This manual describes how to create HALCON Image Acquisition Interfaces, Version 11.0

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission of the publisher.

Edition 1    July 1997
Edition 2    November 1997
Edition 3    March 1998        (HALCON 5.1)
Edition 4    April 1999       (HALCON 5.2)
Edition 5    November 2000    (HALCON 6.0)
Edition 6    June 2002        (HALCON 6.1)
Edition 7    December 2003    (HALCON 7.0)
Edition 7a   April 2005       (HALCON 7.0.2)
Edition 8    June 2007        (HALCON 8.0)
Edition 9    December 2008    (HALCON 9.0)
Edition 10   November 2011    (HALCON 10.0)
Edition 10a  November 2011    (HALCON 10.0.2)
Edition 11   May 2012         (HALCON 11.0)

Copyright © 1997-2012 by MVTec Software GmbH, München, Germany


All other nationally and internationally recognized trademarks and tradenames are hereby recognized.

More information about HALCON can be found at: http://www.halcon.com/
About This Manual

This manual describes the basic techniques needed to integrate third-party image acquisition hardware (*frame grabber boards* or digital cameras) into the HALCON system.

The manual is written for the expert HALCON user who wants to integrate a new image acquisition device. The reader should be familiar with the standard HALCON system. Furthermore, C programming skills\(^1\) are required. Finally, detailed knowledge about the API of the used image acquisition device will be necessary.

If you are first interested in the basics of the HALCON image acquisition interface (from the user’s point of view), please refer to the *Solution Guide II-A*.

The manual is divided into the following parts:

- **Introduction**
  This chapter explains the basics of image acquisition and introduces the HALCON image acquisition interface and the underlying concepts.

- **Data Structures**
  In this chapter, the basic data structures of the image acquisition interface are described.

- **Interface Routines**
  This chapter explains all the routines you have to implement inside your acquisition interface.

- **Generating an Acquisition Interface Library**
  This chapter contains information on how to generate a dynamic object encapsulating your acquisition interface.

- **Changes between versions 4 and 5 of the HALCON image acquisition interface**
  This appendix describes the differences between the versions 4 and 5 of the HALCON image acquisition interface.

- **Changes between versions 3 and 4 of the HALCON image acquisition interface**
  This appendix describes the differences between the versions 3 and 4 of the HALCON image acquisition interface.

- **Changes between versions 2 and 3 of the HALCON image acquisition interface**
  This appendix describes the differences between the versions 2 and 3 of the HALCON image acquisition interface.

---

\(^1\)Naturally, this also includes knowledge about the programming environment (how to invoke the compiler/linker etc.).
• Changes between versions 1 and 2 of the HALCON image acquisition interface
  This appendix describes the differences between the versions 1 and 2 of the HALCON image
  acquisition interface (formerly known as HALCON frame grabber integration interface).

• HALCON Error Codes
  This appendix describes all error codes which you may use for programming an acquisition inter-
  face.
Contents

1 Introduction
   1.1 HALCON’s Generic Image Acquisition Interface ................................................. 7
   1.2 Image Acquisition Basics ...................................................................................... 9
   1.3 Synchronous vs. Asynchronous Grabbing ................................................................. 10
   1.4 Buffering Strategies ............................................................................................... 11
   1.5 A/D Conversion and Multiplexing ........................................................................... 11
   1.6 HALCON Image Acquisition Operators .................................................................. 12
      1.6.1 open_framegrabber ......................................................................................... 12
      1.6.2 close_framegrabber ......................................................................................... 13
      1.6.3 close_all_framegrabbers ................................................................................. 13
      1.6.4 info_framegrabber ............................................................................................ 13
      1.6.5 grab_image ........................................................................................................ 14
      1.6.6 grab_image_async ............................................................................................ 14
      1.6.7 grab_image_start .............................................................................................. 14
      1.6.8 grab_data .......................................................................................................... 14
      1.6.9 grab_data_async ............................................................................................... 15
      1.6.10 set_framegrabber_param, get_framegrabber_param ..................................... 15
      1.6.11 set_framegrabber_lut, get_framegrabber_lut ................................................. 15
      1.6.12 set_framegrabber_callback, get_framegrabber_callback ............................ 15
   1.7 Parameter Naming Conventions ............................................................................ 16
   1.8 Optimal Interaction with ActivVisionTools and HDevelop Image Acquisition Assistant . 17
   1.9 Image Acquisition Integration vs. Image Acquisition Interface ............................ 19

2 Data Structures ........................................................................................................ 21
   2.1 Acquisition Interface Classes and Instances ......................................................... 21
      2.1.1 Structure FGClass ............................................................................................ 22
      2.1.2 Structure FGInstance ...................................................................................... 24
   2.2 Recommended Auxiliary Structures ....................................................................... 25
      2.2.1 Structure BoardInfo ....................................................................................... 25
      2.2.2 TFGInstance ................................................................................................... 26

3 Interface Routines ................................................................................................... 29
   3.1 FGInit() .................................................................................................................. 29
   3.2 FGOpenRequest() ................................................................................................. 33
   3.3 FGOpen() .............................................................................................................. 34
3.4  FGClose() .............................................................. 44
3.5  FGInfo() .............................................................. 45
3.6  FGGrab() ............................................................... 51
3.7  Auxiliary Routine: GrabImg() .................................... 54
3.8  FGGrabAsync() ....................................................... 59
3.9  FGGrabStartAsync() ............................................... 59
3.10 FGGrabData() ......................................................... 61
3.11 FGGrabDataAsync() ............................................... 64
3.12 FGSetParam() ........................................................ 65
3.13 FGGetParam() ........................................................ 68
3.14 FGSetLut() ........................................................... 69
3.15 FGGetLut() ........................................................... 70
3.16 FGSetCallback() ...................................................... 70
3.17 FGGetCallback() ...................................................... 71

4  Generating an Acquisition Interface Library .................. 73
4.1  Windows ................................................................. 73
4.2  Linux ..................................................................... 74

A  Changes Between Versions 4 and 5 of the HALCON Image Acquisition Interface 75
B  Changes Between Versions 3 and 4 of the HALCON Image Acquisition Interface 77
C  Changes Between Versions 2 and 3 of the HALCON Image Acquisition Interface 79
D  Changes Between Versions 1 and 2 of the HALCON Image Acquisition Interface 81
E  HALCON Error Codes .................................................. 85

Index ............................................................................... 87
Chapter 1

Introduction

This chapter provides an introduction to the HALCON image acquisition interface and the underlying concepts. It is intended for users who are not familiar with topics like frame grabber hardware, A/D-conversion, synchronous or asynchronous mode of operation, buffering strategies, and the like. Although this manual is not intended to supply you with detailed knowledge about your image acquisition device’s internals, we still want to give explanations of the basic terms and methods. Reading the manuals supplied with your image acquisition device is a necessity, of course, and possibly gives you a much more detailed view on the things being discussed here.

To get familiar with the image acquisition within HALCON from the user’s point of view, it’s strongly recommended to read the corresponding solution guide first. The Solution Guide II-A is part of HALCON’s Solution Guide II and can be found in the directory %HALCONROOT%/doc/pdf/solution_guide.

Unless stated otherwise, all notations refer to Windows conventions. Thus, for example file paths and environment variables are printed like

```
%HALCONEXAMPLES%\ia_integration\source\hAcqTemplate.c
```

If you are using a Linux system you have to consider the corresponding Linux syntax.

1.1 HALCON’s Generic Image Acquisition Interface

HALCON provides a generic image acquisition interface that allows free integration of new frame grabbers or cameras on the fly, that is even without restarting a HALCON application.
The two basic concepts used are

- Encapsulation of the interface code in dynamically loadable modules.\(^1\)
- A set of predefined HALCON operators for image acquisition, including operators for setting and retrieving specific hardware parameters. The latter allow the parameterization of even the most "exotic" acquisition devices.

If you have successfully developed a new HALCON image acquisition interface (based on the detailed information given in this manual), then all you have to do to use your new image acquisition device is

- Plug in the hardware and install the vendor-specific device driver and libraries.
- Copy the new HALCON interface (i.e., the loadable module with the encapsulated hardware-dependent code) to a directory within your search path for DLLs or shared libraries, respectively. For the proper prefix of the filename of the new HALCON interface see chapter 4 on page 73.
- Specify the name of the new image acquisition device (i.e., the name of the corresponding interface) in the open_framegrabber operator.
- Enjoy the performance of all the features you have integrated in your new image acquisition interface.

The HALCON operators used for image acquisition remain the same, so existing application code can be used without modification\(^2\) in most cases. HALCON automatically loads the interface during the first call to info_framegrabber or open_framegrabber. Thus, you can exchange/add image acquisition interfaces even without restarting your application. Special features of different image acquisition devices can be accessed through the general purpose parameter setting mechanism.

Since digital cameras which are connected by USB, IEEE 1394 or GigE are not really based on an actual frame grabber board, we no longer use the term HALCON frame grabber interface. Instead, we use the term HALCON acquisition interface, and the term image acquisition device is used as a substitute for either a frame grabber board or a digital camera. For backwards compatibility reasons, the names of the HALCON operators and also the underlying names for the data structures and error codes have been unchanged. Thus, operator names like open_framegrabber, info_framegrabber, and close_framegrabber may sound a little bit old-fashioned.

**Example files**

As a guideline for the image acquisition integration, the HALCON distribution contains a template for an image acquisition interface (see hAcqTemplate.c in %HALCONEXAMPLES%/\ia_integration\source). It covers most situations you might encounter while programming such an interface (like supporting multiple boards with multiple cameras per board etc.). Although the template is extensively commented, it might be quite tough to understand the code prior to reading this manual. On the other hand, it can provide a powerful skeleton for a wide range of integrations.

For a list of all the image acquisition devices that are currently supported by HALCON, please check http://www.halcon.com/ or contact your local distributor to get the latest releases of the HALCON image acquisition interfaces. You can find an up-to-date list of all currently supported image acquisition devices at http://www.halcon.com/image-acquisition.

---

\(^1\)DLLs for Windows, shared libraries for Linux systems.

\(^2\)Except for the new name of the image acquisition device and its specific parameters used by the operator set_framegrabber_param.
1.2 Image Acquisition Basics

Note that the following sections in this chapter mainly describe the case that an analog camera is connected to a frame grabber board. However, most of the principles are also important in case of digital capture devices like IEEE 1394, USB, or GigE cameras.

Basically, what a frame grabber does is to take a video signal, which can be understood as a continuous stream of video frames, and grab one or more video frames out of the sequence, whenever triggered to do so. In many cases, the video signal will be an analog one, although more professional equipment often uses digital signals nowadays. The most common analog video formats are

- **NTSC**: $640 \times 480$ pixel, 30 frames per second and
- **PAL**: $768 \times 576$ pixel, 25 frames per second.

Both formats carry color information, although many image acquisition devices only deliver grayscale images, even from a color video signal. The following explanations assume that you are using an analog frame grabber board. With digital boards, things may be different.

Let us take a look at the analog input signal: Actually, it is composed of many different signals: There are vertical and horizontal sync signals and, of course, the raw data signals as well. Sometimes, the color and brightness signals are overlaid (composite signal), sometimes they are delivered on separated input lines ($Y/C$, RGB). Since the frame grabber is usually synchronized by the video source, it has to wait for the next vertical sync signal to start grabbing a new image, see figure 1.1.

![Figure 1.1: Grabbing one frame.](image)

This will cause a delay of half a frame on average when grabbing an image of random frames\(^4\). It also implies that you have to start grabbing the next frame immediately\(^5\) after receiving the previous frame if you want to achieve full frame rate. Consequently, there would be no time at all left to process images. In this synchronous mode, the host computer is exclusively busy triggering one grab after another. Therefore, HALCON also supports asynchronous grabbing as explained in the next section.

---

\(^3\)At least if you do not use a setup that supports asynchronous frame resets.

\(^4\)Naturally, this is not true if the camera supports an asynchronous reset mode, i.e. the camera starts the new grab almost immediately.

\(^5\)There is a very short sync period before the next frame starts.
1.3 Synchronous vs. Asynchronous Grabbing

To understand what asynchronous grabbing means, we first should take a look at what the image acquisition device does with a grabbed frame. It is easily understood, that a digitized frame must be stored in some kind of memory. Basically, there are three possibilities:

- Device memory on the image acquisition device
- Device memory on the host machine
- Host memory

Device memory on the board means dedicated memory, physically mounted to the board. This way, the image acquisition device can store the acquired image(s) directly in its own memory, with each process on the host being able to get the data at any time. On the other hand, memory size is fixed. If it is too small, it may not be possible to keep several images in memory. If it is very big, the whole board can get rather expensive. Device memory on the host machine is non-paged system memory dynamically allocated by the frame grabber’s device driver. Thus, the memory size can be easily adjusted. On the other hand, heavy bus traffic is likely to occur, if the image acquisition device is delivering data to the host computer’s memory permanently. Therefore, we usually do not use continuous grabbing modes provided by some image acquisition devices, but grab images only on demand. Host memory is allocated by the user somewhere in the address space of the application. Since this memory might use paging, the images delivered from the image acquisition device in general must be explicitly copied to this memory (since DMA will fail).

The host computer’s job, as mentioned above, is to trigger the image acquisition device when a new image is needed, but it does not necessarily need to wait while the device digitizes the frame. With device memory being on the board, this is self-evident, but also if the target memory is host-based, externally initiated data transfer is usually possible with techniques like DMA. So the “only thing” the host process has to do is to trigger the image acquisition device an average time of 1 1/2 frames before an image is actually needed, and then it can do some other processing while the new frame is captured by the board in the background. This technique is called asynchronous grabbing. It is easily understood that this eases real-time grabbing, since the time needed for frame completion is rather long (40 ms with PAL, 33 ms with NTSC video) compared to the small time gap between two adjacent frames in the video stream.

Most image acquisition devices support asynchronous data transfer. Therefore, HALCON provides both synchronous (grab_image) and asynchronous grabbing (grab_image_async). The reason for supporting the somehow less powerful synchronous mode is the “clearer” semantics: The operator grab_image starts a grab and waits until it is finished. Thus, the delivered image is per definition up to date. Using asynchronous grabbing needs a little bit more insight in the timing of the application. The grabbed images might be too old to be used otherwise. Now let us take a look at some memory management strategies useful for efficient image acquisition in the next section.

---

6When grabbing a PAL signal with a RGB image acquisition device using a 32 bit per pixel representation, more than 42 Megabytes per second have to be transferred to the host.

7Direct Memory Access.
1.4 Buffering Strategies

Let us look back at the real-time grabbing problem: Assuming a board capable of asynchronous transfer, a possible sequence to choose is:

1. Trigger a grab (control returns immediately to the calling process).
2. Wait for the grab to finish.
3. Trigger the next grab.
4. Process the image resulting from step 2.
5. Go back to step 2.

This sequence corresponds to the simple HALCON program

```halcon
while(1)
    grab_image_async(Image,-1)
    < process Image >
end while
```

Since steps 1 and 3 (starting an asynchronous grab) do not block the process, no time is wasted while the image acquisition device is busy. The only topic left to think about is how the memory used for grabbing should be organized: Assuming step 3 makes the image acquisition device deliver data into a dedicated memory area without knowledge about what the host is doing in step 4, it is easily seen, that the image acquisition device must use a different memory area than the host. If not, the device might write to memory the host is reading at the same time and the processed image would be corrupt. The best way to handle this problem is to use two alternative buffers: One to write new data into, the other to hold the previous image. These buffers might be allocated only once before the cycle is started. They exchange their role after every iteration (the buffer the image acquisition device wrote to becomes the process’s reading buffer and vice versa). This is a very common technique whenever asynchronous data transfer is involved and is called double buffering. Since older image data is overwritten, we also use the term volatile grabbing.

A technique like this offers maximum grabbing performance. On the other hand, flexibility decreases. Obviously, the older images are overwritten again and again. So all the “history” is lost. This strict organization is a contradiction to the general HALCON philosophy that allows to create an arbitrary number of iconic objects and to process them in parallel until you decide in the application that you do not need them anymore. Therefore, a HALCON image acquisition interface always should create new image objects by default and offer volatile grabbing only as an additional option, see section 3.6 on page 51.

1.5 A/D Conversion and Multiplexing

Still bearing in mind that we are talking about analog video, we now take a quick look at the interface through which analog and digital domains are connected: the A/D converter. We are not interested
in details, except that a frame grabber’s A/D converter needs to be synchronized to the video signal in order to keep track with subsequent lines (horizontal sync) and frames (vertical sync). The sync information is either encoded in the analog video signal or is delivered to the frame grabber through additional input lines, so the A/D circuitry is able to synchronize itself to the video source. This is important to know, if we consider frame grabber boards having multiple input lines: In most cases, rather expensive and complex additional A/D converters are traded off against one analog multiplexer circuit, allowing multiple video sources to be connected to one A/D converter selectively. This means, that every time a new video source is about to be connected to the A/D converter, the circuit has to re-synchronize itself to the new signal, which usually means one or two frames being lost (in some cases, synchronization can take much longer, up to one second). To avoid this, one might use genlocked cameras. Please keep in mind that in general you have to adapt parameter settings on your frame grabber board whenever you switch between different input lines. HALCON provides a concept for dealing with multiple cameras connected to one frame grabber board (as well as multiple frame grabber boards inside one host computer): Each camera/board pair is represented by an image acquisition handle. Inside HALCON such a handle corresponds to an image acquisition instance. If you would like to support multiple cameras/boards with your image acquisition interface you have to keep track of all instances corresponding to your image acquisition class, see chapter 2 on page 21 and chapter 3 on page 29.

1.6 HALCON Image Acquisition Operators

This section provides a short overview of the HALCON image acquisition operators (please refer to the reference manuals for additional information). These operators are internally mapped to the image acquisition interface routines you have to provide for a new HALCON image acquisition interface, see chapter 3 on page 29.

1.6.1 open_framegrabber

The operator open_framegrabber is used to create a new image acquisition handle. It loads the specified image acquisition interface and accesses the image acquisition device itself. Moreover, the typical parameters for standard cameras are set (like image size and part, color space, port, etc.). If the image acquisition device (that is the driver as well as your interface) supports multiple boards inside one host computer, you also specify the desired board (using the parameter Device). It is also possible to use more than one camera per board. In that case you create an image acquisition handle for each camera by a sequence of open_framegrabber calls (specifying the camera via the parameters Port or LineIn). Note, that you have to handle multiple instances inside your image acquisition interface if you would like to support multiple cameras or boards. If you use digital devices you don’t have to care about A/D conversion or multiplexing. So each device gets its own image acquisition handle and (inside HALCON) an image acquisition instance.

