, the full frame rate is now reached. Note that some frame grabbers fail to reach the full frame rate even with `grab_image_async`; section 5.1.5 shows how to solve this problem.

In the HDevelop example program `hdevelop\real_time_grabbing_ids.dev`, which was already described in section 5.1.1, the reached frame rate for asynchronous processing is determined as follows:

```
grab_image (BackgroundImage, FGHandle)
count_seconds (Seconds1)
for i := 1 to 20 by 1
  grab_image_async (Image, FGHandle, -1)
  sub_image (BackgroundImage, Image, DifferenceImage, 1, 128)
endfor
count_seconds (Seconds2)
TimeGrabImageAsync := (Seconds2-Seconds1)/20
FrameRateGrabImageAsync := 1 / TimeGrabImageAsync
```

As can be seen in figure 10, the first call to `grab_image_async` has a slightly different effect than the following ones, as it also triggers the first grab command to the frame grabber. As an alternative, you can use the operator `grab_image_start` which just triggers the grab command; then, the first call to `grab_image_async` behaves as the other ones. This is visualized, e.g., in figure 11; as you can see, the advantage of this method is that the application can perform some processing before calling `grab_image_async`.

In the example, the processing was assumed to be faster than the acquisition. If this is not the case, the image will already be ready when the next call to `grab_image_async` arrives. In this case, you can specify how "old" the image is allowed to be using the parameter `MaxDelay`. Please refer to section 5.1.7 for details.

Please note that when using `grab_image_async` it is not obvious anymore which image is returned by the operator call, because the call is decoupled from the command to the frame grabber! In contrast to `grab_image`, which always triggers the acquisition of a new image, `grab_image_async` typically returns an image which has been exposed before the operator was called, i.e., the image delay is negative (see figure 10)! Keep this effect in mind when changing parameters dynamically; contrary to intuition, the change will not affect the image returned by the next call of `grab_image_async` but by the following ones! Another problem appears when switching dynamically between cameras (see section 5.3.1).

5.1.5   Continuous Grabbing

For some frame grabbers `grab_image_async` fails to reach the frame rate because the grab command to the frame grabber comes too late, i.e., after the camera has already started to transfer the next image (see figure 11a).

As a solution to this problem, some frame grabber interfaces provide the so-called *continuous grabbing mode* which can be enables only via the operator `set_framegrabber_param` (parameter `'continuous_grabbing'`):

```
set_framegrabber_param (FGHandle, 'continuous_grabbing', 'enable')
```

In this mode, the frame grabber reads images from a free-running camera continuously and transfers them into computer memory as depicted in figure 11b. Thus, the frame rate is reached. If your frame grabber supports continuous grabbing you can test this effect in the example program `hdevelop\real_time_grabbing_ids.dev`, which was already described in the previous sections; the program measures the achievable frame rate for `grab_image_async` without and with continuous grabbing.

We recommend to use continuous grabbing only if you want to process every image; otherwise, images are transmitted over the PCI bus unnecessarily, thereby perhaps blocking other PCI transfers.
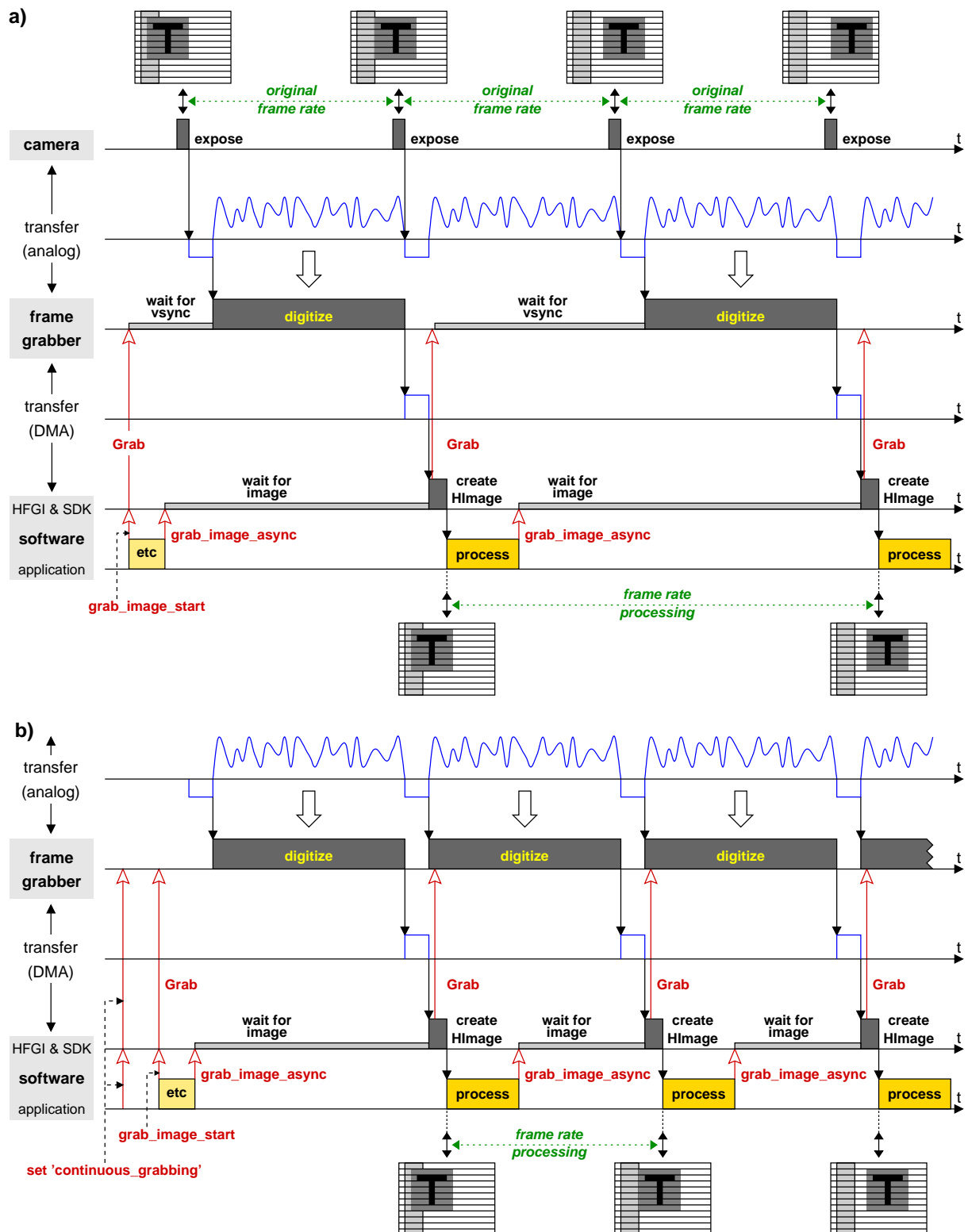
Figure 11: a) `grab_image_async` fails to reach frame rate; b) problem solved using continuous grabbing.

Note that some frame grabber interfaces provide additional functionality in the continuous grabbing mode, e.g., the HALCON `BitFlow` interface. Please refer to the corresponding documentation for more information.
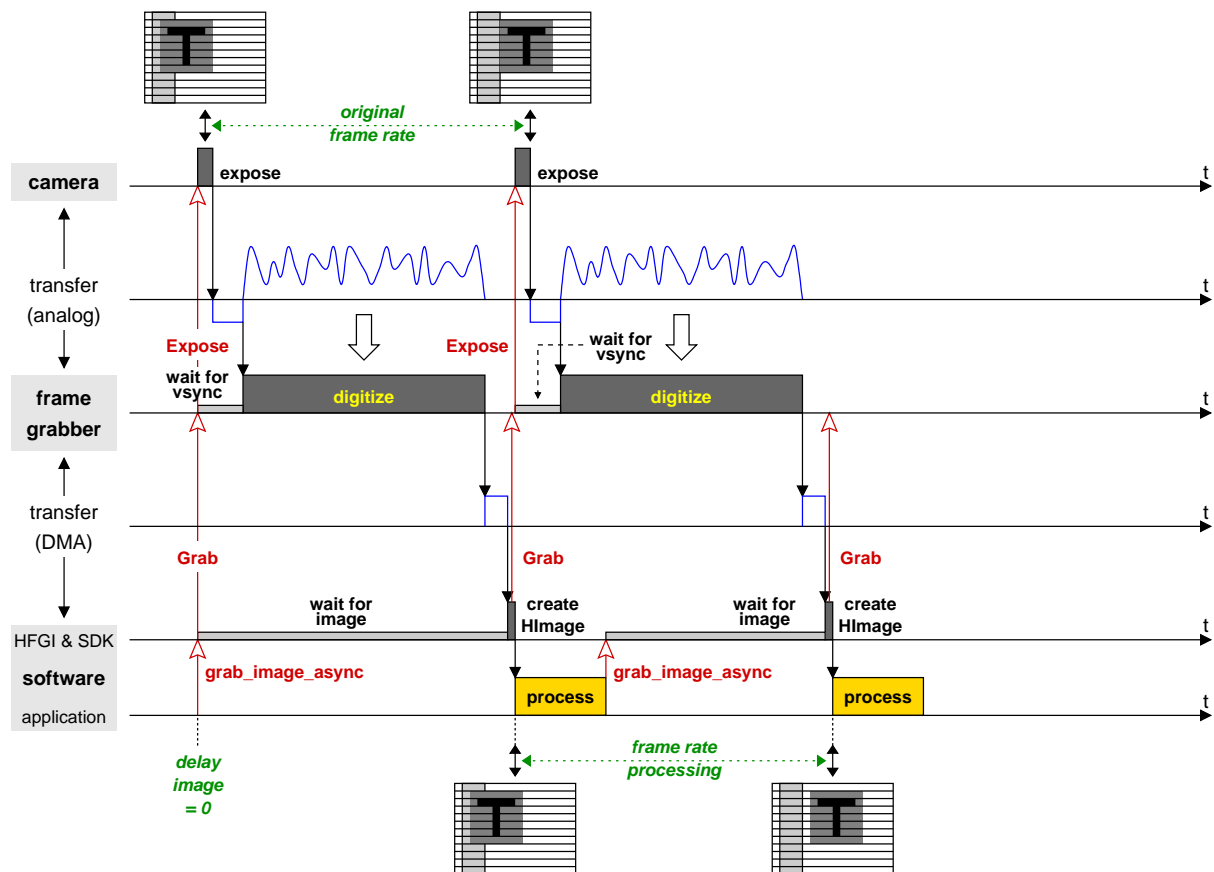
Figure 12: Using a asynchronously resettable camera together with `grab_image_async` (configuration as in figure 9.

