HALCON
the Power of Machine Vision

Programmer's Guide

8.0
About This Manual

This manual describes the programming language interfaces of HALCON and shows how to use HALCON in programming languages like C++, C#, C, or Visual Basic. It contains the necessary information to understand and use the provided data structures and classes in your own programs.

We expect the reader of this manual to be familiar with the programming languages themselves and with the corresponding development tools.

The manual is divided into the following parts:

- **General Issues**
  This part contains information that is relevant for all programming interfaces, e.g., which interface to use for which programming language or how to use Parallel HALCON.

- **Programming With HALCON/C++**
  This part describes the HALCON’s language interface to C++.

- **Programming With HALCON/.NET**
  This part describes the HALCON’s language interface to .NET programming languages (C#, Visual Basic .NET, etc.).

- **Programming With HALCON/COM**
  This part describes the HALCON’s language interface to languages that can handle Microsoft COM, e.g., Visual Basic 6.0 or Delphi.

- **Programming With HALCON/C**
  This part describes the HALCON’s language interface to C.

- **Using HDevEngine**
  This part describes how to use HDevEngine to execute HDevelop programs and procedures from a programming language.
# Contents

## I  General Issues

1 Basic Information About Programming with HALCON  
   1.1 Which HALCON Interface to Use  
   1.2 Platform-Specific HALCON Versions  
   1.3 .NET Framework Security Configuration  

2 Parallel Programming and HALCON  
   2.1 Parallel Programming with Standard HALCON  
   2.2 Parallel HALCON  
   2.3 Automatic Parallelization  
   2.3.1 Initializing Parallel HALCON  
   2.3.2 The Three Methods of Automatic Parallelization  
   2.4 Parallel Programming  
   2.4.1 A Closer Look at Reentrancy  
   2.4.2 Style Guide  
   2.4.3 Examples  
   2.5 Additional Information on Parallel HALCON  
   2.5.1 Customizing the Parallelization Mechanisms  
   2.5.2 Using an Image Acquisition Interface in Parallel HALCON  
   2.5.3 Extension Packages and Parallel HALCON  
   2.5.4 Parallel HALCON and HALCON Spy  

3 Tips and Tricks  
   3.1 Monitoring HALCON Programs with HALCON Spy  

## II  Programming With HALCON/C++

4 Introducing HALCON/C++  
   4.1 A First Example  

5 Basics of the HALCON/C++ Interface  
   5.1 The Namespace HALCON  
   5.2 Calling HALCON Operators  
   5.2.1 A Closer Look at Parameters  
   5.2.2 Calling Operators via Classes  

5.2.3 The Tuple Mode
5.3 Error Handling
5.3.1 Object-Oriented Approach
5.3.2 Procedural Approach
5.4 Memory Management
5.5 How to Combine Procedural and Object-Oriented Code
5.6 I/O Streams

6 The HALCON Parameter Classes
6.1 Iconic Objects
6.1.1 Regions
6.1.2 Images
6.1.3 XLD Objects
6.1.4 Low-Level Iconic Objects
6.2 Control Parameters
6.2.1 The Basic Class for Control Parameters
6.2.2 Tuples
6.2.3 Classes Encapsulating Handles
6.3 Auxiliary Classes

7 Creating Applications With HALCON/C++
7.1 Windows
7.2 UNIX

8 Typical Image Processing Problems
8.1 Thresholding an Image
8.2 Edge Detection
8.3 Dynamic Threshold
8.4 Texture Transformation
8.5 Eliminating Small Objects
8.6 Selecting Oriented Objects
8.7 Smoothing Contours

III Programming With HALCON/.NET
9 Introducing HALCON/.NET
10 Creating Applications With HALCON/.NET
10.1 .NET Framework, Development Environments, and Example Directory Structure
10.1.1 .NET Framework Versions
10.1.2 Example Directory Structure
10.2 Adding HALCON/.NET to an Application
10.2.1 Customizing Visual Studio’s Toolbox
10.2.2 Adding a Reference to HALCON/.NET
10.2.3 Specifying the Namespace
10.3 Adding and Customizing HWindowControl for the Visualization
10.4 Using HALCON/.NET Classes to Call HALCON Operators
### 10.4.1 Online Help

81

### 10.4.2 Calling HALCON Operators

82

### 10.4.3 Declaration, Construction, Initialization, Finalization

83

### 10.4.4 Operator Overloads

85

### 10.5 Working with Tuples

87

#### 10.5.1 Calling HALCON Operators with Single or Multiple Values

87

#### 10.5.2 Iconic Tuples

88

#### 10.5.3 Control Tuples and the Class HTuple

88

### 10.6 Visualization

91

### 10.7 Error Handling

92

### 10.8 Deploying an Application

93

### 11 Additional Information

95

#### 11.1 Provided Examples

95

#### 11.2 HALCON/.NET Applications under UNIX Using Mono

96

##### 11.2.1 Restrictions

96

##### 11.2.2 Deploying HALCON/.NET Applications Created under Windows

97

##### 11.2.3 Compiling HALCON/.NET Applications with Mono

97

##### 11.2.4 Using Other GUI Libraries

98

#### 11.3 Using HDevelop Programs

98

##### 11.3.1 Using the Template Application

98

##### 11.3.2 Combining the Exported Code with the HALCON/.NET Classes

99

#### 11.4 Miscellaneous

100

##### 11.4.1 HALCON/.NET and Remote Access

100

### IV Programming With HALCON/COM

101

#### 12 Introduction

103

##### 12.1 The Microsoft Component Object Model (COM)

103

##### 12.1.1 COM and .NET

103

##### 12.1.2 A Quick Look at Some Programming Aspects

104

##### 12.2 HALCON and COM

105

#### 13 The HALCON/COM Interface

107

##### 13.1 More about Classes

107

###### 13.1.1 Different Types of Classes

108

###### 13.1.2 Classes for Special Purposes

108

##### 13.2 Object Construction and Destruction

111

###### 13.2.1 Construction

111

###### 13.2.2 Destruction

112

##### 13.3 Interfaces and Inheritance

112

##### 13.4 Methods and Properties

113

##### 13.5 A Closer Look at Data Types

113

##### 13.6 Error Handling

114

##### 13.7 HALCON/COM and Visual Basic

114

###### 13.7.1 Object Instantiation

114

###### 13.7.2 Error Handling

115
Part I

General Issues
Chapter 1

Basic Information About Programming with HALCON

This chapter contains basic information:

- which HALCON interface to use for which programming language (section 1.1)
- the available platform-specific HALCON versions, e.g., 32bit or 64bit, SSE2-optimized, etc. (section 1.2)
- special information for developers of .NET applications (section 1.3 on page 6)

1.1 Which HALCON Interface to Use

Since the introduction of HALCON/.NET, for many programming languages you can now use more than one interface. Table 1.1 guides you through these possibilities.

1.2 Platform-Specific HALCON Versions

You can use HALCON under Windows, Linux, and Solaris. The summary of system requirements is listed in table 1.2; more details follow below.

Platform-Specific HALCON Versions

For each of the operating systems listed in table 1.2, platform-specific versions of HALCON’s executables and libraries are provided. But in fact there are even more platform-specific versions: For Windows and Linux systems on processors that support the SSE2 instruction set, there is a corresponding, optimized HALCON version. Besides, for each Linux system you can choose between a HALCON version for gcc 3.3 and gcc 4.0.
Table 1.1: Which interface to use for which programming language.

<table>
<thead>
<tr>
<th>recommendation</th>
<th>alternative(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>HALCON/C</td>
</tr>
<tr>
<td>C++ (unmanaged)</td>
<td>HALCON/C++</td>
</tr>
<tr>
<td>C++ (managed)</td>
<td>HALCON/.NET, HALCON/C++</td>
</tr>
<tr>
<td>C#</td>
<td>HALCON/.NET</td>
</tr>
<tr>
<td>Visual Basic (6.0)</td>
<td>HALCON/COM</td>
</tr>
<tr>
<td>Visual Basic .NET</td>
<td>HALCON/COM</td>
</tr>
<tr>
<td>Delphi</td>
<td>HALCON/COM</td>
</tr>
<tr>
<td>Delphi .NET</td>
<td>HALCON/COM</td>
</tr>
</tbody>
</table>

Table 1.2: Platforms supported by HALCON.

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Processor</th>
<th>Compiler / Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>Intel Pentium or compatible (with and without SSE2)</td>
<td>Microsoft Visual Studio</td>
</tr>
<tr>
<td>Windows x64</td>
<td>Intel EM64T or AMD64</td>
<td>Microsoft Visual Studio 2005</td>
</tr>
<tr>
<td>Linux</td>
<td>Intel Pentium or compatible (with and without SSE2)</td>
<td>gcc 3.3, gcc 4.0</td>
</tr>
<tr>
<td>Linux x86_64</td>
<td>Intel EM64T or AMD64</td>
<td>gcc 3.3, gcc 4.0</td>
</tr>
<tr>
<td>Solaris</td>
<td>SPARC</td>
<td>Sun Studio</td>
</tr>
</tbody>
</table>

Table 1.3 on page 5 lists all platform-specific versions with detailed system requirements. The name of the currently used version is stored in the environment variable HALCONARCH.

Note that HALCON should also run on newer versions of the operating systems than the ones listed; however, we cannot guarantee this.

HALCONARCH appears in several directory paths: Executable HALCON programs like hdevelop, and DLLs like halcon.dll (Windows only), reside in %HALCONROOT%\bin\%HALCONARCH%. On Windows systems, this path is therefore automatically included in the environment variable PATH; on a Linux/UNIX system, you must include it in your login script.

The libraries that you need for linking programs, e.g., halcon.lib (Windows) or halcon.so (Linux/UNIX) reside in the directory %HALCONROOT%\lib\%HALCONARCH%.

Please note that when creating a 64bit application, both the development computer and the computer on which the application will run must be 64bit platforms. Similarly, when using one of the SSE2-optimized HALCON versions, both development and runtime computer must provide SSE2 instructions. If you run an SSE2-optimized application on a computer without SSE2, the application crashes.

On the other hand, you can use a 32bit HALCON version on a 64bit platform, and a non-SSE2-optimized
<table>
<thead>
<tr>
<th>HALCONARCH</th>
<th>Operating System, Processor</th>
<th>Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86sse2-win32</td>
<td>Windows NT/2000/XP/2003/Vista, on x86 processor with SSE2 extension, e.g., Intel Pentium 4 / AMD Athlon64 or newer</td>
<td>Visual Studio 6.0, Visual Studio .NET, Visual Studio 2005</td>
</tr>
<tr>
<td>x64-win64</td>
<td>Windows XP/2003/Vista x64 Edition, on Intel EM64T or AMD64</td>
<td>Visual Studio 2005</td>
</tr>
<tr>
<td>x86-linux2.4-gcc33</td>
<td>Linux, Kernel 2.4 or higher, libc.so.6 (GLIBC_2.3.2 or higher), libstdc++.so.5 (GLIBCPP_3.2 or higher), on Intel Pentium or compatible</td>
<td>gcc 3.2/3.3</td>
</tr>
<tr>
<td>x86-linux2.4-gcc40</td>
<td>Linux, Kernel 2.4 or higher, libc.so.6 (GLIBC_2.3.2 or higher), libstdc++.so.6 (GLIBCXX_3.4 or higher), on Intel Pentium or compatible</td>
<td>gcc 3.4/4.0/4.1</td>
</tr>
<tr>
<td>x86sse2-linux2.4-gcc33</td>
<td>Linux, Kernel 2.4 or higher, libc.so.6 (GLIBC_2.3.2 or higher), libstdc++.so.5 (GLIBCPP_3.2 or higher), on x86 processor with SSE2 extension, e.g., Intel Pentium 4 / AMD Athlon64 or newer</td>
<td>gcc 3.2/3.3</td>
</tr>
<tr>
<td>x86sse2-linux2.4-gcc40</td>
<td>Linux, Kernel 2.4 or higher, libc.so.6 (GLIBC_2.3.2 or higher), libstdc++.so.6 (GLIBCXX_3.4 or higher), on x86 processor with SSE2 extension, e.g., Intel Pentium 4 / AMD Athlon64 or newer</td>
<td>gcc 3.4/4.0/4.1</td>
</tr>
<tr>
<td>x64-linux2.4-gcc33</td>
<td>Linux x86_64, Kernel 2.4 or higher, libc.so.6 (GLIBC_2.3.4 or higher), libstdc++.so.5 (GLIBCPP_3.2 or higher), on Intel EM64T or AMD64</td>
<td>gcc 3.2/3.3</td>
</tr>
<tr>
<td>x64-linux2.4-gcc40</td>
<td>Linux x86_64, Kernel 2.4 or higher, libc.so.6 (GLIBC_2.3.4 or higher), libstdc++.so.6 (GLIBCXX_3.4 or higher), on Intel EM64T or AMD64</td>
<td>gcc 3.4/4.0/4.1</td>
</tr>
<tr>
<td>sparc-sun-solaris9</td>
<td>Solaris 9, on UltraSPARC Workstations</td>
<td>Sun Studio 9 or higher</td>
</tr>
</tbody>
</table>

Table 1.3: Values of HALCONARCH and detailed system requirements.