In detail, this HALCON operator will call your interface routines FGOpenRequest() (see section 3.2 on page 33) and FGOpen() (see section 3.3 on page 34). In addition, FGInit() (see section 3.1 on page 29) will be called when you access a specific image acquisition device for the very first time.
1.6.2 close_framegrabber

The operator close_framegrabber is the counterpart to open_framegrabber. It deallocates an image acquisition handle, releases the associated memory, and unlocks the image acquisition device depending on whatever you program in the underlying interface routine FGCLOSE() (see section 3.4 on page 44).

1.6.3 close_all_framegrabbers

The operator close_all_framegrabbers is a convenience operator that calls close_framegrabber for all image acquisition handles in use. This can be very useful, e.g., if you have forgotten to close an image acquisition instance before loading a new program in HDevelop: The variables containing the old handles are cleared and thus there is no other way left to “unlock” image acquisition devices. However, note that this operator has severe side-effects. It closes all image acquisition devices, but it cannot change the handles in your program. Thus, it is in your responsibility not to use these handles later on.

Since close_all_framegrabbers is based on close_framegrabber you do not have to provide specific routines for this operator inside your image acquisition interface.

1.6.4 info_framegrabber

The operator info_framegrabber is used to access basic information about a specific image acquisition device (and the corresponding interface). Note, that since many parameter settings depend on the specific properties of an image acquisition device, HALCON generally can neither provide meaningful defaults nor check parameters automatically. However, the calls to info_framegrabber should return meaningful values, especially the ones with the value lists for the parameters in open_framegrabber. Please use dynamic lists wherever it is possible.

A call of info_framegrabber should return the same values, regardless if a device has already been opened or not. In special cases more values can be returned if a device handle already exists. Furthermore, most of these return values are used by the HDevelop Image Acquisition Assistant and should work directly in open_framegrabber (see also section 1.8 on page 17). An exception is the parameter info_boards. It has to return a list of the actually available devices, each starting with 'device:' 8. If the return value is empty, the name of the image acquisition interface won’t be listed after an auto-detection or in the HDevelop Image Acquisition Assistant. info_boards can also return additional information about the available devices like, e.g., port, IP address or serial number. Another "special" parameter is camera_type. The return values can vary between video formats, camera models, configuration files and some exotic ones. In case of configuration files there is a syntax, which improves the selection of the file. Therefore, three values have to be returned: 'CAMFILE:', the file extension of the configuration file, and the full path, where to start searching.

This operator will call the routine FGInfo() in your image acquisition interface (see section 3.5 on page 45).

8To handle device strings containing whitespaces each string has to be put between two pipe characters ']'
1.6.5 grab_image

The operator `grab_image` is used to grab a new image *synchronously*, that means a new grab is started and the operator *waits* until this grab has been finished.

This operator will call your interface routine `FGGrab()` (see section 3.6 on page 51).

1.6.6 grab_image_async

The operator `grab_image_async` grabs a new image *asynchronously*. It waits until a pending asynchronous grab has been finished (if you got the timing right this grab should be finished already to prevent wasting time at this point). This image is then returned unless it is older than a specified threshold. Otherwise a new (synchronous) grab is performed. Afterwards, `grab_image_async` triggers a new asynchronous grab and returns without further waiting.

This operator will call the routine `FGGrabAsync()` (see section 3.8 on page 59) in your image acquisition interface. If this routine is missing, the error code `H_ERR_FGASYNC` (“Image acquisition: asynchronous grab not supported”) will be returned. Thus, if you do not want to support asynchronous grabbing, just do not provide `FGGrabAsync()`.

1.6.7 grab_image_start

The operator `grab_image_start` *starts* the asynchronous grabbing of a new image and returns immediately. The image itself is then delivered by the next call to `grab_image_async` or `grab_data_async`. If an asynchronous grab was already started beforehand, `grab_image_async` terminates this pending grab and starts a new one. This operator is useful if your application involves time consuming processing. In this case, asynchronously grabbed images might be too old if you start the grab immediately after grabbing the prior image (via `grab_image_async`). `grab_image_start` allows you to fine-tune the moment you start the grab. In case of a free-running camera call this operator approximately one and a half frames before you need the next image.

This operator will call the routine `FGGetGrabStartAsync()` (see section 3.9 on page 59) in your image acquisition interface. If this routine is missing, the error code `H_ERR_FGASYNC` (“Image acquisition: asynchronous grab not supported”) will be returned.

1.6.8 grab_data

The operator `grab_data` grabs a new image *synchronously*, that means a new grab is started and the operator *waits* until this grab has been finished. It can also return preprocessed data in terms of images, regions, XLD contours, and control data. The kind of preprocessing used is up to *you* (and maybe dependent on some specific hardware features of your image acquisition device).

This operator will call your interface routine `FGGrabData()` (see section 3.10 on page 61). If this routine is missing, the error code `H_ERR_FGFNS` (“Image acquisition: function not supported”) will be returned.
1.6.9 grab_data_async

The operator `grab_data_async` grabs a new image *asynchronously*, that means it waits for a pending grab to finish and starts a new asynchronous grab again before returning. Similar to `grab_data` it can also return preprocessed data in terms of images, regions, XLD contours, and control data. The kind of preprocessing used is up to you (and maybe dependent on some specific hardware features of your image acquisition device).

This operator will call your interface routine `FGGrabDataAsync()` (see section 3.11 on page 64). If this routine is missing, the error code `H_ERR_FGFNS` (“Image acquisition: function not supported”) will be returned.

1.6.10 set_framegrabber_param, get_framegrabber_param

The operators `set_framegrabber_param` and `get_framegrabber_param` are used to set or retrieve specific parameters of an image acquisition instance. They have been designed to allow the "fine-tuning" of your image acquisition hardware. For whatever you can think of as being useful to adjust on your board, just define corresponding parameters. You can either set single parameter values or tuples of parameter values. The latter case might be very useful if some parameters depend on each other and therefore have to be set within one call of `get_framegrabber_param`.

`get_framegrabber_param` and `set_framegrabber_param` do not evaluate the parameters themselves, but only pass them to your interface routines `FGSetParam()` (see section 3.12 on page 65) and `FGGetParam()` (see section 3.13 on page 68). Note, that since the name, values, and semantics of such parameters depend on the specific properties of an image acquisition device, HALCON can neither provide meaningful defaults nor check parameters automatically. This is all up to your image acquisition interface. If `FGSetParam()` or `FGGetParam()` are missing, the error code `H_ERR_FGPARAM` (“Image acquisition: unsupported parameter”) will be returned.

1.6.11 set_framegrabber_lut, get_framegrabber_lut

The operators `set_framegrabber_lut` and `get_framegrabber_lut` are used to set or retrieve color lookup tables of an image acquisition instance (thus supporting things like gamma correction or white balancing).

These operators will call your interface routines `FGSetLut()` (see section 3.14 on page 69) or `FGGetLut()` (see section 3.15 on page 70). If one of these routines is missing, the error code `H_ERR_FGFNS` (“Image acquisition: function not supported”) will be returned.

1.6.12 set_framegrabber_callback, get_framegrabber_callback

The operators `set_framegrabber_callback` and `get_framegrabber_callback` handle callbacks from a specific image acquisition interface. These callbacks should allow the asynchronous notification
of a HALCON application by a specific image acquisition interface or device, e.g., in case of exposure or image transfer end. Note that not all image acquisition interfaces have to support the use of these callbacks. If callbacks are supported, you should also add the parameter ‘available_callback_types’ to the operator get_framegrabber_param to inform the user about the names of the available callbacks.

These operators will call your interface routines FGSetCallback() (see section 3.16 on page 70) or FGGetCallback() (see section 3.17 on page 71). If one of these routines is missing, the error code H_ERR_FGFNS ("Image acquisition: function not supported") will be returned.

### 1.7 Parameter Naming Conventions

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Typical Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>'continuous_grabbing'</td>
<td>'enable','disable'</td>
</tr>
<tr>
<td>'do_abort_grab'</td>
<td>'true'</td>
</tr>
<tr>
<td>'grab_timeout'</td>
<td>&lt;int&gt;</td>
</tr>
<tr>
<td>'image_available'</td>
<td>0, 1</td>
</tr>
<tr>
<td>'num_buffers'</td>
<td>&lt;int&gt;</td>
</tr>
<tr>
<td>'revision'</td>
<td>'4.0','4.1',.....</td>
</tr>
<tr>
<td>'start_async_after_grab_async'</td>
<td>'enable','disable'</td>
</tr>
<tr>
<td>'volatile'</td>
<td>'enable','disable'</td>
</tr>
</tbody>
</table>

Figure 1.2: Standard names and values for recommended image acquisition parameters.

When developing a new frame grabber interface please follow the following notes about parameter names:

- We recommend to use the same parameter names and values as used in other HALCON image acquisition interfaces to ease the handling with different image acquisition interfaces, see also figure 1.2 for a list of recommended parameters. figure 1.3 shows a list of typical names for device specific parameters.

- All parameter names and also the valid string values of parameters should be denoted in lowercase notation.

- All parameters which can be set via set_framegrabber_param should also be asked for their current setting via get_framegrabber_param. This is especially important to achieve compatibility with ActivVisionTools.

- Special parameters (so-called action parameters) which do not set a new parameter value but initiate a certain action should use the prefix 'do_'. Obviously, there’s no need to access the current setting of these parameters by get_framegrabber_param.

- All parameters should have a corresponding parameter with the suffix '_description' containing a meaningful tooltip.
• If applicable, all parameters should have a corresponding parameter with the suffix '_range' or '_values' containing the necessary information about possible ranges or values.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Typical Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>'brightness'</td>
<td>&lt;int&gt;</td>
</tr>
<tr>
<td>'do_flush_buffers'</td>
<td>'true'</td>
</tr>
<tr>
<td>'do_force_trigger'</td>
<td>'true'</td>
</tr>
<tr>
<td>'contrast'</td>
<td>&lt;int&gt;</td>
</tr>
<tr>
<td>'exposure'</td>
<td>&lt;int&gt;</td>
</tr>
<tr>
<td>'frame_rate'</td>
<td>&lt;float&gt;</td>
</tr>
<tr>
<td>'gain'</td>
<td>&lt;int&gt; or &lt;float&gt;</td>
</tr>
<tr>
<td>'shaft_encoder'</td>
<td>'enable','disable'</td>
</tr>
<tr>
<td>'shutter'</td>
<td>&lt;int&gt;</td>
</tr>
<tr>
<td>'strobe'</td>
<td>'enable','disable'</td>
</tr>
<tr>
<td>'strobe_delay'</td>
<td>&lt;int&gt; or &lt;float&gt;</td>
</tr>
<tr>
<td>'strobe_duration'</td>
<td>&lt;int&gt; or &lt;float&gt;</td>
</tr>
<tr>
<td>'timestamp'</td>
<td>&lt;string&gt; or &lt;int&gt;</td>
</tr>
<tr>
<td>'trigger_mode'</td>
<td>'free-run','async'</td>
</tr>
<tr>
<td>'trigger_signal'</td>
<td>'rising','falling'</td>
</tr>
<tr>
<td>'trigger_source'</td>
<td>'line1','line2'</td>
</tr>
</tbody>
</table>

Figure 1.3: Typical names and values for additional image acquisition parameters representing parameters provided by the manufacturer's API.

1.8 Optimal Interaction with ActivVisionTools and HDevelop Image Acquisition Assistant

If you want to use a HALCON image acquisition interface in combination with ActivVisionTools or the HDevelop Image Acquisition Assistant please note the following items:

• Auto Detection

  – The user has the possibility to select an image acquisition interface either manually from the list or restrict this list to all image acquisition interfaces which are actually available on the computer (auto-detection). For all known interfaces which are present in the directory bin/%HALCONARCH% (or lib/$HALCONARCH, respectively) the auto-detection calls info_framegrabber(...,'info_boards',...). If a device is found and the return parameter contains a valid (i.e., non-empty) string, the interface will be added to the list of detected image acquisition interfaces.

  – The implementation of info_framegrabber(...,'info_boards',...) has to check for actually available devices. If multiple devices are present, this operator should return a tuple with all found devices. Each device string should contain the name 'device:' followed...
by the interface-specific device name. Multiple devices should use different names, e.g.,
['device:0 camera_type:Camera-X', 'device:1 camera_type:Camera-Y']. Further
information like port will also be parsed and may be useful inside this string.

- In some cases there is no API function available to call the required information
  without opening the device. In this case it is important that the call of
  \texttt{info\_framegrabber(...,'info\_boards',...)} returns an error if no underlying
driver/API is actually present, especially if there are no dependencies of the image acqui-
sition interface to vendor-specific dynamic libraries!

• **Opening a Device**

- For opening a device the user has to insert the necessary parameters. To provide this param-
eters, the necessary information is queried with calls of \texttt{info\_framegrabber}. All calls of
\texttt{info\_framegrabber} must be also possible without opening the device before.

- The following query types are actually called via \texttt{info\_framegrabber}: 'de-
defaults', 'info\_boards', 'port', 'camera\_type', 'horizontal\_resolution', 'vertical\_resolution',
'color\_space', 'bits\_per\_channel', 'field', 'external\_trigger', 'generic', and 'revision'. In case
of 'camera\_type', the returning string can also indicate that a file selection box should be used
in the GUI, see the template code for an example.

• **Parameterization**

- During the connection process all parameters are queried via
\texttt{info\_framegrabber(...,'parameters',...)} These parameters are displayed in
the parameter window, unless there exists a read-only parameter 'available\_param\_names'
that queries the actual available parameters in case of generic parameters, e.g., in the
GigEVision interface.

- If calling \texttt{get\_framegrabber\_param} for a single parameter returns an error code like
\texttt{H\_ERR\_FGPARAM}, the parameter is discarded.

- For each available parameter also the following parameters are checked automatically:
'*\_range' (if implemented, a slider is will be visible), '*\_values' (if implemented, a combo
box with the returned values will be displayed), and '*\_description' (if implemented, a tooltip
will appear).

- Read-only parameters should also be returned by the call of
\texttt{info\_framegrabber(...,'parameters\_readonly',...)}.

- Write-only parameters should also be returned by the call of
\texttt{info\_framegrabber(...,'parameters\_writeonly',...)}. To display a button
for a write-only parameter, you have to implement also the corresponding '_values'
parameter with exactly one value.

- If you want to exclude some parameters from the GUI, then these parameters should be re-
turned by the call of \texttt{info\_framegrabber(...,'parameters\_hidden',...)} Also tuple
parameters, which are not read-only, should be added to this list.

- Besides '*\_range', '*\_values', and '*\_description', there exist three more specific parameter
extensions: '*\_access' (if implemented, the parameters will be handled as read-only, write-
only, or read- and writable), '*\_category' (if implemented, groups of parameters based on
categories are shown), and '*\_visibility' (if implemented, groups of parameters based on the
possible user skill levels 'Beginner', 'Expert', and 'Guru' are displayed).
1.9  HALCON Image Acquisition Integration Interface vs. HALCON Image Acquisition Interface

The term “HALCON image acquisition interface” typically refers to an external module encapsulating the code that is specific for the image acquisition device; this is the one you have to provide. In contrast, the “HALCON image acquisition integration interface” is the module inside the HALCON library which is responsible for managing and accessing (external) image acquisition interface modules.

Whenever an image acquisition device is accessed for the very first time using open_framegrabber or info_framegrabber the corresponding (external) image acquisition interface, a dynamically loadable module,\(^{10}\) is loaded. For example, a call like

\[
\text{open\_framegrabber('GigEVision', ...)}
\]

will cause the HALCON library to load the module hAcqGigEVision.dll in the case of Windows or hAcqGigEVision.so for Linux systems, respectively. If you use HALCON XL, it will load the library hAcqGigEVisionxl.dll or hAcqGigEVisionxl.so, respectively. Please note, that in order for this mechanism to work all image acquisition libraries need the prefix hAcq (and the suffix xl in case of HALCON XL).

After the first call, the interface routines inside this module (programmed by you) will be called by the corresponding HALCON operators, see section 1.6 on page 12. Before we take a look at the data structures involved, we should bear in mind that some parts of the image acquisition device management take place in the HALCON library and others are up to your image acquisition interface. It is important to keep in mind who is responsible for what:

The HALCON library’s job is to:

- Maintain a list of image acquisition classes,
- Maintain a list of instances for each class, and to
- Decode and pre-process an operator’s parameters.

The interface’s job is to

- “Define” a class (e.g., filling the data structure with appropriate data),
- Manage multiple instances and their mutual dependencies,
- Interpret an operator’s parameters and map them to the underlying hardware, and to
- Grab images based on the image acquisition device’s application programming interface (API).

\(^{10}\) A DLL for Windows or a shared library for Linux systems, respectively.
Chapter 2

Data Structures

This chapter introduces the data structures provided by the HALCON image acquisition interface. Furthermore, it contains some recommendations on how to handle multiple image acquisition instances.

2.1 Acquisition Interface Classes and Instances

The HALCON image acquisition interfaces manage different image acquisition devices using *classes* and *instances*. Since HALCON is designed to access any number and combination of boards and cameras simultaneously\(^1\), situations may occur, where several devices controlled by the same interface, or devices using different interfaces, or a combination of both must be addressed. The mechanism chosen to handle situations like these uses classes and instances of classes:

A **class** represents a specific image acquisition device model (or a family of image acquisition device models) and its interface. The corresponding data structure contains all the function pointers needed to access the routines within the interface and the default parameter settings for `open_framegrabber`. Such an entry exists only once for each type of image acquisition device.

Each new **instance** created from this class represents either a specific board or camera belonging to this class or a multiplexed port on such a board. Figure 2.1 shows a possible configuration. The first two frame grabber boards in our example (a “model 1000” and a “model 2000”) are different boards from the same manufacturer (“foo labs”). Assuming these (similar) boards being controlled by the same image acquisition interface, they belong to the same class. Therefore, the input lines connected to camera 1 and 2 represent two instances of this class. The third frame grabber is a totally different one (bar inc.’s *mega-grabber*) and therefore another interface is needed – the second class. This frame grabber has two (probably multiplexed) ports attached, thus camera 3 and 4 can be understood as two instances of this second class.