5.1.6   Using *grab_image_async* Together With Asynchronously Resettable Cameras

As described in section 5.1.2, you can acquire images without delay by using an asynchronously resettable camera. Figure 12 shows the resulting timing when using such a camera together with `grab_image_async`. When comparing the diagram to the one in figure 9, you can see that a higher frame rate can now be reached, because the processing time is not included anymore.

5.1.7   Specifying a Maximum Delay

In contrast to `grab_image`, the operator `grab_image_async` has an additional parameter `MaxDelay`, which lets you specify how "old" an already grabbed image may be in order to be accepted. Figure 13 visualizes the effect of this parameter. There are two cases to distinguish: If the call to `grab_image` arrives before the next image has been grabbed (first call in the example), the parameter has no effect. However, if an image has been grabbed already (second and third call in the example), the elapsed time since the last grab command to the frame grabber is compared to `MaxDelay`. If it is smaller (second call in the example), the image is accepted; otherwise (third call), a new image is grabbed.

Please note that the delay is not measured starting from the moment the image is exposed, as you might perhaps expect! Currently, only a few frame grabber SDKs provide this information;
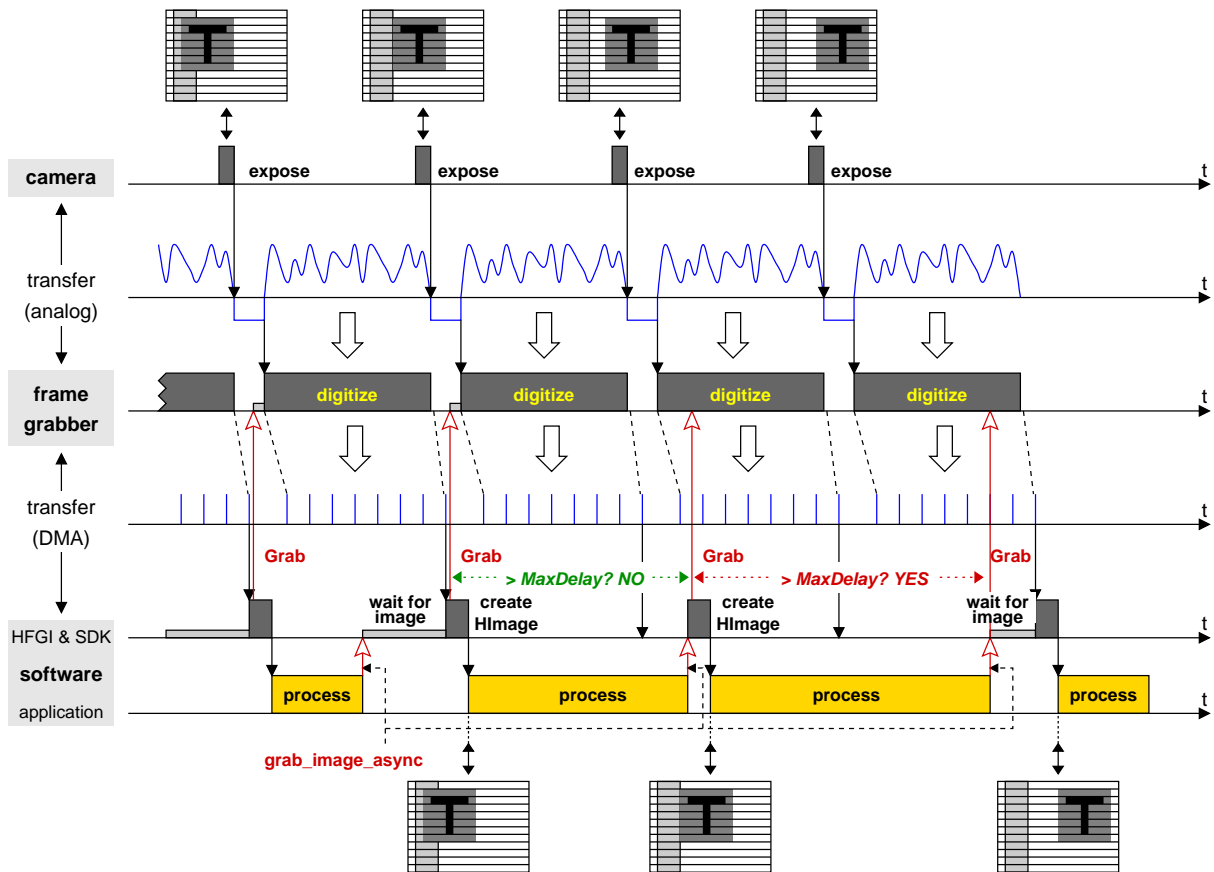
Figure 13: Specifying a maximum delay for `grab_image_async` (using continuous grabbing).

therefore, the last grab command from the interface to the the frame grabber is used as the
starting point instead.

5.2   Using an External Trigger

In the previous section, the software performing the machine vision task decided when to ac-
quire an image (*software trigger*). In industrial applications, however, the moment for image
acquisition is typically specified externally by the process itself, e.g., in form of a hardware
trigger signal indicating the presence of an object to be inspected. Most frame grabber boards
are therefore equipped with at least one input line for such signals, which are called *external
triggers*.

From HALCON's point of view, external triggers are dealt with by the frame grabber board,
the only thing to do is to inform the frame grabber to use the trigger. You can do this simply
by setting the parameter `ExternalTrigger` of `open_framegrabber` to 'true'. Some frame
grabber interfaces also allow to enable or disable the trigger dynamically using the operator
`set_framegrabber_param` (parameter 'external_trigger').

Figure 14a shows the timing diagram when using an external trigger together with `grab_image`
and a free-running camera. After the call to `grab_image`, the frame grabber board waits for
the trigger signal. When it appears, the procedure described in the previous section follows:
The frame grabber waits for the next image, digitizes it, and transfers it into computer memory;

a)