HALCON on an SSE2 platform without problems.

Further note that in order to create .NET applications under UNIX you need Mono.
Platform-Independent Applications

Even when using a platform-specific version of HALCON, you can still create platform-independent applications, in two ways:

- **With HDevelop**, HALCON’s integrated development environment (IDE). HDevelop programs are stored in a platform-independent format, thus, you can run them on any supported platform.

- **With HALCON/.NET**, HALCON’s interface to .NET programming languages. Applications written in .NET languages are stored in a platform-independent intermediate language, which is then converted by the so-called common language runtime into platform-specific code.

  Note that this platform independency is only true when using .NET Framework 2.0, because lower versions do not support 64bit platforms.

You can combine both methods by using HDevEngine/.NET to run HDevelop programs from a HALCON/.NET application.

1.3 .NET Framework Security Configuration

If you want to develop .NET application (independent on the used HALCON interface) in other locations than on a local disk, you must configure the .NET security policy to allow executing code from your desired project location. Otherwise, you get a warning upon creating or loading the project that the location is not trusted and a run-time error upon executing it.

You can configure the .NET security policy in two ways:

- with the command line tool caspol
- with the “Microsoft .NET Framework Configuration”:
  - Open the Control Panel and select the Administrative Tools.
  - Select .NET Framework Configuration (if there is more than one entry, select the .NET Framework version corresponding to your current development environment, see table 10.1 on page 76 for an overview).
  - In the appearing dialog, select My Computer ▶ Runtime Security Policy ▶ Machine ▶ Code Groups ▶ All Code ▶ LocalIntranet_Zone in the left column and then select Edit Code Group Properties in the right column.
  - Another dialog appears; here, select the tab Permission Set and set the permissions to FullTrust.

Note that you need administrator privileges to change the .NET security policy.

Further note that to create .NET applications in Visual Studio, you must be a member of the group 'Debugger User'.

Chapter 2

Parallel Programming and HALCON

This chapter focusses on parallel programming with HALCON in general. Of course, the emphasis lies on using Parallel HALCON (section 2.2). We start, however, with information about using Standard HALCON in parallel programs (section 2.1).

2.1 Parallel Programming with Standard HALCON

Standard HALCON is optimized for running sequential programs on single-processor or single-core computers. But, of course, you can also use it in applications that use parallel programming, e.g., perform image processing in one thread and a different activity in another thread. To enable such applications, Standard HALCON is thread-safe under Windows, Linux, and Solaris – with some exceptions that are listed below.

If, however, two threads use HALCON, they are executed one after the other, because Standard HALCON is not reentrant. In such applications, you should use Parallel HALCON (see section 2.2).

2.2 Parallel HALCON

This section explains how to use Parallel HALCON, concentrating on the main features of Parallel HALCON: automatic parallelization and the support of parallel programming.

Like Standard HALCON, Parallel HALCON can be used in two ways: You can integrate the corresponding libraries into your own C++, C, or COM programs, or you can use the parallelized variant of HDevelop, Parallel HDevelop.
Parallel HALCON and HDevelop

Like the HALCON library, HDevelop exists in two variants: Standard HDevelop uses the standard HALCON library; the corresponding executable is called hdevelop (or hdevelop.exe). The second variant uses Parallel HALCON and therefore automatically parallelizes operators if used on a multi-processor or multi-core hardware. The corresponding executable is called parhdevelop (or parhdevelop.exe); under Windows, you can start it via the menu Start ⊃ Programs ⊃ MVTec HALCON ⊃ Parallel HDevelop.

Note that Parallel HDevelop differs from Standard HDevelop only insofar that it uses the automatic parallelization mechanism. You cannot write parallel programs inside Parallel HDevelop.

Of course, a parallel version is provided not only for HDevelop but also for HDevEngine, the “engine” of HDevelop (see page 152).

Parallel HALCON in a Stand-Alone Application

To create an application using Parallel HALCON instead of Standard HALCON, you just link the parallel version of the libraries to your programs, e.g., parhalconcpp.dll instead of halconcpp.dll under Windows or libparhalcon.so and libparhalconcpp.so instead of libhalcon.so and libhalconcpp.so under UNIX. This is described in detail in the sections on creating applications with the different programming languages.

2.3 Automatic Parallelization

If Parallel HALCON is used on multi-processor or multi-core hardware, it will automatically parallelize image processing operators. Section 2.3.1 describes how to initialize Parallel HALCON in order to use this mechanism. Section 2.3.2 explains the different methods which are used by HALCON operators for their automatic parallelization.

2.3.1 Initializing Parallel HALCON

In order to adapt the parallelization mechanism optimally to the actual hardware, Parallel HALCON needs to examine this hardware once. Afterwards, HALCON programs will be automatically parallelized without needing any further action on your part. Even existing HALCON programs will run and be parallelized without needing to be changed.

You trigger this initial examination by calling the operator check_par_hw_potential (see the corresponding entry in the HALCON Reference Manuals for further information). Note, that this operator will only work correctly if called from Parallel HALCON; if you call the Standard HALCON version (e.g., from Standard HDevelop instead of Parallel HDevelop), a corresponding error message is returned. Similarly, if you call the operator on a single-processor or single-core computer, it will return an error message. As a shortcut, you may call the executable hcheck_parallel which resides in the directory %HALCONROOT%\bin\HALCONARCH.
Upon calling **check_par_hw_potential**, Parallel HALCON examines every operator that can be sped up in principle by an automatic parallelization. Each examined operator is processed several times - both sequentially and in parallel - with a changing set of input parameter values, e.g., images. The latter helps to evaluate dependencies between an operator's input parameter characteristics (e.g. the size of an input image) and the efficiency of its parallel processing. This examination may take up to 10 minutes, depending on your computer.

The extracted information is stored in the registry (under Windows) or in the file `.halcon_par_info` in the directory `$HALCONROOT` (under Linux/UNIX). Please note, that on some operating systems you need special privileges to initialize Parallel HALCON successfully, otherwise the operator **check_par_hw_potential** is not able to store the extracted information. Under Windows 2000, you need administrator privileges or the group privileges 'Power User'. Under Linux/UNIX, per default only the user who installed HALCON can create (or modify) the file `.halcon_par_info`. Please note, that the operator **check_par_hw_potential** returns without an error if it cannot store the extracted information. In contrast, the program `hcheck_parallel` checks privileges and writing permissions beforehand and returns with a corresponding error message.

### 2.3.2 The Three Methods of Automatic Parallelization

For the automatic parallelization of operators, Parallel HALCON exploits **data parallelism**, i.e., the property that parts of the input data of an operator can be processed independently of each other. Data parallelism can be found at three levels:

1. **tuple level**
   If an operator is called with iconic input parameters containing tuples, i.e., arrays of images, regions, or XLDs, it can be parallelized by distributing the tuple elements, i.e., the individual images, regions, or XLDs, on parallel threads. This method requires that all input parameters contain the **same number of tuple elements** (or contain a single iconic object or value).

2. **channel level**
   If an operator is called with input images containing multiple channels, it can be parallelized by distributing the channels on parallel threads. This method requires that all input image objects contain the **same number of channels** or a single channel image.

3. **domain level**
   Every operator can be parallelized by dividing its domain and distributing its parts on parallel threads.

The description of a HALCON operator in the Reference Manuals contains an entry called 'Parallelization Information', which specifies its behavior when using Parallel HALCON. This entry indicates whether the operator will be automatically parallelized by Parallel HALCON and by which method (tuple, channel, or domain).

### 2.4 Parallel Programming Using Parallel HALCON

Parallel HALCON supports parallel programming by being thread-safe and reentrant, i.e., different threads can call HALCON operators simultaneously without having to wait. However, not all operators
are fully reentrant. This section takes a closer look at the reentrancy of Parallel HALCON. Furthermore, it points out issues that should be kept in mind when writing parallel programs that use Parallel HALCON.

The example program `example_multithreaded1.c` in the directory `example\c` shows how to use multithreading to extract different types of components on a board in parallel using Parallel HALCON/C.

### 2.4.1 A Closer Look at Reentrancy

In fact there are different “levels” of reentrancy for HALCON operators:

1. **reentrant**
   
   An operator is fully reentrant if it can be called by multiple threads simultaneously independent of the data it is called with.
   
   Please note that you must take special care when multiple threads use the same data objects, e.g., the same image variable. In this case, you must synchronize the access to this variable manually using the corresponding parallel programming mechanisms (mutexes, semaphores). Better still is to avoid such cases as far as possible, i.e., to use local variables. Note that this is no special problem of HALCON but of parallel programming in general.

2. **local**
   
   This level of reentrancy is only relevant under Windows.
   
   Under Windows, operators marked as *local* should be called only from the thread that instantiates the corresponding objects. Typical examples are operators that use graphical I/O functions, which should only be used in the main thread. The reason is that under Windows, there exists a direct mapping between program threads and graphical elements, such as windows, dialog boxes or button controls. In short, a graphical element only exists in the context of its associated thread. This can cause severe problems (for example, hang the application) if a thread tries to perform user interactions via graphical elements that belong to another thread. For example, you may get a deadlock if one thread opens a window via `open_window` and another thread tries to get input from this window via `draw_circle`.

3. **single write multiple read**
   
   A certain group of operators should be called simultaneously only if the different calling threads work on different data. For example, threads should not try to modify the same template for pattern matching simultaneously by calling `adapt_template` with the same handle. Exactly the same applies to the case that one thread should not modify a data set that is simultaneously read by another thread. Other groups of operators with this behavior are concerned with file I/O (e.g., `write_image` – `read_image`, `fwrite_string` – `fread_string` but also non-HALCON file commands) or background estimation (e.g., `update_bg_esti` – `give_bg_esti`).
   
   As this thread behavior is not recommended quite generally, Parallel HALCON does not actively prevent it and thus saves overhead. This means that if you (accidentally) call such operators simultaneously with the same data no thread will block, but you might get unwelcome effects.

4. **mutual exclusive**
   
   Some operators cannot be called simultaneously by multiple threads but may be executed in parallel to other HALCON operators. Examples for mutual exclusive operators are `combine_roads_xld`, `pouring`, or `concat_ocr_trainf`. 
5. **completely exclusive**
   A group of operators is executed exclusively by Parallel HALCON, i.e., while such an operator is executed, all other threads cannot call another HALCON operator. Examples are all OCR/OCV operators that modify OCR classifiers, all operators that create or delete files, and the operator `reset_obj_db`. For the latter, a programmer has to assure that all operators that are to be executed before `reset_obj_db` are called have already finished, because this operator resets HALCON and therefore strongly influences its behavior.