---

\(^1\)Well, to be honest, there is a limitation: HALCON can handle 128 (`FG_MAX_NUM`) different image acquisition interfaces with up to 256 (`FG_MAX_INST`) instances each *at the same time*. 
2.1.1 Structure FGClass

The data structure FGClass encapsulates the data common to all instances of one image acquisition device model, see figures 2.2, 2.3, and 2.4. It is initialized by your interface routine FGInit(), see section 3.1 on page 29.

```c
typedef struct _FGClass {
    /* -------------------------- internal -------------------------------- */
    char name[MAX_STRING]; /* device name (interface module) */
    void *lib_handle; /* handle of interface library */
    INT interface_version; /* current HALCON image acquisition */
    /* interface version */
    /* ------------------- properties / management ------------------------ */
    HBOOL available; /* supported for the current platform */
    INT instances_num; /* current number instances (INTERNAL!) */
    INT instances_max; /* maximum number of instances */
    FGInstance *instance[FG_MAX_INST]; /* list of instances (INTERNAL!) */
    ...
};
```

You do not have to set all the members of the structure. Especially do not touch the INTERNAL entries like name or lib_handle. They are controlled by the HALCON library exclusively.
typedef struct _FGClass {
...
    /************************************************************************
    * interface-specific functions ---------------------------------------- *
    */
    FGInstance** (*OpenRequest)(Hproc_handle proc_id,FGInstance *fginst);
    Herror (*Open) (Hproc_handle proc_id,FGInstance *fginst);
    Herror (*Close) (Hproc_handle proc_id,FGInstance *fginst);
    Herror (*Grab) (Hproc_handle proc_id,FGInstance *fginst,
                    Himage *image,INT *num_image);
    Herror (*GrabStartAsync) (Hproc_handle proc_id,FGInstance *fginst,
                                  double maxDelay);
    Herror (*GrabAsync) (Hproc_handle proc_id,FGInstance *fginst,
                           double maxDelay,Himage *image,
                           INT *num_image);
    Herror (*GrabData) (Hproc_handle proc_id,FGInstance *fginst,
                        Himage **image,INT **num_channel,INT *num_image,
                        Hrlregion ***region,INT *num_region,
                        Hcont ***cont,INT *num_cont,
                        Hcpar **data,INT *num_data);
    Herror (*GrabDataAsync) (Hproc_handle proc_id,FGInstance *fginst,
                              double maxDelay,Himage **image,
                              INT **num_channel,INT *num_image,
                              Hrlregion ***region,INT *num_region,
                              Hcont ***cont,INT *num_cont,
                              Hcpar **data,INT *num_data);
    Herror (*Info) (Hproc_handle proc_id,INT queryType,
                    char **info,Hcpar **values,INT *numValues);
    Herror (*SetParam) (Hproc_handle proc_id, FGInstance *fginst,
                        char *param,Hcpar *value,INT num);
    Herror (*GetParam) (Hproc_handle proc_id, FGInstance *fginst,
                        char *param,Hcpar *value,INT *num);
    Herror (*SetLut) (Hproc_handle proc_id,FGInstance *fginst,
                      INT4_8 *red,INT4_8 *green,INT4_8 *blue,INT num);
    Herror (*GetLut) (Hproc_handle proc_id,FGInstance *fginst,
                      INT4_8 *red,INT4_8 *green,INT4_8 *blue,INT *num);
...

Figure 2.3: The data structure FGClass defined in include\hlib\CIOFrameGrab.h (to be continued).

We will not discuss each member of the structure. However, it may be useful to look at the different types of fields:

- **Internal / Management**
  name, lib_handle, available (for backward compatibility only), instances_num etc.: These are the internal and some additional entries used for managing instances, see chapter 3 on page 29.

- **Interface-specific functions**
  Open etc.: Pointers to the interface routines you provide, see chapter 3 on page 29 and figure 2.5.

- **Default values**
  image_width, image_height, etc.: The default values for the standard parameters used in
typedef struct _FGClass {
    ... /* -------------------------- default values -------------------------- */
    INT horizontal_resolution, /* desired resolution delivered */
        vertical_resolution; /* by the board */
    INT image_width, image_height; /* desired image size */
    INT start_row, start_col; /* upper left corner */
    INT field; /* even- or odd-field, full image mode */
    INT bits_per_channel; /* color depth per pixel & channel */
    char color_space[MAX_STRING]; /* "gray", "rgb", "yuv", ... */
    float gain; /* for backwards compatibility only! */
    HBOOL external_trigger; /* trigger mode */
    char camera_type[MAX_STRING]; /* used camera type (fg-specific!) */
    char device[MAX_STRING]; /* device name */
    INT port; /* port to use */
    INT line_in; /* multiplexer input line */
    Hcpar *generic; /* generic parameter */
    INT num_generic;
    /* ------------------------- miscellaneous ---------------------------- */
    void *reserved[FG_NUM_RESERVED];
} FGClass;

Figure 2.4: The data structure FGClass defined in include\hlib\CIOFrameGrab.h (continued).

open_framegrabber. Whenever the user specifies “default” (or -1, respectively) in this operator, the HALCON library will pass the corresponding values inside FGClass to the interface routine FGOpen(), see section 3.3 on page 34.

Please see also figure 3.51 on page 66.

2.1.2 Structure FGInstance

There is a data structure called FGInstance (see figure 2.6) you will encounter very often when programming an interface since almost every routine you provide (see chapter 3 on page 29) expects a pointer to the image acquisition instance it should work with.

The structure FGInstance contains the actual parameters for a specific image acquisition instance. The corresponding default values for the underlying image acquisition device model are stored in FGClass. Moreover, additional information concerning asynchronous grabbing might be stored in this structure (async_grab etc.). Finally, if you want to insert raw data allocated with other than the HALCON memory management routines into HALCON images, you must specify halcon_malloc and clear_proc, see FGGrab() in section 3.6 on page 51.

Please note, that FGInstance also contains a generic pointer (gen_pointer), which is very useful for example to link the structure to a structure like TFGInstance (section 2.2.2 on page 26) holding additional information for an instance, see also FGOpenRequest() in section 3.2 on page 33.
2.2 Recommended Auxiliary Structures

The structures FGClass and FGInstance provide data relevant to the HALCON library (that is outside of your interface) common to all different image acquisition types. However, to handle a specific image acquisition device you will need additional data structures. We recommend to adapt the following two structures BoardInfo and TFGInstance for your needs.

2.2.1 Structure BoardInfo

We suggest to define a structure BoardInfo to hold all data relevant to a specific image acquisition device (that is the physical instance of a frame grabber or camera connected to your computer). Figure 2.7 shows a typical example.

DeviceId is an entry in this structure you will need for every image acquisition device we know (although the corresponding data type will vary). It is used to store a handle returned by the device API to access a board. If you decide to support multiple boards you might also want to hold a device name for each board. Moreover, it might be a good idea to share\(^2\) buffer memory among all frame grabber instances using the same board. The other entries in the example refer to the management of these buffers and the instances using the board. Please refer to chapter 3 on page 29 and the example template file hAcqTemplate.c for details.

For every image acquisition device type you have connected to your computer you should allocate one structure BoardInfo. In general there might be more than one image acquisition instance operating on such a physical device. Thus, we recommend to store instance-specific data in another structure called

\(^2\)Since most frame grabbers have only one A/D converter you have to synchronize the grabbing by different instances anyway.
typedef struct _FGInstance {
    struct _FGClass *fgclass; /* a pointer to the corresponding class */
    /* ---------------------- regular parameters -------------------------- */
    INT    horizontal_resolution, /* desired resolution delivered */
            vertical_resolution; /* by the board */
    INT    image_width, image_height; /* desired image size */
    INT    start_row, start_col; /* upper left corner */
    INT    field; /* even- or odd-field, full image mode */
    INT    bits_per_channel; /* color depth per pixel & channel */
    char   color_space[MAX_STRING]; /* "gray", "rgb", "yuv", ... */
    float  gain; /* for backwards compatibility only! */
    HBOOL  external_trigger; /* trigger mode */
    char   camera_type[MAX_STRING]; /* used camera type (fg-specific!) */
    char   device[MAX_STRING]; /* device name */
    INT    port; /* port to use */
    INT    line_in; /* multiplexer input line */
    Hcpar  *generic; /* generic parameter */
    INT    num_generic;
    /* --------------------- miscellaneous parameters --------------------- */
    INT4_8  mode; /* current operating mode */
    INT    width_max, height_max; /* maximum image size */
    INT    num_channels; /* number of image channels */
    HBOOL  async_grab; /* TRUE <=> async grabbing engaged */
    Himage *image; /* image objects to grab into (aux.) */
    void   *gen_pointer; /* generic pointer (auxiliary) */
    /* ------------------- external memory allocation --------------------- */
    HBOOL  halcon_malloc; /* TRUE <=> standard memory allocation */
    /* by HNewImage */
    DBFreeImageProc clear_proc; /* pointer to specific clear function */
    /* (if halcon_malloc==FALSE) */
} FGInstance;

Figure 2.6: The data structure FGInstance defined in include\hlib\CIOFrameGrab.h.

TFGInstance (see below). In some cases, if only a digital camera is used (without a frame grabber), there is no need to use the BoardInfo structure.

2.2.2 Structure TFGInstance

We recommend to use a data structure called TFGInstance to extend the data provided by FGInstance for each image acquisition instance. Figure 2.8 shows a typical example.

Obviously, it is very convenient to hold references to both the assigned physical device (board) and the corresponding instance data from the HALCON library (instance). Moreover, to handle asynchronous grabbing, entries like busy (indicating that a grab is still pending), maxAge (holding the current setting for the maximum tolerated “age” of an asynchronously grabbed image), or grabStarted (containing a timestamp) might be a good idea. If you would like to support volatile grabbing, i.e., to let the image
2.2 Recommended Auxiliary Structures

typedef struct {
    char DeviceName[255]; /* assign a name to each board */
    INT4_8 DeviceId; /* some sort of handle (specific to the frame grabber API) */
    HBYTE *BoardFrameBuffer[MAX_BUFFERS]; /* buffers assigned to the board, that is to ALL TFGInstances */
    INT currBuffer; /* index of the active buffer */
    INT sizeBuffer; /* size of each buffer */
    INT refBuffer; /* number of references to the buffers (from TFGInstance(s)) */
    INT refInst; /* number of instances assigned to this board */
} BoardInfo;

Figure 2.7: An example for the recommended auxiliary data structure BoardInfo.

typedef struct {
    BoardInfo *board; /* the 'physical' board this instance is attached to */
    HBOOL busy; /* useful, if you plan to support asynchronous grabbing (is the last grab still running?) */
    INT instance; /* a useful backreference to the general Halcon instance information: The instance index */
    INT currBuffer; /* you probably use more than one buffer: Index of the active buffer */
    #ifdef WIN32
    struct _timeb grabStarted; /* just to check the timeout: the timestamp when the last grab was started */
    #else
    struct timeval grabStarted; /* the same for UNIX systems ... */
    struct timezone tzp;
    #endif
    HBYTE *InstFrameBuffer[MAX_BUFFERS]; /* buffers assigned to this instance */
    HBOOL allocBuffer; /* TRUE <=> buffers are allocated per instance, not only references to the buffers in "board" */
    HBOOL volatileMode; /* TRUE <=> pass buffer memory directly to a HALCON image (possibly "overwriting" older images) */
} TFGInstance;

Figure 2.8: An example for the recommended auxiliary data structure TFGInstance.
acquisition buffers (containing the image data) insert\textsuperscript{3} into HALCON images, a flag like volatileMode is useful. In this case, but also if you encounter image acquisition instances using different image sizes, buffer memory cannot be shared among all instances assigned to a board. Allocate buffers for each instance instead (using entries like InstFrameBuffer and allocBuffer). Please refer to chapter 3 on page 29 and hAcqTemplate.c for more details on how to use TFGInstance.

It is hard to provide a framework for all possible image acquisition devices in this manual. If you would like to develop an optimal interface you will always have to adapt the example code to the specific API of the image acquisition device and to its specific hardware features. In general, this will also mean to include additional parameters (in TFGInstance and BoardInfo) to allow the fine-tuning of the hardware.

\textsuperscript{3}This prevents an additional copy of data. However, as a side-effect old images are overwritten.
Chapter 3

Interface Routines

This chapter explains all the routines you have to implement inside your image acquisition interface in order to support the corresponding HALCON image acquisition operators, see section 1.6 on page 12. The example code of the next sections can also be found in %HALCONEXAMPLES%\ia_integration\source\hAcqTemplate.c.

3.1 FGInit()

FGInit() as defined in figure 3.1 is called by the HALCON operators open_framegrabber or info_framegrabber when you access a specific image acquisition device for the very first time.

```c
#include "Halcon.h"
#include "hlib/CIOFrameGrab.h"

Herror FGInit(Hproc_handle proc_id, FGClass *fg)
{
    /* initialize the data structure FGClass and the acquisition interface */
    return(H_MSG_OK);
}
```

Figure 3.1: The prototype of FGInit().

In order to do so, the routine FGInit() must be accessible from outside, that is the HALCON library must be able to find the symbol and call the routine inside the DLL or shared library. In Windows you have to export the symbol explicitly:

```c
extern HUserExport Herror FGInit(Hproc_handle proc_id, FGClass *fg);
```

\(^1\) Or if you access an image acquisition device again after closing all instances.
By the way, FGInit() is the only restriction concerning symbol names: The names of all other procedures, variables, and macros you use inside your interface is up to you, but never change the name of this routine. Otherwise, HALCON will fail to access your image acquisition interface.

The routine FGInit() must perform three basic tasks:

- Initialize the data structure FGClass (see section 2.1.1 on page 22). Especially the function pointers to all the other routines within the interface must be inserted.
- Provide default values for the standard parameters used in the HALCON operator open_framegrabber.
- Initialize the data structures inside the image acquisition interface and link them to the corresponding data structures in the HALCON library, if necessary.

An example for the first two tasks is given in figures 3.2 and 3.3; an example for the latter is shown in figure 3.5 on page 32.

```c
Herror FGInit(Hproc_handle proc_id, FGClass *fg)
{
    ...
    /* -------------------------------- management ----------------------------- */
    /* For backward compatibility: */
    fg->available = TRUE;
    /* Do not change the next line or modify fg->instances_num anywhere else */
    /* in the interface (otherwise HALCON will fail to unload the interface */
    /* DLL properly!) */
    fg->instances_num = 0;
    /* Tell HALCON how many instances you are willing to support */
    fg->instances_max = FG_MAX_INST;
    /* -------------------------------- interface-specific functions ---------------- */
    fg->OpenRequest = FGOpenRequest;
    fg->Open = FGOpen;
    fg->Close = FGClose;
    fg->Info = FGInfo;
    fg->Grab = FGGrab;
    fg->GrabStartAsync = FGGrabStartAsync;
    fg->GrabAsync = FGGrabAsync;
    fg->GrabData = FGGrabData;
    fg->GrabDataAsync = FGGrabDataAsync;
    fg->SetParam = FGSetParam;
    fg->GetParam = FGGetParam;
    fg->SetLut = FGSetLut;
    fg->GetLut = FGGetLut;
    fg->SetCallback = FGSetCallback
    fg->GetCallback = FGGetCallback
    ...
```

Figure 3.2: Example code for FGInit(): Initialize FGClass (continued in figure 3.3).

Note that in figure 3.2 the function pointers inside FGClass are assigned to the specific routines you provide in the image acquisition interface. Thus, you can choose arbitrary names for the latter. However,


```c
... /* --------------------------- default values -------------------------- */
/* The following defaults will be delivered to FGOpen(), if "default" */
/* or -1 is specified in open_framegrabber() */
fg->horizontal_resolution = 1;
fg->vertical_resolution = 1;
fg->image_width = fg->image_height = 0;
fg->start_row = fg->start_col = 0;
fg->field = FG_FULL_FRAME;
fg->bits_per_channel = 8;
strcpy(fg->color_space,"gray");
fg->gain = 1.0f; /* only for backwards compatibility */
fg->external_trigger = FALSE;
strcpy(fg->camera_type,"auto");
strcpy(fg->device,"0");
fg->port = 1;
fg->line_in = 1;
...
return(H_MSG_OK);
}
```

Figure 3.3: Example code for FGInit(): Initialize FGClass (continued from figure 3.2).

we recommend to stick to the names used in this manual to ease the understanding of different interface codes. Some of these function pointers are optional, see figure 3.4. If you do not assign anything (or assign a NULL pointer) to these function pointers, HALCON will return an error while executing the corresponding operators, see section 1.6 on page 12.

<table>
<thead>
<tr>
<th>Interface routine</th>
<th>Function pointer</th>
<th>Error code</th>
</tr>
</thead>
<tbody>
<tr>
<td>FGGrabAsync()</td>
<td>fg-&gt;GrabAsync</td>
<td>H_ERR_FGASYNC</td>
</tr>
<tr>
<td>FGGrabStartAsync()</td>
<td>fg-&gt;GrabStartAsync</td>
<td>H_ERR_FGASYNC</td>
</tr>
<tr>
<td>FGGrabData()</td>
<td>fg-&gt;GrabData</td>
<td>H_ERR_FGFNS</td>
</tr>
<tr>
<td>FGGrabDataAsync()</td>
<td>fg-&gt;GrabDataAsync</td>
<td>H_ERR_FGFNS</td>
</tr>
<tr>
<td>FGSetParam()</td>
<td>fg-&gt;SetParam</td>
<td>H_ERR_FGPARAM</td>
</tr>
<tr>
<td>FGGetParam()</td>
<td>fg-&gt;GetParam</td>
<td>H_ERR_FGPARAM</td>
</tr>
<tr>
<td>FGSetLut()</td>
<td>fg-&gt;SetLut</td>
<td>H_ERR_FGFNS</td>
</tr>
<tr>
<td>FGGetLut()</td>
<td>fg-&gt;GetLut</td>
<td>H_ERR_FGFNS</td>
</tr>
<tr>
<td>FGSetCallback()</td>
<td>fg-&gt;SetCallback</td>
<td>H_ERR_FGFNS</td>
</tr>
<tr>
<td>FGGetCallback()</td>
<td>fg-&gt;GetCallback</td>
<td>H_ERR_FGFNS</td>
</tr>
</tbody>
</table>

Figure 3.4: Optional interface routines and the corresponding error codes returned by the HALCON library if the routines are missing.
static TFGInstance FGInst[FG_MAX_INST]; /* all possible instances */
static INT numInstance = 0;        /* # current instances */
static FGClass *fgClass;          /* pointer to the class struct */

Herror FGInit(Hproc_handle proc_id, FGClass *fg)
{
    ...  
    INT i;

    /* Initialize the instance data structure inside of this interface */
    for (i=0; i < FG_MAX_INST; i++)
    {
        memset(&(FGInst[i]), 0, sizeof(TFGInstance));
        FGInst[i].instance = i;
    }

    numInstance = 0;
    ...

    /* ------------------ store the class information ---------------------- */  
    fgClass = fg;
    ...
    return(H_MSG_OK);
}

Figure 3.5: Example code for FGInit(): Initialize FGClass.