b)
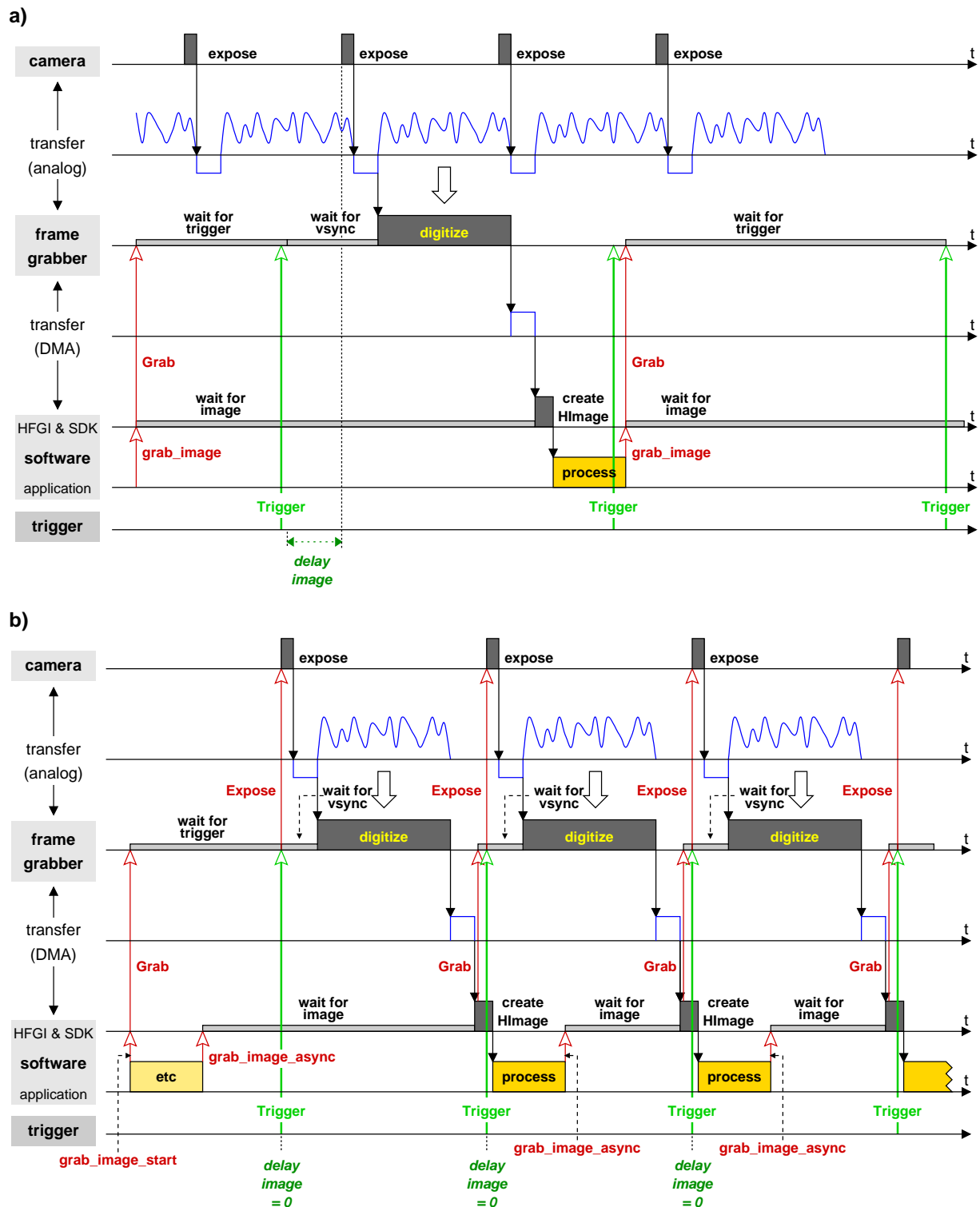
Figure 14: Using an external trigger together with: a) free-running camera and `grab_image`; b) asynchronously resettable camera and `grab_image_async` .

then, the HALCON frame grabber interface transforms it into a HALCON image and returns the control to the application which processes the image and then calls `grab_image` again, which causes the frame grabber board to wait for the next trigger signal.

The (bad) example in figure 14a was chosen on purpose to show an unsuitable configuration

for using an external trigger: First of all, because of the free-running camera there is a non-deterministic delay between the arrival of the trigger signal and the exposure of the image, which may mean that the object to be inspected is not completely visible anymore. Secondly, because grab_image is used, trigger signals which arrive while the application is processing an image are lost.

Both problems can easily be solved by using an asynchronously resettable camera together with the operator grab_image_async as depicted in figure 14b.

The C++ example program cpp\error_handling_timeout_picport.cpp (preconfigured for the HALCON PicPort interface) shows how simple it is to use an external trigger: The connection is opened with ExternalTrigger set to 'true':

```
HFramegrabber      framegrabber;

framegrabber.OpenFramegrabber(fgname, 1, 1, 0, 0, 0, 0, "default", -1,
                              "gray", -1, "true", camtype, device,
                              -1, -1);
```

Then, images are grabbed:

```
HImage             image;

do
{
  image = framegrabber.GrabImageAsync(-1);
} while (button == 0);
```

The example contains a customized error handler which checks whether there is an external trigger; this part is described in detail in section 6.2.3.

5.2.1 Special Parameters for External Triggers

Most frame grabber interfaces allow to further configure the use of external triggering via the operator set_framegrabber_param. As mentioned in section 4.2, some interfaces allow to enable and disable the external trigger dynamically via the parameter 'external_trigger'. Another useful parameter is 'grab_timeout', which sets a timeout for the acquisition process (some interfaces provide an additional parameter 'trigger_timeout' just for triggered grabbing). Without such a timeout, the application would hang if for some reason no trigger signal arrives. In contrast, if a timeout is specified, the operators grab_image and grab_image_async only wait the specified time and then return an error code or raise an exception, depending on the programming language used. Section 6.2 shows how to handle such errors.

Other parameters allow to further specify the form of the trigger signal ('trigger_signal'), e.g., whether the falling or the rising edge is used as the trigger, select between multiple trigger input lines, or even filter trigger signals. Some frame grabber interfaces also allow to influence the exposure via the trigger signal.

5.3 Acquiring Images From Multiple Cameras

The timing diagrams shown in the previous sections depicted the case of a single camera. Below we discuss some issues which arise when acquiring images from multiple cameras (see

section 3.2 for possible configurations).

5.3.1   Dynamic Port Switching and Asynchronous Grabbing

If you switch dynamically between multiple cameras connected to a single board as described in section 3.2.4 you must be careful when using `grab_image_async`: By default, the frame grabber interface commands the frame grabber board to grab the next image automatically after it received the current image — but before the next call of `grab_image_async`! If you switched to another camera before this call, the frame grabber might already be busy grabbing an image from the first camera.

Some frame grabber interfaces solve this problem by providing the parameter `'start_async_after_grab_async'` for the operator `set_framegrabber_param` which allows to disable the automatic grab command to the frame grabber board.

5.3.2   Simultaneous Grabbing

Some frame grabber interfaces provide special functionality to grab images *simultaneously* from multiple (synchronized) cameras. Typically, the cameras are connected to a single frame grabber board; the `PicPort` interface also allows to grab simultaneously from cameras connected to multiple boards. As described in section 3.2.5, the images are grabbed by a single call to `grab_image` or `grab_image_async`, which return them in form of a multi-channel image. Depending on the frame grabber interface, it may be necessary to switch on the simultaneous grabbing via the operator `set_framegrabber_param`.

Please keep in mind that even if a HALCON frame grabber interface supports simultaneous grabbing, this might not be true for every frame grabber board the interface supports! In order to grab multiple images simultaneously, a frame grabber board must be equipped with multiple "grabbing units"; for example, an analog frame grabber board must be equipped with multiple A/D converters. Please check this in the documentation of your frame grabber board.

Even if a HALCON frame grabber interface does not provide the special simultaneous grabbing mode, you can realize a similar behavior "manually", e.g., by connecting each (asynchronously resettable) camera to a single frame grabber board and then using a common external trigger signal to synchronize the grabbing.

6 Miscellaneous

6.1 Acquiring Images From Unsupported Frame Grabbers

If you want to use a frame grabber which is currently not supported by HALCON, i.e., for which no HALCON interface exists there exist two principal ways: First, you can create your own HALCON frame grabber interface; how to do this is described in detail in the Frame Grabber Integration Programmer's Manual .

As an alternative, you can pass externally created images, i.e., the raw image matrix, to HALCON using the operators gen_image1, gen_image3, or gen_image1_extern, which create a corresponding HALCON image. The main difference between the operators gen_image1 and gen_image1_extern is that the former copies the image matrix when creating the HALCON image, whereas the latter doesn't, which is useful if you want to realize *volatile grabbing* as described in section 5.1.3.

The C example program c\use_extern_image.c shows how to use the operator gen_image1_extern to pass standard gray value images to HALCON. In this case, the image matrix consists of 8 bit pixels (bytes), which can be represented by the data type unsigned char. At the beginning, the program calls a procedure which allocates memory for the images to be "grabbed"; in a real application this corresponds to the image buffer(s) used by the frame grabber SDK.

```
unsigned char *image_matrix_ptr;
long          width, height;

InitializeBuffer(&image_matrix_ptr, &width, &height);
```

The example program "simulates" the grabbing of images with a procedure which reads images from an image sequence and copies them into the image buffer. Then, the content of the image buffer is transformed into a HALCON image (type byte) via gen_image1_extern. The parameter ClearProc is set to 0 to signal that the program itself takes care of freeing the memory. The created HALCON image is then displayed. The loop can be exited by clicking into the HALCON window with any mouse button.

```
Hobject       image;
long          window_id;

open_window (0, 0, width, height, 0, "visible", "", &window_id);
while (!ButtonPressed(window_id))
{
  MyGrabImage((const unsigned char **) &image_matrix_ptr);
  gen_image1_extern(&image, "byte", width, height,
                    (long) image_matrix_ptr, (long) 0);
  disp_obj(image, window_id);
}
```

If your frame grabber supplies images with more than 8 bit pixels, you must adapt both the data type for the image matrix and the type of the created HALCON image (parameter Type of gen_image1_extern). In case of color images HALCON expects the image data in form of three separate image matrices. You can create a HALCON image either by calling the operator gen_image3 with the three pointers to the matrices, or by calling the operator
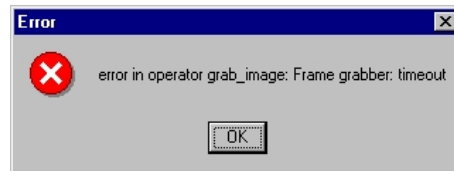
Figure 15: Popup dialog in HDevelop signaling a timeout.