As mentioned already, the description of a HALCON operator in the Reference Manuals contains an entry called 'Parallelization Information', which specifies its behavior when using Parallel HALCON. This entry specifies the level of reentrancy as described above.

### 2.4.2 Style Guide

The following tips are useful for multithreaded programming in general:

- **Number of threads \( \leq \) number of processors or cores**
  If you use more threads than there are processors or cores, your application might actually be slower than before because of the synchronization overhead. Note that when counting threads only the so-called worker threads are relevant, i.e., threads that are running / working continuously.

- **Local variables**
  If possible, use local variables, i.e., instantiate variables in the thread that uses them. If multiple threads use the same variable, you must synchronize their access to the variable using the appropriate parallel programming constructs (mutexes, semaphores; please refer to the documentation of your programming language for details).

  An exception are COM applications that use the so-called “Single-Threaded Apartment” mode, because here calls are synchronized automatically. This mode, however, has other disadvantages, as described in more detail in the tip for Parallel HALCON/COM below.

When using Parallel HALCON, please keep the following tips in mind:

- **Initialization**
  Before calling HALCON operators in parallel in a multithreaded program, you have to call one operator exclusively. This is necessary to allow HALCON to initialize its internal data structures.

- **I/O**
  Under Windows, use I/O operators (including graphics operators like `open_window` or `disp_image`) only in the main thread, otherwise you might get a deadlock. This means that you should not open a window in one thread and request a user interaction in it from another thread. In the Reference Manual, these operators are marked as *locally reentrant* (see section 2.4.1 on page 10).

  Keep in mind that operators which create or delete files work exclusively, i.e., other threads have to wait.

  The programmer has to assure that threads do not access the same file (or handle) simultaneously!

- **Multithreading vs. automatic parallelization**
  If you explicitly balance the load on multiple processors or cores in a multithreaded program, we recommend to switch off the automatic parallelization mechanism in order to get an optimal
Parallel programming and HALCON

Performance (or reduce the number of threads used by it so that the sum of threads does not exceed the number of processors or cores). How to switch off the automatic parallelization or reduce the number of threads is described in section 2.5.1 on page 13.

- **Parallel HALCON/COM**
  Please note that in COM applications threads are created by default using the so-called “Single-Threaded Apartment” (STA) mode. In this mode, calls to COM objects (and thereby all calls of HALCON operators) are synchronized automatically with the windows message queue. This has the drawback that all calls are executed slower than in Standard HALCON because of the synchronization overhead.

  Furthermore, in this apartment model calls to a COM object are always executed by the thread where the object was instantiated. Thus, if you instantiate HFramegrabberX in one thread and call GrabImageAsync in another, both threads are blocked during image acquisition! Therefore, it is very important to use local variables, i.e., instantiate objects in the thread that uses them (see above).

### 2.4.3 Examples

HALCON currently provides the following examples for parallel programming:

**HALCON/C**

- examples\c\example_multithreaded1.c
  two threads extract different elements on a board in parallel

**HALCON/.NET**

- examples\c#\MultiThreading (C#)
  performs image acquisition, processing, and display in three threads

**HALCON/C++**

- examples\mfc\FGMultiThreading (using MFC)
  performs image acquisition / display and processing in two threads
- examples\mfc\MultiThreading (using MFC)
  performs image acquisition, processing, and display in three threads

### 2.5 Additional Information

This section contains additional information that helps you to use Parallel HALCON.
2.5.1 Customizing the Parallelization Mechanisms

With the help of HALCON’s system parameters, which can be set and queried with the operators \texttt{set\_system} and \texttt{get\_system}, respectively, you can customize the behavior of the parallelization mechanisms.

First of all, you can check whether the application uses Parallel HALCON (and not Standard HALCON) by calling

\begin{Verbatim}
get\_system\texttt{('parallel\_halcon', Information)}
\end{Verbatim}

If Parallel HALCON is used, Information contains ‘true’, else ‘false’.

You can also query the number of processors (or cores) by calling

\begin{Verbatim}
get\_system\texttt{('processor\_num', Information)}
\end{Verbatim}

You can switch off parts of the features of Parallel HALCON with the help of the operator \texttt{set\_system}. To switch off the automatic parallelization mechanism, call (HDevelop notation, see the Reference Manual for more information)

\begin{Verbatim}
set\_system\texttt{('parallelize\_operators','false')}
\end{Verbatim}

To switch off reentrancy, call

\begin{Verbatim}
set\_system\texttt{('reentrant','false')}
\end{Verbatim}

Of course, you can switch on both behaviors again by calling \texttt{set\_system} with ‘true’ as the second parameter. Please note that when switching off reentrancy you also switch off automatic parallelization, as it requires reentrancy.

Switch off these features only if you are sure you don’t need them but want to save the corresponding computing overhead. If you write sequential programs to run on a single-processor or single-core computer, the best way to save overhead is to use Standard HALCON, of course.

A reason for switching off the automatic parallelization mechanism could be if your multithreaded program does its own scheduling and does not want Parallel HALCON to interfere via automatic parallelization. Note that you do not need to switch off automatic parallelization when using Parallel HALCON on a single-processor or single-core computer; Parallel HALCON does so automatically if it detects only one processor or core.

With the system parameter ‘parallelize\_operators’ you can customize the automatic parallelization mechanisms in more detail. Please see the description of \texttt{set\_system} for more information.

Finally, you can influence the creation of threads used for automatic parallelization with the parameters ‘thread\_num’ and ‘thread\_pool’ (\texttt{set\_system}). Reducing the number of threads is useful if you also perform a manual parallelization in your program. If you switch off automatic parallelization permanently, you should also switch off the thread pool to save resources of the operating system.
2.5.2 Using an Image Acquisition Interface in Parallel HALCON

All image acquisition devices supported by HALCON can be used in Parallel HALCON as well. Please note, that none of the corresponding operators is automatically parallelized. Most of the operators are reentrant, only the operators concerned with the connection to the device (open_framegrabber, info_framegrabber, close_framegrabber, and close_all_framegrabbers) are processed completely exclusively. Furthermore, these operators are local, i.e., under Windows they should be called from the thread that instantiates the corresponding object (see section 2.4.1 on page 10).

Under Windows, each image acquisition interface comes as two DLLs\(^1\), one for Standard HALCON (prefix hAcq) and one for Parallel HALCON (prefix parhAcq). Both HALCON versions automatically load “their” interface libraries, so you do not need to worry about it.

Under Linux/UNIX, both Standard HALCON and Parallel HALCON work with the same shared library (prefix hAcq).

2.5.3 Extension Packages and Parallel HALCON

To be used in Parallel HALCON, extension packages must be provided in a second version including the prefix par in the name of the libraries. For example, if the package is called halconuser, Parallel HALCON looks for the libraries parhalconuser.dll under Windows or libparhalconuser.so under Linux/UNIX, respectively.

More information about extension packages in Parallel HALCON can be found in the Extension Package Programmer’s Manual.

2.5.4 Parallel HALCON and HALCON Spy

As HALCON Spy (see section 3.1) is working sequentially, it cannot be used to monitor a Parallel HALCON program.

---

\(^1\) see the Installation Guide, section A.1 on page 52, for the file structure of HALCON
Chapter 3

Tips and Tricks

3.1 Monitoring HALCON Programs with HALCON Spy

HALCON Spy helps you to debug image processing programs realized with HALCON operators by monitoring calls to HALCON operators and displaying their input and output data in graphical or textual form. Furthermore, it allows you to step through HALCON programs. Note that under Windows HALCON Spy does only work in combination with a console application, i.e., you can not use it together with HDevelop or Parallel HDevelop. Furthermore, HALCON Spy does not work properly together with Parallel HALCON.

HALCON Spy is activated within a HALCON program by inserting the line

```
set_spy('mode','on')
```

Alternatively, you can activate HALCON Spy for an already linked program by defining the environment variable HALCONSPY (i.e., by setting it to any value). How to set environment variables is described in the Installation Guide, section A.2 on page 54.

You specify the monitoring mode by calling the operator `set_spy` again with a pair of parameters, for example

```
set_spy('operator','on')
set_spy('input_control','on')
```

to be informed about all operator calls and the names and values of input control parameters. The monitoring mode can also be specified via the environment variable HALCONSPY. Under Windows, set HALCONSPY to the value

```
operator=on;input_control=on
```

to get the same mode as in the example above. Under Linux/UNIX, use a `colon` instead of a semicolon to separate options.
Please take a look at the entry for `set_spy` in the HALCON Reference Manuals for detailed information on all the debugging options.
Part II

Programming With HALCON/C++
Chapter 4

Introducing HALCON/C++

HALCON/C++ is the interface of the image analysis system HALCON to the programming language C++. Together with the HALCON library, it allows to use the image processing power of HALCON inside C++ programs.

This part is organized as follows: We start with a first example program to show you how programming with HALCON/C++ looks like. Chapter 5 on page 21 then takes a closer look at the basics of the HALCON/C++ interface, while chapter 6 on page 37 gives an overview of the HALCON’s parameter classes HImage, HRegion, HWindow etc. Chapter 7 on page 61 shows how to create applications based on HALCON/C++. Finally, chapter 8 on page 67 presents typical image processing problems and shows how to solve them using HALCON/C++.

4.1 A First Example

The input image is shown in figure 4.1 on the left side. The task is to find the eyes of the monkey by segmentation. The segmentation of the eyes is performed by the C++ program listed in figure 4.2, the result of the segmentation process is shown in figure 4.1 on the right side.

The program is more or less self-explaining. The basic idea is as follows: First, all pixels of the input image are selected which have a gray value of at least 128, on the assumption that the image Mandrill is a byte image with a gray value range between 0 and 255. Secondly, the connected component analysis is performed. The result of the HALCON operator is an array of regions. Each region is isolated in the sense that it does not touch another region according to the neighbourhood relationship. Among these regions those two are selected which correspond to the eyes of the monkey. This is done by using shape properties of the regions, the size and the anisometry.

This example shows how easy it is to integrate HALCON operators in any C++ program. Their use is very intuitive: You don’t have to care about the underlying data structures and algorithms, you can ignore specific hardware requirements, if you consider e.g. input and output operators. HALCON handles the memory management efficiently and hides details from you, and provides an easy to use runtime system.
Figure 4.1: The left side shows the input image (a mandrill), and the right side shows the result of the image processing: the eyes of the monkey.

```cpp
#include "HalconCpp.h"

main()
{
    using namespace Halcon;

    HImage Mandrill("monkey"); // read image from file "monkey"
    HWindow w; // window with size equal to image

    Mandrill.Display(w); // display image in window
    w.Click(); // wait for mouse click

    HRegion Bright = Mandrill >= 128; // select all bright pixels
    HRegionArray Conn = Bright.Connection(); // get connected components

    // select regions with a size of at least 500 pixels
    HRegionArray Large = Conn.SelectShape("area","and",500,90000);

    // select the eyes out of the instance variable Large by using
    // the anisometry as region feature:
    HRegionArray Eyes = Large.SelectShape("anisometry","and",1,1.7);

    Eyes.Display(w); // display result image in window
    w.Click(); // wait for mouse click
}
```

Figure 4.2: This program extract the eyes of the monkey.
Chapter 5

Basics of the HALCON/C++ Interface

In fact, the HALCON/C++ interface provides two different approaches to use HALCON’s functionality within your C++ program: a procedural and an object-oriented approach. The procedural approach corresponds to calling HALCON operators directly as in C or HDevelop, e.g.:

```cpp
Hobject original_image, smoothed_image;
read_image(&original_image, "monkey");
mean_image(original_image, &smoothed_image, 11, 11);
```

In addition to the procedural approach, HALCON/C++ allows to call HALCON operators in an object-oriented way, i.e., via a set of classes. For example, the code from above can be “translated” into:

```cpp
HImage original_image("monkey"), smoothed_image;
smoothed_image = original_image.MeanImage(11, 11);
```

This simple example already shows that the two approaches result in clearly different code: Besides the different operator names (procedural: small letters and underscores; object-oriented: capitals), the operator calls differ in the number and type of parameters. Furthermore, functionality may be available in different ways; for example, images can be read from files via a constructor of the class HImage. In general, we recommend to use the object-oriented approach. Note, however, that HDevelop can export programs only into procedural C++ code. Section 5.5 on page 34 shows how to combine procedural with object-oriented code.

