HALCON Version 6.0

MVTec Software GmbH

Frame Grabber Integration

Programmer's Manual

This manual describes the integration of user-specific frame grabbers into the HALCON system, Version 6.0

1. Edition	July 1997
2. Edition	November 1997
3. Edition	March 1998
4. Edition	April 1999
5. Edition	November 2000

More information about HALCON can be found at:

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

About This Manual

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

The manual is written for the expert HALCON user who wants to integrate a new frame grabber board. The reader should be familiar with the standard HALCON system. Furthermore, C programming skills[1] are required. Finally, detailed knowledge about the frame grabber API will be necessary.

If you are first interested in the basics of the HALCON frame grabber interface (from the user's point of view) you can also have a look in section 4.3 *The HALCON Frame Grabber Interface* of the **Getting Started with HALCON** User's Manual.

The manual is divided into the following parts:

- **Introduction**
  This chapter explains the basics of image acquisition and introduces the HALCON frame grabber integration interface and the underlying concepts.
- **Data Structures**
  In this chapter, the basic data structures of the frame grabber integration interface are described.
- **Interface Routines**
  This chapter explains all the routines you have to implement inside your frame grabber interface.
- **Generating a Frame Grabber Interface Library**
  This chapter contains information on how to generate a dynamic object encapsulating your frame grabber interface.
- **Appendix A: Changes between versions 1 and 2 of the HALCON frame grabber integration interface**
  This section describes the differences between the versions 1 and 2 of the HALCON frame grabber integration interface.
- **Appendix B: HALCON Error Codes**
  This section describes all error codes which you may use for programming a frame grabber interface.
- **Appendix C: Interface Template CIOFGTemplate.c**
  This section contains a source code template for a frame grabber interface.

[1]Naturally, this also includes knowledge about the programming environment (how to invoke the compiler/linker etc.).

Release Notes

Please note the latest updates of this manual:

- 5^{th} **Edition, HALCON 6.0 (November 2000)**
  The manual has been adapted to the syntactic and semantic changes of the new HALCON frame grabber integration interface version 2. A summary of changes can be found in Appendix A. Besides, the list of currently supported frame grabbers (see Fig. 1.1) has been updated. Furthermore, a small number of syntactic corrections of this manual has taken place.

- 4^{th} **Edition, HALCON 5.2 (March 1999)**
  Some clarifications in the introduction have been made, especially an updated list of currently supported frame grabbers (see Fig. 1.1). The order of the allocated image regions in `FGGrabRegion` is fixed (see new hint in Fig. 3.43). Furthermore, a small number of syntactical corrections have taken place.

- 3^{rd} **Edition, HALCON 5.1 (March 1998)**
  The manual has been revised completely regarding both the structure and the content. The HALCON frame grabber interface was extended by the operators `set_framegrabber_param`, `get_framegrabber_param`, `grab_region`, and `grab_region_async`, which do correspond to the routines `FGSetParam()`, `FGGetParam()`, `FGGrabRegion()`, and `FGGrabRegionAsync()` within the frame grabber interface to be programmed.

Contents

Chapter 1

Introduction

This chapter provides an introduction to the HALCON frame grabber integration 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 frame grabber's internals, we still want to give explanations of the basic terms and methods. Reading the manuals supplied with your frame grabber is a necessity, of course, and possibly gives you a much more detailed view on the things being discussed here.

Unless stated otherwise, all notations refer to Windows NT / 2000 conventions. Thus, for example file paths and environment variables are printed like

```
%HALCONROOT%\examples\fg_integration\CIOFGTemplate.c
```

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

1.1 HALCON's Generic Frame Grabber Interface

HALCON provides a generic frame grabber interface that allows free integration of new frame grabbers 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” boards.

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

[1]*DLL*s for Windows NT / 2000, *shared libraries* for UNIX systems.

- Plug in the hardware and install the vendor-specific device driver, libraries, etc. shipped with the board.
- 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.
- Specify the *name* of the new frame grabber (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 frame grabber 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 `open_framegrabber`. Thus, you can exchange/add frame grabber interfaces even without restarting your application. Special features of different frame grabber boards can be accessed through the general purpose parameter setting mechanism.

Example files

As a guideline for the frame grabber integration, the HALCON distribution contains a template for a frame grabber interface (see `CIOFGTemplate.c` in `examples\fg_integration` and also in Appendix C). 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.

You will also find two specific example interfaces in the HALCON distribution: A very simple one (`MySlicVideo.c`) for the SLIC-Video SBus frame grabber from Multimedia Access Corporation and a fairly complex one (`MyDT3155.c`) for the DT3155 board by Data Translation. These two examples have been implemented in a straightforward way. For the sake of simplicity they do not follow the general design of the template in `CIOFGTemplate.c`. They might be more suitable as starting points for a first prototypical implementation. However, if you are going to design the final HALCON interface, we strongly recommend to use `CIOFGTemplate.c`.

For a list of all the frame grabbers that are currently already supported by HALCON, see Fig.1.1. Please check `http://www.mvtec.com/halcon/` or contact your local distributor to get the latest releases of the HALCON frame grabber interfaces. You can find an up-to-date list of all currently supported frame grabbers (and also a further list of new experimental frame grabber interfaces) at `http://www.mvtec.com/halcon/framegrabber/`.

1.2 Image Acquisition Basics

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,

[2]Except for the new name of the frame grabber and framegrabber-specific parameters used by the operator `set_framegrabber_param`.

Frame Grabber	**Operating System**
BitFlow Raven and RoadRunner	Windows NT / 2000
Cheops Ramses-1	Windows NT / 2000
Data Translation DT3152/DT3153/DT3155	Windows NT / 2000
Eltec PCEye	Windows NT / 2000
IDS FALCON, FALCONplus and EAGLE	Windows NT / 2000
Imagenation PX510/610/610A, PXC200 and PXD	Windows NT / 2000
Integral FlashBus	Windows NT / 2000
Leutron PicPort	Windows NT / 2000, Linux
LinX GINGA	Windows NT / 2000
MATRIX Vision MVdelta, MVsigma, PCimage, MVtitan	Windows NT / 2000
Matrox Meteor-I	Windows NT / 2000, Linux
Mikrotron Inspecta-2	Windows NT / 2000
MRT VideoPort Professional	Windows NT / 2000
Opteon	Windows NT / 2000
The Imaging Source DFG/LC1, DFG/LC2, DFG/BW1, DFG VideoPort	Windows NT / 2000
TWAIN interface	Windows NT / 2000
Unibrain FireBoard 400	Windows NT / 2000
File	Virtual frame grabber interface for accessing image files and sequences Windows NT / 2000: also AVI files

Figure 1.1: Frame grabbers integrated into the HALCON system (November 2000).

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**: 640×480 pixel, 30 frames per second and
- **PAL**: 768×576 pixel, 25 frames per second.

Both formats carry color information, although many frame grabber boards 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[3] for the next vertical sync signal to start grabbing a new image, see Fig. 1.2.

[3]At least if you do not use a setup that supports asynchronous frame resets.

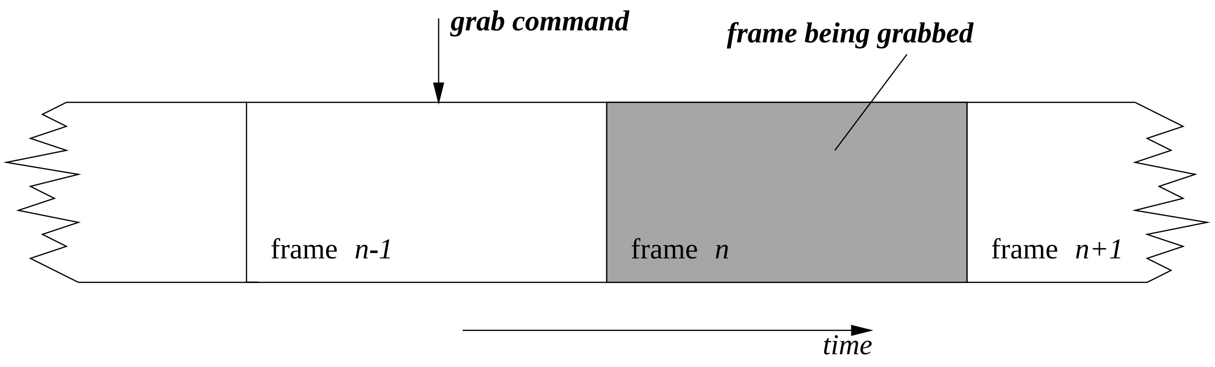

Figure 1.2: Grabbing one frame.

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.

1.3 Synchronous vs. Asynchronous Grabbing

To understand what asynchronous grabbing means, we first should take a look at what the frame grabber 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 frame grabber board
- Device memory on the host machine
- Host memory

Device memory on the board means dedicated memory, physically mounted to the board. This way, the frame grabber 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 frame grabber is delivering data to the host computer's memory *permanently*.[6] Therefore, we usually do not use continuous grabbing modes provided by some frame grabbers, 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 be pagable, the images delivered from the frame grabber in general must be explicitly copied to this memory (since DMA will fail).

[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.

[6]When grabbing a PAL signal with a RGB frame grabber using a 32 bit per pixel representation, more than 42 Megabytes per second have to be transferred to the host.

The host computer's job, as mentioned above, is to trigger the frame grabber when a new image is needed, but it does not necessarily need to *wait* while the board 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[7]. So the "only thing" the host process has to do is to trigger the frame grabber board an average time of $1\frac{1}{2}$ 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 msec with PAL, 33 msec with NTSC video) compared to the small time gap between two adjacent frames in the video stream.

Most frame grabber boards 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.

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

```
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 frame grabber is busy. The only topic left to think about is how the memory used for grabbing should be organized: Assuming step 3 makes the frame grabber deliver data into a dedicated memory area *without* knowledge about what the host is doing in step 4, it is easily seen, that the frame grabber must use a different memory area than the host. If not, the frame

[7]*Direct Memory Access.*

grabber 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 frame grabber 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 frame grabber interface always should create *new* image objects by default and offer volatile grabbing only as an additional option, see section 3.6.

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 a *frame grabber handle*. Inside HALCON such a handle corresponds to a frame grabber *instance*. If you would like to support multiple cameras/boards with your frame grabber interface you have to keep track of all instances corresponding to your frame grabber *class*, see chapters 2 and 3.

1.6 HALCON Frame Grabber Operators

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

1.6.1 open_framegrabber

The operator open_framegrabber is used to create a new frame grabber handle. It loads the specified frame grabber interface and accesses the frame grabber board itself. Moreover, the typical parameters for standard cameras are set (like image size and part, color space, frame grabber port, etc.). If the frame grabber (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 a frame grabber 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 frame grabber instances inside your frame grabber interface if you would like to support multiple cameras or boards.

In detail, this HALCON operator will call your interface routines FGOpenRequest() (see section 3.2) and FGOpen() (see section 3.3). In addition, FGInit() (see section 3.3) will be called when you access a specific frame grabber for the very first time.

1.6.2 close_framegrabber

The operator close_framegrabber is the counterpart to open_framegrabber. It deallocates a frame grabber handle, releases the associated memory, and unlocks the frame grabber board depending on whatever *you* program in the underlying interface routine FGClose() (see section 3.4).

1.6.3 close_all_framegrabbers

The operator close_all_framegrabbers is a convenience operator that calls close_framegrabber for all frame grabber handles in use. This can be very useful, e.g., if you have forgotten to close a frame grabber (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” frame grabbers. However, note that this operator has severe side-effects. It closes *all* frame grabbers, 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 frame grabber interface.

1.6.4 info_framegrabber

The operator info_framegrabber is used to access basic information about a specific frame grabber board (and the corresponding interface). Note, that since many parameter settings depend on the specfic properties of a frame grabber, HALCON can neither provide meaningful defaults nor check parameters automatically.

This operator will call the routine FGInfo() in your frame grabber interface (see section 3.5).

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).

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) in your frame grabber interface. If this routine is missing, the error code `H_ERR_FGASYNC` ("Frame grabber: 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[8] itself is then delivered by the next call to `grab_image` or `grab_image_async` unless it is older than the specified threshold. 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 `FGGrabStartAsync()` (see section 3.9) in your frame grabber interface. If this routine is missing, the error code `H_ERR_FGASYNC` ("Frame grabber: asynchronous grab not supported") will be returned.

1.6.8 `grab_region`

The operator `grab_region` grabs a new image *synchronously*, but does not return the image itself. Instead, a segmentation (that is, image *regions*) based on this image is delivered. The kind of segmentation used is up to *you* (and maybe dependent on some specific hardware features of your frame grabber).

This operator will call your interface routine `FGGrabRegion()` (see section 3.10). If this routine is missing, the error code `H_ERR_FGFNS` ("Frame grabber: function not supported") will be returned.

[8] Alternatively, a segmentation of the image will be returned by `grab_region` or `grab_region_async`.

1.6.9 `grab_region_async`

The operator `grab_region_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_region` it does not return the image itself, but a segmentation (that is, image regions) based on this image. The kind of segmentation used is up to *you* (and maybe dependent on some specific hardware features of your frame grabber).

This operator will call your interface routine `FGGrabRegionAsync()` (see section 3.11). If this routine is missing, the error code `H_ERR_FGFNS` ("Frame grabber: 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 a frame grabber instance. They have been designed to allow the fine-tuning of "exotic" 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`[9] and `set_framegrabber_param` do not evaluate the parameters themselves, but only pass them to *your* interface routines `FGSetParam()` (see section 3.12) and `FGGetParam()` (see section 3.13). Note, that since the name, values, and semantics of such parameters depend on the specific properties of a frame grabber, HALCON can neither provide meaningful defaults nor check parameters automatically. This is all up to your frame grabber interface. If `FGSetParam()` or `FGGetParam()` are missing, the error code `H_ERR_FGPARAM` ("Frame grabber: 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 a frame grabber instance (thus supporting things like gamma correction or white balancing).

These operators will call your interface routines `FGSetLut()` (see section 3.14) or `FGGetLut()` (see section 3.15). If one of these routines is missing, the error code `H_ERR_FGFNS` ("Frame grabber: function not supported") will be returned.

1.7 HALCON Frame Grabber Integration Interface versus HALCON Frame Grabber Interface

The term "HALCON frame grabber interface" refers to an external module encapsulating the frame grabber specific code; this is the one *you* have to provide. In contrast, the "HALCON

[9]Actually this is not completely true. `get_framegrabber_param` automatically returns the current settings for the standard parameters you specify with `open_framegrabber`.

frame grabber integration interface" is the module inside the HALCON library which is responsible for managing and accessing (external) frame grabber interface modules.

Whenever a frame grabber is accessed for the very first time using `open_framegrabber` or `info_framegrabber` the corresponding (external) frame grabber interface, a dynamically loadable module,[10] is loaded. For example, a call like