`gen_image1_extern` three times and then using the operator `channels_to_image` to combine the three images into a multi-channel image. Please refer to appendix A for more information about HALCON images in general.

6.2 Error Handling

Just as the HALCON frame grabber interfaces encapsulate the communication with a frame grabber board, they also encapsulate occurring errors within the HALCON error handling mechanism. How to catch and react to these errors is described below, for HDevelop programs and also for programs using HALCON's programming language interfaces.

Some HALCON frame grabber interfaces provide special parameters for `set_framegrabber_param` which are related to error handling. The most commonly used one is the parameter `'grab_timeout'` which specifies when the frame grabber should quit waiting for an image. The examples described in the following sections show how to handle the corresponding HALCON error.

Note that all example programs enable the signaling of low level errors via the operator `set_system`, e.g., in HDevelop syntax via

```
set_system ('do_low_error', 'true')
```

In this mode, low level errors occurring the frame grabber SDK (or in the HALCON interface) in are signaled by a message box.

6.2.1 Error Handling in HDevelop

The HDevelop example `hdevelop\error_handling_timeout_picport.dev` shows how to handle HALCON errors in a HDevelop program. To "provoke" an error, `open_framegrabber` is called with `ExternalTrigger` = `'true'`. If there is no trigger, a call to `grab_image` results in a timeout; HDevelop reacts to this error with the popup dialog shown in figure 15 and stops the program.

```
open_framegrabber (FGName, 1, 1, 0, 0, 0, 0, 'default', -1, 'default', -1,
                   'true', CameraType, Device, -1, -1, FGHandle)
set_framegrabber_param (FGHandle, 'grab_timeout', 2000)
grab_image (Image, FGHandle)
```

HALCON lets you modify the reaction to an error with the operator `set_check` (in HDevelop: `dev_set_check`). If you set it to `'~give_error'`, the program does not stop in case of an error but only stores its cause in form of an error code. To access this error code in HDevelop,

you must define a corresponding variable using the operator `dev_error_var`. Note that this variable is updated after each operator call; to check the result of a single operator we therefore recommend to switch back into the standard error handling mode directly after the operator call as in the following lines:

```
dev_error_var (ErrorNum, 1)
dev_set_check ('~give_error')
grab_image (Image, FGHandle)
dev_error_var (ErrorNum, 0)
dev_set_check ('give_error')
```

To check whether a timeout occurred, you compare the error variable with the code signaling a timeout (5322); a list of error codes relating to image acquisition can be found in the Frame Grabber Integration Programmer's Manual . In the example, the timeout is handled by disabling the external trigger mode via the operator `set_framegrabber_param` (parameter `'external_trigger'`). Then, the call to `grab_image` is tested again.

```
if (ErrorNum = 5322)
  set_framegrabber_param (FGHandle, 'external_trigger', 'false')
  dev_error_var (ErrorNum, 1)
  dev_set_check ('~give_error')
  grab_image (Image, FGHandle)
  dev_error_var (ErrorNum, 0)
  dev_set_check ('give_error')
endif
```

Now, the error variable should contain the value 2 signaling that the operator call succeeded; for this value HDevelop provides the constant `H_MSG_TRUE`. If you get another error code, the program accesses the corresponding error text using the operator `get_error_text`.

```
if (ErrorNum # H_MSG_TRUE)
  get_error_text (ErrorNum, ErrorText)
endif
```

If your frame grabber interface does not provide the parameter `'external_trigger'`, you can realize a similar behavior by closing the connection and then opening it again with `ExternalTrigger` set to `'false'`.

6.2.2   Error Handling Using HALCON/C

The mechanism for error handling in a program based on HALCON/C is similar to the one in HDevelop; in fact, it is even simpler, because each operator automatically returns its error code. However, if a HALCON error occurs in a C program, the default error handling mode causes the program to abort.

The C example program `c\error_handling_timeout_picport.c` performs the same task as the HDevelop program in the previous section; if the call to `grab_image` succeeds, the program grabs and displays images in a loop, which can be exited by clicking into the window. The following lines show how to test whether a timeout occurred:

```
    set_check ("~give_error");
    error_num = grab_image (&image, fghandle);
    set_check ("give_error");
    switch (error_num)
    {
    case H_ERR_FGTIMEOUT:
```

As you see, in a C program you can use prefined constants for the error codes (see the Frame Grabber Integration Programmer's Manual for a list of image acquisition error codes and their corresponding constants).

6.2.3   Error Handling Using HALCON/C++

If your application is based on HALCON/C++, there are two methods for error handling: If you use operators in their C-like form, i.e., preceeded by a double colon (e.g., ::grab_image), you can apply the same procedure as described for HALCON/C in the previous section.

In addition, HALCON/C++ provides an exception handling mechanism based on the class HException, which is described in the HALCON/C++ User's Manual . Whenever a HAL-CON error occurs, an instance of this class is created. The main idea is that you can specify a procedure which is then called automatically with the created instance of HException as a parameter. How to use this mechanism is explained in the C++ example program cpp\error_handling_timeout_picport.cpp, which performs the same task as the examples in the previous sections.

In the example program cpp\error_handling_timeout_picport.cpp (preconfigured for the HALCON PicPort interface), the procedure which is to be called upon error is very simple: It just raises a standard C++ exception with the instance of HException as a parameter.

```
 void MyHalconExceptionHandler(const HException& except)
 {
   throw except;
 }
```

In the program, you "install" this procedure via a class method of HException:

```
 int  main(int argc, char *argv[])
 {
   HException::InstallHHandler(&MyHalconExceptionHandler);
```

Now, you react to a timeout with the following lines:

```
    try
    {
      image = framegrabber.GrabImage();
    }
    catch (HException except)
    {
      if (except.err == H_ERR_FGTIMEOUT)
      {
        framegrabber.SetFramegrabberParam("external_trigger", "false");
```

As already noted, if your frame grabber interface does not provide the parameter `'external_trigger'`, you can realize a similar behavior by closing the connection and then opening it again with `ExternalTrigger` set to `'false'`:

```
if (except.err == H_ERR_FGTIMEOUT)
{
  framegrabber.OpenFramegrabber(fgname, 1, 1, 0, 0, 0, 0, "default",
                                -1, "gray", -1, "false", camtype,
                                "default", -1, -1);
```

Note that when calling `OpenFramegrabber` via the class `HFramegrabber` as above, the operator checks whether it is called with an already opened connection and automatically closes it before opening it with the new parameters.

6.2.4 Error Handling Using HALCON/COM

The HALCON/COM interface uses the standard COM error handling technique where every method call passes both a numerical and a textual representation of the error to the calling framework. How to use this mechanism is explained in the Visual Basic example program `vb\error_handling_timeout_picport\error_handling_timeout_picport.vbp`, which performs the same task as the examples in the previous sections.

For each method, you can specify an error handler by inserting the following line at the beginning of the method:

```
On Error GoTo ErrorHandler
```

At the end of the method, you insert the code for the error handler. If a runtime error occurs, Visual Basic automatically jumps to this code, with the error being described in the variable `Err`. However, the returned error number does not correspond directly to the HALCON error as in the other programming languages, because low error numbers are reserved for COM. To solve this problem HALCON/COM uses an offset which must be subtracted to get the HALCON error code. This offset is accessible as a property of the class `HSystemX`:

```
ErrorHandler:
  Dim sys As New HSystemX
  ErrorNum = Err.Number - sys.ErrorBaseHalcon
```

The following code fragment checks whether the error is due to a timeout. If yes, the program disables the external trigger mode and tries again to grab an image. If the grab is successful the program continues at the point the error occurred; otherwise, the Visual Basic default error handler is invoked. Note that in contrast to the other programming languages HALCON/COM does not provide constants for the error codes.

```
If (ErrorNum = 5322) Then
  Call FG.SetFramegrabberParam("external_trigger", "false")
  Set Image = FG.GrabImage
  Resume Next
```

If the error is not caused by a timeout, the error handler raises it anew, whereupon the Visual Basic default error handler is invoked.

```
  Else
    Err.Raise (Err.Number)
  End If
```

If your frame grabber interface does not provide the parameter 'external_trigger', you can realize a similar behavior by closing the connection and then opening it again with `ExternalTrigger` set to 'false'. Note that the class `HFramegrabberX` does not provide a method to close the connection; instead you must destroy the variable with the following line:

```
    Set FG = Nothing
```

6.3   Line Scan Cameras

From the point of view of HALCON there is no difference between area and line scan cameras: Both acquire images of a certain width an height; whether the height is 1, i.e., a single line, or larger does not matter. In fact, in many line scan applications the frame grabber combines multiple acquired lines to form a so-called *page* which further lessens the difference between the two camera types.

The main problem is therefore whether your frame grabber supports line scan cameras. If yes, you can acquire images from it via HALCON exactly as from an area scan camera. With the parameter `ImageHeight` of the operator `open_framegrabber` you can sometimes specify the height of the page; typically, this information is set in the camera configuration file. Some HALCON frame grabber interfaces allow to further configure the acquisition mode via the operator `set_framegrabber_param`.