In figure 3.5 we have assumed that you followed our suggestion to provide a data structure TFGInstance to hold additional device-specific information about an instance, see also section 2.2.2 on page 26. Note, that in this example we have also assumed that you are willing to support multiple instances. If you would like to start with a simple image acquisition device integration supporting only one instance, you can simplify FGInit(): In that case it might be enough to hold all device-specific information in one global data structure (or a bunch of global variables) inside the interface. Thus, you could skip the suggested array

TFGInstance FGInst[FG_MAX_INST];

and all the code in figure 3.5.
3.2 FGOpenRequest()

The routine `FGOpenRequest()` as defined in figure 3.6 is called by the HALCON operator `open_framegrabber` prior to calling `FGOpen()`, see section 3.3. Its only task is:

- Return the next available instance (i.e., one of the instance pointers inside the `FGClass` data structure, see section 2.1.1 on page 22).

```c
#include "Halcon.h"
#include "hlib/CIOFrameGrab.h"

static FGInstance **FGOpenRequest(Hproc_handle proc_id, FGInstance *fginst)
{
    /* return an available instance prior to FGInit() */
    return (&(fgClass->instance[0]));
}
```

Figure 3.6: The prototype for `FGOpenRequest()`.

If the instance you return is already assigned to an acquisition handle, this “old instance” is automatically closed using `FGClose()`. If you return a `NULL` pointer, HALCON will return the error code `H_ERR_FGDV` (“Image acquisition: device busy”) as result\(^2\) of `open_framegrabber`.

If you support multiple instances you can use the example code listed in figure 3.7, otherwise use the code in figure 3.8.

```c
static FGInstance **FGOpenRequest(Hproc_handle proc_id, FGInstance *fginst)
{
    INT i;

    if (numInstance >= FG_MAX_INST)
        return(NULL); /* too many instances ... */
    else
    {
        /* search for next unused instance */
        for (i=0; i < FG_MAX_INST; i++)
        {
            if (!FGInst[i].board)
                break;
        }
        fginst->gen_pointer = (void*)&FGInst[i];
        return (&(fgClass->instance[i]));
    }
}
```

Figure 3.7: Example code for `FGOpenRequest()`: Multiple instances.

\(^2\)There is no instance available – thus, the device is “busy”.

static FGInstance **FGOpenRequest(Hproc_handle proc_id, FGInstance *fginst)
{
    fginst->gen_pointer = (void*)&FGInst[0];
    return (&(fgClass->instance[0]));
}

Figure 3.8: Example code for FGOpenRequest(): Only one instance.

Note, that we use the global pointer fgClass that has been set to the FGClass structure assigned to this interface in FGinit() as suggested in figure 3.5 on page 32. Alternatively, you could also use fginst->fgclass, which is also a pointer to the same structure.

Note further, that in figure 3.7 the generic pointer gen_pointer inside the data structure FGInstance (see section 2.1.2 on page 24) is used to establish a link between the exterior structure fginst provided by the HALCON library and the TFGInstance structure FGInst[i] inside the interface.

The example code in figure 3.8 will cause HALCON to automatically close the old instance whenever you request a new instance using open_framegrabber. This is very convenient for interactive programming with HDevelop, but obviously leads to a severe side-effect. Thus, you might also check whether there is an active instance (using a boolean flag) and return NULL in case the frame grabber is busy.

You should not bother too much about this routine. In most cases you can use one of the two examples provided without any changes.

### 3.3 FGOpen()

The routine FGOpen() as defined in figure 3.9 is called by the HALCON operator open_framegrabber, see section 1.6 on page 12. It has to perform the following tasks:

- Check all parameters specified in the FGInstance structure fginst.
- Check the availability of the specified image acquisition device and initialize the device according to the parameter settings.
- Allocate buffers if necessary.

Please refer to hAcqTemplate.c for a detailed example of such a routine.

At the very beginning of FGOpen() we suggest to access the internal TFGInstance structure corresponding to the specified instance fginst and set some defaults as shown in figure 3.10. All the examples in this section are aiming at a complete image acquisition integration. If you are only interested in a basic integration, many of the things discussed here are unnecessary. Thus, the resulting code might be much shorter and easier to understand, but also much less general and flexible.

Note, that all the parameters you specify in the HALCON operator open_framegrabber are passed to FGOpen() in the FGInstance structure pointed to by fginst, see also section 2.1.2 on page 24. Whenever you specify “default” (for string parameters) or -1 in this operator, the corresponding default value you provided in FGInit() (see figure 3.3 on page 31) will be passed in fginst.
#include "Halcon.h"
#include "hlib/CIOFrameGrab.h"

static Herror FGOpen(Hproc_handle proc_id, FGInstance *fginst) {
    /* initialize the new image acquisition instance fginst */
    return(H_MSG_OK);
}

Figure 3.9: The prototype for FGOpen().

static Herror FGOpen(Hproc_handle proc_id, FGInstance *fginst) {
    TFGInstance *currInst = (TFGInstance *)fginst->gen_pointer;

    fginst->async_grab = FALSE;
    currInst->busy = FALSE;
    currInst->allocBuffer = FALSE;
    currInst->currBuffer = 0;
    currInst->volatileMode = FALSE;
    ...
    return(H_MSG_OK);
}

Figure 3.10: Example code for FGOpen(): Accessing the corresponding TFGInstance structure and setting
some defaults.

The parameter checking is rather straightforward: Test whether the specified values are reasonable for
your frame grabber board or not. Please note, that the HALCON library itself cannot do much of such
plausibility tests since the hardware capabilities of different image acquisition devices differ too much.
Whenever you detect an incorrect request, return one of the error codes listed in appendix E on page 85.
Note that returning from an arbitrary position inside your interface code might lead to memory leaks
or “blocked” image acquisition devices. Therefore, we recommend to do as much parameter checks as
possible before accessing the physical device or allocating memory. On the other hand, some of these
tests must be delayed until the device itself is initialized and an analysis of the video signal is possible for
example. In this case, be sure to deallocate all the memory and to unlock the image acquisition device
before returning an error code.

If you support multiple instances per physical image acquisition device (e.g., in case of more than one
camera connected to a single board) you should spend some time on the design of the “availability
checks”. A fairly simple solution is to enforce different ports (fginst->port) (the parameter line_in
to denote a multiplexed input line has historical reasons and is not recommended anymore), but identical
values for the rest of the parameters (image size, color depth, etc.). Otherwise, you have to reset all these
parameters in the image acquisition device whenever you grab for one instance or the other. This might
be both a lot of work to program and time consuming during the application. The latter might be partly
compensated by storing the current parameter settings of the device within the BoardInfo structure

3 In any case you have to set the corresponding input line prior to grabbing.
static Herror FGOpen(Hproc_handle proc_id, FGInstance *fginst)
{
    ...
    if (!currInst->board)
    {
        HckP(HAlloc(proc_id,(size_t)sizeof(BoardInfo),&currInst->board));

        /* init the struct currInst->board, e.g., */
        memset(currInst->board, 0, sizeof(BoardInfo));
        strcpy(currInst->board->DeviceName, fginst->device);

        /* open the board for the 1st time ... */
        currInst->board->DeviceId = ...
    }
    ...
    return(H_MSG_OK);
}

Figure 3.11: Example code for FGOpen(): Access a physical image acquisition device for the very first time.

(see section 2.2.1 on page 25) and comparing them to corresponding settings in the FGInstance (see section 2.1.2 on page 24) or TFGInstance (see section 2.2.2 on page 26) structures. Obviously, you only have to reset the parameters that differ.

#include "Halcon.h"

Herror HAlloc(Hproc_handle proc_id, size_t size, void **pointer)
{
    /* allocate memory on the heap */
    return(H_MSG_OK);
}

Figure 3.12: The prototype for the HALCON extension package interface routine HAlloc().

If you encounter the first instance to be assigned to a physical image acquisition device you have to access the device using the device API and you might have to allocate the corresponding BoardInfo structure if you follow our suggestions, see also section 2.2.1 on page 25. Figure 3.11 shows some example code dealing with this. The routine HAlloc() as defined in figure 3.12 is provided by the HALCON extension package interface, see the Extension Package Programmer’s Manual, section 3.2.2 on page 52, for details. It is used to allocate memory on the heap.

Once you have selected and initialized the image acquisition device you should analyze the video signal. Many frame grabber APIs provide routines to do this automatically. If such a functionality is missing or the analysis is very time consuming, you might want to specify the video norm in open_framegrabber. Use the camera type parameter fginst->camera_type for this purpose, see figure 3.13.

Some image acquisition devices use specific camera configuration files to store all relevant data about the desired camera configuration. Typically, the configuration files are edited with a special camera editor
3.3 FGOpen()

```c
#define STR_CASE_CMP(S1, S2) stricmp(S1, S2)
#endif

static Herror FGOpen(Hproc_handle proc_id, FGInstance *fginst)
{
    INT norm;
    ...
    if (!STR_CASE_CMP(fginst->camera_type, "auto"))
    { /* use special routines provided by your image acquisition device */
        /* to analyze the video signal ... */
    }
    else if (!STR_CASE_CMP(fginst->camera_type, "ntsc"))
    {
        norm = FG_NTSC;
        fginst->width_max = 640;
        fginst->height_max = 480;
    }
    else if (!STR_CASE_CMP(fginst->camera_type, "pal"))
    {
        norm = FG_PAL;
        fginst->width_max = 768;
        fginst->height_max = 576;
    }
    else
        /* well, whatever! */
    ...
    return(H_MSG_OK);
}
```

Figure 3.13: Example code for FGOpen(): Determine the video norm and the maximum image size to be delivered by the image acquisition device.

program. In these cases it's sufficient to specify the complete file name (including the path) of the desired configuration file in the camera type parameter fginst->camera_type and read the configuration from the file.

The next step is to determine the scaling of the image and thus the desired image size to be delivered by the image acquisition device. Note, that we only support subsampling by a factor of 2 or 4 in the example code in figure 3.14. After this, analyze the specified part of the image to be delivered as HALCON image by the grabbing routines, see figure 3.15. Once you have determined both the image size and the part of the image to be grabbed you have to set the video scaler of the image acquisition device according to this values. Obviously, how to do this depends on the device API. Thus, we cannot provide source code.

There is one very important topic left to be discussed for the implementation of FGOpen(): the allocation of buffer memory. Now you have two possibilities to do this: Use a ring buffer with a fixed number of
static Herror FGOpen(Hproc_handle proc_id, FGInstance *fginst)
{
    ... 
    INT widthScale = fginst->horizontal_resolution;
    INT heightScale = fginst->vertical_resolution;

    if (widthScale == fginst->width_max)
        widthScale = 1;
    if (heightScale == fginst->height_max)
        heightScale = 1;
    if (widthScale == fginst->width_max/2)
        widthScale = 2;
    if (heightScale == fginst->height_max/2)
        heightScale = 2;
    if (widthScale == fginst->width_max/4)
        widthScale = 4;
    if (heightScale == fginst->height_max/4)
        heightScale = 4;

    if (! (widthScale == 1 || widthScale == 2 || widthScale == 4))
        return(H_ERR_FGWR); /* wrong resolution */
    if (! (heightScale == 1 || heightScale == 2 || heightScale == 4))
        return(H_ERR_FGWR); /* wrong resolution */

    fginst->horizontal_resolution = fginst->width_max / widthScale;
    fginst->vertical_resolution = fginst->height_max / heightScale;
    ... 
    return(H_MSG_OK);
}

Figure 3.14: Example code for FGOpen(): Determine the desired scaling and thus the image size to be delivered by the image acquisition device.

buffers (MAX_BUFFERS)\(^4\) or let the user decide how many buffers he wants to use (generic parameter num_buffers). Obviously you have to allocate such buffers once per instance if you would like to support different image sizes (or color depths) or volatile grabbing, see section 1.4 on page 11 and section 2.2.2 on page 26. On the other hand, if several instances are assigned to the same physical image acquisition device and they have to share a single A/D converter, you have to synchronize the grabbing by these instances. Thus, they could actually share the buffers as well to reduce the demand for (typically non-paged) buffer memory. Therefore, we suggest to allocate shared buffers per board if possible and per instance only if necessary. Variant to this suggestion, you can adapt the condition of the first if command in figure 3.17 to your needs. The allocation itself usually is done using device-specific API calls. In the following examples we will use the HALCON extension package interface routine HAlloc() as a template. Please note, that in most cases you will have to replace these HAlloc calls by other routines.

\(^4\)Please define MAX_BUFFERS according to your needs inside the interface. In most cases two buffers will be sufficient.
static Herror FGOpen(Hproc_handle proc_id, FGInstance *fginst)
{
    ...
    if (fginst->image_width == 0)
        fginst->image_width = fginst->horizontal_resolution - 2*fginst->start_col;
    if (fginst->image_height == 0)
        fginst->image_height = fginst->vertical_resolution - 2*fginst->start_row;
    if((fginst->start_col+fginst->image_width > fginst->horizontal_resolution) ||
        (fginst->start_row+fginst->image_height > fginst->vertical_resolution))
        /* wrong part */
        return(CleanupFGOpen(proc_id,currInst,newBoardalloc,H_ERR_FGWP));
    ...
    return(H_MSG_OK);
}

Figure 3.15: Example code for FGOpen(): Determine the desired image part to be delivered as HALCON image.

static Herror FGOpen(Hproc_handle proc_id, FGInstance *fginst)
{
    ...
    /* increase the number of instances assigned to this board ... */
    currInst->board->refInst++;
    /* ... and the overall number of instances */
    numInstance++;
    return(H_MSG_OK);
}

Figure 3.16: Example code for FGOpen(): Final settings.

Figures 3.17 and 3.18 demonstrate this strategy. Note, that the data structures BoardInfo and TFGInstance as introduced in section 2.2.1 on page 25 and section 2.2.2 on page 26 have been designed for that purpose. They include all the information needed to handle the buffers (board->sizeBuffer, board->refBuffer, board->BoardFrameBuffer, and currInst->InstFrameBuffer). The macro HckP is used to return from a function in case of an error, see the Extension Package Programmer’s Manual, section 5.6.1 on page 101, for details. Note, that returning from FGOpen() from a point like this can lead to a memory leak as discussed before. In the “real” interface code you should keep this in mind. In case you let the user decide the number of used buffers the needed code is a little bit different. Since there is no extra parameter in FGOpen to do this, you have to use the generic parameter\(^5\), see figures 3.19. It is also very important to change the structure of InstFrameBuffer in TFGInstance from

\[\text{HBYTE } \ast \text{InstFrameBuffer}[\text{MAX_BUFFER}]\]


to

\[\text{HBYTE } \ast \ast \text{InstFrameBuffer}.\]

\(^5\)With the Generic parameter you can apply settings that have to be done before opening the image acquisition device.
static Herror FGOpen(Hproc_handle proc_id, FGInstance *fginst)
{
    INT4_8 sizeBuffer;
    ...
    sizeBuffer = fginst->image_width*fginst->image_height *
        ((fginst->bits_per_channel+7) / 8)*fginst->num_channels;
    if (1)
    {
        /* share the buffers with other instances */
        BoardInfo *board = currInst->board;
        currInst->allocBuffer = FALSE;

        if (!board->sizeBuffer)
        {
            /* that's the very first time such buffers (per board) are requested! */
            for (i=0; i < MAX_BUFFERS; i++)
            {
                err = HAlloc (proc_id,(size_t)sizeBuffer,&board->BoardFrameBuffer[i]);
                if (err != H_MSG_OK)
                    return(CleanupFGOpen(proc_id,currInst,newBoardalloc,err));
            }
            board->sizeBuffer = sizeBuffer;
        }
    }
    ...  

Figure 3.17: Example code for FGOpen(): Allocate buffers (continued in figure 3.18).

Now you know how many buffers will be needed and they can be allocated, see figures 3.20. One more note concerning volatile grabbing: If you decide to create new HALCON images not based on the standard HALCON extension package interface routine HNewImage(), but “insert” buffers directly, you should not forget to let the system know this! Set fginst->halcon_malloc to FALSE and fginst->clear_proc to NULL in this case. See figure 3.16 and the discussion of FGGrab() in section 3.6 on page 51 for details. Finally, after successfully initializing the new instance, you should increase both  the overall counter for instances numInstance and the number of instances for the corresponding physical image acquisition device currInst->board->refInst. Again, this is only of importance if you would like to support multiple instances.

6Note, that these counters have been suggested in figure 3.5 on page 32 and section 2.2.1 on page 25. It is up to you whether you follow this recommendation or come up with another solution.
else if (board->sizeBuffer != sizeBuffer)
{
    /* bad luck! The size of the shared buffers does not match */
    /* the required size! */
    currInst->allocBuffer = TRUE;
}
if (!currInst->allocBuffer)
{
    /* insert references: */
    for (i=0; i < MAX_BUFFERS; i++)
    {
        currInst->InstFrameBuffer[i] = board->BoardFrameBuffer[i];
    }
    board->refBuffer++; /* one more instance that uses the board buffers*/
}
else
    currInst->allocBuffer = TRUE;

if (currInst->allocBuffer)
{
    /* do not use shared buffers, but allocate the buffers for this new */
    /* instance */
    for (i=0; i < MAX_BUFFERS; i++)
    {
        HckP(HAlloc(proc_id,(size_t)sizeBuffer,&currInst->InstFrameBuffer[i]));
    }
}
...
return(H_MSG_OK);

Figure 3.18: Example code for FGOpen(): Allocate buffers (continued from figure 3.17).
currInst->num_buffers = MAX_BUFFERS;
...
if (!(fginst->numGeneric == 1 && fginst->generic->par.l == -1) &&
    !(fginst->numGeneric == 1 && fginst->generic->par.f == -1.0))
{
    for (i=0; i<fginst->numGeneric; i++)
    {
        if (fginst->generic[i].type == STRING_PAR)
        {
            char *pos = NULL;
            int len = 0;
            /* ignore empty string */
            len = (INT)strlen(fginst->generic->par.s);
            if (!len)
                continue;
            else
                HCKP(HAllocLocal(proc_id, (1+len) * sizeof(char), &pos));
        }
        else
        {
            HCKP(HAllocLocal(proc_id, sizeof(fginst->generic->par), &pos));
            if (strstr(fginst->generic[i].par.s, FG_PARAM_NUM_BUFFERS))
            {
                if ((pos = strstr(fginst->generic[i].par.s, "=")) != NULL)
                {
                    currInst->num_buffers = atoi(pos+1);
                    if (currInst->num_buffers < 1 || currInst->num_buffers > MAX_BUFFERS)
                    {
                        MY_PRINT_ERROR_MESSAGE("Invalid number of buffers!");
                        return H_ERR_FGNI;
                    }
                }
                else
                {
                    MY_PRINT_ERROR_MESSAGE("No argument for parameter 'num_buffers' found!");
                    return(CleanupFGOpen(proc_id, currInst, newBoardalloc, H_ERR_FGPARV));
                }
            }
            else
            {
                MY_PRINT_ERROR_MESSAGE("Unknown generic parameter!");
                return(CleanupFGOpen(proc_id, currInst, newBoardalloc, H_ERR_FGPARAM));
            }
        }
    }
    else
    {
        MY_PRINT_ERROR_MESSAGE("Generic parameter accepts only string values!");
        return(CleanupFGOpen(proc_id, currInst, newBoardalloc, H_ERR_FGPART));
    }
} /* end for */

Figure 3.19: Example code for FGOpen(): Use the generic parameter to allocate buffers.
/* do not use shared buffers, but allocate the buffers for this new */
/* instance */
err = HAlloc (proc_id, sizeof(HBYTE*)*currInst->num_buffers,
              &currInst->InstFrameBuffer);
if (err != H_MSG_OK)
    return(CleanupFGOpen(proc_id,currInst,newBoardalloc,err));

for (int i=0; i<currInst->num_buffers; i++)
{
    HCKP(HAlloc(proc_id, (size_t)sizeBuffer,
              &currInst->InstFrameBuffer[i]));
}

Figure 3.20: Example code for FGOpen(): Allocate buffers depending on user input.
3.4 FGClose()

The routine FGClose() as defined in figure 3.21 is called by the HALCON operator close_framegrabber, see section 1.6 on page 12. It has to perform the following tasks:

- Terminate pending asynchronous grabs.
- Deallocate the buffers.
- “Close” or “unlock” the physical image acquisition device if the instance to be closed is the last one assigned to this board.
- Mark the instance as “free”.

```c
#include "Halcon.h"
#include "hlib/CIoFrameGrab.h"

static Herror FGClose(Hproc_handle proc_id, FGInstance *fginst)
{
    /* close the specified image acquisition instance fginst */
    return(H_MSG_OK);
}
```

Figure 3.21: The prototype for FGClose().