```
open_framegrabber('PicPort', ... )
```

will cause the HALCON library to load the module `HFGPicPort.dll` in the case of Windows NT / 2000 or `HFGPicPort.so` for UNIX systems, respectively. If you use Parallel HALCON under Windows NT or Windows 2000, it will load the library `parHFGPicPort.dll`. Please note, that in order for this mechanism to work all frame grabber libraries need the prefix `HFG` (and `parHFG` under Windows NT / 2000).

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. Before we take a look at the data structures involved, we should bear in mind that some parts of the frame grabber management take place in the HALCON library and others are up to your frame grabber interface. It is important to keep in mind who is responsible for what:

The HALCON library's job is to:

- Maintain a list of frame grabber *classes*,
- Maintain a list of *instances* for each class, and to
- Decode and preprocess 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,
- Interprete an operator's parameters and map them to the underlying hardware, and to
- Grab images based on the frame grabber's application programming interface (*API*).

1.8 Additional Sources of Information

For further information, please consult one of the following manuals:

- **Getting Started with HALCON**
  An introduction to HALCON in general, including how to install and configure HALCON.
- **HDevelop User's Manual**
  An introduction to the graphical development environment of the HALCON system.

[10]A *DLL* for Windows NT / 2000 or a *shared library* for UNIX systems, respectively.

- **HALCON/C++ User's Manual**
  How to use the HALCON library in your C++ programs.
- **HALCON/C User's Manual**
  How to use the HALCON library in your C programs.
- **HALCON/COM User's Manual**
  How to use the HALCON library in your COM programs, e.g., in Visual Basic.
- **Extension Package Programmer's Manual**
  How to extend the HALCON system with your own operators.
- **HALCON/HDevelop, HALCON/C++, HALCON/C, HALCON/COM**
  The reference manuals for all HALCON operators (versions for HDevelop, C++, C, and COM).

All these manuals are available as PDF documents. The reference manuals are available as HTML documents as well. For the latest version of the manuals please check

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

Please see also the frame grabber interface template and the example files

`CIOFGTemplate.c`, `MyDT3155.c`, and `MySlicVideo.c`

in `%HALCONROOT%\examples\fg_integration`.

Chapter 2

Data Structures

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

2.1 Frame Grabber Classes and Instances

The HALCON frame grabber interfaces manage frame grabbers using *classes* and *instances*. Since HALCON is designed to access any number and combination of boards simultaneously[1], situations may occur, where several boards controlled by the same interface, or boards 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 frame grabber model[2] 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 frame grabber.

Each new **instance** created from this class represents either a specific board belonging to this class or a multiplexed port on such a board. Fig. 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 frame grabber 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.

2.1.1 Structure `FGClass`

The data structure `FGClass` encapsulates the data common to *all* instances of a one frame grabber model, see Fig. 2.2 to 2.4. It is initialized by *your* interface routine `FGInit()`, see sec-

[1] Well, to be honest, there is a limitation: HALCON can handle 32 (`FG_MAX_NUM`) different frame grabbers with up to 32 (`FG_MAX_INST`) instances each *at the same time*.

[2] Or a *family* of frame grabber models like `DT3152`, `DT3153`, and `DT3155` by Data Translation.

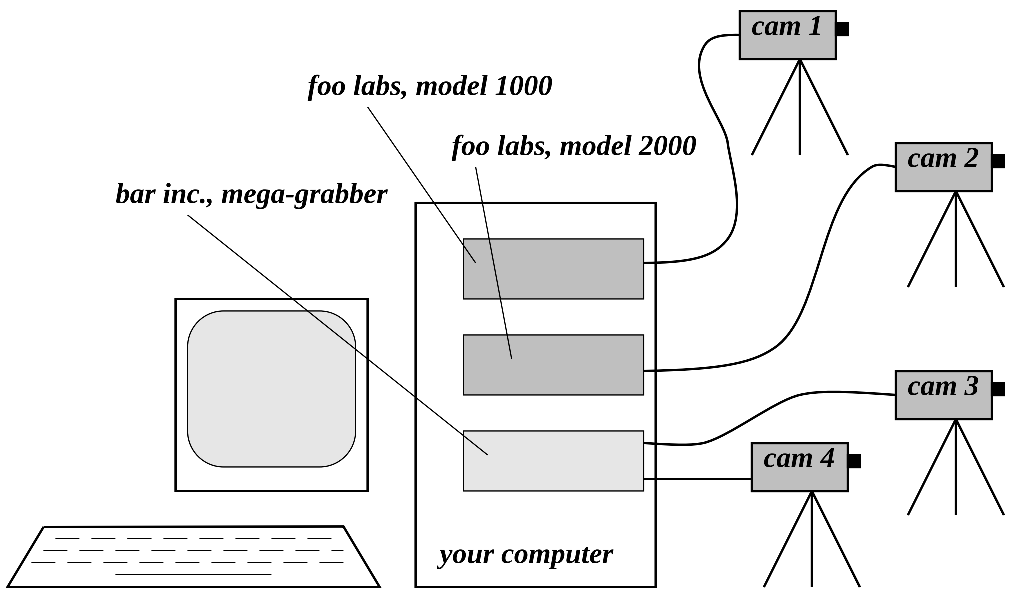

Figure 2.1: A possible configuration with multiple frame grabbers.

tion 3.1.

```
typedef struct _FGClass {
  /* -------------------------- internal -------------------------------- */
  char  name[MAX_STRING];        /* frame grabber name (interface module) */
  void  *lib_handle;             /* handle of interface library           */
  INT   interface_version;       /* current HALCON frame grabber          */
                                 /* 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!)     */
  ...
```

Figure 2.2: The data structure `FGClass` defined in `include\hlib\CIOFrameGrab.h` (to be continued).

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.

We won't 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.

- **Interface-specific functions**
  `Open` etc.: Pointers to the interface routines you provide, see chapter 3 and Fig. 2.5.

```
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 (*GrabRegion)       (Hproc_handle proc_id,FGInstance *fginst,
                              Hrlregion **region,INT *num_region,
                              INT *rlalloc_type);
  Herror (*GrabRegionAsync)  (Hproc_handle proc_id,FGInstance *fginst,
                              double maxDelay,Hrlregion **region,
                              INT *num_region,INT *rlalloc_type);
  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).

- **Default values**
  `image_width`, `image_height`, etc.: The default values for the standard parameters used in `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.

Please see also Fig. 3.47 on page 58.

2.1.2 Structure `FGInstance`

There is a data structure called `FGInstance` (see Fig. 2.6) you will encounter very often when programming an interface since almost every routine you provide (see chapter 3) expects a pointer to the frame grabber instance it should work with.

The structure `FGInstance` contains the actual parameters for a specific frame grabber instance. The corresponding default values for the underlying frame grabber model are stored

```
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;                    /* video-preamp gain                   */
  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              */
  /* ------------------------ miscellaneous --------------------------- */
  void   *reserved[FG_NUM_RESERVED];
} FGClass;
```

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

HALCON operator	**corresponding function pointer**
`open_framegrabber`	`FGClass.OpenRequest`, `FGClass.Open`
`close_framegrabber`	`FGClass.Close`
`grab_image`	`FGClass.Grab`
`grab_image_async`	`FGClass.GrabAsync`
`grab_region`	`FGClass.GrabRegion`
`grab_region_async`	`FGClass.GrabRegionAsync`
`grab_image_start`	`FGClass.GrabStartAsync`
`info_framegrabber`	`FGClass.Info`
`set_framegrabber_lut`	`FGClass.SetLut`
`get_framegrabber_lut`	`FGClass.GetLut`
`set_framegrabber_param`	`FGClass.SetParam`
`get_framegrabber_param`	`FGClass.GetParam`

Figure 2.5: HALCON operators and the corresponding `FGClass` members.

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.

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) holding additional information for an instance, see also `FGOpenRequest()`in section 3.2.

```
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;                    /* video-preamp gain                    */
  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               */
  /* --------------------- 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`.

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 frame grabber types. However, to handle a specific frame grabber 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 for a specific frame grabber board (that is the physical instance of a frame grabber model you plugged into your computer). Fig. 2.7 shows a typical example.

`DeviceId` is an entry in this structure you will need for every frame grabber we know (although the corresponding data *type* will vary). It is used to store a handle returned by the frame grabber API to access a board. If you decide to support multiple boards you might also want to hold a

```
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`.

device name for each board. Moreover, it might be a good idea to share[3] 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 and the example template file `CIOFGTemplate.c` for details.

For every frame grabber board you have installed in your computer you should allocate one structure `BoardInfo`. In general there might be more than one frame grabber instance operating on such a physical board. Thus, we recommend to store instance-specific data in another structure called `TFGInstance` (see below).

2.2.2 Structure `TFGInstance`

We recommend to use a data structure called `TFGInstance` to extend the data provided by `FGInstance` for each frame grabber instance. Fig. 2.8 shows a typical example.

Obviously, it is very convenient to hold references to both the assigned physical board (`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), `timeout` (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 frame grabber buffers (containing the image data) insert[4] into HALCON images, a flag like `volatileMode` is useful. In this case, but also if you encounter frame grabber 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 and `CIOFGTemplate.c` for more details on how to use `TFGInstance`.

It is hard to provide a framework for all possible frame grabbers in this manual. If you would like to develop with an optimal interface you will always have to adapt the example code to the specific API of the frame grabber and to its specific hardware features. In general, this

[3]Since most frame grabbers have only one A/D converter you have to synchronize the grabbing by different instances anyway.

[4]This prevents an additional copy of data. However, as a side-effect old images are overwritten.

```
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       */
                        /* (0 to FG_MAX_INST-1 )                           */
  INT4_8     timeout;   /* useful for async grabbing: timeout threshold   */
                        /* for "images too old"                            */
  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.

will also mean to include additional parameters (in TFGInstance and BoardInfo) to allow the fine-tuning of the hardware.

Chapter 3

Interface Routines

This chapter explains all the routines you have to implement inside your frame grabber interface in order to support the corresponding HALCON frame grabber operators, see section 1.6. The example code of the next sections can also be found in `%HALCONROOOT%\examples\fg_integration\CIOFGTemplate.c`.

3.1 `FGInit()`

`FGInit()` as defined in Fig. 3.1 is called by the HALCON operators `open_framegrabber` or `info_framegrabber` when you access a specific frame grabber for the very first[1] time.

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

Herror FGInit(Hproc_handle proc_id, FGClass *fg)
{
  /* initialize the data structure FGClass and the frame grabber 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 UNIX it is sufficient just to avoid declaring the routine as `static`. In Windows NT / 2000 you have to export the symbol explicitly:

```
extern __declspec(dllexport) Herror FGInit(Hproc_handle proc_id, FGClass *fg);
```

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 frame grabber interface.

The routine `FGInit()` must perform three basic tasks:

[1]Or if you access a frame grabber again after closing all instances.

- Initialize the data structure `FGClass` (see section 2.1.1). 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 frame grabber 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 Fig. 3.2 and 3.3; an example for the latter is shown in Fig. 3.5.

```
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->GrabRegion            = FGGrabRegion;
  fg->GrabRegionAsync       = FGGrabRegionAsync;
  fg->SetParam              = FGSetParam;
  fg->GetParam              = FGGetParam;
  fg->SetLut                = FGSetLut;
  fg->GetLut                = FGGetLut;
  ...
  return(H_MSG_OK);
}
```

Figure 3.2: Example code for `FGInit()`: Initialize `FGClass` (to be continued).

Note, that in Fig. 3.2 the function pointers inside `FGClass` are assigned to the specific routines you provide in the frame grabber interface. Thus, you can choose arbitrary names for the latter. However, we recommend to stick to the names used in this manual to ease understanding different interface code. Some of these function pointers are optional, see Fig. 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.

In Fig. 3.5 we have assumed that you followed our suggestion to provide a data structure `TFGInstance` to hold additional framegrabber-specific information about an instance, see also

```
Herror FGInit(Hproc_handle proc_id, FGClass *fg)
{
  ...
  /* --------------------------- 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;
  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).

Interface routine	Function pointer	Error code
`FGGrabAsync()`	`fg->GrabAsync`	H_ERR_FGASYNC
`FGGrabStartAsync()`	`fg->GrabStartAsync`	H_ERR_FGASYNC
`FGGrabRegion()`	`fg->GrabRegion`	H_ERR_FGFNS
`FGGrabRegionAsync()`	`fg->GrabRegionAsync`	H_ERR_FGFNS
`FGSetParam()`	`fg->SetParam`	H_ERR_FGPARAM
`FGGetParam()`	`fg->GetParam`	H_ERR_FGPARAM
`FGSetLut()`	`fg->SetLut`	H_ERR_FGFNS
`FGGetLut()`	`fg->GetLut`	H_ERR_FGFNS

Figure 3.4: Optional interface routines and the corresponding error codes returned by the HALCON library if the routines are missing.

section 2.2.2. 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 frame grabber integration supporting only *one* instance, you can simplify `FGInit()`: In that case it might be enough to hold all framegrabber-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 Fig. 3.5.

```
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`.

3.2 FGOpenRequest()

The routine FGOpenRequest() as defined in Fig. 3.6 is called by the HALCON operator open_framegrabber prior to calling FGOpen(), see section 3.3. It has to perform only one task:

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

```
#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 a frame grabber handle, this “old instance” is automatically closed using FGClose(). If you return a NULL pointer, HALCON will return the error code H_ERR_FGDV (“Frame grabber: device busy”) as result[2] of open_framegrabber.

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

```
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 (that is to this frame grabber model) in `FGinit()` as suggested in Fig. 3.5. Alternatively, you could also use `fginst->fgclass`, which is also a pointer to the same structure.

Note further, that in Fig. 3.7 the generic pointer `gen_pointer` inside the data structure `FGInstance` (see section 2.1.2) 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 Fig. 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 Fig. 3.9 is called by the HALCON operator open_framegrabber, see section 1.6. It has to perform the following tasks:

- Check all parameters specified in the FGInstance structure fginst.
- Check the availability of the specified frame grabber board and initialize the board according to the parameter settings.
- Allocate buffers if necessary.

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

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

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

Figure 3.9: The prototype for FGOpen().

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 Fig. 3.10. All the examples in this section are aiming at a *complete* frame grabber 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 (e.g., see MySlicVideo.c), but also much less general and flexible.

```
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.

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.

Whenever you specify “default” in this operator, the corresponding default value *you* provided in `FGInit()` (see Fig. 3.3) will be passed in `fginst`.

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 frame grabbers differ too much. Whenever you detect an incorrect request, return one of the error codes listed in appendix B. Note that returning from an arbitrary position inside your interface code might lead to memory leaks or “blocked” frame grabber boards. Therefore, we recommend to do as much parameter checks as possible *before* accessing the physical board or allocating memory. On the other hand, some of these tests must be delayed until the board 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 frame grabber board before returning an error code.

```
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 frame grabber board for the very first time.

If you support multiple instances per physical frame grabber board (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[3] all these parameters in the frame grabber board 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 board within the `BoardInfo` structure (see section 2.2.1) and comparing them to corresponding settings in the `FGInstance` (see section 2.1.2) or `TFGInstance` (see section 2.2.2) structures. Obviously, you only have to reset the parameters that differ.

If you encounter the first instance to be assigned to a physical frame grabber board you have to access the board using the frame grabber API and you might have to allocate the corresponding

[3]In any case you have to set the corresponding input line prior to grabbing.

```
#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()`.

`BoardInfo` structure if you follow our suggestions, see also section 2.2.1. Fig. 3.11 shows some example code dealing with this. The routine `HAlloc()` as defined in Fig. 3.12 is provided by the HALCON extension package interface, see the **Extension Package Programmer's Manual** for details. It is used to allocate memory on the heap.

```
#ifdef WIN32
#define STR_CASE_CMP(S1,S2)   stricmp(S1,S2)
#else
#define STR_CASE_CMP(S1,S2)   strcasecmp(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 frame grabber 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 frame grabber.

Once you have selected and initialized the frame grabber board 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 Fig. 3.13.

```
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 frame grabber.

The next step is to determine the scaling of the image and thus the desired image size to be delivered by the frame grabber. Note, that we only support subsampling by a factor of 2 or 4 in the example code in Fig. 3.14. After this, analyze the specified part of the frame grabber image to be delivered as HALCON image by the grabbing routines, see Fig. 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 frame grabber according to this values. Obviously, how to do this depends on the frame grabber 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. In the following we assume that you use a ring buffer with

```
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.

`MAX_BUFFERS`[4] entries. 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 sections 1.4 and 2.2.2. On the other hand, if several instances are assigned to the same physical frame grabber board 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 Fig. 3.17 to your needs. The allocation itself usually is done using framegrabber-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.**

Fig. 3.17 demonstrates this strategy. Note, that the data structures `BoardInfo` and `TFGInstance` as introduced in the sections 2.2.1 and 2.2.2 have been designed for that purpose. They include all the informations 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** 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.

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 Fig. 3.16 and the discussion of `FGGrab()` in section 3.6 for details.

[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)
{
  ...
  /* 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.

Finally, after successfully initializing the new instance, you should increase both[5] the overall counter for instances `numInstance` and the number of instances for the corresponding physical frame grabber board `currInst->board->refInst`. Again, this is only of importance if you would like to support multiple instances.

[5]Note, that these counters have been suggested in Fig. 3.5 and section 2.2.1. It is up to you whether you follow this recommendation or come up with another solution.

```
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;
    }
    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*/
    }
  }
  ...
}
```

Figure 3.17: Example code for `FGOpen()`: Allocate buffers (to be continued).

```
static Herror FGOpen(Hproc_handle proc_id, FGInstance *fginst)
{
  ...
  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).

3.4 FGClose()

The routine `FGClose()` as defined in Fig. 3.19 is called by the HALCON operator `close_framegrabber`, see section 1.6. It has to perform the following tasks:

- Terminate pending asynchronous grabs.
- Deallocate the buffers.
- "Close" or "unlock" the physical frame grabber board if the instance to be closed is the last one assigned to this board.
- Mark the instance as "free".

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

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

Figure 3.19: The prototype for `FGClose()`.

A pending asynchronous grab is indicated by

```
currInst->busy == TRUE,
```

see also `FGGrabAsync()` in section 3.8. In this case you should use the appropriate frame grabber API call to terminate the grab. Otherwise, you may encounter severe troubles if you deallocate the buffers the frame grabber grabs into. The latter has to be done in accordance with the strategies for buffer allocation in `FGOpen`, see section 3.3. Fig. 3.20 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 frame grabber API.

If the frame grabber instance to be closed is the last one assigned to a specific physical board, you have to "close" or "unlock" the board (using the appropriate API calls) and to deallocate the `BoardInfo` structure (if any), see Fig. 3.21. 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 board (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--;`).

```
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;
      }
    }
  }
  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])
        {
          HCkP( 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 3.20: Example code for `FGClose()`: Deallocate buffers.

```
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()    */
    HCkP(HFree(proc_id,currInst->board));
  }
  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 3.21: Example code for `FGClose()`: Deallocate board.

3.5 FGInfo()

The routine FGInfo() as defined in Fig. 3.22 is called by the HALCON operator info_framegrabber, see section 1.6. It has to perform the following task:

- Return framegrabber-specific informations depending on the specified query.

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

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

Figure 3.22: The prototype for FGInfo().

Currently, the queries listed in Fig. 3.23 should be supported. Thus, a reasonable skeleton for this routine might look like the example in Fig. 3.24. Please see CIOFGTemplate.c for a detailed example.

queryType	Semantics
FG_QUERY_GENERAL	General information (vendor etc.)
FG_QUERY_PORT	Description of the ports (signal, connectors)
FG_QUERY_CAMERA_TYPE	Description of the camera type parameter
FG_QUERY_DEFAULTS	Default values for open_farmegrabber
FG_QUERY_PARAMETERS	Names of the supported non-standard parameters
FG_QUERY_INFO_BOARDS	Information about the installed boards

Figure 3.23: Queries that should be supported by FGInfo().