In the following sections, we take a closer look at various issues regarding the use of the HALCON/C++ interface; chapter 6 on page 37 describes the provided classes in more detail.

5.1 The Namespace HALCON

Starting with HALCON 7.1, all functions and classes of HALCON/C++ use the namespace HALCON to prevent potential name conflicts with other C++ libraries. This means that the code examples on the
previous page are incomplete, because the classes and operator calls cannot be used without specifying their namespace.

You can specify (“use”) the namespace in three ways:

- **specifically**, by prefixing each class name or operator call with the namespace
  
  ```cpp
  Halcon::Hobject original_image, smoothed_image;
  Halcon::read_image(&original_image, "monkey");
  ```

- **locally**, by placing the directive `using namespace Halcon;` at the beginning of a block, e.g., at the beginning of a function:
  
  ```cpp
  int main(int argc, char *argv[])
  {
    using namespace Halcon;

    Hobject original_image, smoothed_image;
    read_image(&original_image, "monkey");
  }
  ```

  Then, you can use HALCON’s classes and functions without prefix inside this block.

- **globally**, by placing the directive `using namespace Halcon;` directly after including `HalconCpp.h`. Then, you don’t need the prefix in your whole application.
  
  ```cpp
  #include "HalconCpp.h"
  using namespace Halcon;
  ```

Which method is the most suitable depends on your application, more exactly on what other libraries it includes and if there are name collisions.

Please note that the namespace is not mentioned in the operator descriptions in the reference manual in order to keep it readable. Similarly, in the following sections the namespace is left out.

## 5.2 Calling HALCON Operators

How a HALCON operator can be called via the HALCON/C++ interface is described in detail in the HALCON/C++ reference manual. As an example, figure 5.1 shows parts of the entry for the operator `mean_image`.

Please note that the reference manual does not list all possible signatures of the operators. A complete list can be found in the file `include/cpp/HCPPGlobal.h`.

Below, we take a closer look at the parameters of an operator call, describe how to call operators via classes, and explain another special HALCON concept, the *tuple mode*.

### 5.2.1 A Closer Look at Parameters

HALCON distinguishes two types of parameters: *iconic* and *control* parameters. *Iconic parameters* are related to the original image (images, regions, XLD objects), whereas *control parameters* are all kinds of alphanumerical values, such as integers, floating-point numbers, or strings.
5.2 Calling HALCON Operators

Error mean_image ( Hobject Image, Hobject *ImageMean, const HTuple &MaskWidth, const HTuple &MaskHeight )

HImage HImage::MeanImage ( const HTuple &MaskWidth, const HTuple &MaskHeight ) const

HImageArray HImageArray::MeanImage ( const HTuple &MaskWidth, const HTuple &MaskHeight ) const

HImage original_image("monkey"), smoothed_image;
smoothed_image = original_image.MeanImage(11, 11);

In contrast to input parameters, output parameters are always modified, thus they must be passed by reference. Note that operators expect a pointer to an already declared variable! For example, when calling the operator Find1dBarCode as in the following lines of code, variables of the class HTuple are declared before passing the corresponding pointers using the operator &.

Figure 5.1: The head and parts of the parameter section of the reference manual entry for mean_image.

A special form of control parameters are the so-called handles. A well-known representative of this type is the window handle, which provides access to an opened HALCON window, e.g., to display an image in it. Besides, handles are used when operators share complex data, e.g., the operators for shape-based matching which create and then use the model data, or for accessing input/output devices, e.g., image acquisition devices. Classes encapsulating handles are described in detail in section 6.2.3 on page 57.

Both iconic and control parameters can appear as input and output parameters of a HALCON operator. For example, the operator mean_image expects one iconic input parameter, one iconic output parameter, and two input control parameters (see figure 5.1); figure 5.2 shows an operator which has all four parameter types. Note how some parameters “disappear” from within the brackets if you call an operator via a class; this mechanism is described in more detail in section 5.2.2 on page 25.

An important concept of HALCON’s philosophy regarding parameters is that input parameters are not modified by an operator. As a consequence, they are passed by value (e.g., Hobject Image in figure 5.1) or via a constant reference (e.g., const HTuple &MaskWidth). This philosophy also holds if an operator is called via a class, with the calling instance acting as an input parameter. Thus, in the following example code the original image is not modified by the call to MeanImage; the operator’s result, i.e., the smoothed image, is provided via the return value instead:

In contrast to input parameters, output parameters are always modified, thus they must be passed by reference. Note that operators expect a pointer to an already declared variable! For example, when calling the operator Find1dBarCode as in the following lines of code, variables of the class HTuple are declared before passing the corresponding pointers using the operator &.
Herror find_1d_bar_code ( Hobject Image, Hobject * CodeRegion, const HTuple & BarCodeDescr, const HTuple & GenericName, const HTuple & GenericValue, HTuple * BarcodeFound, HTuple * BarCodeElements, HTuple * Orientation )

HRegion HImage::Find1dBarCode ( const HTuple & BarCodeDescr, const HTuple & GenericName, const HTuple & GenericValue, HTuple * BarcodeFound, HTuple * BarCodeElements, HTuple * Orientation ) const

HRegion HBarCode1D::Find1dBarCode ( const HImage & Image, const HTuple & GenericName, const HTuple & GenericValue, HTuple * BarcodeFound, HTuple * BarCodeElements, HTuple * Orientation ) const

Image (input_object) .............................................. image \( \sim\) Hobject: HImage (byte / uint2)

CodeRegion (output_object) ...................................... region \( \sim\) Hobject * : HRegion

BarCodeDescr (input_control) .............. barcode_1d-array \( \sim\) HTuple.const char * / Hlong / double

GenericName (input_control) ................. attribute.name(-array) \( \sim\) HTuple.const char *

GenericValue (input_control) ..................... attribute.value(-array) \( \sim\) HTuple.double / Hlong

BarcodeFound (output_control) ...................... integer \( \sim\) HTuple.Hlong *

BarCodeElements (output_control) ................. number-array \( \sim\) HTuple.double *

Orientation (output_control) ......................... angle.rad \( \sim\) HTuple.double *

Figure 5.2: The head and parts of the parameter section of the reference manual entry for find_1d_bar_code.

HImage image("barcode/ean13/ean1301");
HBarCode1D barcode("EAN 13", 13, 13);
HRegion code_region;
HTuple isfound, elements;

code_region = barcode.Find1dBarCode(image, HTuple(), HTuple(),
&isfound, &elements, (HTuple*) _);

The above example shows two other interesting aspects of output parameters: When calling operators via classes, one output parameter becomes the return value (see section 5.2.2 on page 25 for more details); in the example, Find1dBarCode returns the bar code region. Secondly, a special variable called _ was specified for the last parameter of Find1dBarCode. This variable is pre-declared; it tells the operator that it does not need to output the corresponding parameter.

Many HALCON operators accept more than one value for certain parameters. For example, you can call the operator MeanImage with an array of images (see figure 5.1); then, an array of smoothed images is returned. This is called the *tuple mode*; see section 5.2.3 on page 28 for more information.

Please note that **output parameters of the type string need special attention**: First of all, you must allocate memory for them yourself, e.g., by declaring them as character arrays; secondly, you don’t pass them by reference, because such parameters are pointers already. In the following example code, the operator InfoFramegrabber (see also figure 5.3) is called with two output string parameters to query the currently installed image acquisition board:
5.2 Calling HALCON Operators

```c++
Error info_framegrabber ( const HTuple &Name, const HTuple &Query, HTuple *Information, 
                        HTuple *ValueList )
```

Name (input_control) .................................................. string  \( \rightarrow \) HTuple.const char *
Query (input_control) ................................................... string  \( \rightarrow \) HTuple.const char *
Information (output_control) ......................................... string  \( \rightarrow \) HTuple.char *
ValueList (output_control) ............................................. string-array  \( \rightarrow \) HTuple.char */ Hlong */ double *

Figure 5.3: The head and parts of the parameter section of the reference manual entry for info_framegrabber.

```c++
char sInfo[MAX_STRING], sValue[MAX_STRING];

info_framegrabber(FGName, "info_boards", sInfo, sValue);
```

Note that it isn’t necessary to allocate memory for output string parameters in the already mentioned tuple mode, i.e., when using instances of the class HTuple instead of “plain” strings (also see section 5.2.3 on page 28 and section 6.2.2 on page 54):

```c++
HTuple tInfo, tValues;

info_framegrabber(FGName, "info_boards", &tInfo, &tValues);
```

5.2.2 Calling Operators via Classes

As already described in the previous section, the HALCON/C++ reference manual shows via which classes an operator can be called. For example, Find1dBarCode can be called via objects of the class HImage or HBarCode1d (see figure 5.2 on page 24). In both cases, the corresponding input parameter (Image or BarCodeDescr, respectively) does not appear within the brackets anymore as it is replaced by the calling instance of the class (this).

There is a further difference to the procedural operator signature: The first output parameter (in the example the bar code region CodeRegion) also disappears from within the brackets and becomes the return value instead of the error code (more about error handling can be found in section 5.3 on page 32).

Figure 5.4 depicts code examples for the three ways to call Find1dBarCode. When comparing the object-oriented and the procedural approach, you can see that the calls to the operators read_image and gen_1d_bar_code_descr are replaced by special constructors for the classes HImage and HBarCode1d, respectively. This topic is discussed in more detail below.

Please note that the two object-oriented methods seem to be “asymmetric”: If you call Find1dBarCode via HImage, the reference manual seems to suggest that you must pass the handle instead of an instance of HBarCode1d. In fact, you can pass both a handle and a class instance, because the latter is automatically “casted” into a handle; the signature was not changed to keep the HALCON/C++ interface backward compatible as far as possible.
26 Basics of the HALCON/C++ Interface

```cpp
HImage    image("barcode/ean13/ean1301");
HBarCode1D barcode("EAN 13", 13, 13);
HRegion   code_region;
HTuple    isfound, elements;

code_region = barcode.Find1dBarCode(image, HTuple(), HTuple(),
        &isfound, &elements, (HTuple*) _);

code_region = image.Find1dBarCode(barcode.GetHandle(), HTuple(), HTuple(),
        &isfound, &elements, (HTuple*) _);

Hobject   image;
HTuple    barcode;
Hobject   code_region;
HTuple    isfound, elements;

read_image(&image, "barcode/ean13/ean1301");
gen_1d_bar_code_descr("EAN 13", 13, 13, &barcode);
find_1d_bar_code(image, &code_region, barcode, HTuple(), HTuple(),
        &isfound, &elements, (HTuple*) _);
```

Figure 5.4: Using Find1dBarCode via HBarCode1d, via HImage, or in the procedural approach.

5.2.2.1 Constructors

As can be seen in figure 5.4, the HALCON/C++ parameter classes provide additional constructors which are based on suitable HALCON operators. The constructors for HImage and HBarCode1d used in the example are based on read_image and gen_1d_bar_code descr, respectively.

Please note that in the current HALCON version constructors are provided inconsistently for the different classes. Below we take a brief look at the most important classes. A complete and up-to-date list of available constructors can be found in the corresponding header files in %HALCONROOT%/include/cpp.