A pending asynchronous grab is indicated by currInst->busy == TRUE, see also FGGrabAsync() in section 3.8 on page 59. In this case you should use the appropriate device API call to terminate the grab. Otherwise, you may encounter severe troubles if you deallocate the buffers the image acquisition device grabs into. The latter has to be done in accordance with the strategies for buffer allocation in FGOpen, see section 3.3 on page 34. Figures 3.22 and 3.23 may serve as a template for this. Note, that we use the HALCON extension package interface routine HFree() to deallocate memory delivered by HAlloc(). However, in most cases you will have to replace HAlloc() and thus HFree() by specific routines of your device API.

If the image acquisition instance to be closed is the last one assigned to a specific physical device, you have to “close” or “unlock” the device (using the appropriate API calls) and to deallocate the BoardInfo structure (if any), see figures 3.24 and 3.25. Note, that you might have to consider the special case that after closing the specified instance there is only one more instance left assigned to the board. This case is special in that sense that other routines like FGGrab() will assume that all parameters of this single remaining instance have already been set on the device (during FGOpen()). Thus, they might skip resetting these parameters again, which could lead to unexpected grabbing results. There are two obvious solutions to this problem: One is to restore the parameters of the last remaining instance in FGClose(), the other is to check for discrepancies between board settings and instance settings prior to each grab in any case.

Finally, you should mark the internal instance currInst as “free” again (currInst->board = NULL;) and decrease the number of active instances (numInstance--).
3.5 FGInfo()

The routine FGInfo() as defined in figure 3.26 is called by the HALCON operator info_framegrabber, see section 1.6 on page 12. It has to perform the following task:

- Return device-specific information depending on the specified query.

Note that the return values of these queries are essential if you want to use the new image acquisition interface also with ActivVisionTools or the Image Acquisition Assistant. The reason for this is that the auto-detect mechanism of ActivVisionTools and the Image Acquisition Assistant makes use of these queries to determine the list of all available image acquisition devices and also to provide specific value lists for the graphical user interface.

Currently, the queries listed in figure 3.27 should be supported. Optionally, you can also implement the query types FG_QUERY_IMAGE_HEIGHT, FG_QUERY_IMAGE_WIDTH, FG_QUERY_START_ROW, and FG_QUERY_START_COLUMN. A reasonable skeleton for this routine might look like the example in figure 3.28. Please see hAcqTemplate.c for a detailed example.

FGInfo() has two output parameters: A string containing a textual description of the desired information, and optional a list of parameter values. The latter can, for example, hold the values "auto", "pal", and "ntsc" as possible values of the camera type parameter in open_framegrabber, if you decide to use this parameter with this specific semantics, see figure 3.29 on page 50 and figure 3.13 on page 37. Hcpar is a HALCON data structure for storing control parameters (integer, strings, or floating point numbers), see the Extension Package Programmer's Manual, section 4.4 on page 65, for details.

In our example values is used to return three strings. Thus, the type tag type in the Hcpar structure is set to STRING_PAR. The corresponding settings for integers and floating pointer numbers is LONG_PAR and FLOAT_PAR, respectively. The parameter value should be written to par.s (strings), par.l (integers of type long), or par.f (floating point numbers of type double). Note, that neither the string info

```c
static Herror FGClose(Hproc_handle proc_id, FGInstance *fginst)
{
    TFGInstance *currInst = (TFGInstance *)fginst->gen_pointer;
    ...
    if (currInst->allocBuffer)
    {
        /* buffers have been allocated for this instance exclusively -- get rid */
        /* of them! */
        for (i=0; i < MAX_BUFFERS; i++)
        {
            if (currInst->InstFrameBuffer[i])
            {
                HCKP( HFree(proc_id,currInst->InstFrameBuffer[i]));
                currInst->InstFrameBuffer[i] = NULL;
            }
        }
    }
    ...
}
```

Figure 3.22: Example code for FGClose(): Deallocate buffers (continued in figure 3.23).
else
{
    BoardInfo *board = currInst->board;

    /* the instance shared the board buffers with other instances */
    if (board->refBuffer == 1)
    {
        /* This is the last instance which uses the board frame buffer, */
        /* therefore delete the buffer now. */
        for (i=0; i < MAX_BUFFERS; i++)
        {
            if (board->BoardFrameBuffer[i])
            {
                HCheck(HFree(proc_id,board->BoardFrameBuffer[i]));
                board->BoardFrameBuffer[i] = NULL;
            }
        }
        board->sizeBuffer = 0;
    }
    /* otherwise: Do not touch the buffers -- they are still in use! */
    board->refBuffer--;
}
... 
return(H_MSG_OK);

Figure 3.23: Example code for FGClose(): Deallocate buffers (continued from figure 3.22).

static Herror FGClose(Hproc_handle proc_id, FGInstance *fginst)
{
... 
    if (currInst->board->refInst <= 1)
    {
        /* "close" the board (using the appropriate API call) ... */
        /* ... and deallocate the BoardInfo you have allocated in FGOpen() */
        HCheck(HFree(proc_id,currInst->board));
    }
... 

Figure 3.24: Example code for FGClose(): Deallocate board (continued in figure 3.25).

nor the array of Hcpar structures have been allocated prior to calling FGInit(). Please use the HALCON extension package interface routine HAlloc() exclusively to allocate the latter, see figure 3.29. Otherwise, you will encounter system crashes when info_framegrabber deallocates the array using HFree().
... else {
    currInst->board->refInst--;
    if (currInst->board->refInst == 1) {
        /* This is sort of a special situation: See the text */
        ...
    }
    currInst->board = NULL;
    numInstance--;
    ...
    return(H_MSG_OK);
}

Figure 3.25: Example code for FGClose(): Deallocate board (continued from figure 3.24).

#include "Halcon.h"
#include "hlib/CIOFrameGrab.h"

static Herror FGInfo(Hproc_handle proc_id, INT queryType,
                     char **info, Hcpar **values, INT *numValues)
{
    /* return some device-specific information */
    return(H_MSG_OK);
}

Figure 3.26: The prototype for FGInfo().

Figure 3.30 shows another example. For the query FG_QUERY_PARAMETERS FGInfo() has to return a list of all names of additional device-specific parameters supported by your image acquisition device, see also FGSetParam() in section 3.12 on page 65 and FGGetParam() in section 3.13 on page 68. In our example we assume that there are only three additional parameters: “volatile” to control volatile grabbing, see also figure 3.52 on page 67 and figure 3.53 on page 68, “grab_timeout” to specify the desired timeout value for aborting a pending grab, and “revision” to inquire the current interface revision.
<table>
<thead>
<tr>
<th>queryType</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>FG_QUERY_GENERAL</td>
<td>General information</td>
</tr>
<tr>
<td>FG_QUERY_DEFAULTS</td>
<td>Default values for open_framegrabber</td>
</tr>
<tr>
<td>FG_QUERY_PARAMETERS</td>
<td>Additional parameters</td>
</tr>
<tr>
<td>FG_QUERY_INFO_BOARDS</td>
<td>Information about actually installed boards or cameras 7</td>
</tr>
<tr>
<td>FG_QUERY_REVISION</td>
<td>Current interface revision</td>
</tr>
<tr>
<td>FG_QUERY_BITS_PER_CHANNEL</td>
<td>Value list for BitsPerChannel</td>
</tr>
<tr>
<td>FG_QUERY_CAMERA_TYPE</td>
<td>Value list for CameraType</td>
</tr>
<tr>
<td>FG_QUERY_COLOR_SPACE</td>
<td>Value list for ColorSpace</td>
</tr>
<tr>
<td>FG_QUERY_DEVICE</td>
<td>Value list for Device</td>
</tr>
<tr>
<td>FG_QUERY_EXT_TRIGGER</td>
<td>Value list for ExternalTrigger</td>
</tr>
<tr>
<td>FG_QUERY_FIELD</td>
<td>Value list for Field</td>
</tr>
<tr>
<td>FG_QUERY_PORT</td>
<td>Value list for Port</td>
</tr>
<tr>
<td>FG_QUERY_PARAMETERS_RO</td>
<td>Value list for read-only parameters</td>
</tr>
<tr>
<td>FG_QUERY_PARAMETERS_WO</td>
<td>Value list for write-only parameters</td>
</tr>
<tr>
<td>FG_QUERY_HORIZONTAL_RESOLUTION</td>
<td>Value list for HorizontalResolution</td>
</tr>
<tr>
<td>FG_QUERY_VERTICAL_RESOLUTION</td>
<td>Value list for VerticalResolution</td>
</tr>
</tbody>
</table>

Figure 3.27: Queries that should be supported by FGInfo().
static Herror FGInfo (Hproc_handle proc_id, INT queryType,
    char **info, Hcpar **values, INT *numValues)
{
    switch(queryType)
    {
    case FG_QUERY_GENERAL:
        *info = "HALCON image acquisition interface template.";
        *values = NULL;
        *numValues = 0;
        break;
    case FG_QUERY_DEFAULTS:
        *info = "Default values (as used for open_framegrabber).";
        HckP( HFgGetDefaults(proc_id,fgClass,values,numValues));
        break;
    case FG_QUERY_PARAMETERS:
    ...
        break;
    case FG_QUERY_INFO_BOARDS:
        *info = "Info about installed boards.";
        ...
        break;
    case FG_QUERY_REVISION:
        *info = "Current interface revision.";
        HckP( HAlloc (proc_id,(size_t)(sizeof(Hcpar)),&val));
        val[0].par.s = INTERFACE_REVISION;
        val[0].type = STRING_PAR;
        *values = val;
        *numValues = 1;
        break;
    case FG_QUERY_BITS_PER_CHANNEL:
    ...
        break;
    ...
    case FG_QUERY_PORT:
    ...
        break;
    default:
        *info = "Unsupported query!";
        *values = NULL;
        *numValues = 0;
    }
    return(H_MSG_OK);
}

Figure 3.28: Example code for FGInfo(): Parsing the query.
static Herror FGInfo(Hproc_handle proc_id, INT queryType,
    char **info, Hcpar **values, INT *numValues)
{
    Hcpar *val;
    ...
    case FG_QUERY_CAMERA_TYPE:
        *info = "Value list for camera type parameter.";
        HckP( HAalloc (proc_id,(size_t)(3*sizeof(*val)),&val));
        val[0].par.s = "ntsc";
        val[1].par.s = "pal";
        val[2].par.s = "auto";
        val[0].type = val[1].type = val[2].type = STRING_PAR;
        *values = val;
        *numValues = 3;
        break;
    ...
    return(H_MSG_OK);
}

Figure 3.29: Example code for FGInfo(): The query FG_QUERY_CAMERA_TYPE.

#define FG_PARAM_VOLATILE "volatile"

static Herror FGInfo(Hproc_handle proc_id, INT queryType,
    char **info, Hcpar **values, INT *numValues)
{
    Hcpar *val;
    ...
    case FG_QUERY_PARAMETERS:
        *info = "Additional parameters for this image acquisition interface.";
        HckP( HAalloc (proc_id,(size_t)(3*sizeof(*val)),&val));
        val[0].par.s = FG_PARAM_VOLATILE;
        val[0].type = STRING_PAR;
        val[1].par.s = FG_PARAM_GRAB_TIMEOUT;
        val[1].type = STRING_PAR;
        val[2].par.s = FG_PARAM_REVISION
        val[2].type = STRING_PAR;
        *values = val;
        *numValues = 3;
        break;
    ...
    return(H_MSG_OK);
}

Figure 3.30: Example code for FGInfo(): The query FG_QUERY_PARAMETERS.
3.6 FGGrab()

The routine FGGrab() as defined in figure 3.31 is called by the HALCON operator grab_image\textsuperscript{8}, see section 1.6 on page 12. It has to perform the following tasks:

- Terminate pending asynchronous grabs.
- (Re-)set the parameters on the image acquisition device\textsuperscript{9}.
- Grab an image synchronously.
- Return a HALCON image containing the grabbed raw data.

```c
#include "Halcon.h"
#include "hlib/CIOFrameGrab.h"

static Herror FGGrab (Hproc_handle proc_id, FGInstance *fginst,
                     Himage *image, INT *num_image)
{
    /* grab an image synchronously */
    return(H_MSG_OK);
}
```

Figure 3.31: The prototype for FGGrab().

Note, that there might be an asynchronous grab pending when entering this routine (if the application called grab_image_async or grab_image_start prior to grab_image). Since you want to grab an image synchronously now you should terminate these grabs and launch a new grab. However, if you have a closer look at the semantics of the grab routines you will notice that it is quite easy to implement FGGrabAsync() based on FGGrab() if you include some additional branches in the code. Furthermore, FGGrabData() and FGGrabDataAsync() also share the basic task of grabbing an image to a buffer with the other two routines. Thus, we suggest to implement an auxiliary routine GrabImg() underlying all four of them, see figure 3.32.

```c
#include "Halcon.h"
#include "hlib/CIOFrameGrab.h"

static Herror GrabImg (Hproc_handle proc_id, FGInstance *fginst,
                      INT *readBuffer)
{
    /* grab an image to the current buffer */
    return(H_MSG_OK);
}
```

Figure 3.32: The prototype for the auxiliary routine GrabImg().

\textsuperscript{8}In the framework suggested in this manual it is also called by FGGrabAsync() see section 3.8 on page 59.

\textsuperscript{9}with the function SetInstParam. Mostly it is only needed for multiple devices sharing one frame grabber board, see also figure 3.39.
static Herror FGGrab(Hproc_handle proc_id, FGInstance *fginst,
                   Himage *image, INT *num_image)
{
    TFGInstance *currInst = (TFGInstance *)fginst->gen_pointer;
    INT readBuffer;

    HckP(GrabImg(proc_id, fginst, &readBuffer));

    if (currInst->volatileMode)
    {
        /* Insert the 8 bit image buffer directly into a HALCON object */
        HckP(HNewImagePtr(proc_id, &image[0], BYTE_IMAGE,
                           fginst->image_width, fginst->image_height,
                           (void*)currInst->InstFrameBuffer[readBuffer],
                           FALSE));
        *num_image = 1;
    }
    else
    {
        num_image = fginst->num_channels;
        for (i=0; i < *num_image; i++)
        {
            HckP(HNewImage(proc_id, &image[i], BYTE_IMAGE,
                           fginst->image_width, fginst->image_height));
        }
        if (fginst->num_channels == 1)
            memcpy((void *)image[0].pixel.b,
                    currInst->InstFrameBuffer[readBuffer],
                    fginst->image_width * fginst->image_height);
        else
            HckP((ExtractChannelsFromRGB(fginst,
                                          currInst->InstFrameBuffer[readBuffer],
                                          image[0].pixel.b,
                                          image[1].pixel.b,
                                          image[2].pixel.b));
    }
    fginst->async_grab=FALSE;
    return(H_MSG_OK);
}

Figure 3.33: Example code for FGGrab(): The basic structure.

Based on this auxiliary routine FGGrab() might look as shown in figure 3.33. Note, that we only support one channel grayscale and three channels color images of 8 bit scale depth per channel in this example. We further assume\textsuperscript{10} that color images are delivered in an “interleaved” format with RGB triples per pixel. Thus, this raw data must be separated into three image channels (ExtractChannelsFromRGB). As a consequence, volatile grabbing does not make sense for color images.

\textsuperscript{10}Actually, most image acquisition devices we know do not allow to grab three separated channels.
Before we have a closer look at GrabImg() let us finish the discussion of FGGrab(). For the moment we just assume that GrabImg() delivered the grabbed image in the buffer currInst->InstFrameBuffer[readBuffer]. There are two possibilities to allocate a HALCON image of type Himage; for details about the data structure and the allocation routines, please refer to the Extension Package Programmer’s Manual, section 4.1 on page 59 and section 3.2.2 on page 52, respectively.

HNewImagePtr() as defined in figure 3.34 initializes the data structure, but does not allocate memory for the image matrix, that is the gray values or the raw data itself. Instead, only a pointer to the data is inserted. Note that you have to set initImg to FALSE to avoid an initialization of the image matrix with 0 which would wipe out the grabbed image.

```c
#include "Halcon.h"

Herror HNewImagePtr (Hproc_handle proc_id, Himage *image, INT kind,
                     INT width, INT height, void *data, HBOOL initImg)
{
    /* initialize "image" and insert the pointer "data" as image matrix */
    return(H_MSG_OK);
}
```

Figure 3.34: The prototype for the extension package interface routine HNewImagePtr().

Inserting the image buffer assigned to the image acquisition instance into the new HALCON image avoids the overhead introduced by a memcpy. On the other hand, this is a severe side-effect. Older HALCON images will be overwritten. In any case, the HALCON library that calls FGGrab() must know that you did not use the HALCON memory management to allocate the image matrix. Otherwise, the system will crash when the image object encapsulating the returned Himage structure is cleared from the data base. Thus, for volatile grabbing fginst->halcon_malloc must be set to FALSE, and fginst->clear_proc must be set to NULL. In the examples provided in this manual this was done in FGInit(), see figure 3.16 on page 39.

The standard routine to initialize a Himage structure is HNewImage() as defined in figure 3.35. This routine allocates a new image matrix (using HAlloc). Thus, either a memcpy or a call to ExtractChannelsFromRGB() is necessary in figure 3.33 to fill the matrix with the grabbed image. This induces a small overhead. On the other hand, the resulting HALCON images are independent, which conforms to the HALCON philosophy: The user should decide how long he/she would like to use an image. It should not be overwritten as a side-effect of calling another HALCON operator. Thus, we strongly recommend to implement this non-volatile grabbing strategy as default.