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 Fig. 3.25 and Fig 3.13. Hcpar is a CameraType data structure for storing *control* parameters (integer, strings, or floating point numbers), see the HALCON **Extension Package Programmer's Manual** 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 nor the array of Hcpar structures have been allocated prior to calling FGInit(). Please use the HALCON extension package interface routine HAlloc()

```
  static Herror FGInfo (Hproc_handle proc_id, INT queryType,
                        char **info, Hcpar **values, INT *numValues)
  {
    switch(queryType)
    {
      case FG_QUERY_GENERAL:
        *info      = "HALCON frame grabber interface template, vendor: MVTec.";
        *values    = NULL;
        *numValues = 0;
        break;
      case FG_QUERY_PORT:
        ...
        break;
      case FG_QUERY_CAMERA_TYPE:
        ...
        break;
      case FG_QUERY_DEFAULTS:
        *info = "Default values (as used for open_framegraber).";
        HCkP( HFgGetDefaults(proc_id,fgClass,values,numValues));
        break;
      case FG_QUERY_PARAMETERS:
        ...
        break;
      case FG_QUERY_INFO_BOARDS:
        *info = "Info about installed boards.";
...
        break;
      default:
        *info      = "Unsupported query!";
        *values    = NULL;
        *numValues = 0;
    }
    return(H_MSG_OK);
  }
```

Figure 3.24: Example code for `FGInfo()`: Parsing the query.

exclusively to allocate the latter, see Fig. 3.25. Otherwise, you will encounter system crashes when `info_framegrabber` deallocates the array using `HFree()`.

Figure 3.26 shows another example. For the query `FG_QUERY_PARAMETERS` `FGInfo()` has to return a list of all names of additional framegrabber-specific parameters supported by your frame grabber, see also `FGSetParam()` in section 3.12 and `FGGetParam()` in section 3.13. In our example we assume that there is only one additional parameter controlling volatile grabbing, see also Fig. 3.48 on page 59 and Fig. 3.49 on page 60.

```
static Herror FGInfo(Hproc_handle proc_id, INT queryType,
                     char **info, Hcpar **values, INT *numValues)
{
  Hcpar *val;
  ...
    case FG_QUERY_CAMERA_TYPE:
      *info = "Video Signal of the camera ('ntsc','pal','auto').";
      HCkP( HAlloc (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 3.25: 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 frame grabber.";
      HCkP( HAlloc (proc_id,(size_t)(1*sizeof(*val)),&val));
      val[0].par.s = FG_PARAM_VOLATILE;
      val[0].type  = STRING_PAR;
      *values      = val;
      *numValues   = 1;
      break;
  ...
  return(H_MSG_OK);
}
```

Figure 3.26: Example code for `FGInfo()`: The query `FG_QUERY_PARAMETERS`.

3.6 FGGrab()

The routine FGGrab() as defined in Fig. 3.27 is called by the HALCON operator grab_image,[6], see section 1.6. It has to perform the following tasks:

- Terminate pending asynchronous grabs.
- (Re-)set the parameters on the frame grabber board.
- Grab an image *synchronously*.
- Return a HALCON image containing the grabbed raw data.

```
#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 3.27: 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_imgage_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, FGGrabRegion() and FGGrabRegionAsync() 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 Fig. 3.28.

```
#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 3.28: The prototype for the auxiliary routine GrabImg().

Based on this auxiliary routine FGGrab() might look as shown in Fig. 3.29. Note, that we only support one channel grayscale and three channels color images of 8 bit scale depth per channel

[6]In the framework suggested in this manual it is also called by FGGrabAsync() see section 3.8.

```
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 3.29: Example code for `FGGrab()`: The basic structure.

in this example. We further assume[7] 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.

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

[7]Actually most frame grabbers we know do not allow to grab three separated channels.

`currInst->InstFrameBuffer[readBuffer]`.

There are two possibilities to allocate a HALCON image of type `Himage`, see the **Extension Package Programmer's Manual** for details on both the data structure and the allocation routines.

`HNewImagePtr()` as defined in Fig. 3.30 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.

```
#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 3.30: The prototype for the extension package interface routine `HNewImagePtr()`.

Inserting the image buffer assigned to the frame grabber 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 Fig. 3.16.

The standard routine to initialize a `Himage` structure is `HNewImage()` as defined in Fig. 3.31. This routine allocates a new image matrix (using `HAlloc`). Thus, either a memcpy or a call to `ExtractChannelsFromRGB()` is necessary in Fig. 3.29 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 Fig. 3.32. Please make sure to restore the old setting before returning from `FGGrab()`. This implies not to use `HCkP` directly as shown in Fig. 3.32.

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**.

Splitting "interleaved" raw color data into three separated image channels is straightforward, see Fig. 3.33. 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.

```
#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 3.31: 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 3.32: Example code for `FGGrab()`: Avoid initialization of the new image matrix in `HNewImage()`.

```
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 3.33: Example code for the auxiliary routine `ExtractChannelsFromRGB()`.

3.7 Auxiliary Routine: `GrabImg()`

We suggest to implement an auxiliary routine `GrabImg()` as defined in Fig. 3.28 on page 41 as basis for the grabbing routines `FGGrab()` and `FGGrabRegion()` and thus also for `FGGrabAsync()` and `FGGrabRegionAsync()`. It has to perform the following tasks:

- 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.

Fig. 3.34 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 frame grabber. Most of Fig. 3.34 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 frame grabber board 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[8] 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 Fig. 3.35. 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 to 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 frame grabber. 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 Fig. 3.17.

[8]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;

  < terminate pending grabs of other instances using the same ADC >;
  if ((!currInst->busy) && (currInst->board->refInst > 1))
    HCkP(SetInstParam(fginst));
  if (currInst->busy)
  {
    if (!fginst->async_grab)
    {
      < cancel the pending asynchronous 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 3.34: 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_region_async` allow

```
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 3.35: The prototype for `SetInstParam()`.

to specify the maximum 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 Fig. 3.36. 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 pending (`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 timeout 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 Fig. 3.34 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->timeout)
          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);
}
```

Figure 3.36: Example code for `GrabImg()`: Check the age of images.

3.8 FGGrabAsync()

The routine FGGrabAsync() as defined in Fig. 3.37 is called by the HALCON operator grab_image_async, see section 1.6. It has to perform the following tasks:

- (Re-)set the parameters on the frame grabber board.
- 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 aynchronous grab (without waiting).
- Return a HALCON image containing the grabbed raw data.

```
#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->timeout  = (INT4_8)(maxDelay+0.5);
  fginst->async_grab = TRUE;

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

  return(H_MSG_OK);
}
```

Figure 3.37: The prototype for FGGrabAsync() based on FGGrab().

Since we have chosen an implementation of FGGrab() in section 3.6 based on GrabImg() in section 3.7, which is more general than necessary for a pure synchronous grabbing, we can easily implement FGGrabAsync() based on FGGrab(), see Fig. 3.37. 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->timeout), see also Fig. 3.36.

3.9 FGGrabStartAsync()

The routine FGGrabStartAsync() as defined in Fig. 3.38 is called by the HALCON operator grab_image_start, see section 1.6. 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 frame grabber board.
- Start an asynchronous grab.

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

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

Figure 3.38: The prototype for FGGrabStartAsync().

The implementation of this routine is rather straightforward, see Fig. 3.39. Please also take a look at the auxiliary routine GrabImg() in section 3.7 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 converter with the current instance. Terminating other instances obviously is a side-effect[9]. Thus, you might consider to return an error code instead (H_ERR_FGDV – "Device busy").

Note further, that the threshold currInst->timeout is used to determine whether an asynchronously grabbed image is too old to be delivered, see also Fig. 3.36.

[9]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_region, or grab_region_async.

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

  currInst->timeout = (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 3.39: Example code for `FGGrabStartAsync()`.

3.10 FGGrabRegion()

The routine `FGGrabRegion()` as defined in Fig. 3.40 is called by the HALCON operator `grab_region`[10], see section 1.6. It has to perform the following tasks:

- Terminate pending asynchronous grabs.
- (Re-)set the parameters on the frame grabber board.
- Grab an image *synchronously*.
- Return a segmentation based on the grabbed raw data.

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

static Herror FGGrabRegion (Hproc_handle proc_id, FGInstance *fginst,
                            Hrlregion **region, INT *num_region,
                            INT *rlalloc_type)
{
  /* grab an image synchronously and segment it */
  return(H_MSG_OK);
}
```

Figure 3.40: The prototype for `FGGrabRegion()`.

A routine like this should be implemented if the frame grabber hardware offers some specific features that support an image segmentation. It might also be more efficient to segment color images within the frame grabber interface even without hardware support, because in this case one can avoid the channel splitting (see Fig. 3.33) and work on the original raw data instead. However, please note, that neither `grab_region` nor `grab_region_async` return the image itself. Thus, the visualization of the segmentation results cannot use the underlying image.

The implemented segmentation is up to *you* (and maybe dependent on some specific hardware features of your frame grabber). We cannot provide example source code for that. However, we will indicate how to allocate image regions encoded in the data structure `Hrlregion`. Please see the HALCON **Extension Package Programmer's Manual** 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 for the segmentation process. We suggest to use `FGSetParam()` for that purpose, see section 3.12.

The HALCON library passes an array of `MAX_OBJ_PER_PAR`[11] pointers to `Hrlregion` in `FGGrabRegion()`. However, the region data itself is *not* allocated automatically. This has to be done within your interface. There are several methods to do so.

[10] In the framework suggested in this manual it is also called by `FGGrabRegionAsync()`, see section 3.11.

[11] In the current version this define is set to 100 000. That should be more than enough to hold all reasonable segmentation results.

```
#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 3.41: The prototype for the extension package interface routine `HAllocRLNumLocal()`.

```
static Herror FGGrabRegion (Hproc_handle proc_id, FGInstance *fginst,
                            Hrlregion **region, INT *num_region,
                            INT *rlalloc_type)
{
  TFGInstance *currInst = (TFGInstance *)fginst->gen_pointer;
  INT         readBuffer;

  HCkP(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)                   */

  HCkP(HAllocRLNumLocal(proc_id,&region[0],
       fginst->image_width*fginst->image_height/2));
  HCkP(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 3.42: Example code for `FGGrabRegion()`: The basic structure.

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 `FGGrabRegion()` and should therefore be preferred. The most flexible memory allocation method is the second one,[12] which is also used in the example in Fig. 3.42. The Extension Package Interface routine `HAllocRLNumLocal()` as defined in Fig. 3.41 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 Fig. 3.42. Note, that you can change this number dynamically using `HReallocRLNumLocal()`.

`rlalloc_type`	Semantics
`FG_RLALLOC_TMP`	Temporary data on stacks allocated with `HAllocRLTmp()` or `HAllocRLNumTmp()` **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!
`FG_RLALLOC_LOCAL`	Temporary data on the heap allocated with `HAllocRLLocal()` or `HAllocRLNumLocal()`
`FG_RLALLOC_PERMANENT`	Permanent data on the heap allocated with `HAllocRL()` or `HAllocRLNum()`

Figure 3.43: Defines for indicating the memory allocation strategy for regions in `FGGrabRegion()`.

Whatever you decide to use, you have to indicate the memory allocation strategy to the HALCON library using one of the defines listed in Fig. 3.43 as return value for the parameter `rlalloc_type`. If you fail to do so, you will encounter program crashes.

[12]Please refer to `HAllocRLLocal()`, `HAllocRLNumLocal()`, and `HReallocRLLocal()` in the **Extension Package Programmer's Manual**.

3.11 FGGrabRegionAsync()

The routine `FGGrabRegionAsync()` as defined in Fig. 3.44 is called by the HALCON operator `grab_region_async`, see section 1.6. It has to perform the following tasks:

- (Re-)set the parameters on the frame grabber board.
- 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 aynchronous grab (without waiting).
- Return a segmentation based on the grabbed raw data.

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

static Herror FGGrabRegionAsync (Hproc_handle proc_id, FGInstance *fginst,
                                 double maxDelay, Hrlregion **region,
                                 INT *num_region, INT *rlalloc_type)
{
  /* grab an image asynchronously and segment it */
  TFGInstance *currInst = (TFGInstance *)fginst->gen_pointer;

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

  HCkP(FGGrabRegion(proc_id, fginst, region, num_region, rlalloc_type));

  return(H_MSG_OK);
}
```

Figure 3.44: The prototype for `FGGrabRegionAsync()` based on `FGGrabRegion()`.

Since we have chosen an implementation of `FGGrabRegion()` in section 3.10 based on `GrabImg()` in section 3.7, which is more general than necessary for a pure synchronous grabbing, we can easily implement `FGGrabRegionAsync()` based on `FGGrabRegion()`, see Fig. 3.44. Please see also `FGGrabAsync()` in Fig. 3.37 which is the corresponding routine to grab *images* instead of regions asynchronously.

3.12 FGSetParam()

The routine FGSetParam() as defined in Fig. 3.45 is called by the HALCON operator set_framegrabber_param, see section 1.6. 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 3.45: The prototype for FGSetParam().

define	name	type
FG_PARAM_HORIZONTAL_RESOLUTION	"horizontal_resolution"	LONG_PAR
FG_PARAM_VERTICAL_RESOLUTION	"vertical_resolution"	LONG_PAR
FG_PARAM_IMAGE_WIDTH	"image_width"	LONG_PAR
FG_PARAM_IMAGE_HEIGHT	"image_height"	LONG_PAR
FG_PARAM_START_ROW	"start_row"	LONG_PAR
FG_PARAM_START_COL	"start_column"	LONG_PAR
FG_PARAM_FIELD	"field"	STRING_PAR
FG_PARAM_BITS_PER_CHANNEL	"bits_per_channel"	LONG_PAR
FG_PARAM_COLOR_SPACE	"color_space"	STRING_PAR
FG_PARAM_GAIN	"gain"	FLOAT_PAR
FG_PARAM_EXT_TRIGGER	"external_trigger"	STRING_PAR
FG_PARAM_CAMERA_TYPE	"camera_type"	STRING_PAR
FG_PARAM_DEVICE	"device"	STRING_PAR
FG_PARAM_PORT	"port"	LONG_PAR
FG_PARAM_LINE_IN	"line_in"	LONG_PAR

Figure 3.46: 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 Fig. 3.46. Please note, that "field" (corresponding to fginst->field) is externally defined as string, but internally as integer using the conversion indicated in Fig. 3.47. Note, further that `"external_trigger"` is externally defined as string (`"true"` or `"false"`) but is internally defined as boolean value of type `HBOOL`.

The parameter value to be set is passed in a structure of type `Hcpar`. Please refer to the **Extension Package Programmer's Manual** for a detailed description of this structure. Some comments have been made on page 38 in this manual as well.

external define	external name	internal define
`FG_FIRST_FIELD_TXT`	`"first"`	`FG_FIRST_FIELD`
`FG_SECOND_FIELD_TXT`	`"second"`	`FG_SECOND_FIELD`
`FG_NEXT_FIELD_TXT`	`"next"`	`FG_NEXT_FIELD`
`FG_FULL_FRAME_TXT`	`"interlaced"`	`FG_FULL_FRAME`
`FG_PROGRESSIVE_FRAME_TXT`	`"progressive"`	`FG_PROGRESSIVE_FRAME`

Figure 3.47: 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 frame grabber. Please do not forget to return these parameter names for the query `FG_QUERY_PARAMETERS` in `FGInfo()`, see Fig. 3.26 on page 40.

The HALCON frame grabber 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. Fig. 3.48 shows example code[13] 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 Fig. 3.17 in section 3.6.

[13]Please refer to `CIOFGTemplate.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 specfic 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);
}
```

Figure 3.48: Example code for `FGSetParam()`: Activate volatile grabbing.

3.13 FGGetParam()

The routine FGGetParam() as defined in Fig. 3.49 is called by the HALCON operator get_framegrabber_param, see section 1.6. 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.

```
#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 3.49: The prototype for FGGetParam() and a simple example.

This routine is the counterpart of FGSetParam(), see section 3.12. 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 Fig. 3.46 on page 57. 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 framegrabber-specific parameters you introduced in FGSetParam().[14] The parameter value has to be returned in a structure of type Hcpar. Please see the **Extension Package Programmer's Manual** for a detailed description of this structure. A short description is given on page 38. 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. Fig. 3.49 shows a simple example assuming that there is only one additional parameter controlling volatile grabbing, see also Fig. 3.48.

[14]Remember that the names of these parameters must be returned for the query FG_QUERY_PARAMETERS by FGInfo(), see Fig. 3.26 on page 40.

3.14 `FGSetLut()`

The routine `FGSetLut()` as defined in Fig. 3.50 is called by the HALCON operator `set_framegrabber_lut`, see section 1.6. It has to perform the following task:

- Set the lookup table for the specified instance.

```
#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 3.50: The prototype for `FGSetLut()`.

A modification of a frame grabber'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 frame grabber (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 Fig. 3.51 is called by the HALCON operator `get_framegrabber_lut`, see section 1.6. It has to perform the following task:

- Return the lookup table for the specified instance.

```
#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 3.51: The prototype for `FGGetLut()`.

This routine is the counterpart to `FGSetLut()` in section 3.14. 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`[15] 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 frame grabber (and its API). Therefore, we cannot provide source code for this task.