- **Images:**
  
The class HImage provides constructors based on the operators read_image, gen_image1, gen_image1_extern, and gen_image_const.

  Please **beware of the following pitfall** when using the operators themselves via HImage: Contrary to intuition, the operators do not modify the instance they are called from; instead, the created image is the return value of the operator! Thus, after the following code the image is still uninitialized:

  ```cpp
  HImage    image;
  image.ReadImage("barcode/ean13/ean1301"); // incorrect
  ```

  The correct way to call ReadImage is as follows:

  ```cpp
  image = HImage::ReadImage("barcode/ean13/ean1301"); // correct
  ```

  Note that this pitfall concerns all operators where HImage appears as an output parameter, e.g., GrabImage. More information about HImage can be found in section 6.1.2 on page 44.
5.2 Calling HALCON Operators

- **Regions:**
  The class HRegion provides constructors based on operators like gen_rectangle2 or gen_circle. However, instead of the parameters of these operators, the constructors expect instances of auxiliary classes like HRectangle2 or HCircle (see section 6.3 on page 59 for more information about these classes).

  Please note that HRegion presents the same pitfall as HImage, i.e., operators like GenRectangle2 do not modify the calling instance of HRegion but return the created region! More information about HRegion can be found in section 6.1.1 on page 37.

- **XLDs:**
  The classes for XLDs (HXLD, HXLDCont, etc., see section 6.1.3 on page 52 for more information) do not provide constructors based on operators.

- **Windows:**
  The class HWindow provides constructors based on the operators open_window and newExternWindow. Note that the former is realized with default values for all parameters, thus becoming the default constructor, i.e., all window instances are already opened upon construction!

  Of course, you can close a window using CloseWindow and then open it again using OpenWindow. In contrast to the iconic parameter classes, you can call the “constructor-like” operator OpenWindow via an instance of HWindow in the intuitive way, i.e., the calling instance is modified; in addition the corresponding handle is returned. HWindow is described in more detail in section 6.2.3.1 on page 57.

- **Other Handle Classes:**
  The other classes encapsulating handles, e.g., HBarCode1d or HFramegrabber, provide constructors in a systematic way: If a class appears as an output parameter in an operator, there automatically exists a constructor based on this operator. Thus, instances of HBarCode1d can be constructed based on gen_1d_bar_code_descr as shown in figure 5.4 on page 26, instances of HShapeModel based on create_shape_model, instances of HFramegrabber based on open_framegrabber and so on.

  In contrast to the iconic parameter classes, handle classes allow to call constructor-like operators via instances of the class in the intuitive way, i.e., the calling instance is modified. For example, you can create an instance of HBarCode1d with the default constructor and then initialize it using Gen1dBarCodeDescr as follows:

  ```
  HBarCode1D barcode;
  barcode.Gen1dBarCodeDescr("EAN 13", 13, 13);
  ```

  If the instance was already initialized, the corresponding data structures are automatically destroyed before constructing and initializing them anew (see also section 5.2.2.2). The handle classes are described in more detail in section 6.2.3.2 on page 58.

5.2.2.2 Destructors

All HALCON/C++ classes provide default destructors which automatically free the corresponding memory. For some classes, the destructors are based on suitable operators:

- **Windows:**
  The default destructor of the class HWindow closes the window based on close_window. Note
### Basics of the HALCON/C++ Interface

| Error char_threshold ( Hobject Image, Hobject HistoRegion, Hobject *Characters, const HTuple &Sigma, const HTuple &Percent, Hlong *Threshold ) |
| Error char_threshold ( Hobject Image, Hobject HistoRegion, Hobject *Characters, const HTuple &Sigma, const HTuple &Percent, HTuple *Threshold ) |
| HRegion HIImage::CharThreshold ( const HRegion &HistoRegion, const HTuple &Sigma, const HTuple &Percent, Hlong *Threshold ) const |
| HRegionArray HIImageArray::CharThreshold ( const HRegion &HistoRegion, const HTuple &Sigma, const HTuple &Percent, HTuple *Threshold ) const |

| Image (input_object) ................................. image(-array)  \( \rightarrow \) Hobject: HIImage(Array) ( byte ) |
| HistoRegion (input_object) ................................ region(-array)  \( \rightarrow \) Hobject: HRegion |
| Characters (output_object) ................................ region(-array)  \( \rightarrow \) Hobject*: HRegion(Array) |
| Sigma (input_control) ..................................... number  \( \rightarrow \) HTuple.double |
| Percent (input_control) ................................... number  \( \rightarrow \) HTuple.double / Hlong |
| Threshold (output_control) .............................. integer(-array)  \( \rightarrow \) (HTuple.) Hlong * |

**Figure 5.5:** The head and parts of the parameter section of the reference manual entry for CharThreshold.

That the operator itself is no destructor, i.e., you can close a window with **CloseWindow** and then open it again using **OpenWindow**.

- **Other Handle Classes:**
  The default destructors of the other classes encapsulating handles, e.g., HShapeModel or HFramegrabber, apply operators like **clear_shape_model** or **close_framegrabber**, respectively. In contrast to **close_window**, these operators cannot be called via instances of the class, as can be seen in the corresponding reference manual entries; the same holds for operators like **clear_all_shape_models**. In fact, there is no need to call these operators as you can initialize instances anew as described in **section 5.2.2.1**.

Please note that you must not use operators like **clear_shape_model**, **clear_all_shape_models**, or **close_framegrabber** together with instances of the corresponding handle classes!

### 5.2.3 The Tuple Mode

As already mentioned in **section 5.2.1** on page 22, many HALCON operators can be called in the so-called **tuple mode**. In this mode, you can, e.g., apply an operator to multiple images or regions with a single call. The standard case, e.g., calling the operator with a single image, is called the **simple mode**. Whether or not an operator supports the tuple mode can be checked in the reference manual. For example, take a look at **figure 5.5**, which shows an extract of the reference manual entry for the operator **char_threshold**. In the parameter section, the parameter **Image** is described as an image(-array); this signals that you can apply the operator to multiple images at once.
5.2 Calling HALCON Operators

```c++
HImage image("alpha1");
HRegion region;
long threshold;

region = image.CharThreshold(image.GetDomain(), 2, 95, &threshold);
image.Display(window);
region.Display(window);
cout << "Threshold for 'alpha1': " << threshold;
```

```c++
Hobject image;
Hobject region;
long num;
long threshold;

read_image(&image, "alpha1");
char_threshold(image, image, &region, 2, 95, &threshold);
disp_obj(image, window);
disp_obj(region, window);
cout << "Threshold for 'alpha1': " << threshold;
```

Figure 5.6: Using CharThreshold in simple mode, via HImage, or in the procedural approach (declaration and opening of window omitted).

If you call `char_threshold` with multiple images, i.e., with an image tuple, the output parameters automatically become tuples as well. Consequently, the parameters `Characters` and `Threshold` are described as `region(-array)` and `integer(-array)`, respectively.

The head section of the reference entry in figure 5.5 shows how simple and tuple mode are reflected in the operator’s signatures. In the procedural approach, the simple and tuple mode methods of calling `char_threshold` differ only in the type of the output parameter `Threshold`: a pointer to a long or to a `HTuple` of long values, respectively. Note that the class `HTuple` can also contain arrays (tuples) of control parameters of mixed type; please refer to section 6.2.2 on page 54 for more information about this class. In contrast to the control parameters, the iconic parameters remain instances of the class `Hobject` in both modes, as this class can contain both single objects and object arrays (see also section 6.1.4 on page 52).

In the object-oriented approach, simple mode and tuple mode methods use different classes for the iconic parameters: `HImage` and `HRegion` vs. `HImageArray` and `HRegionArray` (see section 6.1.1 on page 37 and section 6.1.2 on page 44 for more information about these classes). As in the procedural approach, control parameters can be of a basic type (simple mode only) or instances of `HTuple` (simple and tuple mode).

After this rather theoretic introduction, let’s take a look at example code. In figure 5.6, `char_threshold` is applied in simple mode, i.e., to a single image, in figure 5.7 to two images at once. Both examples are realized both in the object-oriented and in the procedural approach. The examples highlight some interesting points:

- **Creation and initialization of iconic arrays:**
  In the object-oriented approach, the image array can be constructed very easily by assigning the individual images to certain positions in the array using the well-known array operator[]. In the
procedural approach, you must explicitly create an empty object using `gen_empty_obj` and then add the images via `concat_obj`.

- **Access to iconic objects:**
  As expected, in the object-oriented approach, the individual images and regions are accessed via the array operator `[]`; the number of objects in an array can be queried via the method `Num()`. In the procedural approach, objects must be selected explicitly using the operator `select_obj`; the number of objects can be queried via `count_obj`.

- **Polymorphism of HObject:** (part I)
  As already noted, instances of `HObject` can be used both in simple and in tuple mode. In contrast, you must use different classes when switching from simple to tuple mode in the object-oriented approach.

- **Polymorphism of HObject:** (part II)
  The class `HObject` is used for all types of iconic objects. What’s more, image objects can be used for parameters expecting a region, as in the call to `char_threshold` in the examples; in this case, the `domain` of the image, i.e., the region in which the pixels are “valid”, is extracted automatically. In the object-oriented approach, you must extract the domain explicitly via the operator `GetDomain`.

- **Array (tuple) indices:**
  Object-oriented iconic arrays start with the index 0, the same is true for `HTuple`. In contrast, `HObject` arrays start with the index 1!!

Most of the time you will call operators in tuple mode without noticing: As soon as you divide a region into connected components via the operator `Connection`, you end up with a `HRegionArray` – thus, any subsequent processing, e.g., morphological operations like `DilationCircle` or the calculation of the region’s position using `AreaCenter` is automatically performed on all regions in the array, i.e., in tuple mode. Thus, the tuple mode is a simple mode after all!
HImageArray images;
HRegionArray regions;
HTuple thresholds;

for (int i=1; i<=2; i++)
{
    images[i-1] = HImage::ReadImage(HTuple("alpha") + i);
}

regions = images.CharThreshold(images[0].GetDomain(), 2, 95, &thresholds);

for (int i=0; i<images.Num(); i++)
{
    images[i].Display(window);
    regions[i].Display(window);
    cout << "Threshold for 'alpha" << i+1 << ": " " << thresholds[i].L();
    window.Click();
}

Hobject images, image;
Hobject regions, region;
long num;
HTuple thresholds;
generate_empty_obj(&images);

for (int i=1; i<=2; i++)
{
    read_image(&image, HTuple("alpha") + i);
    concatenate_obj(images, image, &images);
}

char_threshold(images, image, &regions, 2, 95, &thresholds);
count_obj(images, &num);

for (int i=0; i<num; i++)
{
    select_obj(images, &image, i+1);
    display_obj(image, window);
    select_obj(regions, &region, i+1);
    display_obj(region, window);
    cout << "Threshold for 'alpha" << i+1 << ": " " << thresholds[i].L();
}

Figure 5.7: Using CharThreshold in tuple mode, via HImageArray, or in the procedural approach (declaration and opening of window omitted).
5.3 Error Handling

In case of a runtime error, HALCON/C++ by default prints a corresponding error message and terminates the program. In some applications, however, it might be useful to slacken this rule for certain errors. For example, if an application allows the user to specify an image file to read interactively, it would be inconvenient if the application terminates because the user misspelled the file name. Therefore, HALCON/C++ allows to integrate your own error handling. How to do this in the object-oriented and in the procedural approach is described in the following sections. Please note that you cannot mix object-oriented and procedural error handling.

5.3.1 Object-Oriented Approach

If a runtime error occurs in an object-oriented operator call, an instance of the class \texttt{HException} is created (see figure 5.8 for the declaration of the class). This instance contains all information concerning the error. The important members of an exception are:

- \texttt{line}: Number of the program line in which the error occurred
- \texttt{file}: Name of the file in which the error occurred
- \texttt{proc}: Name of the actual HALCON operator
- \texttt{err}: Number of the error, see below
- \texttt{message}: Error text

After the generation, the instance of \texttt{HException} is passed to a so-called \textit{exception handler}. HALCON’s default exception handler prints the corresponding error message and terminates the program.