The images acquired from a line scan camera can then be processed just like images from area scan cameras. However, line scan images often pose an additional problem: The objects to inspect may be spread over multiple images (pages). To solve this problem, HALCON provides special operators: `tile_images` allows to merge images into a larger image, `merge_regions_line_scan` and `merge_cont_line_scan_xld` allow to merge the (intermediate) processing results of subsequent images.

How to use these operators is explained in the HDevelop example program `hdevelop\line_scan.dev`. The program is based on an image file sequence which is read using the HALCON virtual frame grabber interface `File`; the task is to extract paper clips and calculate their orientation. Furthermore, the gray values in a rectangle surrounding each clip are determined.

An important parameter for the merging is over how many images an object can be spread. In the example, a clip can be spread over 4 images:

```
 MaxImagesRegions := 4
```

The continuous processing is realized by a simple loop: At each iteration, a new image is grabbed, and the regions forming candidates for the clips are extracted using thresholding.

```
while (1)
  grab_image (Image, FGHandle)
  threshold (Image, CurrRegions, 0, 80)
```

The current regions are then merged with ones extracted in the previous image using the operator `merge_regions_line_scan`. As a result, two sets of regions are returned: The parameter
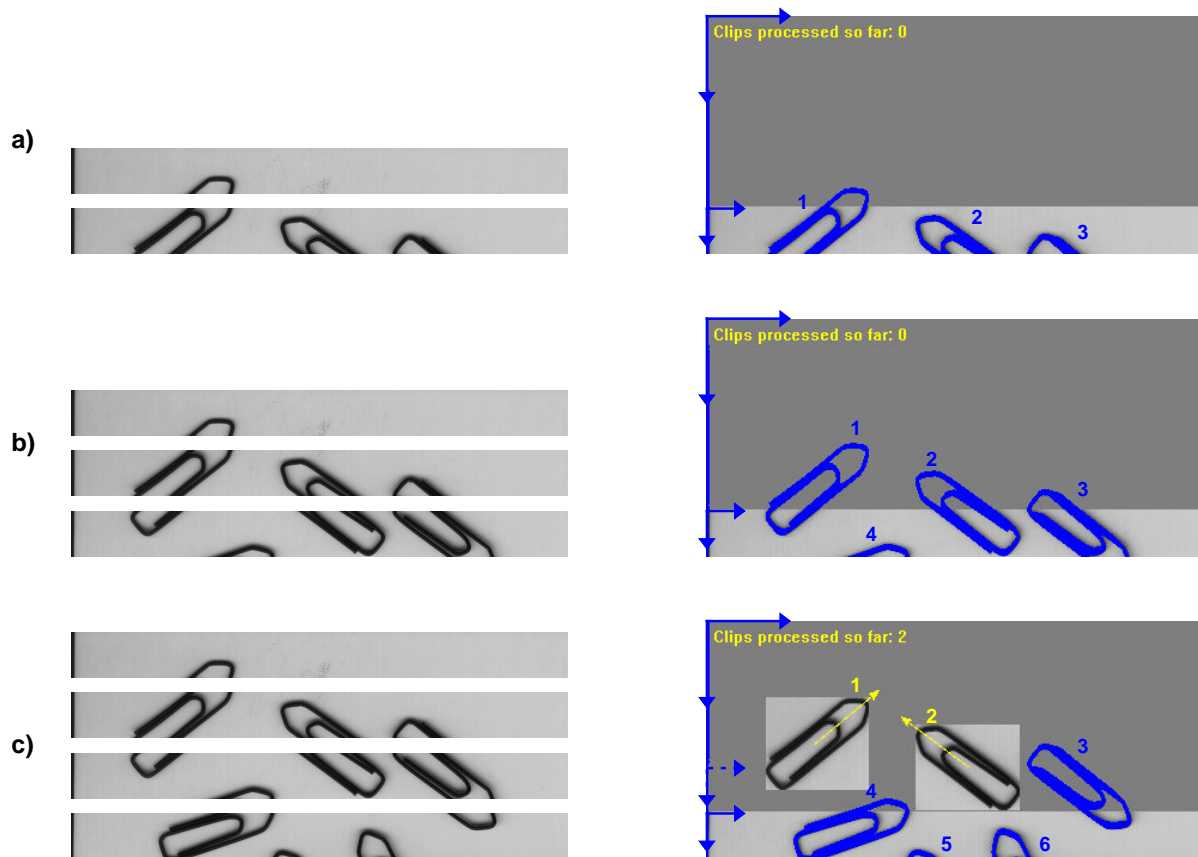
Figure 16: Merging regions extracted from subsequent line scan images: state after a) 2, b) 3, c) 4 images (large coordinate system: tiled image; small coordinate systems: current image or most recent image).

CurrMergedRegions contains the current regions, possibly extended by fitting parts of the previously extracted regions, whereas the parameter PrevMergedRegions contains the rest of the previous regions.

```
merge_regions_line_scan (CurrRegions, PrevRegions, CurrMergedRegions,
                         PrevMergedRegions, ImageHeight, 'top',
                         MaxImagesRegions)
connection (PrevMergedRegions, ClipCandidates)
select_shape (ClipCandidates, FinishedClips, 'area', 'and', 4500, 7000)
```

The regions in PrevMergedRegions are "finished"; from them, the program selects the clips via their area further processes them later, e.g., determines their position and orientation. The regions in CurrMergedRegions are renamed and now form the previous regions for the next iteration.

```
  copy_obj (CurrMergedRegions, PrevRegions, 1, -1)
endwhile
```

Note that the operator copy_obj does not copy the regions themselves but only the corresponding HALCON objects, which can be thought of as references to the actual region data.

Before we show how to merge the images let's take a look at figure 16, which visualizes the whole process: After the first two images CurrMergedRegions contains three clip parts; for the first one a previously extracted region was merged. Note that the regions are described in

the coordinate frame of the current image; this means that the merged part of clip no. 1 has negative coordinates.

In the next iteration (figure 16b), further clip parts are merged, but no clip is finished yet. Note that the coordinate frame is again fixed to the current image; as a consequence the currently merged regions seem to move into negative coordinates.

After the fourth image (figure 16c), clips no. 1 and 2 are completed; they are returned in the parameter `PrevMergedRegions`. Note that they are still described in the coordinate frame of the previous image (depicted with dashed arrow); to visualize them together with `CurrMergedRegions` they must be moved to the coordinate system of the current image using the operator `move_region`:

```
   move_region (FinishedClips, ClipsInCurrentImageCoordinates,
                -ImageHeight, 0)
```

Let's get back to the task of merging images: To access the gray values around a clip, one must merge those images over which the `PrevMergedRegions` can be spread. At the beginning, therefore an empty image is created which can hold 4 images:

```
 gen_image_const (TiledImage, 'byte', ImageWidth,
                  ImageHeight * MaxImagesRegions)
```

At the end of each iteration, the "oldest" image, i.e., the image at the top, is cut off from the tiled image using `crop_part`, and the current image is merged at the bottom using `tile_images_offset`:

```
   crop_part (TiledImage, TiledImageMinusOldest, ImageHeight, 0,
              ImageWidth, (MaxImagesRegions - 1) * ImageHeight)
   ImagesToTile := [TiledImageMinusOldest,Image]
   tile_images_offset (ImagesToTile, TiledImage, [0,
                       (MaxImagesRegions-1)*ImageHeight], [0, 0], [-1,
                       -1], [-1, -1], [-1, -1], [-1, -1], ImageWidth,
                       MaxImagesRegions * ImageHeight)
```

As noted above, the regions returned in `PrevMergedRegions` are described in the coordinate frame of the most recent image (depicted with dashed arrow in figure 16c); to extract the corresponding gray values from the tiled image, they must first be moved to its coordinate system (depicted with longer arrows) using the operator `move_region`. Then, the surrounding rectangles are created using `shape_trans`, and finally the corresponding gray values are extracted using `add_channels`:

```
   move_region (FinishedClips, ClipsInTiledImageCoordinates,
                (MaxImagesRegions-1) *  ImageHeight, 0)
   shape_trans (ClipsInTiledImageCoordinates, AroundClips, 'rectangle1')
   add_channels (AroundClips, TiledImage, GrayValuesAroundClips)
```

Appendix

A   HALCON Images

In the following, we take a closer look at the way HALCON represents and handles images. Of course, we won't bother you with details about the low-level representation and the memory management; HALCON takes care of it in a way to guarantee optimal performance.

A.1   The Philosophy of HALCON Images

There are three important concepts behind HALCON's image objects:

1. **Multiple channels**

   Typically, one thinks of an image as a matrix of pixels. In HALCON, this matrix is called a *channel*, and images may consist of one or more such channels. For example, gray value images consist of a single channel, color images of three channels.

   The advantage of this representation is that many HALCON operators automatically process all channels at once; for example, if you want to subtract gray level or color images from another, you can apply `sub_image` without worrying about the image type. Whether an operator processes all channels at once can be seen in the parameter description in the reference manual: If an image parameter is described as (multichannel-)image or (multichannel-)image(-array) (e.g., the parameter `ImageMinuend` of `sub_image`), all channels are processed; if it is described as image or image(-array) (e.g., the parameter `Image` of `threshold`), only the first channel is processed.