For reasons of backward compatibility, a flag like initImg in HNewImagePtr() is missing in HNewImage(). To surely avoid an (unnecessary) initialization of the new image matrix, you might use the code fragment in figure 3.36. Please make sure to restore the old setting before returning from FGGrab(). This implies not to use HCkP directly as shown in figure 3.36.

In case of byte images the image matrix in a Himage structure is accessed via pixel.b. For a discussion on other supported image types please refer to the Extension Package Programmer’s Manual, section 4.1 on page 59.
#include "Halcon.h"

Herror HNewImage(Hproc_handle proc_id, Himage *image, INT kind,
                  INT width, INT height)
{
    /* initialize "image" and allocate a new image matrix */
    return(H_MSG_OK);
}

Figure 3.35: The prototype for the extension package interface routine HNewImage().

static Herror FGGrab(Hproc_handle proc_id, FGInstance *fginst,
                      Himage *image, INT *num_image)
{
    Herror err;
    INT save;
...
    HReadSysComInfo(proc_id, HGInitNewImage, &save);
    HWriteSysComInfo(proc_id, HGInitNewImage, FALSE);
    for (i=0; i < *num_image; i++)
    {
        err = HNewImage(proc_id,&image[i],BYTE_IMAGE,
                         fginst->image_width,fginst->image_height);
        if (err != H_MSG_OK)
        {
            HWriteSysComInfo(proc_id, HGInitNewImage, save);
            return(err);
        }
    }
    HWriteSysComInfo(proc_id, HGInitNewImage, save);
    HCkP(err);
    ... return(H_MSG_OK);
}

Figure 3.36: Example code for FGGrab(): Avoid initialization of the new image matrix in HNewImage().

Splitting “interleaved” raw color data into three separated image channels is straightforward, see figure 3.37. Please refer to the API manual of your frame grabber to learn about the specific data representation. In our example we assumed a 24 bit per pixel representation of RGB triples.

3.7 Auxiliary Routine: GrabImg()

We suggest to implement an auxiliary routine GrabImg() as defined in figure 3.32 on page 51 as basis for the grabbing routines FGGrab() and FGGrabData() and thus also for FGGrabAsync() and FGGrabDataAsync(). It has to perform the following tasks:
static Herror ExtractChannelsFromRGB(FGInstance *fginst, HBYTE *data, 
    HBYTE *r_img, HBYTE *g_img, HBYTE *b_img)
{
    INT4_8 i,size = fginst->image_width*fginst->image_height;

    for (i=0; i < size; i++)
    {
        *r_img++ = *data++;
        *g_img++ = *data++;
        *b_img++ = *data++;
    }
    return(H_MSG_OK);
}

Figure 3.37: Example code for the auxiliary routine ExtractChannelsFromRGB().

- Terminate pending grabs of other instances in case they use the same A/D converter.
- If there is an asynchronous grab pending: Terminate it in case of a synchronous grab command and wait for its end otherwise.
- If synchronous grabbing is requested or an asynchronously grabbed image is too old: Grab a new image.
- If asynchronous grabbing is requested: Start the next grab (but do not wait for the end of the grab).
- Switch to the next buffer.

Figure 3.38 shows the basic structure of such a routine. Naturally, there is a lot of pseudo-code indicated by < ...> since the grabbing routines etc. depend on the specific API of your image acquisition device. Most of figure 3.38 should be rather self-explaining. However, some specific topics should be discussed in detail.

First of all, if multiple instances are assigned to the same physical device with only one A/D converter, you have to synchronize grabs by these instances. If you enter GrabImg() with asynchronous grabs of other instances pending, you have to cancel these jobs or return an error code (H_ERR_FGDV – “Device busy”). Moreover, in case of multiple instances you have to make sure that the board is correctly parameterized for grabbing by a specific instance. We suggest to provide an auxiliary routine SetInstParam() for this purpose, see figure 3.39. Within this routine you have to reset all parameters that can differ from instance to instance. If this is a time consuming task it might be a good idea to store the current settings of the board in the corresponding BoardInfo structure and set only those values again which differ from the requested values in fginst or additional parameters in the TFGInstance structure currInst.

Throughout this section we assume a ring buffer to which images should be delivered by the image acquisition device. The current buffer for grabbing is indicated by currInst->currBuffer. This index should be returned in the parameter readBuffer. The next grab should be done to the buffer currInst->currBuffer + 1 or 0 if there is a wrap around in the ring structure of the buffers. The corresponding frame buffers, that is pointers to the memory, are accessible via currInst->InstFrameBuffer[i], see also figure 3.17 on page 40.

We recommend to cancel the other jobs. They will be started again when the user requests the corresponding image.
static Herror GrabImg(Hproc_handle proc_id, FGInstance *fginst, 
   INT *readBuffer)
{
   TFGInstance       *currInst = (TFGInstance *)fginst->gen_pointer;
   HBOOL               done = FALSE;
   HBOOL               newGrab = FALSE;
   HBOOL               checkTimeAgain = FALSE;

   if ((!currInst->busy) && (currInst->board->refInst > 1))
      HCkP(SetInstParam(fginst));
   if (currInst->busy)
   {
      if (!fginst->async_grab)
      {
         newGrab = TRUE;
      }
      else
      {
         done = < test whether the pending grab is already finished >;
         if (done)
            newGrab = < test whether the grabbed image is too old >;
         else
            checkTimeAgain = TRUE;
      }
   }
   else
      newGrab = TRUE;
   if (newGrab)
   {
      < grab a new image >;
      done = TRUE;
   }
   if (!done)
      < wait for the end of the current grab >;
   if (checkTimeAgain)
      < test if the new image is too old and grab a new one if necessary >;
   *readBuffer = currInst->currBuffer;
   currInst->currBuffer++;
   if (currInst->currBuffer >= MAX_BUFFERS) currInst->currBuffer = 0;
   if (fginst->async_grab)
      < start the next asynchronous grab >;
   return(H_MSG_OK);
}

Figure 3.38: Example code for GrabImg(): The basic structure.

Note, that asynchronous grabbing might lead to “old” images returned by the grabbing operators. Therefore, grab_image_start, grab_image_async and grab_data_async allow to specify the maximum
static Herror SetInstParam(FGInstance *fginst) {
    TFGInstance *currInst = (TFGInstance *)fginst->gen_pointer;

    /* Everything that you allow to be different for instances */
    /* of the same board (like port and input line etc.) must be */
    /* checked and set again if necessary. */
    /* Note: If this is very time consuming, you might want to */
    /* store the current parameter settings of the board in */
    /* currInst->board and check whether they differ from the */
    /* values in currInst / fginst */
    /* example: */
    /* */
    /* if (currInst->board->port != fginst->port) */
    /* { */
    /* < set the port fginst->port >; */
    /* /* currInst->board->port = fginst->port; */
    /* /* } */
    return(H_MSG_OK);
}

Figure 3.39: The prototype for SetInstParam().

tolerated age of an image, see the HALCON reference manual for details. Consequently, you should
store a timestamp whenever you start grabbing an image, see figure 3.40. Then, before returning an
asynchronously grabbed image check whether too much time has passed and grab a new image again
if necessary. There is one special configuration worth thinking about it for a minute: If you enter
GrabImg() in asynchronous mode (fginst->grab_async is TRUE) with an asynchronous grab pend-
ing (currInst->busy is TRUE) which is not finished up to now (done is FALSE), you have to decide
what to do. If the grab already lasts for too long, you can cancel it and start a new one. However, if the
duration of the grab is still below the maxAge threshold, it is impossible to say whether the image will
be too old or not after completion of the grab. Therefore, we delayed the time check in the example in
figure 3.38 in this specific case by setting checkTimeAgain = TRUE.
static Herror GrabImg (Hproc_handle proc_id, FGInstance *fginst, 
    INT *readBuffer)
{
    #ifdef WIN32
    struct _timeb now;
    #else
    struct timeval now;
    struct timezone tzp;
    #endif
    INT4_8 time_diff;
    ...
    /* test whether the grabbed image is too old: * /
    #ifdef WIN32
    _ftime(&now);
    time_diff = now.millitm - currInst->grabStarted.millitm + 
                1000*(now.time - currInst->grabStarted.time);
    #else
    gettimeofday(&now,&tzp);
    time_diff = (INT4_8)
                (((double)now.tv_sec*1000.0 + (double)now.tv_usec/1000.0) - 
                 ((double)currInst->grabStarted.tv_sec*1000.0 + 
                 (double)currInst->grabStarted.tv_usec/1000.0) + 0.5 );
    #endif
    if (time_diff > currInst->maxAge)
        newGrab = TRUE;
    ...
    if (fginst->async_grab)
    {
        < start the next asynchronous grab >;
        #ifdef WIN32
        _ftime(&currInst->grabStarted);
        #else
        gettimeofday(&currInst->grabStarted,&currInst->tzp);
        #endif
    }
    ... 
    return(H_MSG_OK);
}
3.8 FGGrabAsync()

The routine FGGrabAsync() as defined in figure 3.41 is called by the HALCON operator grab_image_async, see section 1.6 on page 12. It has to perform the following tasks:

- (Re-)set the parameters on the image acquisition device.
- Wait until a pending asynchronous grab is finished or grab a new image if there is no pending job.
- Check if the asynchronously grabbed image is too old. If this is the case, grab a new image.
- Start a new asynchronous grab (without waiting).
- Return a HALCON image containing the grabbed raw data.

```c
#include "Halcon.h"
#include "hlib/CIOFrameGrab.h"

static Herror FGGrabAsync (Hproc_handle proc_id, FGInstance *fginst,
                           double maxDelay, Himage *image, INT *num_image)
{
    /* grab an image asynchronously */
    TFGInstance *currInst = (TFGInstance *)fginst->gen_pointer;

    currInst->maxAge = (INT4_8)(maxDelay+0.5);
    fginst->async_grab = TRUE;

    HCkP(FGGrab(proc_id, fginst, image, num_image));

    return(H_MSG_OK);
}
```

Figure 3.41: The prototype for FGGrabAsync() based on FGGrab().

Since we have chosen an implementation of FGGrab() in section 3.6 on page 51 based on GrabImg() in section 3.7 on page 54, which is more general than necessary for a pure synchronous grabbing, we can easily implement FGGrabAsync() based on FGGrab(), see figure 3.41. All we have to do is to set the asynchronous grabbing mode (fginst->async_grab is TRUE) and to update the threshold for the decision whether an asynchronously grabbed image is too old and thus has to be replaced by a new image (currInst->maxAge), see also figure 3.40.

3.9 FGGrabStartAsync()

The routine FGGrabStartAsync() as defined in figure 3.42 is called by the HALCON operator grab_image_start, see section 1.6 on page 12. It has to perform the following tasks:

- Terminate pending asynchronous grabs of all instances assigned to the current board.
- (Re-)set the parameters on the image acquisition device.
- Start an asynchronous grab.

```c
#include "Halcon.h"
#include "hlib/CIOFrameGrab.h"

static Herror FGGrabStartAsync (Hproc_handle proc_id, FGInstance *fginst,
                                 double maxDelay)
{
    /* start an asynchronous grab */
    return(H_MSG_OK);
}
```

Figure 3.42: The prototype for FGGrabStartAsync().

The implementation of this routine is rather straightforward, see figure 3.43. Please also take a look at the auxiliary routine GrabImg() in section 3.7 on page 54 which is the counterpart to FGGrabStartAsync() finishing the grab started here. Note, that in general all pending grabs of instances assigned to the same board have to be canceled: A pending job of the current instance, since we want to start a new grab, but also grabs started by other instances, since in most cases they share the A/D

```c
static Herror FGGrabStartAsync(Hproc_handle proc_id, FGInstance *fginst,
                                 double maxDelay)
{
    TFGInstance *currInst = (TFGInstance *)fginst->gen_pointer;

    currInst->maxAge = (INT4_8)(maxDelay + 0.5);

    < terminate pending grabs of other instances using the same ADC >;
    if (currInst->busy)
        < terminate the current grab >;
    else
        HCKP(SetInstParam(fginst));

    < start the new asynchronous grab >;
    #ifdef WIN32
        _ftime(&currInst->grabStarted);
    #else
        gettimeofday(&currInst->grabStarted,&currInst->tzp);
    #endif

    currInst->busy = TRUE;

    return(H_MSG_OK);
}
```

Figure 3.43: Example code for FGGrabStartAsync().
3.10 FGGrabData()

The routine FGGrabData() as defined in figure 3.44 is called by the HALCON operator grab_data\textsuperscript{13}, see section 1.6 on page 12. It has to perform the following tasks:

- Terminate pending asynchronous grabs.
- (Re-)set the parameters on the image acquisition device\textsuperscript{14}.
- Grab an image \emph{synchronously}.
- Return preprocessed image data based on the grabbed raw data.

```c
#include "Halcon.h"
#include "hlib/CIOFrameGrab.h"

static Herror FGGrabData(Hproc_handle proc_handle, FGInstance *fginst, 
                        Himage **image, INT **num_channel, INT *num_image, 
                        Hrlregion ***region, INT *num_region, 
                        Hcont ***cont, INT *num_cont, 
                        Hcpar **data, INT *num_data)
{
    /* grab an image synchronously and return preprocessed image data */
    /* in terms of images, regions, XLD contours, and control data */
    return(H_MSG_OK);
}
```

Figure 3.44: The prototype for FGGrabData().

A routine like this should be implemented if the image acquisition device hardware offers some specific features that support an image preprocessing. It might also be more efficient to segment color images within the image acquisition interface even without hardware support, because in this case one can avoid the channel splitting (see figure 3.37 on page 55) and work on the original raw data instead.

The implemented preprocessing is up to \emph{you} (and maybe dependent on some specific hardware features of your image acquisition device). We cannot provide example source code for that. However, we

\textsuperscript{12}However, only the performance is affected. If you terminate an asynchronous grab, a new grab will be launched when you access the corresponding image with one of the HALCON grabbing operators grab_image, grab_image_async, grab_data, or grab_data_async.

\textsuperscript{13}In the framework suggested in this manual it is also called by FGGrabDataAsync(), see section 3.11 on page 64.

\textsuperscript{14}with the function SetInstParam. Mostly it is only needed for multiple devices sharing one frame grabber board, see also figure 3.39.
will indicate how to allocate image regions encoded in the data structure Hr1region. Please see the Extension Package Programmer’s Manual, section 4.2 on page 61 and chapter 5 on page 69, for both a discussion of this data type and routines to manipulate it. Please note, that in most cases you will have to provide additional parameters to control the creation of the preprocessed image data. We suggest to use FGSetParam() for that purpose, see section 3.12 on page 65.

```c
#include "Halcon.h"

Herror HAllocRLNumLocal (Hproc_handle proc_id, Hrlregion **region,
                      size_t len)
{
    /* initialize "region" and temporarily allocate memory for "len" chords */
    return(H_MSG_OK);
}
```

Figure 3.45: The prototype for the extension package interface routine HAllocRLNumLocal().

```c
static Herror FGGrabData(Hproc_handle proc_handle,FGInstance *fginst,
                           Himage **image,INT **num_channel,INT *num_image,
                           Hrlregion ***region,INT *num_region,
                           Hcont ***cont,INT *num_cont,
                           Hcpar **data,INT *num_data)
{
    TFGInstance *currInst = (TFGInstance *)fginst->gen_pointer;
    INT    readBuffer;

    Hchk(GrabImg (proc_id, fginst, &readBuffer));

    /* example: Allocate two regions (e.g. one for all image parts of a */
    /* specific color and one for the rest of the image) */
    Hchk(HAllocRLNumLocal(proc_id,&region[0],
                           fginst->image_width*fginst->image_height/2));
    Hchk(HAllocRLNumLocal(proc_id,&region[1],
                           fginst->image_width*fginst->image_height/2));
    *rlalloc_type = FG_RLALLOC_LOCAL;
    *num_region = 2;

    /* Well the segmentation itself is up to you :-) */
    ...

    fginst->async_grab = FALSE;
    return(H_MSG_OK);
}
```

Figure 3.46: Example code for FGGrabData(): The basic structure.
The HALCON library passes an array of MAX_OBJ_PER_PAR\textsuperscript{15} pointers to Hrlregion in FGGrabData(). However, the region data itself is not allocated automatically. This has to be done within your interface. There are several methods to do so.

You can allocate region data

- temporarily on stacks inside the HALCON library
- temporarily on the heap
- permanently on the heap

The first two methods include an automatic garbage collection in case you return an error as a result of FGGrabData() and should therefore be preferred. The most flexible memory allocation method is the second one,\textsuperscript{16} which is also used in the example in figure 3.46. The Extension Package Interface routine HAllocRLNumLocal() as defined in figure 3.45 is used to temporarily allocate memory for the specified number of chords and to initialize the Hrlregion structure. Since we do not know the number of chords in advance we have used a rather conservative estimate in figure 3.46. Note, that you can change this number dynamically using HReallocRLNumLocal().

<table>
<thead>
<tr>
<th>rlalloc_type</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>FG_RLALLOC_TMP</td>
<td>Temporary data on stacks allocated with HAllocRLTmp() or HAllocRLNumTmp() \textbf{Attention:} In this case you MUST allocate the image regions in ascending order, because in the HALCON interface the corresponding freeing is done in descending order!</td>
</tr>
<tr>
<td>FG_RLALLOC_LOCAL</td>
<td>Temporary data on the heap allocated with HAllocRLLocal() or HAllocRLNumLocal()</td>
</tr>
<tr>
<td>FG_RLALLOC_PERMANENT</td>
<td>Permanent data on the heap allocated with HAllocRL() or HAllocRLNum()</td>
</tr>
</tbody>
</table>

Figure 3.47: Defines for indicating the memory allocation strategy for regions in FGGrabData().

Whatever you decide to use, you have to indicate the memory allocation strategy to the HALCON library using one of the defines listed in figure 3.47 as return value for the parameter rlalloc_type. If you fail to do so, you will encounter program crashes.

Beside region data FGGrabData can also return images, contours, and data in form of strings or numbers. Different to FGGrab you can return tuples of (multi-channel) images with different sizes and types. Especially for 3D data images, which contain floating point data in most cases, this is very useful.

\textsuperscript{15}In the current version this define is set to 100 000. That should be more than enough to hold all reasonable segmentation results.