[15]In the current version `FG_MAX_LUT_LENGTH` is 4096 corresponding to a maximum resolution of 12 bits per channel.

Chapter 4

Generating a Frame Grabber Interface Library

Whenever a frame grabber is accessed for the very first time by using `open_framegrabber` or `info_framegrabber`, the corresponding HALCON frame grabber 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.

Generating a Frame Grabber Interface Under Windows NT / 2000

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

- the HALCON library `halcon.lib` and
- the frame grabber libraries provided by the frame grabber manufacturer

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 frame grabber interface in Parallel HALCON as well, you must create a second DLL which is linked to the Parallel HALCON library `parhalcon.lib` instead of `halcon.lib`.

In order to be automatically loadable by HALCON or Parallel HALCON, the name of the frame grabber interface library must start with the prefix `HFG` or `parHFG`, respectively. 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 `HFGMegaGrabber.dll` (and `parHFGMegaGrabber.dll`), you access the frame grabber by calling

```
open_framegrabber('MegaGrabber', ... )
```

[1]A *DLL* for Windows NT / 2000 or a *shared library* for UNIX systems, respectively.

Do not forget to export the symbol FGInit with the line

```
extern __declspec(dllexport) Herror FGInit(Hproc_handle proc_id, FGClass *fg);
```

in your interface code, see also section 3.1.

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

Do not copy a frame grabber DLL into the Windows system directories, as it would be loaded twice in this case!

Generating a Frame Grabber Interface Under UNIX

To build a HALCON frame grabber interface you have to generate a *shared library* from the file containing the source code of your interface (like CIOFGTemplate.c) by linking the corresponding object file(s) using ld.

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 frame grabber interface library must start with the prefix HFG. 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 HFGMegaGrabber.so, you access the frame grabber by calling

```
  open_framegrabber('MegaGrabber', ... )
```

Note, that the location of the generated HALCON frame grabber interface must be included in the search path for dynamic objects, i.e., the variable LD_LIBRARY_PATH. The same might be true for any frame grabber library provided by the manufacturer of the frame grabber which is used by your HALCON interface.

In contrast to Windows NT / 2000, both standard HALCON and Parallel HALCON can use one and the same HALCON frame grabber interface library.

Appendix A

Changes between Versions 1 and 2 of the HALCON Frame Grabber Integration Interface

This section summarizes all syntactic and semantic differences between the HALCON frame grabber integration interface version 1 and version 2. Please note, that because of these changes **older frame grabber interfaces won't work together with HALCON 6.0 and vice versa.** This applies to every supported operating system since the library symbols of the integration interface have changed.

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

Version 1	Version 2
`bits`	`bits_per_channel`
`generic`	`camera_type`
`start_line`	`start_row`
`internal_width`	`horizontal_resolution`
`internal_height`	`vertical_resolution`
`width`	`image_width`
`height`	`image_height`
`sel_input`	`line_in`

In addition, the structure `FGClass` does not contain the variables `bw_available`, `color_available`, `gray_available`, `width_max`, `height_max`, `width_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:

	Version 1		Version 2	
	bits	color_space	bits_per_channel	color_space
8 bit gray value image	8	gray	8	gray
10 bit gray value image	10	gray	10	gray
12 bit gray value image	12	gray	12	gray
RGB image, 8 bit per channel	24	rgb	8	rgb
RGB image, 5 bit per channel	16	rgb	5	rgb

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).

To distinguish color spaces, you now set `fginst->color_space` `fginst->num_channels` while evaluating the parameters `fginst->bits_per_channel` and in `FGOpen()` (see section 3.3).

Correspondingly, the names of the following defines have changed:

Version 1	Version 2
FG_QUERY_GENERIC	FG_QUERY_CAMERA_TYPE
FG_PARAM_FGWIDTH	FG_PARAM_HORIZONTAL_RESOLUTION
FG_PARAM_FGHEIGHT	FG_PARAM_VERTICAL_RESOLUTION
FG_PARAM_WIDTH	FG_PARAM_IMAGE_WIDTH
FG_PARAM_HEIGHT	FG_PARAM_IMAGE_HEIGHT
FG_PARAM_BITS	FG_PARAM_BITS_PER_CHANNEL
FG_PARAM_GENERIC	FG_PARAM_CAMERA_TYPE
FG_PARAM_LINE	FG_PARAM_LINE_IN
H_ERR_FGGP	H_ERR_FGCT

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). 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

```
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 the chapters 3.12 and 3.13 for more details.

To fix a bug of the old `CIOFGTemplate.c` delete the line

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

at the end of the functions `FGGrabStartAsync()`, `FGGrabAsync()` and `FGGrabRegionAsync()`.

Appendix B

HALCON Error Codes

In this chapter all HALCON error codes relevant for programming a frame grabber interface are summarized. Please refer to `CIOFGTemplate.c` for a discussion when to use which error code.

Error Name	Code	Description
`H_ERR_NFS`	5300	No frame grabber opened
`H_ERR_FGWC`	5301	Wrong color depth
`H_ERR_FGWD`	5302	Wrong device
`H_ERR_FGVF`	5303	Determination of video format not possible
`H_ERR_FGNV`	5304	No video signal
`H_ERR_UFG`	5305	Unknown frame grabber
`H_ERR_FGF`	5306	Failed grabbing of an image
`H_ERR_FGWR`	5307	Wrong resolution chosen
`H_ERR_FGWP`	5308	Wrong image part chosen
`H_ERR_FGWPR`	5309	Wrong pixel ratio chosen
`H_ERR_FGWH`	5310	Handle not valid
`H_ERR_FGCL`	5311	Instance not valid (already closed?)
`H_ERR_FGNI`	5312	Frame grabber cannot be initialized
`H_ERR_FGET`	5313	External triggering not supported
`H_ERR_FGLI`	5314	Wrong camera input line (multiplex)
`H_ERR_FGCS`	5315	Wrong color space
`H_ERR_FGPT`	5316	Wrong port
`H_ERR_FGCT`	5317	Wrong camera type
`H_ERR_FGTM`	5318	Maximum number of frame grabber classes exceeded
`H_ERR_FGDV`	5319	Device busy
`H_ERR_FGASYNC`	5320	Asynchronous grab not supported
`H_ERR_FGPARAM`	5321	Unsupported parameter
`H_ERR_FGTIMEOUT`	5322	Timeout
`H_ERR_FGGAIN`	5323	Invalid gain
`H_ERR_FGFIELD`	5324	Invalid field

H_ERR_FGPART	5325	Invalid parameter type
H_ERR_FGPARV	5326	Invalid parameter value
H_ERR_FGFNS	5327	Function not supported
H_ERR_FGIVERS	5328	Incompatible interface version
H_ERR_DNA	5104	Device or operator not available
H_ERR_MEM	6001	Not enough memory available

Appendix C

Interface Template CIOFGTemplate.c

This chapter contains a listing of the interface template source code `CIOFGTemplate.c`. Please see also

`%HALCONROOT%\examples\framegrabber\CIOFGTemplate.c`

```
/*****************************************************************************
 * CIOFGTemplate.c
 *****************************************************************************
 *
 * Project:      HALCON
 * Author(s):    Th.Bandlow, Ch.Zierl
 * Description:  General purpose frame grabber interface version 2 template
 *
 * (c) 1996-2000 by MVTec Software GmbH
 *                  www.mvtec.com
 *
 *****************************************************************************
 *
 * See also:  - "Frame Grabber Integration Programmer's Manual"
 *            - "Extension Package Interface Programmer's Manual"
 *
 *****************************************************************************
 *
 * Procedures:
 *
 *        Herror FGInit   (Hproc_handle proc_id, FGClass *fg)
 * static Herror FGOpen   (Hproc_handle proc_id, FGInstance *fg)
 * static Herror FGClose  (Hproc_handle proc_id, FGInstance *fg)
 * static Herror FGGrabStartAsync  (Hproc_handle proc_id, FGInstance *fginst,
 *                                   double maxDelay)
 * static Herror FGGrab            (Hproc_handle proc_id, FGInstance *fginst,
 *                                   Himage *image, INT *num_image)
 * static Herror FGGrabAsync       (Hproc_handle proc_id, FGInstance *fginst,
 *                                   double maxDelay, Himage *image,
 *                                   INT *num_image)
 * static Herror FGGrabRegion      (Hproc_handle proc_id, FGInstance *fginst,
 *                                   Hrlregion **region, INT *num_region,
 *                                   INT *rlalloc_type)
 * static Herror FGGrabRegionAsync (Hproc_handle proc_id, FGInstance *fginst,
 *                                   Hrlregion **region, INT *num_region,
 *                                   INT *rlalloc_type)
 * static Herror FGInfo   (Hproc_handle proc_id, INT queryType, char **info,
 *                         Hcpar **values, INT *numValues)
 * static Herror FGSetLut (Hproc_handle proc_id, FGInstance *fginst,
 *                          INT4_8 *red, INT4_8 *green, INT4_8 *blue,
 *                          INT num)
 * static Herror FGGetLut (Hproc_handle proc_id, FGInstance *fginst,
 *                          INT4_8 *red, INT4_8 *green, INT4_8 *blue,
 *                          INT *num)
 * 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)
 * static FGInstance** FGOpenRequest (Hproc_handle proc_id,
 *                                    FGInstance *fginst)
 *
 ***************************************************************************/

/***************************************************************************/
/***   TODO: adapt INTERFACE_REVISION appropriately                     ***/
/***************************************************************************/

#define INTERFACE_REVISION "2.x"

/***************************************************************************/
/***    TODO: If you provide software for different architectures       ***/
/***          you might have to encapsulate code sections like this ... ***/
/***************************************************************************/

#ifdef WIN32
#include <sys/timeb.h>
#else
#include <time.h>
#endif


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

/***************************************************************************/
/***    TODO: Place your vendor-specific #include's here...             ***/
/***************************************************************************/

/* e.g.
#include "file1.h"
#include "file2.h"
#include "file3.h"
*/

/***************************************************************************/
/***       TODO: place your vendor-specific #define's here...           ***/
/***************************************************************************/

/* e.g.
#define MY_BUFFER_SIZE 0xffff
*/

/* These defines be a good idea, if you would like to support             */
/* subsampling...                                                          */
#define FG_FULL_RESOLUTION    0
#define FG_HALF_RESOLUTION    1
#define FG_QUARTER_RESOLUTION 2
#define FG_OTHER_RESOLUTION   3

/* You might want to define additional parameters to be handled by        */
/* FGSetParam(), FGGetParam()                                              */
#define FG_PARAM_VOLATILE     "volatile"
#define FG_PARAM_REVISION     "revision"
#define FG_PARAM_NUM 2

/* Typically, you will need two buffers to grab to (alternatively)        */
/* so let's set MAX_BUFFERS to 2 for the moment...                         */
#define MAX_BUFFERS 2

/* Also quite convenient ... */
#define FG_PAL          0
#define FG_NTSC         1
#define FG_SPECIAL_NORM 2

/* Use this Macro to display error messages                                */
#define MY_PRINT_ERROR_MESSAGE(ERR) { \
  if (HDoLowError) IOPrintErrorMessage(ERR); }
```

```
#ifdef WIN32
#define STR_CASE_CMP(S1,S2)   stricmp(S1,S2)
#else
#define STR_CASE_CMP(S1,S2)   strcasecmp(S1,S2)
#endif

/*************************************************************************/
/***       DON'T TOUCH THE NEXT LINE !!!                              ***/
/*************************************************************************/
/* Make the Procedure 'FGInit' visible outside the DLL...               */
/*                                                                       */
/* NOTE: If you're using C++ you have to use                            */
/*               extern "C" __declspec(dllexport)                        */
/* instead of                                                            */
/*               extern __declspec(dllexport)                            */
/* in the following declaration.                                         */
/*************************************************************************/
#ifdef WIN32
extern __declspec(dllexport) Herror FGInit(Hproc_handle proc_id, FGClass *fg);
#endif


/*************************************************************************/
/***    TODO: Adapt the following structs to match your frame grabber's ***/
/***          hardware features.                                      ***/
/*************************************************************************/

/* The 'BoardInfo' struct:                                               */
/* is used one per physical board. It contains hardware features tightly  */
/* coupled with the board, e.g., a board handle, memory mapping, ...     */

/* The 'TFGInstance' struct:                                             */
/* is the internal representation of a 'frame grabber handle', thus it    */
/* contains board-related data for one instance; for example, different   */
/* instances may represent different multiplexed inputs on one single     */
/* board or, on the other hand, maybe different physical boards with only */
/* one physical input each.                                              */

typedef struct
{
  /* Note: The following struct members will only be used once per board  */

  /***********************************************************************/
  /***        TODO: place your FG-specific entries here...            ***/
  /***********************************************************************/

  /*      examples: */
  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                 */
  /*      more examples: */
  HBOOL   doesPhysicalSubsampling;
  INT     maxBitsPerChannel;

} BoardInfo;

typedef struct
{
  /* Note: The following struct members will be used once per instance    */
  /* (that is more than one per board, e.g., if you use several cameras   */
  /* per frame grabber)                                                    */

  /***********************************************************************/
  /***          TODO: Place your FG-specific data here...             ***/
  /***********************************************************************/
  /*  examples:  */

  BoardInfo  *board;    /* the 'physical' board this instance is          */
                        /* attached to                                    */
  HBOOL      busy;      /* useful, if you plan to support asynchr.        */
                        /* grabbing (is the last grab still running?)     */
  INT        instance;  /* a useful backreference to the general HALCON   */
                        /* instance information: The instance index       */
                        /* (0 to FG_MAX_INST-1 )                          */
  INT4_8     timeout;   /* useful for async grabbing: timeout threshold   */
                        /* for "images too old"                           */
  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;


/*************************************************************************/
/***       We recommend that you leave these untouched:               ***/
/*************************************************************************/

static FGClass   *fgClass; /* a pointer to the frame grabber class struct */

static TFGInstance FGInst[FG_MAX_INST];  /* all possible instances        */
static INT numInstance = 0;              /* # current instances           */


/* Some useful general-purpose buffers (e.g., for status messages):       */
static char errMsg[512];




/* ====================================================================
 *
 *                  HBOOL KillAllOtherJobs (...)
 *
 * ====================================================================
 *
 * Terminates jobs (pending asynchronous grabs) started on the same
 * physical board (another board is no problem, since that usually has
 * its own memory).
 * Note that if the board has more than one A/D converter the other jobs
 * in general won't have to be killed.
 *
 * This routine might be useful for FGOpen(); please see the FGopen()
 * routine prior to studying this one here ...
 *
 * ================================================================= */

static HBOOL KillAllOtherJobs(FGInstance *fginst)
{
  TFGInstance *currInst = (TFGInstance *)fginst->gen_pointer;
  INT          i;
  HBOOL        killedSomebody=FALSE;

  if (numInstance > 1)
  {
    for (i=0; i < FG_MAX_INST; i++)
    {
      if (FGInst[i].board && (currInst->instance != i))
      {
        /* ok, FGInst[i] is in use and is NOT the current Instance.     */
        /* Now let's see, if it references the same board as the        */
        /* current instance ...                                          */
        if (FGInst[i].board == currInst->board && FGInst[i].busy)
        {
          /* There might be a problem: Another asynchronous grab using  */
          /* the same frame grabber board ...                            */

          /**********************************************************/
          /***  TODO: terminate this async job                    ***/
          /**********************************************************/

          killedSomebody = TRUE;
          FGInst[i].busy = FALSE;
        }
      }
    }
  }
  return(killedSomebody);
} /* KillAllOtherJobs */



/* ======================================================================
 *
 *                           CleanupFGOpen (...)
 *
 * ======================================================================
 *
 * Auxiliary routine for FGOpen (...)
 *
 * ======================================================================
 */

/* If you have to exit FGOpen() in case of an error you might have to do */
/* some cleaning up before returning the error code ...                  */
/* Note that in case of newBoardalloc == TRUE (that is, you have         */
/* initialized the frame grabber board for the first time), you might    */
/* also have to "close" or "unlock" the frame grabber again ...          */

static Herror CleanupFGOpen(Hproc_handle proc_id, TFGInstance *currInst,
                            HBOOL newBoardalloc, Herror err)
{
  INT j;

  if ((!currInst->allocBuffer) && (!currInst->board->refBuffer))
  {
    for (j=0; j < MAX_BUFFERS; j++)
    {
      if (currInst->board->BoardFrameBuffer[j])
      {
        (void)HFree(proc_id,currInst->board->BoardFrameBuffer[j]);
        currInst->board->BoardFrameBuffer[j] = NULL;
      }
    }
  }
  if (currInst->allocBuffer)
  {
    INT j;
    for (j=0; j < MAX_BUFFERS; j++)
    {
      if (currInst->InstFrameBuffer[j])
      {
        (void)HFree(proc_id,currInst->InstFrameBuffer[j]);
        currInst->InstFrameBuffer[j] = NULL;
      }
    }
  }
  if (newBoardalloc)
    HFree(proc_id,currInst->board);
  currInst->board = NULL;

  return(err);
```

```
} /* CleanupFGOpen */



/* ======================================================================
 *
 *                        Herror FGOpen (...)
 *
 * ======================================================================
 *
 * Initialize a new frame grabber instance via open_framegrabber (...)
 *
 * ======================================================================
 */