   For more information about channels please refer to appendix A.3.2.

2. **Various pixel types**

   Besides the standard 8 bit (type `byte`) used to represent gray value image, HALCON allows images to contain various other data, e.g. 16 bit integers (type `int2` or `uint2`) or 32 bit floating point numbers (type `real`) to represent derivatives.

   Most of the time you need not worry about pixel types, because HALCON operators that output images automatically use a suitable pixel type. For example, the operator `derivate_gauss` creates a `real` image to store the result of the derivation. As another example, if you connect to a frame grabber selecting a value > 8 for the parameter `BitsPerChannel`, a subsequent `grab_image` returns an `uint2` image.

3. **Arbitrarily-shaped region of interest**

   Besides the pixel information, each HALCON image also stores its so-called *domain* in form of a HALCON region. The domain can be interpreted as a region of interest, i.e., HALCON operators (with some exceptions) restrict their processing to this region.

   The image domain inherits the full flexibility of a HALCON region, i.e., it can be of arbitrary shape and size, can have holes, or even consist of unconnected points. For more information about domains please refer to appendix A.3.3.

The power of HALCON's approach lies in the fact that it offers full flexibility but does not require you to worry about options you don't need at the moment. For example, if all you do is grab and process standard 8 bit gray value images, you can ignore channels and pixel types. At the moment you decide to use color images instead, all you need to do is to add some lines to decompose the image into its channels. And if your camera / frame grabber provides images with more than 8 bit pixel information, HALCON is ready for this as well.

A.2   Image Tuples (Arrays)

Another powerful mechanism of HALCON is the so-called *tuple processing*: If you want to process multiple images in the same way, e.g., to smooth them, you can call the operator (e.g., `mean_image`) once passing it all images as a tuple (array), instead of calling it multiple times. Furthermore, some operators always return image tuples, e.g., `gen_gauss_pyramid` or `inspect_shape_model`.

Whether an operator supports tuple processing can be seen in the parameter description in the reference manual: If an input image parameter is described as `image(-array)` or `(multichannel-)image(-array)` (e.g., the parameter `Image` of `mean_image`), it supports tuple processing; if it is described as `image` or `(multichannel-)image` (e.g., the parameter `Image` of `find_1d_bar_code`), only one image is processed.

For information about creating or accessing image tuples please refer to appendix A.3.6.

A.3   HALCON Operators for Handling Images

Below you find a brief overview of operators that allow to create HALCON images or to modify "technical aspects" like the image size or the number of channels.

A.3.1   Creation

HALCON images are created automatically when you use operators like `grab_image` or `read_image`. You can also create images from scratch using the operators listed in the HDevelop menu Operators ▷ Image ▷ Creation, e.g., `gen_image_const` or `gen_image1_extern` (see also section 6.1).

A.3.2   Channels

Operators for manipulating channels can be found in the HDevelop menu Operators ▷ Image ▷ Channel. You can query the number of channels of an image with the operator `count_channels`. Channels can be accessed using `access_channel` (which extracts a specified channel without copying), `image_to_channels` (which converts a multi-channel image into an image tuple), or `decompose2` etc. (which converts a multi-channel image into 2 or more single-channel images). Vice versa, you can create a multi-channel image using `channels_to_image` or `compose2` etc., and add channels to an image using `append_channel`.

A.3.3   Domain

Operators for manipulating the domain of an image can be found in the HDevelop menu
`Operators ▷ Image ▷ Domain`. Upon creation of an image, its domain is set to the full im-
age size. You can set it to a specified region using `change_domain`. In contrast, the operator
`reduce_domain` takes the original domain into account; the new domain is equal to the inter-
section of the original domain with the specified region. Please also take a look at the operator
`add_channels`, which can be seen as complementary to `reduce_domain`.

A.3.4   Access

Operators for accessing infomation about a HALCON image can be found in the HDevelop
menu `Operators ▷ Image ▷ Access`. For example, `get_image_pointer1` returns the size of
an image and a pointer to the image matrix of its first channel.

A.3.5   Manipulation

You can change the size of an image using the operators `change_format` or `crop_part`,
or other operators from the HDevelop menu `Operators ▷ Image ▷ Format`. The menu
`Operators ▷ Image ▷ Type-Conversion` lists operators which change the pixel type, e.g.,
`convert_image_type`. Operators to modify the pixel values, can be found in the menu
`Operators ▷ Image ▷ Manipulation`, e.g., `paint_gray`, which copies pixels from one im-
age into another.

A.3.6   Image Tuples

Operators for creating and accessing image tuples can be found in the HDevelop menu
`Operators ▷ Object ▷ Manipulation`. Image tuples can be created using the operators
`gen_empty_obj` and `concat_obj`, while the operator `select_obj` allows to access an indi-
vidual image that is part of a tuple.

B   Parameters Describing the Image

When opening a connection with `open_framegrabber`, you can specify the desired image format, e.g., ist size or the number of bits per pixel, using 9 parameters, which are described in the following.

B.1   Image Size

The following 6 parameters influence the size of the grabbed images: `HorizontalResolution` and `VerticalResolution` specify the *spatial resolution* of the image in relation to the original size. For example, if you choose `VerticalResolution` = 2, you get an image with half the height of the original as depicted in figure 17b. Another name for this process is (vertical and horizontal) *subsampling*.

With the parameters `ImageWidth`, `ImageHeight`, `StartRow`, and `StartColumn` you can grab only a part of the (possibly subsampled) image; this is also called *image cropping*. In figure 17, image part to be grabbed is marked with a rectangle in the original (or subsampled) image; to the right, the resulting image is depicted. Note that the resulting HALCON image always starts with the coordinates (0,0), i.e., the information contained in the parameters `StartRow` and `StartColumn` cannot be recovered from the resulting image.

Depending on the involved components, both subsampling and image cropping may be executed at different points during the transfer of an image from the camera into HALCON: in the camera, in the frame grabber, or in the software. Please note that in most cases you get no direct effect on the performance in form of a higher frame rate; exceptions are CMOS cameras which adapt their frame rate to the requested image size. Subsampling or cropping on the software side has no effect on the frame rate; besides, you can achieve a similar result using `reduce_domain`. If the frame grabber executes the subsampling or cropping you may get a positive effect if the PCI bus is the bottleneck of your application and prevents you from getting the
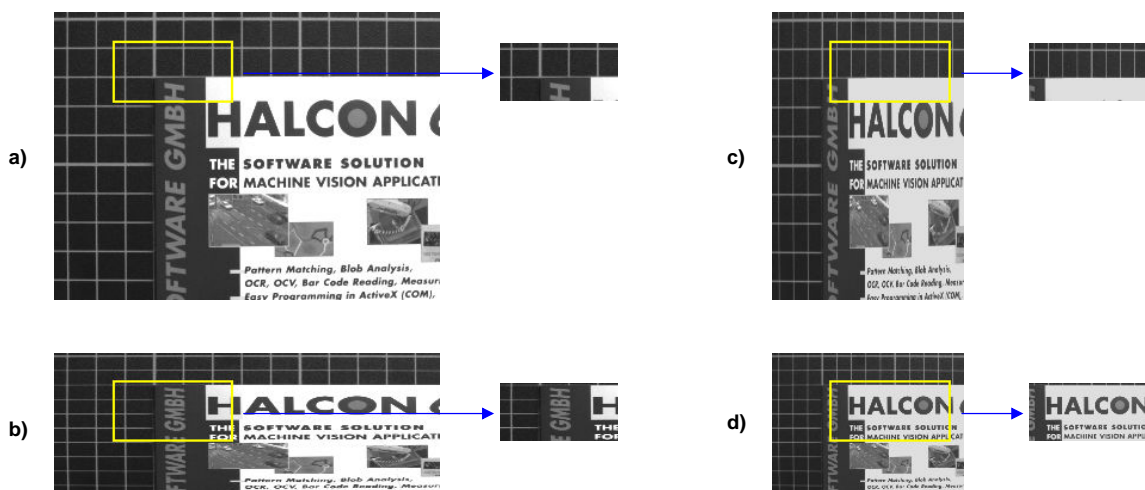

Figure 17: The effect of image resolution (subsampling) and image cropping (`ImageWidth` = 200, `ImageHeight` = 100, `StartRow` = 50, `StartColumn` = 100): a) `HorizontalResolution` (HR) = `VerticalResolution` (VR) = 1; b) HR = 1, VR = 2; c) HR = 2, VR = 1; d) HR = VR = 2.

full frame rate. Some frame grabber interfaces allow *dynamic image cropping* via the operator `set_framegrabber_param`.

Note that HALCON itself does not differentiate between area and line scan cameras as both produce images – the former in form of frames, the latter in form of so-called *pages* created from successive lines (number specified in the parameter `ImageHeight`). Section 6.3 contains additional information regarding the use of line scan cameras.

B.2 Frames vs. Fields

The parameter `Field` is relevant only for analog cameras that produce signals following the video standards originally developed for TV, e.g., NTSC or PAL. In these standards, the camera transmits images (also called *frames*) in form of two so-called *fields*, one containing all odd lines of a frame, the other all even lines of the next frame. On the frame grabber board, these two fields are then *interlaced*; the resulting frame is transferred via the PCI bus into the computer memory using DMA (*direct memory access*).