As an alternative, you can implement and use your own exception handler. In order to act as a HALCON exception handler, a procedure must have the following signature:
You “install” your exception handler procedure via `HException`’s class method `InstallHHandler` (see figure 5.8). In case of a runtime error, HALCON then calls your procedure, passing the instance of the actual exception as a parameter.

The following example shows how to use a user-specific exception handler together with the standard C++ exception handling mechanism (try...catch). The corresponding program `example_errorhandling.cpp` can be found in the subdirectory `%HALCONROOT%/examples/cpp`. It realizes the application mentioned above: You can type in image files to load; if a file does not exist, the program prints a corresponding message but continues nevertheless.

At the beginning of the program, a user-specific exception handler is installed with the following line:

```cpp
HException::InstallHHandler(&MyHalconExceptionHandler);
```

The installed procedure simply hands the exception object to the C++ exception handling via `throw`:

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

The call to `ReadImage` is then encapsulated by a try block; a possibly ensuing exception is then evaluated in a corresponding catch block:

```cpp
Herror error_num;

try
{
    image = HImage::ReadImage(filename);
}
catch (HException &except)
{
    error_num = except.err;
    return error_num;
}
return H_MSG_TRUE;
```

### 5.3.2 Procedural Approach

As can be seen in the extracts of the reference manual in section 5.2 on page 22, in the procedural approach operators return a value of the type `Herror`. This value can fall into two categories: messages `H_MSG_*` and errors `H_ERR_*`. There are four different messages:

- `H_MSG_TRUE`: The operator terminated without an error and the result value is the boolean value `true`.
- `H_MSG_FALSE`: The operator terminated without an error and the result value is the boolean value `false`. 
**5.4 Memory Management**

All of HALCON’s classes, i.e., not only HImage, HRegion, HTuple, HFramegrabber etc., but also the class HObject used when calling operators in the procedural approach, release their allocated memory automatically in their default destructor (see also section 5.2.2.2 on page 27). Furthermore, when constructing instances anew, e.g., by calling Gen1dBarCodeDescr via an already initialized instance as mentioned in section 5.2.2.1 on page 26, the already allocated memory is automatically released before allocating it anew. Thus, there is no need to call the operator clear_obj in HALCON/C++; what’s more, if you do use it HALCON will complain about already released memory.

The only occasion for explicit memory management on your part is when using handles in the procedural approach: The memory allocated when creating a handle, e.g., with open_framegrabber, is only released when calling the “complementary” operator, in the example close_framegrabber — or at the end of the program.

**5.5 How to Combine Procedural and Object-Oriented Code**

As already noted, we recommend to use the object-oriented approach wherever possible. However, there are some reasons for using the procedural approach, e.g., if you want to quickly integrate code that is...
exported by HDevelop, which can only create procedural code. Besides, currently some operators are only available in procedural form, e.g., operators creating affine transformations like `vector_to_rigid`.

The least trouble is caused by the basic control parameters as both approaches use the elementary types `long` etc. and the class `HTuple`. Iconic parameters and handles can be converted as follows:

- **Converting `Hobject` into iconic parameter classes**

```cpp
Hobject p_image;
read_image(&p_image, "barcode/ean13/ean1301");

HImage o_image(p_image);
```

Iconic parameters can be converted from `Hobject` to, e.g., `HImage` simply by calling the constructor with the procedural variable as a parameter.

- **Converting handles into handle classes**

```cpp
HTuple p_barcode;
gen_1d_bar_code descr("EAN 13", 13, 13, &p_barcode);

HBarCode1D o_barcode;
o_barcode.SetHandle(p_barcode);
o_code_region = o_barcode.Find1dBarCode(o_image, HTuple(), HTuple(),
    &isfound, &elements,
    (HTuple*)_);
```

Handles cannot be converted directly via a constructor; instead, you call the method `SetHandle()` with the procedural handle as a parameter.

- **Converting handle classes into handles**

```cpp
p_barcode = o_barcode.GetHandle();
decode_1d_bar_code(elements, p_barcode, (HTuple*)_ , &result, (HTuple*)_);
```

Similarly, a handle can be extracted from the corresponding class via the method `GetHandle()`. You can even omit the method, as the handle classes provide cast operators which convert them automatically into handles.

```cpp
p_barcode = o_barcode;
```

- **Converting iconic parameter classes into `Hobject`**

```cpp
Hobject p_code_region = o_code_region.Id();
```

Iconic parameters can be converted from classes like `HRegion` back into `Hobject` via the method `Id()`. In contrast to the handle classes no cast operator is provided.
• **Converting HWindow into a window handle**

```c
long p_window;
open_window(0, 0, width/2, height/2, 0, "visible", ",", &p_window);

HWindow o_window(0, 0, 100, 100, 0, "visible", ",");

p_window = o_window.WindowHandle();
disp_obj(p_code_region, p_window);
```

In contrast to other handles, procedural window handles cannot be converted into instances of the class HWindow! However, you can extract the handle from an instance of HWindow via the method WindowHandle().

As already remarked in section 5.2.2.2 on page 27, you must not use operators like `clear_shape_model`, `clear_all_shape_models`, or `close_framegrabber` together with instances of the corresponding handle classes!

### 5.6 I/O Streams

Starting with HALCON 7.1, HALCON/C++ no longer provides iostream operators by default because this created problems for applications that want to use the older iostream interface (i.e., that want to use `<iostream.h>` instead of `<iostream>`). Consequently, the HALCON/C++ library now does not provide operators for either interface.

To enable backwards compatibility, the iostream operators for the class HTuple are now provided as in-line functions in a new header file HIOStream.h, which can optionally be included. HIOStream.h uses the newer interface `<iostream>` by default, but uses the older interface `<iostream.h>` if USE_IOSTREAM_H is defined before HIOStream.h is included. Note that it may be necessary to include the statement `using namespace std;`.

To use the iostream operators, we recommend to insert the following lines of code after including HalconCpp.h:

```c
#include "HalconCpp.h"
#include "HIOStream.h"
#if !defined(USE_IOSTREAM_H)
using namespace std;
#endif
```
Chapter 6

The HALCON Parameter Classes

6.1 Iconic Objects

The base class of the iconic parameter classes in HALCON/C++ is the (abstract) class \texttt{HObject} which manages entries in the database, i.e., the copying or releasing of objects. The entries themselves are represented by the class \texttt{HObject} (see also section 6.1.4 on page 52). The classes \texttt{HObject} and \texttt{Hobject} can contain all types of iconic objects. This has the advantage that important methods like \texttt{Display()} can be applied to all iconic objects in the same manner.

Three classes are derived from the root class \texttt{HObject}:

- Class \texttt{HImage} for handling images.
- Class \texttt{HRegion} for handling regions.
- Class \texttt{HXLD} for handling polygons.

These classes are described in more detail below.

6.1.1 Regions

A region is a set of coordinates in the image plane. Such a region does not need to be connected and it may contain holes. A region can be larger than the actual image format. Regions are represented by the so-called runlength coding in HALCON. The class \texttt{HRegion} represents a region in HALCON/C++. Besides those operators that can be called via \texttt{HRegion} (see also section 5.2.2 on page 25), \texttt{HRegion} provides the following member functions:

- \texttt{HRegion(\texttt{void})}
  Default constructor. It creates an empty region, i.e., the area of this region is zero. Not all operators can handle the empty region as input, e.g. some shape property operators.

- \texttt{HRegion(const HDChord &line)}
  Constructing a region from a chord. A chord is a horizontal line.
- `HRegion(const HDPoint2D &point)`
  Constructing a region from a discrete 2-dimensional point.

- `HRegion(const HRectangle1 &rect)`
  Constructing a region from a rectangle parallel to the coordinate axis. The coordinates do not need to be discrete.

- `HRegion(const HRectangle2 &rect)`
  Constructing a region from an arbitrarily oriented rectangle. The coordinates do not need to be discrete.

- `HRegion(const HCircle &circle)`
  Constructing a region from a circle. The radius and center do not need to be discrete.

- `HRegion(const HEllipse &ellipse)`
  Constructing a region from an arbitrarily oriented ellipse. The radii and center do not need to be discrete.

- `HRegion(const char *file)`
  Constructing a region by reading the representation from file. This file can be generated by the member function `WriteRegion`.

- `HRegion(const HRegion &reg)`
  Copy constructor.

- `HRegion &operator = (const HRegion &reg)`
  Assignment operator.

- `~HRegion(void)`
  Destructor.

- `void Display(const HWindow &w) const`
  Output of the region in a window.

- `HRegion operator * (double scale) const`
  Zooming the region by an arbitrary factor. The center of scaling is the origin (0, 0).

- `HRegion operator >> (double radius) const`
  `HRegion &operator >>= (double radius)`
  Erosion of the region with a circle of radius `radius`, see reference manual entry of `erosion_circle`.

- `HRegion operator << (double radius) const`
  `HRegion &operator <<= (double radius)`
  Dilation of the region with a circle of radius `radius`, see reference manual entry `dilation_circle`.

- `HRegion operator + (const HDPoint2D &point) const`
  `HRegion &operator += (const HDPoint2D &point)`
  Translating the region by a 2-dimensional point.

- `HRegion &operator ++ (void)`
  Dilation of the region with a cross containing five points.
- HRegion operator + (const HRegion &reg) const
  HRegion &operator += (const HRegion &reg)
  Minkowsky addition of the region with another region, see reference manual entry of minkowski_add1.

- HRegion operator - (const HRegion &reg) const
  HRegion &operator -= (const HRegion &reg)
  Minkowsky subtraction of the region with another region, see reference manual entry of minkowski_sub1.

- HRegion &operator -- (void)
  Erosion of the region with a cross containing five points.

- HRegion operator ~ (void) const
  Complement of the region, see reference manual entry of complement.

- HRegion operator ! (void) const
  Transpose the region at the origin, see reference manual entry of transpose_region.

- HRegion operator & (const HRegion &reg) const
  HRegion &operator &= (const HRegion &reg)
  Intersection of the region with another region, see reference manual entry of intersection.

- HRegion operator | (const HRegion &reg) const
  HRegion &operator |= (const HRegion &reg)
  Union of the region with another region, see reference manual entry of union2.

- HRegion operator / (const HRegion &reg) const
  HRegion &operator /= (const HRegion &reg)
  Subtract another region from the region, see reference manual entry of difference.

- HBool operator == (const HRegion &reg) const
  Boolean test if two regions are identical, see reference manual entry of test_equal_region.

- HBool operator >= (const HRegion &reg) const
  HBool operator > (const HRegion &reg) const
  HBool operator <= (const HRegion &reg) const
  HBool operator < (const HRegion &reg) const
  Boolean test if another region is included in the region by using the subset of the corresponding coordinates.

- double Phi(void) const
  Orientation of the region by using the angle of the equivalent ellipse, see reference manual entry of elliptic_axis.

- double Ra(void) const
  Length of the major axis of the equivalent ellipse of the region, see reference manual entry of elliptic_axis.

- double Rb(void) const
  Length of the minor axis of the equivalent ellipse of the region, see reference manual entry of elliptic_axis.
The HALCON Parameter Classes

- long Area(void) const
  Area of the region, i.e., number of pixels, see reference manual entry of area_center.

- double X(void) const
double Y(void) const
  Center point of the region, see reference manual entry of area_center.

- double Contlength(void) const
  Length of the contour of the region, see reference manual entry of contlength.

- double Compactness(void) const
  Compactness of the actual region, see reference manual entry of compactness.

- double Anisometry(void) const
double Bulkiness(void) const
double StructureFactor(void) const
  Shape factors, see reference manual entry of eccentricity.