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

  /* other local variables ... */
  INT     i;
  INT     norm;
  INT     widthScale;
  INT     heightScale;
  HBOOL   newBoardalloc = FALSE;
  Herror  err;
  INT4_8  sizeBuffer;

  /***********************************************************************/
  /***    TODO: Set defaults                                          ***/
  /***********************************************************************/

  fginst->async_grab     = FALSE;

  currInst->busy         = FALSE;
  currInst->allocBuffer  = FALSE;
  currInst->currBuffer   = 0;
  currInst->volatileMode = FALSE;

  /***********************************************************************/
  /***    TODO: Place initialization code here; the complexity of this    ***/
  /***          task depends on the features you want to support; e.g.,   ***/
  /***          if you allow only one board with only one input line this ***/
  /***          is quite simple; if, however, you want to support multiple***/
  /***          boards with different features using one API with every   ***/
  /***          board having many ports and multiplexed input lines,      ***/
  /***          things could get a bit tricky...                          ***/
  /***                                                                    ***/
  /***    NOTE: The following example fragments assume that you want      ***/
  /***          it the hard way (multiple boards, multiple ports,         ***/
  /*            multiple input lines, different board types)...          ***/
  /***********************************************************************/

  if (currInst->busy) /* kill my own job */
  {
    /* This should not happen (but who knows ...)                         */
    /*********************************************************************/
    /***  TODO: terminate 'my own' async job (the one belonging to this   ***/
    /***  instance)                                                     ***/
    /*********************************************************************/
  }

  /***********************************************************************/
  /***    TODO: Select frame grabber board                            ***/
  /***********************************************************************/

  /* The desired frame grabber board is specified by fginst->device.        */
  /* If "default" is used in open_framegrabber() the string YOU provided    */
  /* for fg->device in FGInit() will be passed in fginst->device. If the    */
  /* default value depends on the CURRENT configuration of the system,      */
  /* specify "default" in FGInit(), get the corresponding information NOW,  */
  /* and overwrite the device name in fginst->device.                       */
  /* In case of foolish inputs: return H_ERR_FGWD -- wrong device           */
  /* Note: In many cases there is a call in the frame grabber API to ASK for*/
/* all available frame grabbers in the system.                        */

if (!strcmp(fginst->device,"default"))
  strcpy(fginst->device, "1");         /* example: default device "1" */
else if (strcmp(fginst->device,"1") && strcmp(fginst->device,"2"))
  return(H_ERR_FGWD);                  /* example: only "1" or "2"  */

/************************************************************************/
/* NOTE: Some of the following parameter checks might be possible without */
/* accessing the frame grabber hardware; in other cases it might be      */
/* necessary to initialize the hardware and ASK the specific frame grabber*/
/* about its abilities. We suggest to make parameter tests as soon as     */
/* possible and as late as necessary ...                                  */
/************************************************************************/

/************************************************************************/
/***    TODO: Check the desired port / multiplexed input line          ***/
/************************************************************************/

/* The desired physical port into which your camera is plugged is        */
/* typically passed in fginst->port; if there is a multiplexer available  */
/* at this port, the desired input line is passed in fginst->line_in      */
/* If -1 (for "default") is used in open_framegrabber() the values YOU    */
/* provided for fg->port and fg->line_in in FGInit() will be passed in    */
/* fginst->port and fginst->line_in.                                      */

/* You have to check the values of both parameters and return appropriate */
/* error codes in case of a failure.                                      */
/* If the desired port is invalid return H_ERR_FGPT -- wrong port;        */
/* If the desired input line is invalid return H_ERR_FGLI -- wrong line in*/


if ((fginst->port < 1) || (fginst->port > 3))
  return(H_ERR_FGPT);  /* example: available ports: [1,2,3] */

if (fginst->line_in != 1)
  return(H_ERR_FGLI);  /* example: no MUX */

/************************************************************************/
/***    TODO: Check number of bits per channel                         ***/
/************************************************************************/

/* The desired number of bits per image channel is passed in             */
/* fginst->bits_per_channel (eg. use 8 bits for 8-8-8 rgb-image or 8 bit  */
/* grayscale); If -1 (for "default") is used in open_framegrabber() the   */
/* values YOU provided for fg->bits_per_channel in FGInit() will be passed*/
/* in fginst->bits_per_channel.                                           */
/* If the default value depends on the CURRENT configuration of the       */
/* system, specify -1 in FGInit(), get the corresponding information NOW, */
/* and overwrite the default value in fginst->bits_per_channel.           */
/* If you encounter unreasonable requests, return H_ERR_FGWC -- wrong     */
/* color depth                                                            */

if ((fginst->bits_per_channel != 8) && (fginst->bits_per_channel != 12))
  /* example: Allow only 8 and 12 bits per image channel */
  return (H_ERR_FGWC);

/************************************************************************/
/***    TODO: Check the desired color space                            ***/
/************************************************************************/

/* The desired color space (eg. 'gray' or 'rgb') is passed  in           */
/* fginst->color_space. What we have to do is to test for each valid color*/
/* space the fginst->bits_per_channel parameter on valid value and set    */
/* fginst->num_channels on the number of image channels of the resulting  */
/* HALCON image.                                                          */
/*   NOTE: within this concept we can't distinguish between rgb images of */
/* 24 and 32 bits per pixel if both modes should be offered. This is      */
/* because both modes result in a 8-8-8 rgb HALCON image. In this case    */
/* we recommend to prefer the 24 bit mode as it implicates a lower load on*/
/* the PCI-bus. If you have to provide 32 bits per pixel, set             */
/* fginst->num_channels = 4. If you want to offer both modes anyhow, we   */
/* recommend to initiate a separate color space "xrgb" for the 32 bit mode*/
/* and use the "rgb" value to denote the 24 bit mode.                     */
/*   If "default" is used in open_framegrabber() the string YOU provided  */
/* for fg->color_space in FGInit() will be passed. If the default value   */
/* depends on the CURRENT configuration of the system, specify "default"  */
/* in FGInit(), get the corresponding information NOW, and overwrite the  */
/* default value in fginst->color_space.                                  */
/* In case of unreasonable requests: Return H_ERR_FGCS -- invalid color   */
/* space                                                                  */
if (!STR_CASE_CMP(fginst->color_space,"default"))
  strcpy(fginst->color_space, "rgb");          /* example: default "rgb" */
else if (STR_CASE_CMP(fginst->color_space,"gray"))
{
  if ((fginst->bits_per_channel != 8) ||
      (fginst->bits_per_channel != 16))
    /*example: allow only 8 or 16 bit grayscale images */
    return (H_ERR_FGWC);
  fginst->num_channels = 1; /* grayscale means a one channel HALCON image*/
}
else if (STR_CASE_CMP(fginst->color_space,"rgb"))
{
  if (fginst->bits_per_channel != 5 || fginst->bits_per_channel != 8)
    /*example: allow only 5-6-5 and  8-8-8 rgb images */
    return (H_ERR_FGWC);
  fginst->num_channels = 3; /* rgb means a three channel HALCON image */
}
else if (STR_CASE_CMP(fginst->color_space,"xrgb"))
{/*example:  32 bits per pixel rgb image*/
  if (fginst->bits_per_channel != 8)
    return (H_ERR_FGWC);
  fginst->num_channels = 4; /* denote the 32 bit color mode by four   */
  /* channels; the mode results also in a three channel, 8 bit HALCON */
  /* image.                                                           */
}

else
  return (H_ERR_FGCS);


/************************************************************************/
/***    TODO: Check the desired video gain                           ***/
/************************************************************************/

/* The desired video gain is passed in fginst->gain. Just ignore this     */
/* parameter if your board does not support any gain setting.             */
/* If -1.0 (for "default") is used in open_framegrabber() the values YOU  */
/* provided for fg->gain in FGInit() will be passed in fginst->gain.      */
/* If you encounter unreasonable requests, return H_ERR_FGGAIN -- invalid */
/* video gain                                                             */

if (fginst->gain < 0.0)
  return(H_ERR_FGGAIN);   /* example: gain must be positive */

/************************************************************************/
/***    TODO: Check for external triggering                          ***/
/************************************************************************/

/* If external triggering is desired, fginst->external_trigger is set to  */
/* TRUE, otherwise to FALSE.                                              */
/* If "default" is used in open_framegrabber() the value YOU provided     */
/* for fg->external_trigger in FGInit() will be passed.                   */
/* If the frame grabber does not support external triggering, return      */
/* H_ERR_FGET -- external triggering not supported.                       */

if (fginst->external_trigger)
  return(H_ERR_FGET);        /* example: Do not allow external triggering */

/************************************************************************/
/***    TODO: Check the desired field                                ***/
/************************************************************************/

/* The desired field to be grabbed is specified in fginst->field:         */
/* FG_FIRST_FIELD       -- grab first (even) field                        */
/* FG_SECOND_FIELD      -- grab second (odd) field                        */
/* FG_NEXT_FIELD        -- grab arbitrary next field                      */
/* FG_FULL_FRAME        -- grab a full frame (interlaced)                 */
/* FG_PROGRESSIVE_FRAME -- grab a full frame (progressive scan)          */
/* If "default" is used in open_framegrabber() the value YOU provided     */
/* for fg->field in FGInit() will be passed.                              */
/* In case of unreasonable values, return H_ERR_FGFIELD --  invalid field */

/* Note: This parameter is tightly coupled with the specified image size  */
/* So, many HALCON frame grabber interfaces IGNORE this parameter and     */
/* decide what to do exclusively by the size parameters ...               */

if ((fginst->field != FG_FIRST_FIELD) &&
    (fginst->field != FG_FULL_FRAME))
  return(H_ERR_FGFIELD);     /* example: first field or interleaved frame */

/*************************************************************************/
/***    TODO: Check availability                                       ***/
/*************************************************************************/

for (i=0; i < FG_MAX_INST; i++)
{
  if (FGInst[i].board && (currInst->instance != i))
  {
    /* ok, FGInst[i] is in use and is NOT the current Instance,           */
    /* now let's see, if the associated board is the requested one...     */

    if (!strcmp(FGInst[i].board->DeviceName, fginst->device))
    {
      /* The selected board is already in use! Decide, whether you        */
      /* can allow parallel usage by this new instance (for example       */
      /* using a different port or line in ...)                           */

      /* Note, that you can access the other HALCON FGInstance structs    */
      /* using FGInst[i].fginst if you follow our suggestions, or you     */
      /* provide additional information concerning port/MUX settings in   */
      /* the struct TFGInstance, or you ask the frame grabber (API) ...   */

      /* If you detect an incompatibility:                                */
      /* return H_ERR_FGDV -- device busy                                 */
      /* otherwise: currInst->board = FGInst[i].board;                    */

      if (1)
        return(H_ERR_FGDV);    /* example: Only one instance per board    */
      else
        currInst->board = FGInst[i].board;
    }
  }
}

/*************************************************************************/
/****    NOTE: From this time on you should reset currInst->board to    ***/
/****          NULL before exiting in case of an error (in order to     ***/
/***           unlock the instance again)                               ***/
/*************************************************************************/

/* Note: There might be asynchronous grabs pending on the same     */
/* board you would like to use. In general, most frame grabbers    */
/* have only one A/D converter - thus, you might have to cancel    */
/* all these old jobs before doing anything else.                  */
/* Obviously, this operation has nasty side-effects! So you could  */
/* also return the error H_ERR_FGDV -- device busy in that case    */

KillAllOtherJobs(fginst);

/*************************************************************************/
/***    TODO: Allocate BoardInfo / initialize frame grabber hardware    ***/
/*************************************************************************/

if (!currInst->board)
{
  /* All right, it seems that the desired board isn't yet open...       */
  /* allocate space for the BoardInfo struct member, e.g., like this:   */
  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 this device (fginst->device) for the 1st time ...       */
  /* PLEASE REFER TO THE API MANUAL OF YOUR FRAME GRABBER FOR DETAILS */
  /* query frame grabber capabilities (and store them in the struct   */
  /* currInst->board ...)                                         */
  /*****************************************************************/

  newBoardalloc = TRUE;
}

/* Note: In some cases parameter checks are not possible until now, */
/* that is until the specific board has been initialized and can be */
/* ASKED about its abilities (this is typically true for evaluating */
/* the image size and the camera_type parameter)                    */

/**********************************************************************/
/****    NOTE: From this time on you might have to do some cleaning up  ***/
/****          before exiting in case of an error (deallocate the      ***/
/***           BoardInfo, unlock the frame grabber etc.);              ***/
/***           see CleanupFGOpen() as a reference                      ***/
/**********************************************************************/

/* Ok, at this point we know that the frame grabber board IS available    */
/* and initialized.                                                       */
/* Note that some parameters like number of bits_per_channel, field to be */
/* grabbed etc. typically only influence the GRABBING, that is FGGrab(),  */
/* FGGrabStartAsync(), and FGGrabAsync().                                 */
/* Others have to be SET right now ...                                    */

/**********************************************************************/
/***    TODO: Set port / input line                                 ***/
/**********************************************************************/

/* we did the checks already - now we set these values on the board ...   */

/**********************************************************************/
/***    TODO: Set video gain                                        ***/
/**********************************************************************/

/* We did the check already - now we set this value on the board ...      */

/**********************************************************************/
/***    TODO: Set external triggering                               ***/
/**********************************************************************/

if (fginst->external_trigger)
{
  /* Well, whatever the frame grabber API requests you to do ...         */
  ;
}

/**********************************************************************/
/***    TODO: Evaluate the camera_type parameter                    ***/
/**********************************************************************/

/* HALCON provides one camera_type string parameter in open_framegrabber()*/
/* passed in fginst->camera_type.                                         */
/* Use this parameter to specify some frequently used additional settings */
/* that are not available among the standard parameters. Note that you    */
/* can provide arbitrary additional parameters to be set via              */
/* set_framegrabber_param() AFTER open_framegrabber(), see FGSetParam().  */
/* If "default" is used in open_framegrabber() the value YOU provided     */
/* for fg->camera_type in FGInit() will be passed.                        */

/* A typical application for the camera_type parameter is to specify the  */
/* video norm ("ntsc", "pal", "auto") ...                                 */
/* Another possibility is to specify a camera configuration file which    */
/* many frame grabbers use for configuration.                             */
/* It's up to YOU to decide what semantics to assign to this parameter!   */
/* If you encounter an unreasonable value, return H_ERR_FGCT              */

/**********************************************************************/
```

```
/***    TODO: Determine the video norm (pal,  ntsc, ...)              ***/
/************************************************************************/

/* This might be done by analyzing the video signal or by evaluating the  */
/* camera_type parameter (see above). This might look like this:          */
if (!STR_CASE_CMP(fginst->camera_type, "auto"))
{
  /* use special routines provided by your frame grabber to analyze the   */
  /* the video signal ...                                                  */
}
else if (!STR_CASE_CMP(fginst->camera_type, "ntsc"))
  norm = FG_NTSC;
else if (!STR_CASE_CMP(fginst->camera_type, "pal"))
  norm = FG_PAL;
else
  norm = FG_SPECIAL_NORM;

/************************************************************************/
/***    TODO: Set fginst->width_max / fginst->height_max              ***/
/************************************************************************/

/* Depending on the video norm, set the maximum allowed image size in     */
/* fginst:                                                                 */

switch (norm)
{
  case FG_PAL:
    fginst->width_max  = 768;
    fginst->height_max = 576;
    break;
  case FG_NTSC:
    fginst->width_max  = 640;
    fginst->height_max = 480;
    break;
  case FG_SPECIAL_NORM:
  default:
    /* well, whatever! */
    fginst->width_max  = 768;
    fginst->height_max = 576;
}

/************************************************************************/
/***    TODO: Evaluate the image size / part                          ***/
/************************************************************************/

/* This is going to be tricky!                                            */
/* There are six parameters to be evaluated:                              */
/* fginst->horizontal_resolution/vertical_resolution specifying the image */
/* size/subsampling for the frame grabber, and fginst->start_row/start_col*/
/* + fginst->image_width/image_height specifying a part of this image to  */
/* be delivered in the HALCON image. In many cases, frame grabbers can be */
/* used for NTSC and PAL signals. Thus, reasonable values for these       */
/* parameters depend on the camera signal (see fginst->width_max/         */
/* height_max). This is inconvenient for the user. Therefore, we          */
/* recommend not only to support the "absolute" values, but also the      */
/* following relative values:                                              */
/*                                                                         */
/* horizontal_resolution/vertical_resolution = 1,2,4: Full/half/quarter   */
/*                                                    resolution          */
/* image_width,image_height                  = 0: Return "full" image     */
/* Note that due to this convention, you cannot grab images of size       */
/* 1x1, 2x2, or 4x4 pixels (we are certain that this loss is              */
/* acceptable ...)                                                         */

/* The following code segments assumes that you already know what video   */
/* norm to use (that is fginst->width_max/height_max is set properly)     */
/* Now, some arithmetic: Check & set the required cropping / subsampling  */
/* values. The purpose of this is to deliver a unified internal           */
/* representation of the requested subsampling values: Regardless whether */
/* the user selects '320 and 240' or '2 and 2' for a 'half size' NTSC     */
/* image, the internal representation should always be '2' and '2'        */
/* (half height and half width)                                            */
```

```
widthScale  = fginst->horizontal_resolution;
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;