Figure 18 visualizes this process and demonstrates its major draw back: If a moving object is observed (in the example a dark square with the letter 'T'), the position of the object changes from field to field, the resulting frame shows a distortion at the vertical object boundaries (also called *picket-fence effect*). Such a distortion seriously impairs the accuracy of measurements; industrial vision systems therefore often use so-called *progressive scan* cameras which transfer full frames (see figure 19). Some cameras also "mix" interlacing with progressive scan as depicted in figure 20.

You can also acquire the individual fields by specifying `VerticalResolution` = 2. Via the parameter `Field` you can then select which fields are to be acquired (see also figure 21): If you select 'first' or 'second', all you get odd or all even fields, respectively; if you select 'next', you get every field. The latter mode has the advantage of a higher field rate, at the
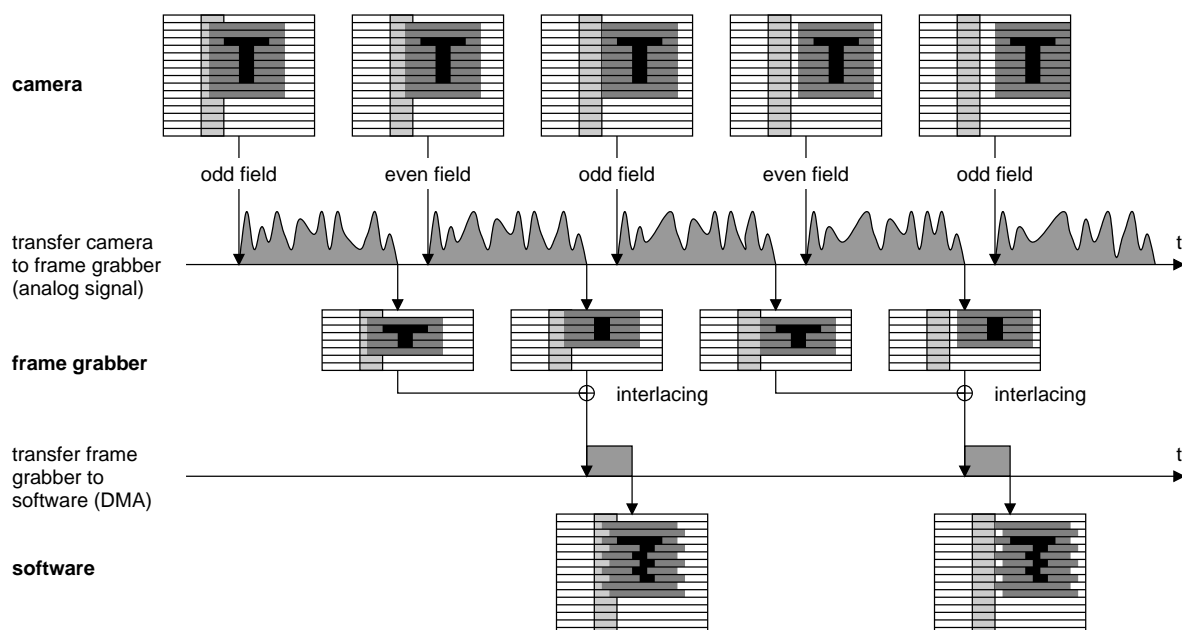

Figure 18: Interlaced grabbing (`Field` = 'interlaced').
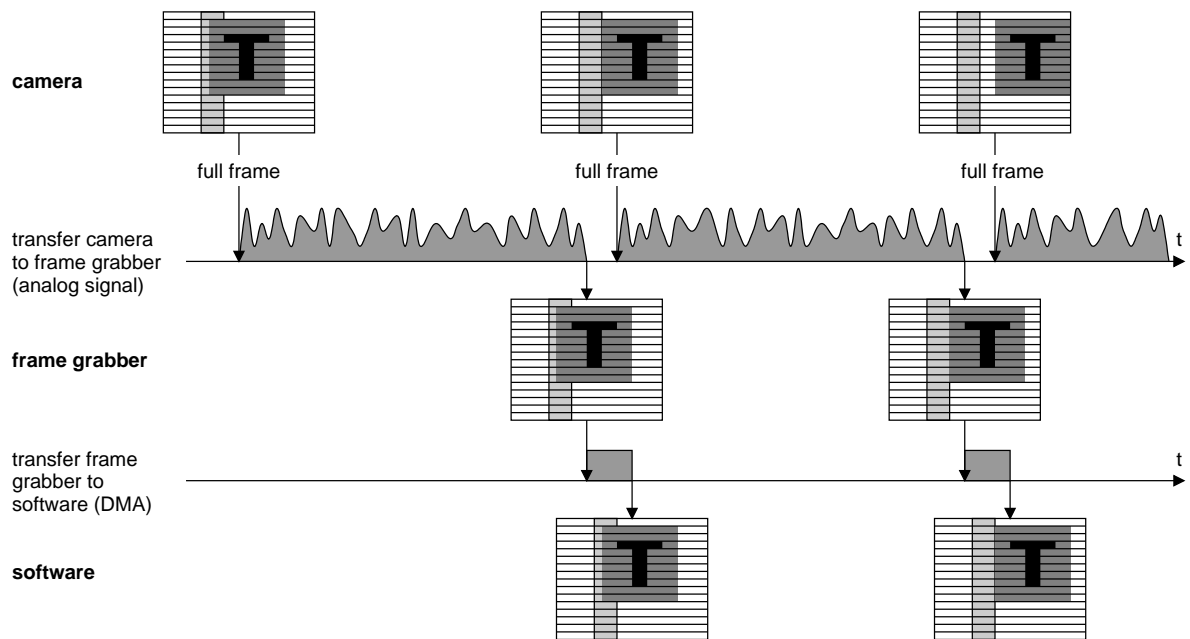
Figure 19: Progressive scan grabbing (`Field` = 'progressive').
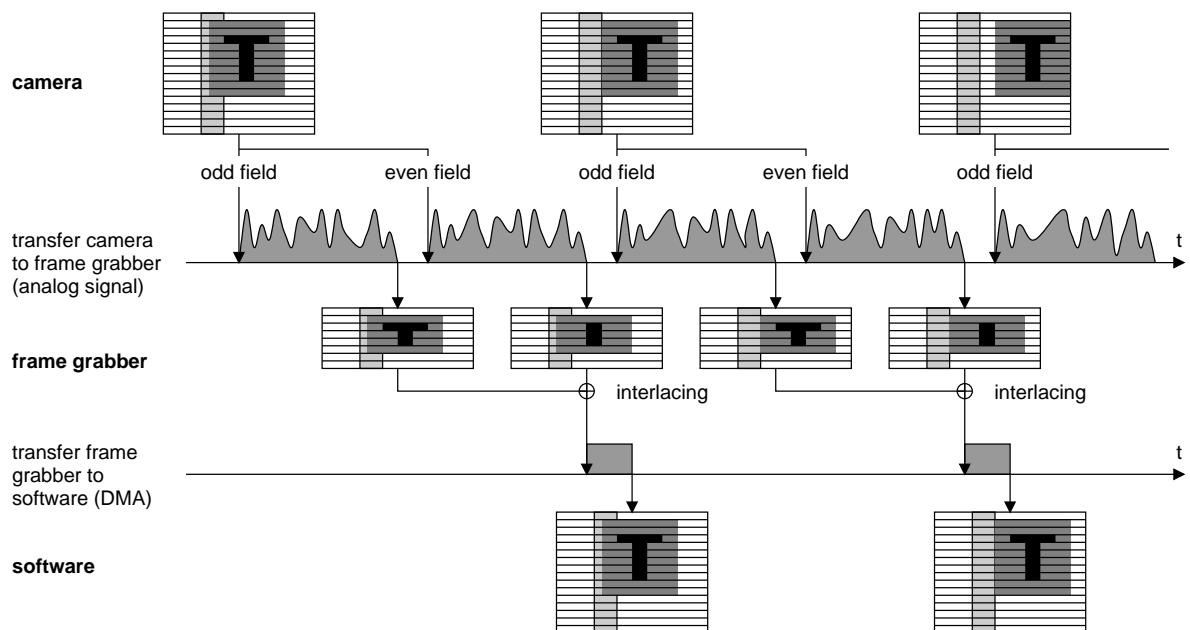

Figure 20: Special form of interlaced grabbing supported by some cameras.

cost, however, of the so-called *vertical jitter*: Objects may seem to move up and down (like the square in figure 21), while structures that are one pixel wide appear and disappear (like the upper part of the 'T').

By specifying `Field` = 'first', 'second', or 'next' for a full resolution image (`VerticalResolution` = 1), you can select with which field the interlacing starts.