- double M11(void) const
double M20(void) const
double M02(void) const
double Ia(void) const
double Ib(void) const
  Moments of the region, see reference manual entry of moments_region_2nd.

- HRectangle1 SmallestRectangle1(void) const
  Smallest surrounding rectangle parallel to the coordinate axis, see reference manual entry of smallest_rectangle1.

- HBool In(const HDPoint2D &p) const
  Boolean test if a point is inside a region, see reference manual entry of test_region_point.

- HBool IsEmpty(void) const;
  Boolean test if the region is empty, i.e., the area of the region is zero.

A program shows the power of the class HRegion, see figure 6.1.

First, an aerial image (mreut.png) is read from a file. All pixels with a gray value ≥ 190 are selected. This results in one region (region).

This region is transformed by the next steps: All holes in the region are filled (Fillup), small parts of the region are eliminated by two morphological operations, first an erosion, a kind of shrinking the region, followed by a dilation, a kind of enlarging the region. The last step is the zooming of the region. For that the region is first shifted by a translation vector (-100, -150) to the upper left corner and then zoomed by the factor two. Figure 6.2 shows the input image and the result of the opening operation.

6.1.1.1 Region Arrays

The class HRegionArray serves as container class for regions. Besides those operators that can be called via HRegionArray (see also section 5.2.2 on page 25), HRegionArray provides the following member functions:
#include "HalconCpp.h"
using namespace Halcon;

main ()
{
    HImage image("mreut"); // Reading an aerial image
    HRegion region = image >= 190; // Calculating a threshold
    HWindow w; // Display window
    w.SetColor("red"); // Set color for regions
    region.Display(w); // Display the region
    HRegion filled = region.FillUp(); // Fill holes in region
    filled.Display(w); // Display the region
    // Opening: erosion followed by a dilation with a circle mask
    HRegion open = (filled >> 4.5) << 4.5;
    w.SetColor("green"); // Set color for regions
    open.Display(w); // Display the region
    HDPoint2D trans(-100,-150); // Vector for translation
    HRegion moved = open + trans; // Translation
    HRegion zoomed = moved * 2.0; // Zooming the region
}

Figure 6.1: Sample program for the application of the class HRegion.

Figure 6.2: On the left the input image (mreut.png), and on the right the region after the opening (open).

- **HRegionArray(void)**
  Constructor for an empty array (Num() is 0).

- **HRegionArray(const HRegion &reg)**
  Constructor with a single region.
• HRegionArray(const HRegionArray &arr)
  Copy constructor.

• "HRegionArray(void)
  Destructor.

• HRegionArray &operator = (const HRegionArray &arr)
  Assignment operator.

• long Num(void)
  Number of regions in the array, largest index is Num() − 1.

• HRegion const &operator [] (long index) const
  Reading the element $i$ of the array. The index is in the range $0 \ldots \text{Num()} - 1$.

• HRegion &operator [] (long index)
  Assigning a region to the element $i$ of the array. The index $i$ can be $\geq \text{Num()}$.

• HRegionArray operator () (long min, long max) const
  Selecting a subset between the lower min and upper max index.

• HRegionArray &Append(const HRegion &reg)
  Appending another region to the region array.

• HRegionArray &Append(const HRegionArray &reg)
  Appending another region array to the region array.

• void Display(const HWindow &w) const
  Display the regions of the array in a window.

• HRegionArray operator << (double radius) const
  Applying a dilation to all regions using a circular mask, see reference manual entry of dilation_circle.

• HRegionArray operator >> (double radius) const
  Applying an erosion to all regions using a circular mask, see reference manual entry of erosion_circle.

• HRegionArray operator + (const HRegion &reg) const
  Applying the Minkowsky addition to all regions using another region as mask, see reference manual entry of minkowski_add1.

• HRegionArray operator - (const HRegion &reg) const
  Applying the Minkowsky subtraction to all regions using another region as mask, see reference manual entry of minkowski_sub1.

• HRegionArray operator ~ (void) const
  Applying the complement operator to each region of the array, see reference manual entry of complement.

• HRegionArray operator & (const HRegionArray &reg) const
  Intersection of each region of the actual array with the union of $\text{reg}$, see reference manual entry of intersection.
• **HRegionArray operator | (const HRegionArray &reg) const**
  Union of each region in the actual array with the union of reg, see reference manual entry of \texttt{union2}.

• **HRegionArray operator / (const HRegionArray &reg) const**
  Difference of each region in the actual array with the union of reg, see reference manual entry of \texttt{difference}.

Most HALCON operators expecting a region for an input parameter accept an instance of \texttt{HRegionArray}, e.g. \texttt{Union1, Intersection, Difference}, etc. The constructor instantiating the region array \texttt{HRegionArray} from a single region \texttt{HRegion} makes it possible to handle operators expecting a single region: Without changing the data structure a \texttt{HRegionArray} can be used as input parameter even in the case of a single region.

\textbf{Figure 6.3} shows a short example how to use the class \texttt{HRegionArray}.

```cpp
#include "HalconCpp.h"
using namespace Halcon;

main ()
{
  HImage image("control_unit"); // Reading an image from file
  // Segmentation by regiongrowing
  HRegionArray regs = image.Regiongrowing(1,1,4,100);
  HWindow w; // Display window
  w.SetColored(12); // Set colors for regions
  regs.Display(w); // Display the regions
  HRegionArray rect; // New array
  for (long i = 0; i < regs.Num(); i++) // For all regions in array
  { // Test size and shape of each region
    if ((regs[i].Area() > 1000) && (regs[i].Compactness() < 1.5)) // If test true, append region
      rect.Append(regs[i]);
  }
  image.Display(w); // Display the image
  rect.Display(w); // Display resulting regions
}
```

\textbf{Figure 6.3}: Sample program for use of the class \texttt{HRegionArray}.

The first step is to read an image. In this case it shows a control unit in a manufacturing environment, see \textbf{figure 6.4} on the left side. By applying a regiongrowing algorithm from the HALCON library the image is segmented into regions. Each region inside the resulting region array \texttt{regs} is now selected according to its size and its compactness. Each region of a size larger than 1000 pixels and of a compactness value smaller than 1.5 is appended to the region array \texttt{rect}. After the processing of the \texttt{for} loop only the regions showing on the right side of \textbf{figure 6.4} are left.
6.1.2 Images

There is more to HALCON images than just a matrix of pixels: In HALCON, this matrix is called a *channel*, and images may consist of one or more such channels. For example, gray value images consist of a single channel, color images of three channels. Channels can not only contain the standard 8 bit pixels (pixel type `byte`) used to represent gray value images, HALCON allows images to contain various other data, e.g. 16 bit integers (type `int2`) or 32 bit floating point numbers (type `real`) to represent derivatives. Besides the pixel information, each HALCON image also stores its so-called *domain* in form of a HALCON region. The domain can be interpreted as a region of interest, i.e., HALCON operators (with some exceptions) restrict their processing to this region.

6.1.2.1 Image Objects

The class `HImage` is the root class for all derived image classes. By using the class `HImage` all different pixel types can be handled in a unique way (polymorphism). The class `HImage` is not virtual, thus it can be instantiated. Besides those operators that can be called via `HRegion` (see also section 5.2.2 on page 25), `HRegion` provides the following member functions:

- `HImage(void)`
  Default constructor, empty image.

- `HImage(const char *file)`
  Constructing an image by reading from a file, see reference manual entry of `read_image`.

- `HImage(int width, int height, const char *type)`
  Constructing an image of a defined size and a specific pixel type, see reference manual entry of `gen_image_const`. 

Figure 6.4: On the left side the input image (`fabrik.png`), and on the right side the selected rectangles.
• **HImage(void *ptr, int width, int height, const char *type)**
  Constructing an image of a defined size and a specific pixel type by copying memory, see reference manual entry of `gen_image1`.

• **HImage(const HImage &image)**
  Copy constructor.

• **virtual ~HImage(void)**
  Destructor.

• **HImage &operator = (const HImage &arr)**
  Assignment operator.

• **virtual const char *PixType(void) const**
  Return the pixel type of the image, see reference manual entry of `get_image_pointer1`.

• **int Width(void) const**
  Return the width of the image, see reference manual entry of `get_image_pointer1`.

• **int Height(void) const**
  Return the height of the image, see reference manual entry of `get_image_pointer1`.

• **HPixVal GetPixVal(int x, int y) const**
  Access a pixel value via the \((x, y)\) coordinates, see reference manual entry of `get_grayval`.

• **HPixVal GetPixVal(long k) const**
  Linear access of a pixel value.

• **virtual void SetPixVal(int x, int y, const HPixVal &val)**
  Set the pixel value via the \((x, y)\) coordinates, see reference manual entry of `set_grayval`.

• **virtual void SetPixVal(long k, const HPixVal &val)**
  Set the pixel value by linear access.

• **virtual void Display(const HWindow &w) const**
  Display an image in a window.

• **HImage operator & (const HRegion &reg) const**
  Reduce the domain of an image, see reference manual entry of `reduce_domain`.

• **HImage operator + (const HImage &add) const**
  Adding two images, see reference manual entry of `add_image`.

• **HImage operator - (const HImage &sub) const**
  Subtracting two images, see reference manual entry of `sub_image`.

• **HImage operator * (const HImage &mult) const**
  Multiplication of two images, see reference manual entry of `mult_image`.

• **HImage operator - (void) const**
  Inverting the values of the image, see reference manual entry of `invert_image`.
HImage operator / (double div) const
Arithmetic operators, see reference manual entry of scale_image.

- HRegion operator >= (const HImage &image) const
  HRegion operator <= (const HImage &image) const
Selecting all pixel with gray values brighter than or equal to (or darker than or equal to, respectively) those of the input image, see reference manual entry of dyn_threshold.

- HRegion operator >= (double thresh) const
  HRegion operator <= (double thresh) const
  HRegion operator == (double thresh) const
  HRegion operator != (double thresh) const
Selecting all pixel with gray values brighter than or equal to (or darker than or equal to, or equal to, or not equal to, respectively) a threshold, see reference manual entry of threshold.

Figure 6.5 gives an example of the use of the class HImage.

```cpp
#include "HalconCpp.h"
using namespace Halcon;
#include "HIOstream.h"
#if !defined(USE_IOSTREAM_H)
using namespace std;
#endif

main ()
{
    HImage image("mreut"); // Aerial image
    HWindow w; // Output window
    image.Display(w); // Display image
    // Returning the size of the image
    cout << "width = " << image.Width();
    cout << "height = " << image.Height() << endl;
    // Interactive drawing of a region by using the mouse
    HRegion mask = w.DrawRegion();
    // Reduce the domain of the image to the mask
    HImage reduced = image & mask;
    w.ClearWindow(); // Clear the window
    reduced.Display(w); // Display the reduced image
    // Applying the mean filter in the reduced image
    HImage mean = reduced.MeanImage(61,61);
    mean.Display(w);
    HRegion reg = bild >= (mean + 3);
    reg.Display(w);
}
```

Figure 6.5: Sample program for the use of the class HImage.

The example starts by reading a byte image from a file. The aim is to extract bright parts from the image. The used filter and the segmentation process itself is applied only in a pre-chosen part of the image in order to accelerate the runtime. This part is selected by drawing an arbitrary region with the mouse. This
6.1 Iconic Objects

Figure 6.6: On the left side the input image (mreut.png), and on the right side the segmented regions in the selected image domain.

region mask serves as input for reducing the domain of the original image (operator &). The mean filter with a mask size of 61 × 61 is applied to the resulting region reduced. Bright pixels are selected by applying the operator >=. All pixels brighter than the filtered part of the image reduced +3 are selected. Figure 6.6 shows the result of the sample program in figure 6.5.