/* Subsampling has to be either 1, 2 or 4, but may be different for the   */
/* x- and y-axes.                                                          */
/* Note: This is the standard behaviour of a HALCON frame grabber          */
/* interface: Allow only full size and subsampling by a factor of 2 or     */
/* 4. Of course, if you feel like supporting some arbitrary weird image    */
/* zooming/scaling also: Go ahead ...                                      */

if (!(widthScale == 1 || widthScale == 2 || widthScale == 4))
  /* wrong resolution */
  return(CleanupFGOpen(proc_id,currInst,newBoardalloc,H_ERR_FGWR));

if (!(heightScale == 1 || heightScale == 2 || heightScale == 4))
  /* wrong resolution */
  return(CleanupFGOpen(proc_id,currInst,newBoardalloc,H_ERR_FGWR));

/* Now that we have a proper representation of the desired values, let's  */
/* compute the effective image size                                        */

fginst->horizontal_resolution = fginst->width_max  / widthScale;
fginst->vertical_resolution   = fginst->height_max / heightScale;

/* It might be useful to store the current subsampling mode, e.g., in     */
/* fginst->mode                                                            */

if ((widthScale  == 1) && (heightScale  == 1))
  fginst->mode = FG_FULL_RESOLUTION;
else if ((widthScale  == 2) && (heightScale  == 2))
  fginst->mode = FG_HALF_RESOLUTION;
else if ((widthScale  == 4) && (heightScale  == 4))
  fginst->mode = FG_QUARTER_RESOLUTION;
else
  fginst->mode = FG_OTHER_RESOLUTION;

/* the subsampling is (hopefully) handled properly now; let's analyze the */
/* image part...                                                           */

/* "full" centred image part: */
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;

/* Now let's check the part -- if not reasonable: Return H_ERR_FGWP --    */
/* wrong image part                                                        */

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));

/************************************************************************/
/***    TODO: Set the image size / part                              ***/
/************************************************************************/

/* Now you have to set the frame grabber scaler etc. in order to deliver  */
/* the image part specified by start_col/start_row and image_width/       */
/* image_height within the image of size horizontal_resolution x          */
/* vertical_resolution. This might also mean to switch between full images*/
/* and single fields (there is no use to grab a full frame if you would   */
/* like to do a subsampling of factor 2 anyway ...), see also           */
/* fginst->field.                                                        */

/* Note: Some frame grabbers do not support cropping an image part in    */
/* hardware. In that case, return H_ERR_FGWP if you are urged to grab    */
/* only part of an image or do it in SOFTWARE (in that case: note, that  */
/* you have to provide a buffer to grab to for the full image            */
/* horizontal_resolution x vertical_resolution, not only image_width x   */
/* image_height).                                                        */

/*************************************************************************/
/***    TODO: Allocate buffers                                        ***/
/*************************************************************************/

/* Oh boy, this is going to be fun again, because there are a lot of     */
/* things to consider:                                                   */

/* (1) The size of the buffers:                                          */
/*     - image_width x image_height x bytes per pixel in case your       */
/*       frame grabber supports grabbing only a part of an image         */
/*     - horizontal_resolution x vertical_resolution x bytes per pixel   */
/*       otherwise                                                       */
/*                                                                       */
/* (2) Allocate buffers per board or per instance?                       */
/*     - If you do not pass this memory directly to HALCON objects, but  */
/*       perform a memcopy (or other "copying" procedures, for example to */
/*       split color raw data into three channels) AND you've got only   */
/*       one A/D converter per board, you can allocate the buffer(s) per  */
/*       board (frame grabber board), e.g, using                         */
/*                       currInst->board->BoardFrameBuffer[]             */
/*     - Otherwise (and also if you allow several instances per board    */
/*       with different image sizes) allocate them per instance.         */
/*     In both cases we would recommend to store at least copies of the  */
/*     pointers to the buffers in                                        */
/*                         currInst->InstFrameBuffer[]                   */
/*     in order to have a unified access to the data. Note, that the     */
/*     recommended entries sizeBuffer and refBuffer in BoardInfo and     */
/*     allocBuffer in TFGInstance might be quite handy to keep track of  */
/*     the current memory configuration.                                 */
/*                                                                       */
/* (3) How to allocate them?                                             */
/*     In most cases the frame grabber API will provide specific routines */
/*     for this task (since buffer memory to grab to at least has to be  */
/*     non-paged etc.)                                                   */
/*                                                                       */
/* (4) How many buffers?                                                 */
/*     In most cases 2 (in order to support asynchronous grabbing), see  */
/*     also the define MAX_BUFFERS                                       */

/* If you fail to allocate buffers return H_ERR_MEM -- not enough memory  */

/*************************************************************************/
/* example (NOTE: YOU WILL HAVE TO CHANGE THIS SECTION IN 99 OF 100 CASES)*/
/*************************************************************************/

/* We assume that we can crop an image part of size fginst->image_width x */
/* image_height in hardware; thus the size of the buffers is something   */
/* like                                                                  */

sizeBuffer = fginst->image_width*fginst->image_height *
              ((fginst->bits_per_channel+7) / 8)*fginst->num_channels;

/* Note, that especially for color frame grabbers                        */
/* ((fginst->bits_per_channel+7) / 8)*fginst->num_channels  might fail,   */
/* e.g., if the frame grabber delivers 32 bits of data instead of 24. In  */
/* this case make sure that you have set the fginst->num_channels        */
/* parameter to 4 above.                                                 */

/* Now we decide whether to use the memory pool of the board (shared by  */
/* maybe more than one instance) or to allocate instance-specific buffers.*/

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;
    }
    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++)
    {
      err = HAlloc (proc_id,(size_t)sizeBuffer,&currInst->InstFrameBuffer[i]);
      if (err != H_MSG_OK)
        return(CleanupFGOpen(proc_id,currInst,newBoardalloc,err));
    }
  }

  currInst->currBuffer = 0;        /* start whith the first buffer */


  /*********************************************************************/
  /***    TODO: Final Settings                                       ***/
  /*********************************************************************/

  /* increase the number of instances assigned to this board ...           */
  currInst->board->refInst++;
  /* ... and the overall number of instances                               */
  numInstance++;

  /* that's it: You finally succeeded! */

  return(H_MSG_OK);

} /* FGOpen */



/* ====================================================================
 *
 *                        Herror SetInstParam (...)
 *
 * ====================================================================
 *
 * Set the instance-specific frame grabber parameters
 *
 * ======================================================================
 */

static Herror SetInstParam (FGInstance *fginst)
{
  TFGInstance *currInst = (TFGInstance *)fginst->gen_pointer;

  /************************************************************/
  /***  TODO: Restore frame grabber settings for the instance ***/
  /************************************************************/

  /* everything that you allow to be different for instances    */
  /* of the same board (like port and input line etc.)          */
  /* 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 / currInst->fginst                      */
  /* example:                                                   */
  /* if (currInst->board->port != fginst->port)                 */
  /* {                                                          */
  /*   ...                                                      */
  /*   currInst->board->port = fginst->port;                    */
  /* }                                                          */

  return(H_MSG_OK);

} /* SetInstParam */




/* ======================================================================
 *
 *                        Herror FGClose (...)
 *
 * ======================================================================
 *
 * Close a frame grabber instance via close_framegrabber (...)
 *
 * ======================================================================
 */

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

  if (currInst->busy)
  {
    /**********************************************************************/
    /***   TODO: terminate the pending asynchronous job (the one          ***/
    /***         belonging to this instance                               ***/
    /**********************************************************************/
    currInst->busy=FALSE;
  }

  /**********************************************************************/
  /***    TODO: Cleanup                                               ***/
  /**********************************************************************/

  /* Basically, you have to deallocate the buffers associated with the      */
  /* instance and maybe you have to deallocate the data associated with the */
  /* board and unlock the frame grabber                                     */

  /***************************************/
  /***    TODO: Deallocate Buffers ?    ***/
  /***************************************/

  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;
      }
    }
  }
  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])
        {
          HCkP( 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--;
  }

  /*****************************************/
  /***    TODO: Deallocate Board ?      ***/
  /*****************************************/

  /* Check if the referenced board is still in use by another instance...   */

  if (currInst->board->refInst <= 1)
  {
    /**********************************************************************/
    /* Ok, here comes the serious part. You must "close" the board itself,  */
    /* because its not in use anymore. But please do not ask how to do this,*/
    /* ask the API manual of the frame grabber instead ...                  */
    /**********************************************************************/

    /* ... and deallocate the BoardInfo you have allocated in FGOpen()      */
    HCkP( HFree(proc_id,currInst->board));
  }
  else
  {
    currInst->board->refInst--;

    if (currInst->board->refInst == 1)
    {
      /* This is sort of a special situation: After you close this instance */
      /* there is only one other instance left using the same frame grabber */
      /* board. Thus, this other instance will rely on the fact that all    */
      /* the frame grabber settings have been done in FGOpen(). It won't set*/
      /* the port and input line etc. again before grabbing. Thus, you have */
      /* to make sure, that these settings are correct NOW:                 */

      for (i=0; i < FG_MAX_INST; i++)
      {
        if (FGInst[i].board && (currInst->instance != i))
        {
          /* Ok, FGInst[i] is in use and is NOT the current instance.       */
          /* Now let's see, if it references the same board as the          */
          /* current instance ...                                           */
          if (FGInst[i].board == currInst->board)
          {
            /****************************************************************/
            /***    TODO: Restore frame grabber settings for the          ***/
            /***          instance FGInst[i].instance/FGInst[i].fginst    ***/
            /****************************************************************/
            /* everything that you allow to be different for instances      */
            /* of the same board (like port and input line etc.)            */

            HCkP(SetInstParam(fginst->fgclass->instance[i]));
            break;
          }
        }
      } /* for (i... */
    } /* currInst->board->refInst == 1 */
  } /* currInst->board->refInst > 1 */

  currInst->board = NULL;
  numInstance--;

  return(H_MSG_OK);

} /* FGClose */



/* ======================================================================
 *
 *                      Herror ExtractChannelsFromRGB32 (...)
 *
 * ======================================================================
 *
 * Typically, a color frame grabber delivers the data as interleaved
 * tuple (e.g., RGB triples per pixel). Thus, you have to split this
 * data into separate channels (conform to the HALCON philosophy). This
 * routine might be a very simple template for such a procedure.
 *
 * ======================================================================
 */

static Herror ExtractChannelsFromRGB32 (FGInstance *fginst, HBYTE *data,
                            HBYTE *r_img, HBYTE *g_img, HBYTE *b_img)
{
  INT4_8 i,size;

  size = fginst->image_width*fginst->image_height;

  /* Assume that the frame grabber delivers 32 bits per pixel, e.g, BGRX */
  for (i=0; i < size; i++)
  {
    *b_img++ = *data++;
    *g_img++ = *data++;
    *r_img++ = *data++;
    data++;
  }

  return(H_MSG_OK);
} /* ExtractChannelsFromRGB32 */




/* ======================================================================
 *
 *                      Herror ExtractChannelsFromRGB24 (...)
 *
 * ======================================================================
 *
 * Typically, a color frame grabber delivers the data as interleaved
 * tuple (e.g., RGB triples per pixel). Thus, you have to split this
 * data into separate channels (conform to the HALCON philosophy). This
 * routine might be a very simple template for such a procedure.
 *
 * ======================================================================
 */

static Herror ExtractChannelsFromRGB24(FGInstance *fginst, HBYTE *data,
                                    HBYTE *r_img, HBYTE *g_img, HBYTE *b_img)
{
  INT4_8 i,size;

  size = fginst->image_width*fginst->image_height;

  /* Assume that the frame grabber delivers 24 bits per pixel, e.g, RGB */
  for (i=0; i < size; i++)
  {
    *r_img++ = *data++;
    *g_img++ = *data++;
    *b_img++ = *data++;
  }

  return(H_MSG_OK);
} /* ExtractChannelsFromRGB24 */


/* ======================================================================
 *
 *                      Herror ExtractChannelsFromRGB16 (...)
 *
 * ======================================================================
 * Typically, a color framegrabber delivers the data as interleaved
 * tuple (e.g., RGB triples per pixel). Thus, you have to split this
 * data into separate channels (conform to the HALCON philosophy). This
 * routine might be a very simple template for such a procedure. It
 * decomposes the rgb color mode 5:6:5, 16bpp, 1-plane, rgb into 3 channels
 * ======================================================================
 */

static Herror ExtractChannelsFromRGB16 (FGInstance *fginst, INT2 *data,
                            HBYTE *r_img, HBYTE *g_img, HBYTE *b_img)
{
  INT4_8 i,size;

  size = fginst->image_width*fginst->image_height;

  /* Assume that the frame grabber delivers 5-6-5 RGB data */
  for (i=0; i < size; i++)
  {
    //copy pixel wise
    *r_img++ = (*data & 0x1F)   << 3;
    *g_img++ = (*data & 0x7E0)  >> 3;
    *b_img++ = (*data & 0xF800) >> 8;
    data++;
  }

  return(H_MSG_OK);
} /* ExtractChannelsFromRGB16 */


/* ======================================================================
 *
 *                     Herror FGGrabStartAsync (...)
 *
 * ======================================================================
 *
 * Start an asynchronous grab via grab_image_start()
 *
 * ======================================================================
 */

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

  /*********************************************************************/
  /* Note: If your frame grabber does not support asynchronous grabbing:   */
  /*       Return H_ERR_FGASYNC                                          */
  /*********************************************************************/

  /* 'currInst->timeout' is the maximum allowed "age" for an image,  */
  /* see FGGrab(); just set the new threshold ...                    */
  currInst->timeout = (INT4_8)(maxDelay + 0.5);
  /* Note: There might be asynchronous grabs pending on the same     */
  /* board you would like to use. In general, most frame grabber     */
  /* have only one A/D converter - thus, you might have to cancel    */
  /* all these old jobs before doing anything else.                  */
  /* Obviously, this operation has nasty side-effects! So you could  */
  /* also return the error H_ERR_FGDV -- device busy in this case    */

  KillAllOtherJobs(fginst);

  if (currInst->busy)
  {
    /*******************************************************************/
    /***    TODO: Cancel pending job                                 ***/
    /*******************************************************************/

    /* there is an asynchronous grab pending - check whether its finished; */
    /* if not, terminate it (you have to start a new grab NOW)!           */
  }
  else if (currInst->board->refInst > 1)
  {
    /* There are other instances using the same board! Thus, you have to   */
    /* set the frame grabber parameters again ...                          */

    /*******************************************************************/
    /***    TODO: Restore frame grabber settings for currInst        ***/
    /*******************************************************************/

    /* Everything that you allow to be different for instances of the same */
    /* board (like port and input line etc.) has to be set again.          */

    HCkP(SetInstParam(fginst));
  }

  /*********************************************************************/
  /***    TODO: start an asynchronous grab                           ***/
  /*********************************************************************/

#ifdef WIN32
  _ftime(&currInst->grabStarted); /* remeber the time the grab was started */
#else
  gettimeofday(&currInst->grabStarted,&currInst->tzp);
#endif

  currInst->busy     = TRUE;

  return(H_MSG_OK);

} /* FGGrabStartAsync */




/* ======================================================================
 *
 *                          Herror GrabImg (...)
 *
 * ======================================================================
 *
 * Grab an image (auxiliary routine).
 *
 * ======================================================================
 */

static Herror GrabImg (Hproc_handle proc_id, FGInstance *fginst,
                       INT *readBuffer)
{
  TFGInstance     *currInst = (TFGInstance *)fginst->gen_pointer;
  HBOOL           done           = FALSE;
  HBOOL           checkTimeAgain = FALSE;
  HBOOL           newGrab        = FALSE;
#ifdef WIN32
  struct _timeb   now;
#else
```

```
  struct timeval  now;
  struct timezone tzp;
#endif
  INT4_8          time_diff;

  /* Note: There might be asynchronous grabs pending on the same     */
  /* board you would like to use. In general, most frame grabber     */
  /* have only one A/D converter - thus, you might have to cancel    */
  /* all these old jobs before doing anything else.                  */
  /* Obviously, this operation has nasty side-effects! So you could  */
  /* also return the error H_ERR_FGDV -- device busy in this case    */

  KillAllOtherJobs(fginst);

  if ((!currInst->busy) && (currInst->board->refInst > 1))
  {
    /* There are other instances using the same board! Thus, you have to   */
    /* set the frame grabber parameters again ...                          */

    /*********************************************************************/
    /***    TODO: Restore frame grabber settings for currInst          ***/
    /*********************************************************************/

    /* everything that you allow to be different for instances of the same */
    /* board (like port and input line etc.) has to be set again.          */

    HCkP(SetInstParam(fginst));
  }

  /***********************************************************************/
  /* Note: If you encounter errors during grabbing, return one of the      */
  /*       following error codes:                                          */
  /*       H_ERR_FGNV      -- no video signal                              */
  /*       H_ERR_FGF       -- grabbing failed (general)                    */
  /*       H_ERR_FGTIMEOUT -- timeout                                      */
  /***********************************************************************/

  if (currInst->busy)
  {
    /* there is an asynchronous job pending for this instance ...          */

    if (!fginst->async_grab)
    {
      /* a SYNCHRONOUS grab was requested ...                              */
      /*******************************************************************/
      /***    TODO: Cancel the current job                             ***/
      /*******************************************************************/

      newGrab = TRUE;
    }
    else
    {
      /* an ASYNCHRONOUS grab was requested ...                            */
      /*******************************************************************/
      /***    TODO: Check, if the previously started grab has finished   ***/
      /***          (there is a pending grab (currInst->busy is TRUE)    ***/
      /*******************************************************************/