Figure 22 shows a timing diagram for using `grab_image` together with an interlaced-scan camera. Here, you can in some cases increase the processing frame rate by specifying 'next' for the parameter `Field`. The frame grabber then starts to digitize an image when the next field
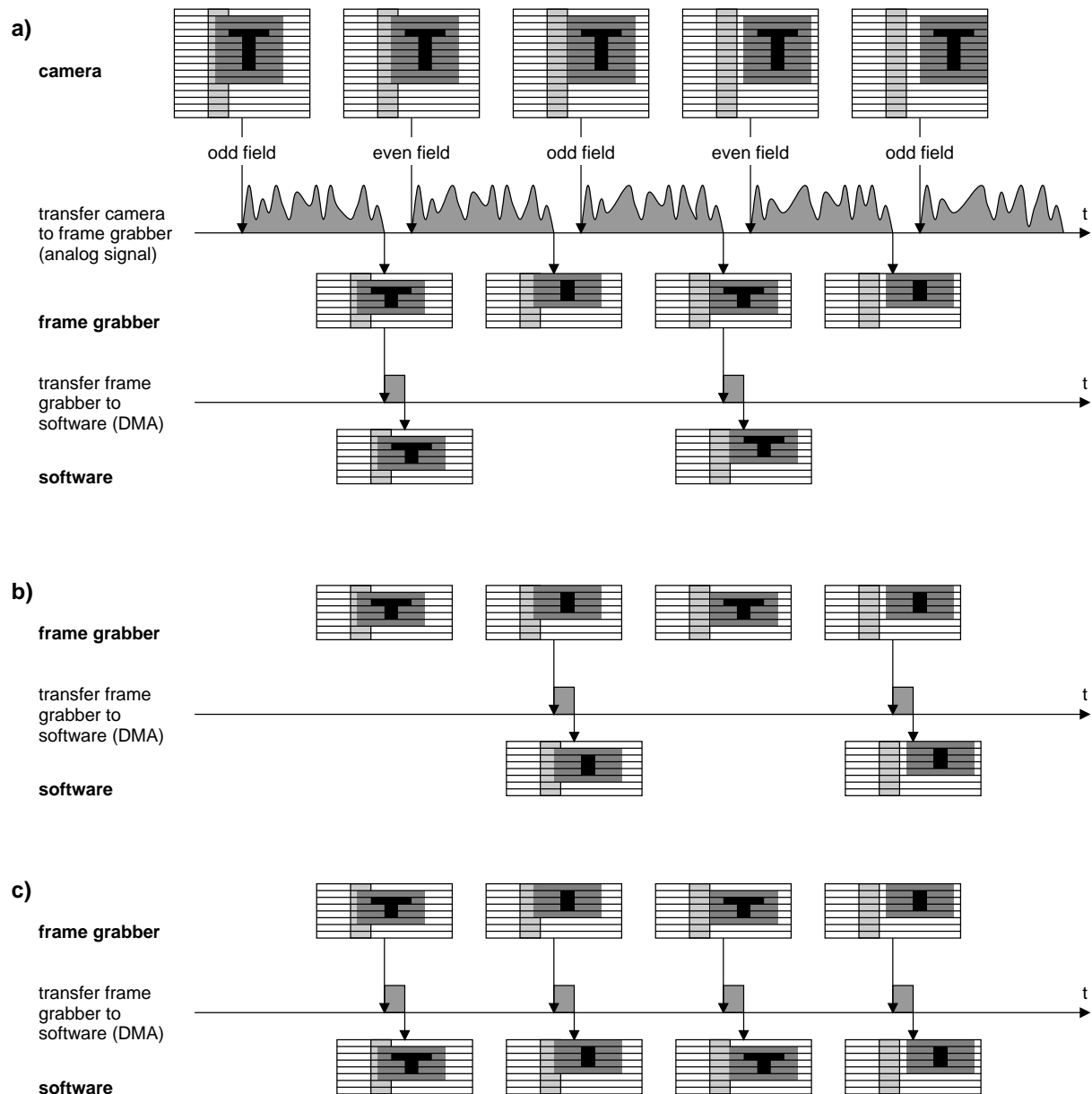
Figure 21: Three ways of field grabbing: a) 'first'; b) 'second'; c) 'next'.

arrives; in the example therefore only one field is lost.

B.3 Image Data

The parameters described in the previous sections concentrated on the size of the images. The *image data*, i.e., the data contained in a pixel, is described with the parameters `BitsPerChannel` and `ColorSpace`. To understand these parameters, a quick look at the HALCON's way to represent images is necessary: A HALCON image consists of one or more matrices of pixels, which are called *channels*. Gray value images are represented as single-channel images, while color images consist of three channels, e.g., for the red, green, and blue part of an RGB image. Each image matrix (channel) consists of *pixels*, which may be of different *data types*, e.g., standard 8 bit (type `byte`) or 16 bit integers (type `int2` or `uint2`) or 32 bit floating
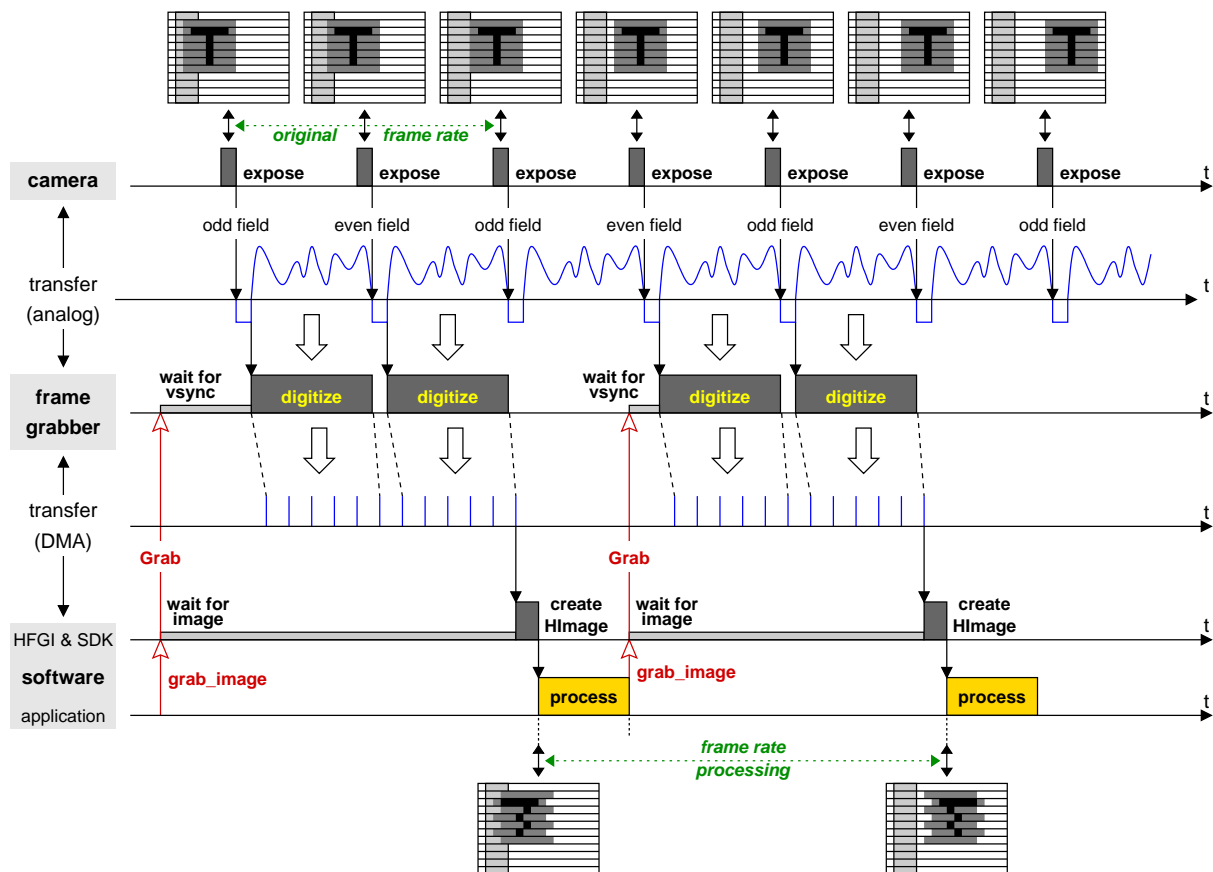
Figure 22: Grabbing interlaced images starting with the 'next' field.

point numbers (type real). For detailed information about HALCON images please refer to appendix A.

The two parameters correspond to the two main aspects of HALCON's images: With the parameter ColorSpace you can select whether the resulting HALCON image is to be a (single-channel) gray value image (value 'gray') or a (multi-channel) color image (e.g., value 'rgb'). The parameter BitsPerChannel specifies how many bits are *transmitted* per pixel per channel from the frame grabber to the computer; the pixel type of the HALCON image is then chosen to accommodate the transmitted number of pixels.

For example, if a frame grabber is able to transmit 10 bit gray value images, one selects ColorSpace = 'gray' and BitsPerChannel = 10 and gets a single-channel HALCON image of the type 'uint2', i.e., 16 bit per channel. Another example concerns RGB images: Some frame grabbers allow the values 8 and 5 for BitsPerChannel. In the first case, $3 \times 8 = 24$ bit are transmitted per pixel, while in the second case only $3 \times 5 = 15$ (padded to 16) bit are transmitted; in both cases, a three-channel 'byte' image results.