\textsuperscript{16}Please refer to the descriptions of HAllocRLLocal(), HAllocRLNumLocal(), and HReallocRLLocal() in the Extension Package Programmer’s Manual, section 3.2.1 on page 49.
3.11 FGGrabDataAsync()

The routine FGGrabDataAsync() as defined in figure 3.48 is called by the HALCON operator grab_data_async, see section 1.6 on page 12. It has to perform the following tasks:

- (Re-)set the parameters on the image acquisition device.
- Wait until a pending asynchronous grab is finished or grab a new image if there is no pending job.
- Check if the asynchronously grabbed image is too old. If this is the case grab a new image.
- Start a new asynchronous grab (without waiting).
- Return preprocessed image data based on the grabbed raw data.

```c
#include "Halcon.h"
#include "hlib/CIOFrameGrab.h"

static Herror FGGrabDataAsync(Hproc_handle proc_handle, FGInstance *fginst,
                              double maxDelay,
                              Himage **image, INT **num_channel, INT *num_image,
                              Hrlregion ***region, INT *num_region,
                              Hcont ***cont, INT *num_cont,
                              Hcpar **data, INT *num_data)
{
    /* grab an image synchronously and return preprocessed image data */
    /* in terms of images, regions, XLD contours, and control data */
    TFGInstance *currInst = (TFGInstance *)fginst->gen_pointer;

    currInst->maxAge = (INT4_8)(maxDelay+0.5);
    fginst->async_grab = TRUE;

    HCKP(FGGrabData(proc_id, fginst, image, num_channel, num_image,
                    region, num_region, cont, num_cont, data, num_data));

    return(H_MSG_OK);
}
```

Figure 3.48: The prototype for FGGrabDataAsync() based on FGGrabData().

Since we have chosen an implementation of FGGrabData() in section 3.10 on page 61 based on GrabImg() in section 3.7 on page 54, which is more general than necessary for a pure synchronous grabbing, we can easily implement FGGrabDataAsync() based on FGGrabData(), see figure 3.48. Please see also FGGrabAsync() in figure 3.41 on page 59 which is the corresponding routine to grab images instead of preprocessed image data asynchronously.
3.12 FGSetParam()

The routine FGSetParam() as defined in figure 3.49 is called by the HALCON operator set_framegrabber_param, see section 1.6 on page 12. It has to perform the following tasks:

- Parse the specified parameter.
- Set the parameter value(s) for the specified instance or return an error code.

```
#include "Halcon.h"
#include "hlib/CIOFrameGrab.h"

static Herror FGSetParam(Hproc_handle proc_id, FGInstance *fginst,
                          char *param, Hcpar *value, INT num)
{
    /* set the specified parameter value for an instance */
    return(H_MSG_OK);
}
```

Figure 3.49: The prototype for FGSetParam().

<table>
<thead>
<tr>
<th>define</th>
<th>name</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FG_PARAM_HORIZONTAL_RESOLUTION</td>
<td>&quot;horizontal_resolution&quot;</td>
<td>LONG_PAR</td>
</tr>
<tr>
<td>FG_PARAM_VERTICAL_RESOLUTION</td>
<td>&quot;vertical_resolution&quot;</td>
<td>LONG_PAR</td>
</tr>
<tr>
<td>FG_PARAM_IMAGE_WIDTH</td>
<td>&quot;image_width&quot;</td>
<td>LONG_PAR</td>
</tr>
<tr>
<td>FG_PARAM_IMAGE_HEIGHT</td>
<td>&quot;image_height&quot;</td>
<td>LONG_PAR</td>
</tr>
<tr>
<td>FG_PARAM_START_ROW</td>
<td>&quot;start_row&quot;</td>
<td>LONG_PAR</td>
</tr>
<tr>
<td>FG_PARAM_START_COL</td>
<td>&quot;start_column&quot;</td>
<td>LONG_PAR</td>
</tr>
<tr>
<td>FG_PARAM_FIELD</td>
<td>&quot;field&quot;</td>
<td>STRING_PAR</td>
</tr>
<tr>
<td>FG_PARAM_BITS_PER_CHANNEL</td>
<td>&quot;bits_per_channel&quot;</td>
<td>LONG_PAR</td>
</tr>
<tr>
<td>FG_PARAM_COLOR_SPACE</td>
<td>&quot;color_space&quot;</td>
<td>STRING_PAR</td>
</tr>
<tr>
<td>FG_PARAM_GENERIC</td>
<td>&quot;generic&quot;</td>
<td>Hcpar</td>
</tr>
<tr>
<td>FG_PARAM_EXT_TRIGGER</td>
<td>&quot;external_trigger&quot;</td>
<td>STRING_PAR</td>
</tr>
<tr>
<td>FG_PARAM_CAMERA_TYPE</td>
<td>&quot;camera_type&quot;</td>
<td>STRING_PAR</td>
</tr>
<tr>
<td>FG_PARAM_DEVICE</td>
<td>&quot;device&quot;</td>
<td>STRING_PAR</td>
</tr>
<tr>
<td>FG_PARAM_PORT</td>
<td>&quot;port&quot;</td>
<td>LONG_PAR</td>
</tr>
<tr>
<td>FG_PARAM_LINE_IN</td>
<td>&quot;line_in&quot;</td>
<td>LONG_PAR</td>
</tr>
</tbody>
</table>

Figure 3.50: Defines for the standard parameters used in open_framegrabber.

A routine like this should be implemented if you would like to use additional parameters to tune specific hardware features or to change the standard parameters specified in open_framegrabber on the fly. The names of the standard parameters are fixed, see figure 3.50. Please note, that "field" (corresponding to fginst->field) is externally defined as string, but internally as integer using the conversion indicated in figure 3.51. Note, further that "external_trigger" is externally defined as string ("true" or "false") but is internally defined as boolean value of type HB00L.
The parameter value to be set is passed in a structure of type Hcpar. Please refer to the Extension Package Programmer’s Manual, section 4.4 on page 65, for a detailed description of this structure. Some comments have been made on page 45 in this manual as well.

Please follow also the remarks about parameter naming (see section 1.7 on page 16) and keep in mind the special remarks about using the image acquisition interface with ActivVisionTools and the HALCON Image Acquisition Assistant, see section 1.8 on page 17.

<table>
<thead>
<tr>
<th>external define</th>
<th>external name</th>
<th>internal define</th>
</tr>
</thead>
<tbody>
<tr>
<td>FG_FIRST_FIELD_TXT</td>
<td>&quot;first&quot;</td>
<td>FG_FIRST_FIELD</td>
</tr>
<tr>
<td>FG_SECOND_FIELD_TXT</td>
<td>&quot;second&quot;</td>
<td>FG_SECOND_FIELD</td>
</tr>
<tr>
<td>FG_NEXT_FIELD_TXT</td>
<td>&quot;next&quot;</td>
<td>FG_NEXT_FIELD</td>
</tr>
<tr>
<td>FG_FULL_FRAME_TXT</td>
<td>&quot;interlaced&quot;</td>
<td>FG_FULL_FRAME</td>
</tr>
<tr>
<td>FG_PROGRESSION_FRAME_TXT</td>
<td>&quot;progressive&quot;</td>
<td>FG_PROGRESSION_FRAME</td>
</tr>
<tr>
<td>FG_FIELD_DEFAULT_TXT</td>
<td>&quot;default&quot;</td>
<td>FG_FIELD_DEFAULT</td>
</tr>
</tbody>
</table>

Figure 3.51: Internal and external representation of values for the parameter Field in open_framegrabber.

Feel free to choose arbitrary names for additional parameters. However, we suggest to try to preserve the look and feel of typical HALCON operators and to choose names in correspondence with the API of the specific image acquisition device. Please do not forget to return these parameter names for the query FG_QUERY_PARAMETERS in FGInfo(), see figure 3.30 on page 50.

The HALCON image acquisition integration interface also provides the opportunity to pass multi-parameter values. This enables you to pass a tuple of values for the parameter param, with num denoting the number of values.

In general you will have to extend the structures BoardInfo and TFGInstance to hold these additional parameters. Figure 3.52 shows example code\(^\text{17}\) for activating volatile grabbing which only uses the entry volatileMode already included in the TFGInstance structure. Please see also the comments on allocating buffers according to figure 3.17 on page 40 in section 3.6 on page 51.

\(^{17}\)Please refer to hAcqTemplate.c for a detailed discussion.
#define FG_PARAM_VOLATILE "volatile"

static Herror FGSetParam(Hproc_handle proc_id, FGInstance *fginst, char *param, Hcpar *value, INT num)
{
    TFGInstance *currInst = (TFGInstance *)fginst->gen_pointer;
    INT i;
    BoardInfo *board = currInst->board;
    INT4_8 sizeBuffer;
    if (!strcmp(param, FG_PARAM_VOLATILE))
    {
        if (value->type != STRING_PAR) return(H_ERR_FGPART);
        if (!strcmp(value->par.s, "enable"))
        {
            if (fginst->num_channels != 1) return(H_ERR_FGPARV);
            if (!currInst->volatileMode)
            {
                if (!currInst->allocBuffer)
                {
                    /* This specific instance uses buffers assigned to the board. */
                    if (board->refBuffer == 1)
                    {
                        /* No other instance uses the board buffer. Just transfer */
                        /* them to the instance: */
                        for (i=0; i < MAX_BUFFERS; i++)
                        {
                            currInst->InstFrameBuffer[i] = board->BoardFrameBuffer[i];
                            board->BoardFrameBuffer[i] = NULL;
                        }
                        board->sizeBuffer = 0;
                    }
                    else
                    {
                        /* There are other instances using the board buffers. */
                        sizeBuffer = fginst->image_width * fginst->image_height * 
                        ((fginst->bits_per_channel+7) / 8)*fginst->num_channels;
                        for (i=0; i < MAX_BUFFERS; i++)
                            HCkP(HAlloc (proc_id,(size_t)sizeBuffer,
                        &currInst->InstFrameBuffer[i]));
                        board->refBuffer--;
                        currInst->allocBuffer = TRUE;
                    }
                }
            }
        }
        currInst->volatileMode = TRUE;
        fginst->halcon_malloc = FALSE;
        fginst->clear_proc = NULL;
    }
    ...
    return(H_MSG_OK);
}
3.13 FGGetParam()

The routine FGGetParam() as defined in figure 3.53 is called by the HALCON operator `get_framegrabber_param`, see section 1.6 on page 12. It has to perform the following tasks:

- Parse the specified parameter.
- Return the current parameter value(s) for the specified instance or return an error code.

```c
#include "Halcon.h"
#include "hlib/CIOFrameGrab.h"

static Herror FGGetParam(Hproc_handle proc_id, FGInstance *fginst, char *param, Hcpar *value, INT *num)
{
    /* return the specified parameter value for an instance */
    TFGInstance *currInst = (TFGInstance *)fginst->gen_pointer;
    *num = 1;
    ...
    if (!strcmp(param, FG_PARAM_VOLATILE))
    {
        value->type = STRING_PAR;
        value->par.s = ( currInst->volatileMode ? "enable" : "disable" );
    }
    else if ...
    else
        /* parameter not supported */
        return(H_ERR_FGPARAM);

    return(H_MSG_OK);
}
```

Figure 3.53: The prototype for FGGetParam() and a simple example.

This routine is the counterpart of FGSetParam(), see section 3.12 on page 65. The values for all standard parameters used in `open_framegrabber` are automatically returned by the HALCON library. So you do not need to provide code for the parameters listed in figure 3.50 on page 65. However, please make sure to replace default values in `fginst` by the current settings if necessary in FGOpen().

FGGetParam() should be able to handle all additional device-specific parameters you introduced in FGSetParam(). The parameter value has to be returned in a structure of type `Hcpar`. Please see the Extension Package Programmer’s Manual, section 4.4 on page 65, for a detailed description of this structure. A short description is given on page 45. Like FGSetParam(), FGGetParam offers the opportunity to return multi-parameter values. Therefore, assign index-wise each parameter to `value[i]`. But do not forget to specify the type of the value in `value[i].type` and to set the function parameter `*num` to the number of returned values. Figure 3.53 shows a simple example assuming that there is only one additional parameter controlling volatile grabbing, see also figure 3.52 on page 67.

---

18 Remember that the names of these parameters must be returned for the query `FG_QUERY_PARAMETERS` by FGInfo(), see figure 3.30 on page 50.
As an additional information to the user all parameters should have a parameter with the suffix
'\_description' containing a meaningful tooltip for the described parameter. For integer and float
values there should also be a parameter with the suffix '\_range', which contains a tuple with at least 4
values: minimum, maximum, step width, and default/current value. To allow automatic value changes,
'auto' could be the fifth entry. The possible values for string parameters are implemented with the
suffix '\_values'. See also section 1.8 on page 17.

3.14 FGSetLut()

The routine FGSetLut() as defined in figure 3.54 is called by the HALCON operator
set\_framegrabber\_lut, see section 1.6 on page 12. It has to perform the following task:

- Set the lookup table for the specified instance.

```c
#include "Halcon.h"
#include "hlib/CIOFrameGrab.h"

static Herror FGSetLut(Hproc_handle proc_id, FGInstance *fginst,
                      INT4_8 *red, INT4_8 *green, INT4_8 *blue, INT num)
{
    /* set the specified lookup table for an instance */
    return(H_MSG_OK);
}
```

Figure 3.54: The prototype for FGSetLut().

A modification of an image acquisition device’s lookup table might be used for a gamma correction
or white balancing. The input to FGSetLut() are three integer arrays for the red, green, and blue
components of the LUT and the number of entries in these arrays. Whether lookup tables are supported
or not and how to handle such lookup tables depends on the specific device (and its API). Therefore, we
cannot provide source code for this task.

Please note, that the modification of a frame grabber’s lookup table will affect other instances assigned
to the same board. Thus, you should think about a mechanism to check whether instance-specific LUTs
differ and to restore them prior to grabbing if necessary.
3.15  FGGetLut()

The routine FGGetLut() as defined in figure 3.55 is called by the HALCON operator get_framegrabber_lut, see section 1.6 on page 12. It has to perform the following task:

- Return the lookup table for the specified instance.

```c
#include "Halcon.h"
#include "hlib/CIOFrameGrab.h"

static Herror FGGetLut (Hproc_handle proc_id, FGInstance *fginst,
                        INT4_8 *red, INT4_8 *green, INT4_8 *blue, INT *num)
{
    /* return the specified lookup table for an instance */
    return(H_MSG_OK);
}
```

Figure 3.55: The prototype for FGGetLut().

This routine is the counterpart to FGSetLut() in section 3.14 on page 69. It has to return three integer arrays for the red, green, and blue components of the LUT and the number of entries in these arrays. Memory for FG_MAX_LUT_LENGTH entries per array has already been allocated by the HALCON library.

Whether lookup tables are supported or not and how to handle such lookup tables depends on the specific device (and its API). Therefore, we cannot provide source code for this task.

3.16  FGSetCallback()

The routine FGSetCallback() as defined in figure 3.56 is called by the HALCON operator set_framegrabber_callback, see section 1.6 on page 12. It has to perform the following task:

- Register a callback function for an image acquisition device.

Note that the execution time of a user-specific callback function should always be as short as possible since during the execution of a callback function the handling of further internal callbacks might be blocked. This can be achieved by removing the actual processing from the user-specific callback function to a separate thread that is controlled via signals or events.

Attention: Since HDevelop is a single-threaded application, it is currently not possible to use the callback functionality within HDevelop. Thus, the callback functionality is only available via the HALCON language interfaces.

The parameter myCallback contains a pointer to the function to register, myFunc specifies to which function of the underlying API the callback should be connected. Suggested values for myFunc are 'exception', 'exposure_end', 'exposure_start', and 'transfer_end'. Depending on the functionality of the underlying API, additional values are possible. The parameter context denotes an optional pointer to user-specific context data.

19In the current version FG_MAX_LUT_LENGTH is 65536 corresponding to a maximum resolution of 16 bits per channel.
#include "Halcon.h"
#include "hlib/CIOFrameGrab.h"

static Herror FGSetCallback (Hproc_handle proc_id, FGInstance *fginst,
    char *myFunc, HAcqCallback myCallback,
    void *context)
{
    /* register the specified callback for an instance */
    return(H_MSG_OK);
}

Figure 3.56: The prototype for FGSetCallback().

### 3.17 FGGetCallback()

The routine FGGetCallback() as defined in figure 3.57 is called by the HALCON operator get_framegrabber_callback, see section 1.6 on page 12. It has to perform the following task:

- Query a callback function of an image acquisition device.

This routine is the counterpart to FGSetCallback() in section 3.16 on page 70. If the specified callback function was not registered before this function returns NULL.

All actually supported callback types of a specific image acquisition device can be queried by calling FGGetParam with the parameter 'available_callback_types'. Once the callback is registered, on every occurrence of the underlying event (e.g., the notification that the exposure has finished) the specified callback function will be called. If the callback function is set to NULL, the corresponding callback will be unregistered.

#include "Halcon.h"
#include "hlib/CIOFrameGrab.h"

static Herror FGGetCallback (Hproc_handle proc_id, FGInstance *fginst,
    char *myFunc, HAcqCallback *myCallback,
    void **context)
{
    /* return the specified callback for an instance */
    return(H_MSG_OK);
}

Figure 3.57: The prototype for FGGetCallback().
Chapter 4

Generating an Acquisition Interface Library

Whenever an image acquisition device is accessed for the very first time by using `open_framegrabber` or `info_framegrabber`, the corresponding HALCON acquisition interface library, a dynamically loadable module,\(^1\) is loaded. This chapter contains information on how to generate such a dynamic object. Please refer to the documentation of your programming environment for details on compiling and linking.

4.1 Generating an Acquisition Interface Under Windows

To build a HALCON acquisition interface you have to generate a DLL from the file containing the source code of your interface (like `hAcqTemplate.c`) by linking the corresponding object file(s) with

- the HALCON library `halcon.lib` and
- the libraries provided by the manufacturer of the image acquisition device

Please note that both an example makefile as well as a workspace for the template code `hAcqTemplate.c` are available in the directory `%HALCONEXAMPLES%\ia_integration\source`.

Make sure that the optimization is switched on for the compilation process (i.e., create a “Release”, not a “Debug” version); otherwise, grabbing, especially of color images, will be slowed down significantly!

If you want to use the new acquisition interface in HALCON XL as well, you must create a second DLL that is linked to the HALCON XL library `halconxl.lib` instead of `halcon.lib`.

In order to be automatically loadable by HALCON or HALCON XL, the name of the acquisition interface library must start with the prefix `hAcq` (and for HALCON XL end with the suffix `xl`). The rest of the library name automatically defines the name of the interface as used in the operator `open_framegrabber`. For example, if your interface library is named `hAcqMyDevice.dll`, you access the image acquisition device by calling

\(^1\)A DLL for Windows or a shared library for Linux systems, respectively.
Generating an Acquisition Interface Library

4.2 Generating an Acquisition Interface Under Linux

To build a HALCON acquisition interface you have to generate a shared library from the file containing the source code of your interface (like hAcqTemplate.c) by linking the corresponding object file(s) using `ld`. Note that an example makefile for the template code `hAcqTemplate.c` is available in the directory $HALCONEXAMPLES/ia_integration/source.

We recommend to use some level of optimization for the compilation process; otherwise, grabbing, especially of color images, will be slowed down significantly!