      /* Assume that the checking routine sets "done" to TRUE/FALSE        */
      if (done)
      {
        /* old grab finished: Check, whether this 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->timeout)
        {
          /* Bad luck! The image is there, but too old ...               */
          /* Thus, you have to grab a new image ...                      */
          /* You can do this synchronously or asynchronously -- you have */
          /* to wait anyway.                                             */
          newGrab = TRUE;
        }
      }
      else
      {
        /* There is an old job pending and the grab has not finished yet.  */
        /* Basically, you can just skip doing anything in this branch and  */
        /* wait until the grab has finished. However, the image still      */
        /* might be too old ...                                            */
        /* This is sort of a timing problem, because you can measure the   */
        /* time since the grab has started HERE and cancel the grab if the */
        /* image is already too old NOW, but its hard to tell when the     */
        /* grab will finish. Maybe its not too old NOW, but it will be too */
        /* old THEN. In this case, there is no use waiting. We could cancel*/
        /* the job right now!                                              */
        checkTimeAgain = TRUE;
      }
    } /* async. grab */
  } /* pending job ... */

  else

  {
    /* There is no asynchronous job pending for this instance ...          */
    /* Thus, you have to grab a new image ...                              */
    /* You can do this synchronously or asynchronously -- you have to wait */
    /* anyway.                                                             */
    newGrab = TRUE;
  }

  if (newGrab)
  {
    /***********************************************************************/
    /***    TODO: grab a new image                                         */
    /***********************************************************************/

    done = TRUE;  /* or FALSE, if you use an API call for grabbing that    */
                  /* does not wait for the end of the grab                 */
  }

  if (!done)
  {
    /***********************************************************************/
    /***    TODO: wait until the (asynchronous) grab has finished          */
    /***********************************************************************/

  }

  if (checkTimeAgain)
  {
    /* old grab finished: Check, whether this 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->timeout)
    {
      /* Bad luck! The image is there, but too old ...                     */
      /*********************************************************************/
      /***    TODO: grab a new image                                       */
      /*********************************************************************/
    }
  }

  /* You've got your image by now                                      */
  /*********************************************************************/
  /***    TODO: Switch the buffer (if you use more than one buffer)      */
  /*********************************************************************/

  *readBuffer = currInst->currBuffer;

  /* Select the next buffer for grabbing                               */
  currInst->currBuffer++;
  if (currInst->currBuffer >= MAX_BUFFERS)
    currInst->currBuffer = 0;

  if (fginst->async_grab)
  {
    /*********************************************************************/
    /***    TODO: Start the next asynchronous grab                         */
    /*********************************************************************/


#ifdef WIN32
    _ftime(&currInst->grabStarted);       /* the time the grab was started */
#else
    gettimeofday(&currInst->grabStarted,&currInst->tzp);
#endif

    currInst->busy = TRUE;
  }

  return(H_MSG_OK);

} /* GrabImg */




/* ===================================================================
 *
 *                          Herror FGGrab (...)
 *
 * ===================================================================
 *
 * Grab an image via grab_image(), that is, synchronously.
 * Note: In most cases you can use this routine also for asynchronous
 *       grabbing, see FGGrabAsync().
 *
 * ===================================================================
 */

static Herror FGGrab (Hproc_handle proc_id, FGInstance *fginst,
                      Himage *image, INT *num_image)
{
  TFGInstance   *currInst = (TFGInstance *)fginst->gen_pointer;
  INT           readBuffer;
  INT           i;
  Herror        err;

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

  /*********************************************************************/
  /***    TODO: Create a HALCON image from the grabbed data              */
  /*********************************************************************/

  /* Note that this might be slightly more difficult as indicated below,   */
  /* if you would like to do subsampling or cropping of image parts in     */
  /* software. For the example we assume that this is done by the          */
  /* frame grabber hardware.                                               */

  if (currInst->volatileMode)
  {
    /*********************************************************************/
    /* Insert the buffer into a HALCON object (fast but with side effects, */
    /* see above.                                                          */
  /*********************************************************************/

  /* In general, the volatile mode works only with non interleaved image */
  /* data. To profit by the volatile mode the buffer data has to be      */
  /* structured in a compatible way to the HALCON image format. If you   */
  /* allow the volatile mode on multi channel images (e.g. RGB) get sure */
  /* that your frame grabber delivers the data for each channel in       */
  /* separate (and in this example consecutive) memory planes. The order */
  /* of the channels is R-G-B with ascending memory adress in this       */
  /* example.                                                            */

  /* Note on this that the decision what type of HALCON image to create  */
  /* is rather crude in this example: It assumes that the frame grabber  */
  /* delivers either 8 bit unsigned, 16 bit signed, or 32 bit signed     */
  /* data; otherwise you would have to convert the buffers. Thus, the    */
  /* volatile mode would be obsolete ...                                 */

  INT4_8  size;
  INT     num_channels;

  if(fginst->num_channels == 4)
    /* If you've allowed 32 bits per pixel while setting the  volatile   */
    /* mode you have to decide how to handle the redundant data. In this */
    /* example we assume an RGBX ordering of the corresponding memory    */
    /* planes. Thus we just take the first three channels.               */
    num_channels = 3;
  else
    num_channels = fginst->num_channels;

  size = fginst->image_width*fginst->image_height;

  if (fginst->bits_per_channel <= 8)
  {
    for (i=0; i<num_channels; i++)
    {
      HCkP(HNewImagePtr(proc_id,
                        &image[i],
                        BYTE_IMAGE,
                        fginst->image_width, fginst->image_height,
                        (void*)currInst->InstFrameBuffer[readBuffer] +
                          i*size*sizeof(BYTE_IMAGE),
                        FALSE));
    }
  }
  else if (fginst->bits_per_channel <= 16)
  {
    for (i=0; i<fginst->num_channels; i++)
    {
      HCkP(HNewImagePtr(proc_id,
                        &image[i],
                        INT2_IMAGE,
                        fginst->image_width, fginst->image_height,
                        (void*)currInst->InstFrameBuffer[readBuffer] +
                          i*size*sizeof(INT2_IMAGE),
                        FALSE));
    }
  }
  else
  {
    for (i=0; i<fginst->num_channels; i++)
    {
      HCkP(HNewImagePtr(proc_id,
                        &image[0],
                        INT4_IMAGE,
                        fginst->image_width, fginst->image_height,
                        (void*)currInst->InstFrameBuffer[readBuffer] +
                          i*size*sizeof(INT4_IMAGE),
                        FALSE));
    }
  }
  *num_image = num_channels;
}

else
```

```
{
  /*********************************************************************/
  /* Copy the buffer into a NEW HALCON object                          */
  /*********************************************************************/

  INT save;

  /* Do not initialize the new images with 0: */
  HReadSysComInfo(proc_id, HGInitNewImage, &save);
  HWriteSysComInfo(proc_id, HGInitNewImage, FALSE);

  /* Note that we support in our example following different image types:*/
  /* 8 bit unsigned, 16 bit signed, or 32 bit signed for one channel     */
  /* grayscale and 5/8bit unsigned for three channel color format.       */
  /* Obviously, the subsequent memcpy for gray value images can only     */
  /* work, if the frame grabber delivers the data in the same format;    */
  /* otherwise you must do some shifting ...                             */
  /* For color images we assume that the channel image data is delivered */
  /* in an interleaved format. The use of memcpy is obsolet therefore.   */

  /*********************************************************************/
  /* (I) Create a NEW HALCON object                                    */
  /*********************************************************************/

  if(fginst->num_channels == 4)
    /* 32 bit par pixel: discard the redundant data and copy the image   */
    /* data to a three channel HALCON image.                             */
    *num_image = 3;
  else
    *num_image = fginst->num_channels;

  if (fginst->bits_per_channel <= 8)
  {
    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;
      }
    }
  }
  else if (fginst->bits_per_channel <= 16)
  {
    for (i=0; i<*num_image; i++)
    {
      err = HNewImage(proc_id,&image[0],INT2_IMAGE,
                      fginst->image_width, fginst->image_height);
      if (err != H_MSG_OK)
      {
        HWriteSysComInfo(proc_id, HGInitNewImage, save);
        return err;
      }
    }
  }
  else if (fginst->bits_per_channel <= 32)
  {
    for (i=0; i<*num_image; i++)
    {
      err = HNewImage(proc_id,&image[0],INT4_IMAGE,
                      fginst->image_width,fginst->image_height);
      if (err != H_MSG_OK)
      {
        HWriteSysComInfo(proc_id, HGInitNewImage, save);
        return err;
      }
    }
  }

  HWriteSysComInfo(proc_id, HGInitNewImage, save);
  /*********************************************************************/
  /* (II) Copy data                                                    */
```

```
    /*********************************************************************/

    if (*num_image == 1)
    {
      if (fginst->bits_per_channel <= 8)
        memcpy ((void *)image[0].pixel.b,
                currInst->InstFrameBuffer[readBuffer],
                fginst->image_width * fginst->image_height);
      else if (fginst->bits_per_channel <= 16)
      {
        memcpy ((void *)image[0].pixel.s.p,
                currInst->InstFrameBuffer[readBuffer],
                fginst->image_width * fginst->image_height * 2);
        image[0].pixel.s.num_bits = 16;
      }
      else if (fginst->bits_per_channel <= 32)
        memcpy ((void *)image[0].pixel.l,
                currInst->InstFrameBuffer[readBuffer],
                fginst->image_width * fginst->image_height * 4);
    }
    else
    {
      /* Note again: Many color frame grabbers deliver the data in an      */
      /* interleaved format incompatible to HALCON. Thus, you typically    */
      /* will have to use something like this:                             */
      if(fginst->num_channels == 4)
      {/* 32 bit RGB format */
        HCkP(ExtractChannelsFromRGB32 (fginst,
                                       currInst->InstFrameBuffer[readBuffer],
                                       image[0].pixel.b,
                                       image[1].pixel.b,
                                       image[2].pixel.b));
      }
      else if (fginst->bits_per_channel == 5)
      {/* 5-6-5 RGB format */
        HCkP(ExtractChannelsFromRGB16 (fginst,
                                       (INT2 *)currInst->InstFrameBuffer[readBuffer],
                                       image[0].pixel.b,
                                       image[1].pixel.b,
                                       image[2].pixel.b));
      }
      else /*fginst->bits_per_channel == 8*/
      {/* 8-8-8 RGB format */
        HCkP(ExtractChannelsFromRGB24 (fginst,
                                       currInst->InstFrameBuffer[readBuffer],
                                       image[0].pixel.b,
                                       image[1].pixel.b,
                                       image[2].pixel.b));
      }
    }
  }/* copy data */

  /* Note that we will use FGGrab() for asynchronous grabbing also.  */
  /* Thus, we set the async_grab here explicitly for synchronous     */
  /* grabbing and reset it to asynchronous grabbing in FGGrabAsync() */
  /* if necessary.                                                   */

  fginst->async_grab=FALSE;

  return(H_MSG_OK);

} /* FGGrab */




/* ======================================================================
 *
 *                          Herror FGGrabAsync (...)
 *
 * ======================================================================
 *
 * Grab an image via grab_image_async(), that is, asynchronously.
 *
 * ======================================================================
 */

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

  /* Set timeout and asynchronous mode    */
  currInst->timeout = (INT4_8)(maxDelay+0.5);
  fginst->async_grab = TRUE;

  /* Get current image and start new grab */
  HCkP(FGGrab(proc_id, fginst, image, num_image));

  return(H_MSG_OK);
} /* FGGrabAsync */



/* ======================================================================
 *
 *                        Herror FGGrabRegion (...)
 *
 * ======================================================================
 *
 * grab region(s) via grab_region(), that is synchronously
 *
 * ======================================================================
 */

static Herror FGGrabRegion (Hproc_handle proc_id, FGInstance *fginst,
                            Hrlregion **region, INT *num_region,
                            INT *rlalloc_type)
{
  TFGInstance *currInst = (TFGInstance *)fginst->gen_pointer;
  INT         readBuffer;

  /**********************************************************************/
  /***    TODO: Grab an image and segment it into region(s)           ***/
  /***          region is an array of MAX_OBJ_PER_PAR pointers to     ***/
  /***          Hrlregion; the Hrlregions themself have not been      ***/
  /***          allocated so far!                                     ***/
  /**********************************************************************/

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

  /* Now you've got an image (in buffer "readBuffer") -- segment it!       */
  /* Note that there are three different ways to allocate region data (see */
  /* the C-Interface Programmer's Manual for details). Since the data you  */
  /* allocate in this routine is copied to the HALCON data base and then   */
  /* deallocated again, the caller of this routine must know, which one    */
  /* you used. This is specified by rlalloc_type:                          */
  /* HAllocRLTmp / HAllocRLNumTmp  (*rlalloc_type = FG_RLALLOC_TMP)        */
  /* HAllocRL / HAllocRLNum        (*rlalloc_type = FG_RLALLOC_PERMANENT)  */
  /* HAllocRLLocal / HAllocRLNumLocal  (*rlalloc_type = FG_RLALLOC_LOCAL)  */

  /* Attention: If you the "Tmp" version, you MUST allocate the image      */
  /*            regions in ascending order, because they're stored on the  */
  /*            stack and the HALCON interface will free them in           */
  /*            descending order!                                          */

  /* We recommend to use the "Local" version: It's more flexible than the  */
  /* "Tmp" version, but still includes an automatic garbage collection.    */

  /* Example: Allocate two regions (e.g. one for all image parts of a      */
  /* specific color and one for the rest of the image)                     */

  HCkP(HAllocRLNumLocal(proc_id, &region[0],
                        fginst->image_width*fginst->image_height/2));
  HCkP(HAllocRLNumLocal(proc_id, &region[1],
                        fginst->image_width*fginst->image_height/2));
  *rlalloc_type = FG_RLALLOC_LOCAL;
  /* Well the segmentation itself is up to you :-)                         */
  /* ...                                                                   */
  *num_region = 2;

  /* Note that we will use FGGrabRegion() for asynchronous grabbing also.  */
  /* Thus, we set the async_grab here explicetly for synchronous           */
  /* grabbing and reset it to asynchronous grabbing in FGGrabRegionAsync() */
  /* if necessary.                                                         */

  fginst->async_grab = FALSE;

  return(H_MSG_OK);

} /* FGGrabRegion */




/* ======================================================================
 *
 *                        Herror FGGrabRegionAsync (...)
 *
 * ======================================================================
 *
 * grab region(s) via grab_region_async(), that is asynchronously
 *
 * ======================================================================
 */

static Herror FGGrabRegionAsync (Hproc_handle proc_id, FGInstance *fginst,
                                 double maxDelay, Hrlregion **region,
                                 INT *num_region, INT *rlalloc_type)
{
  TFGInstance *currInst = (TFGInstance *)fginst->gen_pointer;

  /* Set timeout and asynchronous mode    */
  currInst->timeout = (INT4_8)(maxDelay+0.5);
  fginst->async_grab = TRUE;

  /* Get current image, segment it, and start new grab */
  HCkP(FGGrabRegion(proc_id, fginst, region, num_region, rlalloc_type));

  return(H_MSG_OK);

} /* FGGrabRegionAsync */




/* ======================================================================
 *
 *                        Herror FGInfo (...)
 *
 * ======================================================================
 *
 * Information + value list (if applicable) concerning a specific
 * query for this frame grabber as requested by info_framegrabber().
 *
 * ATTENTION: No memory has been allocated for values!
 *
 * ======================================================================
 */

static Herror FGInfo (Hproc_handle proc_id, INT queryType,
                      char **info, Hcpar **values, INT *numValues)
{
  /* queryType:                                                           */
  /*   FG_QUERY_GENERAL:     General Info (full name , vendor etc.)       */
  /*   FG_QUERY_PORT:        Descript. of the ports (signal, connectors)  */
  /*   FG_QUERY_CAMERA_TYPE: Descript. of the camera_type paramater       */
  /*   FG_QUERY_DEFAULTS:    Default values (see open_framegrabber() )    */
  /*   FG_QUERY_PARAMETERS:  Names of non-standard parameters available   */
  /*                         for set_framegrabber_param()                 */
  /*   FG_QUERY_INFO_BOARDS: Info about installed boards                  */
```

```
Hcpar *val;
INT    i;

switch(queryType)
{
  case FG_QUERY_GENERAL:
    /*******************************************************************/
    /***    TODO: Return general information                         ***/
    /*******************************************************************/

    /* This query typically doesn't return any "values", but only a     */
    /* general description of the board (-family or -model) etc.        */
    *info = "HALCON frame grabber interface template, vendor: MVTec Software GmbH.";
    *values      = NULL;
    *numValues   = 0;
    break;

  case FG_QUERY_PORT:
    /*******************************************************************/
    /***    TODO: Return port description                            ***/
    /*******************************************************************/

    /* Explain what ports are available and how you select them (it is  */
    /* the assignment of port numbers to physical connectors like a     */
    /* S-VHS plugin                                                     */

    /* example: */
    *info = "Port 0 (S-Video), port 1,2 (Composite); 8 and 24 bits available for each port.";
    HCkP( HAlloc (proc_id,(size_t)(3*sizeof(Hcpar)),&val));
    val[0].par.l = 0;
    val[1].par.l = 1;
    val[2].par.l = 2;
    val[0].type  = val[1].type = val[2].type = LONG_PAR;
    *values      = val;
    *numValues   = 3;