In order to be automatically loadable by HALCON, the name of the acquisition interface library must start with the prefix `hAcq` (and for HALCON XL end with the suffix `xl`). The rest of the library name automatically defines the name of the interface as used in the operator `open_framegrabber`. For example, if your interface library is called `hAcqMyDevice.so`, you access the image acquisition device by calling

```c
open_framegrabber('MyDevice', ... )
```

Note, that the location of the generated HALCON acquisition interface must be included in the search path for dynamic objects, i.e., the variable `PATH`. The same might be true for any library provided by the manufacturer of the image acquisition device that is used by your HALCON interface.

Caution: Do not copy an acquisition interface DLL into the Windows system directories, as it would be loaded twice in this case!
Appendix A

Changes Between Versions 4 and 5 of the HALCON Image Acquisition Interface

This chapter summarizes all syntactic and semantic differences between the HALCON image acquisition interface version 3 and version 4. Please note, that because of these changes older image acquisition interfaces do not run with HALCON 11.0 or higher and vice versa.

- INTERFACE_VERSION = 5
  Thus, all interfaces compatible to HALCON 11.0 have a revision number “5.x”, with x starting at 0, see the corresponding define INTERFACE_REVISION in each image acquisition interface.

- The operators info_framegrabber, open_framegrabber, close_framegrabber, and close_all_framegrabbers now have the multithreading type ‘reentrant’ instead of ‘exclusive’, i.e., these operators do not block any other running HALCON operators anymore.

- The maximum number of instances per image acquisition interface (FG_MAX_INST) has been increased from 64 to 256, i.e., for example you can access up to 256 GigE Vision compliant devices simultaneously by the HALCON GigEVision interface.
Appendix B

Changes Between Versions 3 and 4 of the HALCON Image Acquisition Interface

This chapter summarizes all syntactic and semantic differences between the HALCON image acquisition interface version 3 and version 4. Please note, that because of these changes older image acquisition interfaces do not run with HALCON 9.0 or higher and vice versa.

- INTERFACE_VERSION = 4
  Thus, all interfaces compatible to HALCON 9.0 have a revision number “4.x”, with x starting at 0, see the corresponding define INTERFACE_REVISION in each image acquisition interface.

- New operators get_framegrabber_callback and set_framegrabber_callback
  HALCON has been extended with the two new operators get_framegrabber_callback and set_framegrabber_callback. The operator set_framegrabber_callback allows to register a HALCON-specific callback function for an image acquisition device. The operator get_framegrabber_callback queries a callback function. Note that not all image acquisition interfaces actually support the use of these callbacks.

- Parallel HALCON by default
  Parallel HALCON is now the default, i.e., the library that is used as standard corresponds to the library that was previously used for Parallel HALCON. Thus, all image acquisition interfaces have to use the prefix par no more.

- HALCON XL
  HALCON is now able to handle images that are larger than 32767 x 32767. For this, HALCON XL has been introduced. For building image acquisition interfaces for HALCON XL, you have to use the suffix xl, e.g., hAcqGigEVisionxl.dll or hAcq1394IIDCx1.so.

- The maximum number of parameters per image acquisition handle (FG_MAX_PARAM) has been increased from 1024 to 2048.
There are two new error codes regarding image acquisition: H_ERR_FGCALLBACK denotes an unsupported callback type, H_ERR_FGDEVLOST indicates that the device was lost.

The template code for new image acquisition interfaces has been moved from %HALCONEXAMPLES%\ia_integration to %HALCONEXAMPLES%\ia_integration\source. Furthermore, the corresponding project files are now also available for Visual Studio 2005 and include additional configurations for building HALCON XL interfaces.
Appendix C

Changes Between Versions 2 and 3 of the HALCON Image Acquisition Interface

This chapter summarizes all syntactic and semantic differences between the HALCON image acquisition interface version 2 and version 3. Please note, that because of these changes older image acquisition interfaces do not run with HALCON 8.0 or higher and vice versa.

- **New terminology:** *Image acquisition interface*
  
  Since digital cameras are not really based on an actual frame grabber board, we no longer use the term HALCON frame grabber interface. Instead, we use the term *HALCON acquisition interface*, and the term *image acquisition device* is used as a substitute for either a frame grabber board or a digital camera. For backwards compatibility reasons, the names of the HALCON operators and also the underlying names for the data structures and error codes have been unchanged.

- **INTERFACE_VERSION = 3**
  
  Thus, all interfaces compatible to HALCON 8.0 have a revision number “3.x”, with x starting at 0, see the corresponding define INTERFACE_REVISION in each image acquisition interface.

- **New prefix hAcq:** Acquisition interface libraries now start with the prefix hAcq instead of HFG.

- **In the operator open_framegrabber,** the parameter Gain has been replaced by the new parameter Generic, in which a tuple of interface-specific values can be passed that must be set before the device is opened. The default value of Generic is the same as for the former parameter Gain, thus the change is mostly backwards compatible. A new value for the gain can still be set dynamically by calling set_framegrabber_param with the additional parameter ’gain’.

- **New operators grab_data and grab_data_async**
  
  The new operators grab_data and grab_data_async replace the operators grab_region and grab_region_async. These new operators are much more flexible and provide the possibility to return any kind of (preprocessed) image data in terms of images, regions, XLD contours, and control data.
• The operator info_framegrabber provides the new queries 'parameters_readonly', 'parameters_writeonly', 'start_column', 'start_row', 'horizontal_resolution', 'vertical_resolution', 'image_width', and 'image_height'.

• There are two new error codes regarding image acquisition: H_ERR_FGCLOSE indicates that a device could not be closed. H_ERR_FGCAMFILE indicates that a camera configuration file could not be opened.

• The HTML manual pages use the name of the interface, e.g., hAcq1394IIDC.html instead of nt-x86-1394iidc.html.
Appendix D

Changes Between Versions 1 and 2 of the HALCON Image Acquisition Interface

This chapter summarizes all syntactic and semantic differences between the HALCON image acquisition interface version 1 and version 2. Please note, that because of these changes older image acquisition interfaces do not run with HALCON 6.0 or higher and vice versa. This applies to every supported operating system since the library symbols of the HALCON image acquisition interface have changed. Furthermore, image acquisition interfaces built with HALCON 6.0 will only run with HALCON 6.1 or HALCON 7.0 in case that you do not grab multi-channel images!

The following variable names of the structures FGClass and FGInstance have changed:

<table>
<thead>
<tr>
<th>Version 1</th>
<th>Version 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits</td>
<td>bits_per_channel</td>
</tr>
<tr>
<td>generic</td>
<td>camera_type</td>
</tr>
<tr>
<td>start_line</td>
<td>start_row</td>
</tr>
<tr>
<td>internal_width</td>
<td>horizontal_resolution</td>
</tr>
<tr>
<td>internal_height</td>
<td>vertical_resolution</td>
</tr>
<tr>
<td>width</td>
<td>image_width</td>
</tr>
<tr>
<td>height</td>
<td>image_height</td>
</tr>
<tr>
<td>sel_input</td>
<td>line_in</td>
</tr>
</tbody>
</table>

In addition, the structure FGClass does not contain the variables bw_available, color_available, gray_available, width_max, height_max, and mode anymore. The structure FGInstance does not contain the variable threshold anymore. The variable num_channels has been moved from FGClass to FGInstance.
Note that not only the notation has changed but also the meaning of variables: `bits_per_channel` now denotes the number of (actually transferred) bits per pixel for one image channel while `bits` denoted the number of bits per pixel over all channels. The following table shows how typical images are encoded:

<table>
<thead>
<tr>
<th></th>
<th>Version 1</th>
<th></th>
<th>Version 2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>bits</td>
<td>color_space</td>
<td>bits_per_channel</td>
<td>color_space</td>
</tr>
<tr>
<td>8 bit gray value image</td>
<td>8</td>
<td>gray</td>
<td>8</td>
<td>gray</td>
</tr>
<tr>
<td>10 bit gray value image</td>
<td>10</td>
<td>gray</td>
<td>10</td>
<td>gray</td>
</tr>
<tr>
<td>12 bit gray value image</td>
<td>12</td>
<td>gray</td>
<td>12</td>
<td>gray</td>
</tr>
<tr>
<td>RGB image, 8 bit per channel</td>
<td>24</td>
<td>rgb</td>
<td>8</td>
<td>rgb</td>
</tr>
<tr>
<td>RGB image, 5 bit per channel</td>
<td>16</td>
<td>rgb</td>
<td>5</td>
<td>rgb</td>
</tr>
</tbody>
</table>

The number of channels is implicitly encoded in the variable `color_space`: If the variable is set to 'rgb' or 'yuv' for example, the number of channels is 3; if the variable is set to 'gray', the number of channels is 1. We recommend to set the variable `num_channels` to the inferred number of channels in `FGOpen()` while evaluating the parameters `bits_per_channel` and `color_space` (see section 3.3 on page 34). Correspondingly, the names of the following defines have changed:

<table>
<thead>
<tr>
<th>Version 1</th>
<th>Version 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>FG_QUERY_GENERIC</td>
<td>FG_QUERY_CAMERA_TYPE</td>
</tr>
<tr>
<td>FG_PARAM_FGWIDTH</td>
<td>FG_PARAM_HORIZONTAL_RESOLUTION</td>
</tr>
<tr>
<td>FG_PARAM_FGHEIGHT</td>
<td>FG_PARAM_VERTICAL_RESOLUTION</td>
</tr>
<tr>
<td>FG_PARAM_WIDTH</td>
<td>FG_PARAM_IMAGE_WIDTH</td>
</tr>
<tr>
<td>FG_PARAM_HEIGHT</td>
<td>FG_PARAM_IMAGE_HEIGHT</td>
</tr>
<tr>
<td>FG_PARAM_BITS</td>
<td>FG_PARAM_BITS_PER_CHANNEL</td>
</tr>
<tr>
<td>FG_PARAM_GENERIC</td>
<td>FG_PARAM_CAMERA_TYPE</td>
</tr>
<tr>
<td>FG_PARAM_LINE</td>
<td>FG_PARAM_LINE_IN</td>
</tr>
<tr>
<td>H_ERR_FGGP</td>
<td>H_ERR_FGCT</td>
</tr>
</tbody>
</table>

Furthermore, a new define called `FG_QUERY_INFO_BOARDS` has been added. This define is used in a new branch in the function `FGInfo()` to query all the installed frame grabber boards accessible from your interface (see section 3.5 on page 45). The value of the define `FG_INTERFACE_VERSION` should be set from 1.x to 2.0.

The parameter lists of the functions `FGSetParam()` and `FGGetParam()` were extended to handle multi-parameter values. Therefore, the declaration of these functions has changed to

```c
static Herror FGSetParam(Hproc_handle proc_id, FGInstance *fginst,
                         char *param, Hcpar *value, INT num);

static Herror FGGetParam(Hproc_handle proc_id, FGInstance *fginst,
                         char *param, Hcpar *value, INT *num);
```

Make sure that you set `*num` to a reasonable value within `FGSetParam()`. See also section 3.12 on page 65 and section 3.13 on page 68 for more details.
To fix a bug of the old CIOFGTemplate.c delete the line

```c
fginst->async_grab = TRUE
```

at the end of the functions FGGrabStartAsync(), FGGrabAsync() and FGGrabRegionAsync().
# Appendix E

## HALCON Error Codes

In this chapter all HALCON error codes relevant for programming an image acquisition interface are summarized. Please refer to `hAcqTemplate.c` for a discussion when to use which error code.

<table>
<thead>
<tr>
<th>Error Name</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>H_ERR_NFS</td>
<td>5300</td>
<td>No image acquisition device opened</td>
</tr>
<tr>
<td>H_ERR_FGWC</td>
<td>5301</td>
<td>Wrong color depth</td>
</tr>
<tr>
<td>H_ERR_FGWD</td>
<td>5302</td>
<td>Wrong device</td>
</tr>
<tr>
<td>H_ERR_FGVF</td>
<td>5303</td>
<td>Determination of video format not possible</td>
</tr>
<tr>
<td>H_ERR_FGNV</td>
<td>5304</td>
<td>No video signal</td>
</tr>
<tr>
<td>H_ERR_UFG</td>
<td>5305</td>
<td>Unknown image acquisition device</td>
</tr>
<tr>
<td>H_ERR_FGF</td>
<td>5306</td>
<td>Failed grabbing of an image</td>
</tr>
<tr>
<td>H_ERR_FGWR</td>
<td>5307</td>
<td>Wrong resolution chosen</td>
</tr>
<tr>
<td>H_ERR_FGWP</td>
<td>5308</td>
<td>Wrong image part chosen</td>
</tr>
<tr>
<td>H_ERR_FGWPR</td>
<td>5309</td>
<td>Wrong pixel ratio chosen</td>
</tr>
<tr>
<td>H_ERR_FGWH</td>
<td>5310</td>
<td>Handle not valid</td>
</tr>
<tr>
<td>H_ERR_FGCL</td>
<td>5311</td>
<td>Instance not valid (already closed?)</td>
</tr>
<tr>
<td>H_ERR_FGNI</td>
<td>5312</td>
<td>Image acquisition device could not be initialized</td>
</tr>
<tr>
<td>H_ERR_FGET</td>
<td>5313</td>
<td>External triggering not supported</td>
</tr>
<tr>
<td>H_ERR_FGLI</td>
<td>5314</td>
<td>Wrong camera input line (multiplex)</td>
</tr>
<tr>
<td>H_ERR_FGCS</td>
<td>5315</td>
<td>Wrong color space</td>
</tr>
<tr>
<td>H_ERR_FGPT</td>
<td>5316</td>
<td>Wrong port</td>
</tr>
<tr>
<td>H_ERR_FGCT</td>
<td>5317</td>
<td>Wrong camera type</td>
</tr>
<tr>
<td>H_ERR_FGTM</td>
<td>5318</td>
<td>Maximum number of acquisition device classes exceeded</td>
</tr>
<tr>
<td>H_ERR_FGDV</td>
<td>5319</td>
<td>Device busy</td>
</tr>
<tr>
<td>H_ERR_FGASYNC</td>
<td>5320</td>
<td>Asynchronous grab not supported</td>
</tr>
<tr>
<td>H_ERR_FGPARAM</td>
<td>5321</td>
<td>Unsupported parameter</td>
</tr>
<tr>
<td>Error Code</td>
<td>Error Description</td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td>------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>H_ERR_FGTIMEOUT</td>
<td>5322 Timeout</td>
<td></td>
</tr>
<tr>
<td>H_ERR_FGGAIN</td>
<td>5323 Invalid gain</td>
<td></td>
</tr>
<tr>
<td>H_ERR_FGFIELD</td>
<td>5324 Invalid field</td>
<td></td>
</tr>
<tr>
<td>H_ERR_FGPART</td>
<td>5325 Invalid parameter type</td>
<td></td>
</tr>
<tr>
<td>H_ERR_FGPARV</td>
<td>5326 Invalid parameter value</td>
<td></td>
</tr>
<tr>
<td>H_ERR_FGFNS</td>
<td>5327 Function not supported</td>
<td></td>
</tr>
<tr>
<td>H_ERR_FGIVERS</td>
<td>5328 Incompatible interface version</td>
<td></td>
</tr>
<tr>
<td>H_ERR_FGSETPAR</td>
<td>5329 Could not set parameter value</td>
<td></td>
</tr>
<tr>
<td>H_ERR_FGGETPAR</td>
<td>5330 Could not query parameter setting</td>
<td></td>
</tr>
<tr>
<td>H_ERR_FGPARNA</td>
<td>5331 Parameter not available in current configuration</td>
<td></td>
</tr>
<tr>
<td>H_ERR_FGCLOSE</td>
<td>5332 Device could not be closed properly</td>
<td></td>
</tr>
<tr>
<td>H_ERR_FGCAMFILE</td>
<td>5333 Camera configuration file could not be opened</td>
<td></td>
</tr>
<tr>
<td>H_ERR_FGCALLBACK</td>
<td>5334 Unsupported callback type</td>
<td></td>
</tr>
<tr>
<td>H_ERR_FGDEVLOST</td>
<td>5335 Device lost</td>
<td></td>
</tr>
<tr>
<td>H_ERR_DNA</td>
<td>5104 Device or operator not available</td>
<td></td>
</tr>
<tr>
<td>H_ERR_MEM</td>
<td>6001 Not enough memory available</td>
<td></td>
</tr>
</tbody>
</table>
Index

_ftime(), 58

BoardInfo, 27, 36, 66
close_all_framegrabbers, 13
close_framegrabber, 13, 44
DBFreeImageProc, 26
double buffering, 11

ExtractChannelsFromRGB(), 55

FGClass, 22–24, 30
FGClose(), 13, 44–47
FGGetCallback(), 16, 71
FGGetLut(), 15, 70
FGGetParam(), 15, 68
FGGrab(), 14, 51, 52, 54, 59
FGGrabAsync(), 14, 51, 59
FGGrabData(), 14, 51, 61–64
FGGrabDataAsync(), 15, 51, 63, 64
FGGrabStartAsync(), 14, 59, 60
FGInfo(), 13, 45, 47–50
FGInit(), 29–32
FGInstance, 26, 34
FGOpen(), 12, 34–43
FGOpenRequest(), 12, 33, 34
FGSetCallback(), 16, 70, 71
FGSetLut(), 15, 69
FGSetParam(), 15, 65–67

generic_framegrabber_callback, 15, 71
generic_framegrabber_lut, 15, 70
generic_framegrabber_param, 15, 68
gettimeofday(), 58
grab_data, 14, 61
grab_data_async, 15, 64
grab_image, 14
grab_image_async, 14, 59

generic_framegrabber_callback, 15, 71
generic_framegrabber_lut, 15, 70
generic_framegrabber_param, 15, 68
gettimeofday(), 58
grab_data, 14, 61
grab_data_async, 15, 64
grab_image, 14
grab_image_async, 14, 59

HALCON XL, 19, 73
HAlloc(), 36, 38, 44, 46
HAllocRL(), 63
HAllocRLLocal(), 63
HAllocRLNumLocal(), 62, 63
HAllocRLTmp(), 63
HCKP, 39
Hcpar, 45, 66, 68
HFree(), 44
Himage, 53
HNewImage(), 40, 54
HNewImagePtr(), 53
HReallocRLNumLocal(), 63
Hrlregion, 62

image acquisition

class, 21
handle, 12
instance, 12, 21

image acquisition grabber

class, 12
info_framegrabber, 13, 29, 45

LD_LIBRARY_PATH, 74
Linux, 74

makefile, 73, 74

open_framegrabber, 12, 29, 34, 65, 66
Path, 74

SecFGInit(), 12
set_framegrabber_callback, 15, 70
set_framegrabber_lut, 15, 69
set_framegrabber_param, 15, 65
SetInstParam(), 57

TFGInstance, 27, 32, 66

Windows, 29, 73