    /* another example: Only one port (that is: No need to specify a port) */
    *info = "Unused.";
    *values      = NULL;
    *numValues   = 0;

    break;

  case FG_QUERY_CAMERA_TYPE:
    /*******************************************************************/
    /***    TODO: Return a description of the "camera_type" parameter ***/
    /*******************************************************************/

    /* Explain the usage of the "camera_type" parameter in              */
    /* open_framegrabber() and its possible values                      */

    /* example: */
    *info = "Video signal of the camera.";
    HCkP( HAlloc (proc_id,(size_t)(3*sizeof(Hcpar)),&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;

  case FG_QUERY_DEFAULTS:
    /* Just leave this one here like it is ... */
    *info = "Default values (as used for open_framegraber).";
    HCkP( HFgGetDefaults(proc_id,fgClass,values,numValues));
    break;

  case FG_QUERY_PARAMETERS:
    /*******************************************************************/
    /***    TODO: Return the names of non-standard parameters        ***/
    /*******************************************************************/
```

```
      /* What additional parameters are supported for  */
      /* set_framegrabber_param(), see FGSetParam()    */

      /* example: */
      *info = "Additional parameters for this frame grabber.";
      HCkP( HAlloc (proc_id,(size_t)(FG_PARAM_NUM*sizeof(Hcpar)),&val));
      val[0].par.s = FG_PARAM_VOLATILE;
      val[1].par.s = FG_PARAM_REVISION;
      for (i=0; i < FG_PARAM_NUM; i++)
        val[i].type = STRING_PAR;
      *values      = val;
      *numValues   = FG_PARAM_NUM;
      break;

    case FG_QUERY_INFO_BOARDS:
      /******************************************************************/
      /***    TODO: Return the real device numbers of the installed     ***/
      /***          boards or something like that                       ***/
      /******************************************************************/
      *info        = "Info about installed xy boards.";
      *values      = NULL;
      *numValues   = 0;
      break;

    default:
      *info = "Unsupported query!";
      *values      = NULL;
      *numValues   = 0;
  } /* switch */

  return(H_MSG_OK);

} /* FGInfo */




/* ======================================================================
 *
 *                         Herror FGSetLut (...)
 *
 * ======================================================================
 *
 * Set the LUT of the frame grabber via set_framegrabber_lut()
 *
 * ======================================================================
 */

static Herror FGSetLut (Hproc_handle proc_id, FGInstance *fginst,
                        INT4_8 *red, INT4_8 *green, INT4_8 *blue,
                        INT num)
{
  TFGInstance *currInst = (TFGInstance *)fginst->gen_pointer;

  /**********************************************************************/
  /***    TODO: Set the lookup table of your frame grabber              ***/
  /**********************************************************************/

  /* The input to this routine is a RGB lookup table with "num" entries of */
  /* RGB triples. There is not much more than can be said in general ...   */
  /* Sorry, you will have to find out by yourself what your frame grabber  */
  /* supports concerning such a feature ...                                */

  /* Note that setting the frame grabber LUT will have side effects on the */
  /* other instances using the same board. Thus, in case that              */
  /*              fginst->board->refInst > 1                               */
  /* you will have to store the LUT in the TFGInstance struct and set it   */
  /* again and again before grabbing ...                                   */

  return(H_MSG_OK);
} /* FGSetLut */
```

```
/* ======================================================================
 *
 *                        Herror FGGetLut (...)
 *
 * ======================================================================
 *
 * Get the LUT of the frame grabber via get_framegrabber_lut()
 *
 * ======================================================================
 */

static Herror FGGetLut (Hproc_handle proc_id, FGInstance *fginst,
                        INT4_8 *red, INT4_8 *green, INT4_8 *blue,
                        INT *num)
{
  TFGInstance *currInst = (TFGInstance *)fginst->gen_pointer;

  /**********************************************************************/
  /***    TODO: Get the lookup table of your frame grabber            ***/
  /**********************************************************************/

  /* The output of this routine is a RGB lookup table with "num" entries   */
  /* of RGB triples. There is not much more than can be said in general... */
  /* Sorry, you will have to find out by yourself what your frame grabber  */
  /* supports concerning such a feature ...                                */

  *num = 0;
  return(H_MSG_OK);
} /* FGGetLut */



/* ======================================================================
 *
 *                        Herror FGSetParam (...)
 *
 * ======================================================================
 *
 * Set frame grabber specific parameters via set_framegrabber_param()
 *
 * ======================================================================
 */

static Herror FGSetParam (Hproc_handle proc_id, FGInstance *fginst,
                          char *param, Hcpar *value, INT num)
{
  TFGInstance *currInst = (TFGInstance *)fginst->gen_pointer;

  /**********************************************************************/
  /***    TODO: Parse the parameter "param" and set the corresponding ***/
  /***          frame grabber parameter for this instance             ***/
  /**********************************************************************/

  /* The standard parameters specified in open_framegrabber() and evaluated*/
  /* in FGOpen() cannot cover every aspect of the hardware features of all */
  /* available frame grabbers. Therefore, HALCON provides an additional    */
  /* operator to specify settings for frame grabber specific features OR to*/
  /* change the values for standard parameters without closing an opening  */
  /* the frame grabber again. Additional parameters are denoted by an      */
  /* arbitrary string of YOUR choice. However, please make the names of    */
  /* these parameters available via the query 'FG_QUERY_PARAMETERS' in     */
  /* FGInfo() so that the user can access this information online.         */

  /* The STANDARD PARAMETERS can be handled using the following defines    */
  /* (note that you do NOT have to support all these parameters here in    */
  /* this routine. They are set via FGOpen() during the initialization of  */
  /* a new instance. However, you might want to change the original        */
  /* settings dynamically without closing/opening the frame grabber again):*/
  /*                                                                       */
  /*      define              curr. value of the define     type           */
  /* --------------------------------------------------------------------- */
  /* FG_PARAM_HORIZONTAL_RESOLUTION  "horizontal_resolution" LONG_PAR       */
  /* FG_PARAM_VERTICAL_RESOLUTION    "vertical_resolution"   LONG_PAR       */
  /* FG_PARAM_IMAGE_WIDTH            "image_width"           LONG_PAR       */
/* FG_PARAM_IMAGE_HEIGHT          "image_height"          LONG_PAR       */
/* FG_PARAM_START_ROW             "start_row"             LONG_PAR       */
/* FG_PARAM_START_COL             "start_column"          LONG_PAR       */
/* FG_PARAM_FIELD                 "field"                 STRING_PAR     */
/* FG_PARAM_BITS_PER_CHANNEL      "bits_per_channel"      LONG_PAR       */
/* FG_PARAM_COLOR_SPACE           "color_space"           STRING_PAR     */
/* FG_PARAM_GAIN                  "gain"                  FLOAT_PAR      */
/* FG_PARAM_CAMERA_TYPE           "camera_type"           STRING_PAR     */
/* FG_PARAM_DEVICE                "device"                STRING_PAR     */
/* FG_PARAM_PORT                  "port"                  LONG_PAR       */
/* FG_PARAM_LINE_IN               "line_in"               LONG_PAR       */
/*                                                                       */
/* Note that "field" (fginst->field) externally is defined as string,    */
/* but internally as "int" using the following conversion:              */
/* FG_FIRST_FIELD_TXT          "first"         <->  FG_FIRST_FIELD       */
/* FG_SECOND_FIELD_TXT         "second"        <->  FG_SECOND_FIELD      */
/* FG_NEXT_FIELD_TXT           "next"          <->  FG_NEXT_FIELD        */
/* FG_FULL_FRAME_TXT           "interlaced"    <->  FG_FULL_FRAME        */
/* FG_PROGRESSIVE_FRAME_TXT    "progressive"   <->  FG_PROGRESSIVE_FRAME */
/* Note further that "external_trigger" externally is defined as string  */
/* ("true", "false") but internally as HBOOL.                            */

/* The input to FGSetParam() is ONE parameter-value pair. You should     */
/* check the name of the parameter, the type of the corresping value,    */
/* and - of course - the consistency of the specified value. In case of  */
/* unreasonable inputs return the error codes                            */
/* H_ERR_FGPARAM -- parameter not supported                              */
/* H_ERR_FGPART  -- invalid parameter type                               */
/* H_ERR_FGPARV  -- invalid parameter value                              */

/* Note that some of the parameters might have side-effects on other     */
/* instances. Thus, you might have to include these parameters in the    */
/* BoardInfo and TFGInstance structs and set them prior to each grab ... */

/* Example: In our example we will enable/disable the "volatile" mode    */
/* for an instance, see also FGOpen(). This does not change the current  */
/* configuration of the board itself. Thus, we can perform all           */
/* necessary steps right here in this routine (without side-effects on   */
/* other instances).                                                     */



if (!strcmp(param, FG_PARAM_VOLATILE))
{
  /***************************/
  /***    Volatile         ***/
  /***************************/

  /* In the 'volatile' mode we attach the frame grabber's buffer memory to*/
  /* the HALCON image instead of allocating new memory (thus avoiding a   */
  /* memcpy). Note that this grabbing method has a severe side-effect:    */
  /* Older HALCON images are overwritten!                                 */
  /* By the way, many color frame grabbers  deliver interleaved data, that*/
  /* is, e.g., RGB triples instead of three separate channels. In this    */
  /* case you have to split the data explicitly. Thus, a "volatile" mode  */
  /* is not attractive anymore. If YOU do the memory management of the    */
  /* image data inside a HALCON image you MUST let HALCON know (otherwise */
  /* there will be system crashes during deallocating images)             */
  INT       i;
  BoardInfo *board = currInst->board;
  INT4_8    sizeBuffer;

  if (value->type != STRING_PAR)
    return(H_ERR_FGPART);
  if (!strcmp(value->par.s, "enable"))
  {
    /***************************/
    /* Enable the volatile mode */
    /***************************/

    if (fginst->bits_per_channel != 8  ||
        fginst->bits_per_channel != 16 ||
        fginst->bits_per_channel != 32)
    {
        /* There’s no use for the volatile mode grabbing non byte      */
        /* conform channel depth because we have to split the grabbed  */
        /* pixel data into separate bytes ...                          */
        return(H_ERR_FGPARV);
      }
      else if (fginst->num_channels != 1)
      {
        /* We assume for our example that the frame grabber delivers   */
        /* interleaved color data, which is inoperative for ’volatile’ */
        /* mode. This is because we have to split the grabbed raw data */
        /* into separate RGB channels ...                              */
        return(H_ERR_FGPARV);
      }
      if (!currInst->volatileMode)
      {
        /* Otherwise we don’t have to do anything at all since the */
        /* volatile mode already IS enabled.                       */
        if (!currInst->allocBuffer)
        {
          /* This specific instance uses buffers assigned to the board.*/
          /* These buffers might be shared with other instances. To    */
          /* prevent side-effects we have to provide buffers for the   */
          /* current instance exclusively.                             */
          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.     */
            /* Thus, we have to allocate new buffers (typically you   */
            /* will have to use frame grabber specific routines for   */
            /* this task, see FGOpen() also):                         */
            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->allocBuffer */

        currInst->volatileMode = TRUE;
        fginst->halcon_malloc  = FALSE;
        fginst->clear_proc     = NULL;   /* Do not deallocate the     */
                                         /* grabbed image!            */

      } /* !currInst->volatileMode */
    } /* "enable" */

    else if (!strcmp(value->par.s, "disable"))
    {
      /*****************************/
      /* Disable the volatile mode */
      /*****************************/

      if (currInst->volatileMode)
      {
        /* Otherwise we don’t have to do anything at all since the */
        /* volatile mode already IS disabled.                      */

        sizeBuffer = fginst->image_width*fginst->image_height *
          ((fginst->bits_per_channel+7) / 8) * fginst->num_channels;
        if (board->refInst > 1)
        {
          /* There are other instances using the same board. Thus, */
          /* if the size of the board buffers allows to use these  */
          /* for the current instance as well we can deallocate    */
          /* the instance-specific buffers to decrease the memory  */
          /* load:                                                  */
          if (board->sizeBuffer >= sizeBuffer)
          {
            for (i=0; i < MAX_BUFFERS; i++)
            {
              HCkP( HFree(proc_id,currInst->InstFrameBuffer[i]));
              currInst->InstFrameBuffer[i] = board->BoardFrameBuffer[i];
            }
            currInst->allocBuffer = FALSE;
            board->refBuffer++;
          }
        }
        if (board->sizeBuffer == 0)
        {
          /* This is a special case: There are no board buffers so */
          /* far. Transfer the instance buffers to the board in    */
          /* order to make them "shared".                           */
          for (i=0; i < MAX_BUFFERS; i++)
          {
            board->BoardFrameBuffer[i] = currInst->InstFrameBuffer[i];
          }
          currInst->allocBuffer = FALSE;
          board->refBuffer  = 1;
          board->sizeBuffer = sizeBuffer;
        }
        currInst->volatileMode = FALSE;
        fginst->halcon_malloc  = TRUE;

      } /* currInst->volatileMode */
    } /* "disable" */
    else
      return(H_ERR_FGPARV);
  } /* param: FG_PARAM_VOLATILE */

  else
    /* parameter not supported */
    return(H_ERR_FGPARAM);

  return(H_MSG_OK);
} /* FGSetParam */



/* ======================================================================
 *
 *                        Herror FGGetParam (...)
 *
 * ======================================================================
 *
 * Get frame grabber specific parameters via get_framegrabber_param()
 *
 * ======================================================================
 */

static Herror FGGetParam (Hproc_handle proc_id, FGInstance *fginst,
                          char *param, Hcpar *value, INT *num)
{
  TFGInstance *currInst = (TFGInstance *)fginst->gen_pointer;

  *num = 1;
  /**********************************************************************/
  /***    TODO: Parse the parameter "param" and return the corresponding ***/
  /***          frame grabber parameter for this instance              ***/
  /**********************************************************************/

  /* Please see FGSetParam() for a detailed discussion.                */
  /* Note: The standard parameters (encoded in fginst) are already handled */
  /* by the HALCON system. You do NOT have to provide code for these      */
  /* parameters here!                                                  */

  /* Example: Return the "volatile status", see FGSetParam().          */

  if (!strcmp(param, FG_PARAM_VOLATILE))
  {
    /****************************/
    /*         VOLATILE         */
    /****************************/
    value->type  = STRING_PAR;
    value->par.s = ( currInst->volatileMode ? "enable" : "disable" );
  }
  else if (!strcmp(param, FG_PARAM_REVISION))
  {
    /****************************/
    /*         REVISION         */
    /****************************/
    value->type  = STRING_PAR;
    value->par.s = INTERFACE_REVISION;    /* adapt to Revision in header! */
  }
  else
    /* parameter not supported */
    return(H_ERR_FGPARAM);

  return(H_MSG_OK);
} /* FGGetParam */



/* ======================================================================
 *
 *                  FGInstance** FGOpenRequest(void)
 *
 * ======================================================================
 *
 * provide a new instance prior to FGGrab()
 *
 * ======================================================================
 */

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

  if (numInstance >= FG_MAX_INST)
  {
    /* too many instances ... */
    return(NULL);
  }
  else
  {
    /* Note: If you do not want to bother about multiple instances */
    /* just return ALWAYS the same instance:                       */
    /*                                                             */
    /*         fginst->gen_pointer = (void*)&FGInst[0];            */
    /*         return (&(fgClass->instance[0]));                   */
    /*                                                             */
    /* The instance will be closed using FGClose() and re-opened   */
    /* using FGopen()                                              */

    /* retrieve next unused instance */
    for (i=0; i < FG_MAX_INST; i++)
    {
      if (!FGInst[i].board)
        break;
    }
    if (i >= FG_MAX_INST)
      return(NULL); /* this cannot happen, but you know Murphy, don't you? */
    fginst->gen_pointer = (void*)&FGInst[i];
    return (&(fgClass->instance[i]));
  }
} /* FGOpenRequest */
```

```
/* ====================================================================
 *
 *                           Herror FGInit (...)
 *
 * ====================================================================
 *
 * Initialize the frame grabber class. This routine is called by HALCON
 * the very first time a HALCON process wants to access a frame grabber
 * via open_framegrabber() or info_framegrabber()
 *
 * ================================================================== */

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;

  fg->interface_version = FG_INTERFACE_VERSION;/* do not change this line! */

  /*********************************************************************/
  /***    TODO: Provide reasonable defaults etc. for open_framegrabber()   */
  /*********************************************************************/

  /* -------------------------- 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->GrabRegion            = FGGrabRegion;
  fg->GrabRegionAsync       = FGGrabRegionAsync;
  fg->SetParam              = FGSetParam;
  fg->GetParam              = FGGetParam;
  fg->SetLut                = FGSetLut;
  fg->GetLut                = FGGetLut;
  /* --------------------------- 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;
  fg->external_trigger      = FALSE;
  strcpy(fg->camera_type,"auto");
  strcpy(fg->device,"0");
  fg->port                  = 1;
  fg->line_in               = 1;
  /* ----------------- store the class information --------------------- */
  fgClass                   = fg;
  return(H_MSG_OK);
} /* FGInit */


/*****************************************************************************
 *  end of CIOFGTemplate.c
 *****************************************************************************/
```

Index