6.1.2.2 Pixel Values

The class HPixVal is used for accessing the pixel values of the class HImage. Gray values can be set and returned independent of their types:

- HPixVal(void)
  Default constructor.

- HPixVal(const HComplex &Val)
  Constructing a pixel value from a complex number.

- HPixVal(int Val)
  Constructing a pixel value from an integer (int).

- HPixVal(long Val)
  Constructing a pixel value from a long (long).

- HPixVal(HByte Val)
  Constructing a pixel value from a byte (byte).

- HPixVal(double Val)
  Constructing a pixel value from a double (double).
```cpp
#include "HalconCpp.h"
#include "HIOStream.h"
#if !defined(USE_IOSSTREAM_H)
using namespace std;
#endif
using namespace Halcon;

main ()
{
    HByteImage in("mreut");  // Aerial image
    HWindow w;                // Output window
    in.Display(w);           // Displaying the image
    HByteImage out = in;      // Copying the image
    int width = out.Width();  // Width of the image
    int height = out.Height(); // Height of the image
    long end = width * height; // Number of pixel of the image

    // 1. run: linear accessing
    for (long k = 0; k < end; k++) {
        int pix = in.GetPixVal(k); // Reading the pixel
        out.SetPixVal(k, 255 - pix); // Setting the pixel
    }
    // Displaying the transformation
    cout << "Transformed !" << endl; out.Display(w); w.Click();
    cout << "Original !" << endl; in.Display(w); w.Click();

    // 2. run: accessing the image via the coordinates (x,y)
    for (int y=0; y<height; y++) {
        for (int x=0; x<width; x++) {
            int pix = in.GetPixVal(x, y); // Reading the pixel
            out.SetPixVal(x, y, 255 - pix); // Setting the pixel
        }
    }
    // Displaying the transformation
    cout << "Transformed !" << endl; out.Display(w); w.Click();
    cout << "Original !" << endl; in.Display(w); w.Click();
}
```

Figure 6.7: Sample program for the use of the class HPixVal.

- **HPixVal(const HPixVal &Val)**
  Copy constructor.

- **HPixVal &operator = (const HPixVal &grey)**
  Assignment operator.

- **operator HByte(void) const**
  Converting a pixel value to byte (0...255).
• operator int(void) const
Converting a pixel value to int.

• operator long(void) const
Converting a pixel value to long.

• operator double(void) const
Converting a pixel value to double.

• operator HComplex(void) const
Converting a pixel value to Complex.

The handling of the class HPixVal is explained by an example in figure 6.7 which inverts an input image. The input image is a byte image. First, a copy is generated and the image size is determined. In the first run the pixels are accessed linearly. In the second run the pixel are accessed via the \((x, y)\)-coordinates.

### 6.1.2.3 Image Arrays

The same way which was used to define arrays of regions is used to obtain arrays of images. The class is named HImageArray and contains the following member functions (in addition to the operators):

• HImageArray(void)
  Default constructor: empty array, no element.

• HImageArray(const HImage &reg)
  Constructing an image array from a single image.

• HImageArray(const HImageArray &arr)
  Copy constructor.

• ~HImageArray(void)
  Destructor.

• HImageArray &operator = (const HImageArray &arr)
  Assignment operator.

• long Num(void) const
  Returning the number of elements in the array.

• HImage const &operator [] (long index) const
  Reading the element \(i\) of the array. The index is in the range \(0 \ldots \text{Num}() - 1\).

• HImage &operator [] (long index)
  Assigning an image to the element \(i\) of the array. The index \(\text{index}\) can be \(\geq \text{Num}()\).

• HImageArray operator () (long min, long max)
  Selecting a subset between the lower \(\text{min}\) and upper \(\text{max}\) index.

• HImageArray &Append(const HImage &image)
  Appending another image to the image array.

• HImageArray &Append(const HImageArray &images)
  Appending another image array to the image array.
#include "HalconCpp.h"
#include "HIOStream.h"
#if !defined(USE_IOSTREAM_H)
using namespace std;
#endif
using namespace Halcon;

main()
{
    HByteImage in("mreut"); // Aerial image
    HWindow w; // Output window
    in.Display(w); // Displaying the image
    HImageByte out = in; // Copying the image
    int width = out.Width(); // Width of the image
    int height = out.Height(); // Height of the image
    long end = width * height; // Number of pixel of the image

    // 1. run: linear accessing
    for (long k = 0; k < end; k++)
        out[k] = 255 - in[k]; // Reading and setting the pixel

    // Displaying the transformation
    cout << "Transformed!" << endl; out.Display(w); w.Click();
    cout << "Original!" << endl; in.Display(w); w.Click();

    // 2. run: accessing the image via the coordinates (x,y)
    for (int y=0; y<height; y++)
        for (int x=0; x<width; x++)
            out(x,y) = 255 - out(x,y); // Reading and setting the pixel

    // Displaying the transformation
    cout << "Transformed!" << endl; out.Display(w); w.Click();
    cout << "Original!" << endl; in.Display(w); w.Click();
}

Figure 6.8: Sample program for accessing a pixel value using the class HByteImage.

6.1.2.4 Byte Images

For each pixel type, there exists a corresponding image class derived from HImage, e.g., HByteImage for the pixel type byte (standard 8 bit pixels) or HInt2Image for the pixel type int2 (unsigned 16 bit pixels). The most important derived class is naturally HByteImage, as this pixel type still covers the majority of all applications in the field of image processing. The advantage of the class HByteImage in comparison to the class HImage is the simplified access to the pixel values. This is because the class HPixVal is not necessary. Besides the member functions of HImage, the class HByteImage contains the following extensions:

- HByteImage(void)
  Default constructor.
• HByteImage(const char *file)
  Constructing a byte image by reading a file.

• HByteImage(int width, int height)
  Constructing an empty byte image of a given size.

• HByteImage(HByte *ptr, int width, int height)
  Constructing a byte image by copying memory.

• HByteImage(const HByteImage &image)
  Copy constructor.

• virtual ~HByteImage(void)
  Destructor.

• HByte &operator[] (long k)
  Setting a pixel value by linear accessing.

• HByte operator[] (long k) const
  Reading a pixel value by linear accessing.

• HByte &operator() (long k)
  Setting a pixel value by linear accessing.

• HByte operator() (long k) const
  Reading a pixel value by linear accessing.

• HByte &operator()(int x, int y)
  Setting a pixel value by accessing it via \((x, y)\) coordinates.

• HByte operator()(int x, int y) const
  Reading a pixel value by accessing it via \((x, y)\) coordinates.

• HByteImage operator & (int i)
  Applying the logical “and”-operation on each pixel with \(i\).

• HByteImage operator << (int i)
  Applying a left-shift on each pixel with \(i\).

• HByteImage operator >> (int i)
  Applying a right-shift on each pixel with \(i\).

• HByteImage operator ~ (void)
  Complement of each pixel.

• HByteImage operator & (HByteImage &ima)
  Pixel by pixel logical “and”-operation of two images.

• HByteImage operator | (HByteImage &ima)
  Pixel by pixel logical “or”-operation of two images.

• HByteImage operator ^ (HByteImage &ima)
  Pixel by pixel logical “xor”-operation of two images.

The advantage of the class HByteImage can be seen when accessing each pixel, see figure 6.8. The class HPixVal is not necessary in this example. Furthermore, the member functions GetPixVal and
SetPixVal are not used. HByteImage allows to access pixel values in a notation like in the programming language C. The result of the example in figure 6.8 is basically the same as in the example in figure 6.7 on page 48. The program in figure 6.8 is shorter, easy to read, and has a better runtime performance.

6.1.3 XLD Objects

XLD is the abbreviation for eXtended Line Description. This is a data structure used for describing areas (e.g., arbitrarily sized regions or polygons) or any closed or open contour, i.e., also lines. In contrast to regions, which represent all areas at pixel precision, XLD objects provide subpixel precision. There are two basic XLD structures: contours and polygons.

Similarly to images, HALCON/C++ provides both a base class HXLD and a set of specialized classes derived from HXLD, e.g., HXLDCnt for contours or HXLDPoly for polygons. For all classes there exists a corresponding container class, e.g., HXLDArra.

In contrast to the classes described in the previous sections, the XLD classes provide only member functions corresponding to HALCON operators (see also section 5.2.2 on page 25).

6.1.4 Low-Level Iconic Objects

As could be seen in the examples in chapter 5 on page 21, when calling operators in the procedural approach, the class HObject is used for all iconic parameters, be it an image, a region, or even an image array. In fact, the class HObject is HALCON’s basic class for accessing the internal data management, i.e., it handles the keys of the database. Furthermore, HObject serves as the basis for the class HObject and the derived classes, e.g., HImage.

The class HObject has the following member functions:

- HObject(void)
  Default constructor.

- HObject(const HObject &obj)
  Copy constructor.

- virtual ~HObject(void)
  Destructor.

- HObject &operator = (const HObject &obj)
  Assignment operator.

- void Reset(void)
  Freeing the memory and resetting the corresponding database key.

As noted above, an instance of HObject can also contain a tuple (array) of iconic objects. Unfortunately, HObject provides no special member functions to add objects or select them; instead, you must use the operators gen_empty_obj, concat_obj, select_obj, and count_obj as described in section 5.2.3 on page 28.
6.2 Control Parameters

HALCON/C++ can handle different types of alphanumerical control parameters for HALCON operators:

- discrete numbers (long),
- floating point numbers (double), and
- strings (char *).

A special form of control parameters are the so-called handles, which provide access to more complex data structures like windows, image acquisition connections, or models for shape-based matching. Internally, handles are almost always represented by discrete numbers (long); a notable exception is the handle to a bar code descriptor, which is an HTuple, as can be seen in figure 5.2 on page 24. For handles there exist corresponding classes, which are described in section 6.2.3 on page 57.

With the class HTuple, HALCON/C++ provides a container class for control parameters. What’s more, HTuple is polymorphic, i.e., it can also contain arrays of control parameters of mixed type. To realize this, the auxiliary class HCtrlVal is introduced, which is described in the next section.

6.2.1 The Basic Class for Control Parameters

The class HCtrlVal serves as the basis for the class HTuple and is normally hidden from the user because it is only used temporarily for type conversion. The main point is that it can contain the three elementary types of control parameters, i.e., discrete numbers (long), floating point numbers (double), and strings (char *). HCtrlVal provides the following member functions:

- HCtrlVal(void)
  Default constructor.
- HCtrlVal(long l)
  Constructing a value from long.
- HCtrlVal(int l)
  Constructing a value from int.
- HCtrlVal(double d)
  Constructing a value from double.
- HCtrlVal(const char *s)
  Constructing a value from char *.
- HCtrlVal(const HCtrlVal &v)
  Copy constructor.
- ~HCtrlVal(void)
  Destructor.
- HCtrlVal& operator = (const HCtrlVal &v)
  Assignment operator.
- `int ValType() const`
  Type of a value.

- `operator int(void) const`
  Conversion to `int`.

- `operator long(void) const`
  Conversion to `long`.

- `operator double(void) const`
  Conversion to `double`.

- `operator const char*(void) const`
  Conversion to `char *`.

- `double D() const`
  Accessing a value and conversion to `double`.

- `long L() const`
  Accessing a value and conversion to `long`.

- `int I() const`
  Accessing a value and conversion to `int`.

- `const char *S() const`
  Accessing a value and conversion to `char *`.

- `HCtrlVal operator + (const HCtrlVal &val) const`
  Adding two values.

- `HCtrlVal operator - (const HCtrlVal &val) const`
  Subtracting two values.

- `HCtrlVal operator * (const HCtrlVal &val) const`
  Multiplying two values.

- `HCtrlVal operator / (const HCtrlVal &val) const`
  Division of two values.

### 6.2.2 Tuples

The class `HTuple` is built upon the class `HCtrlVal`; it implements an array of dynamic length for instances of the class `HCtrlVal`. The default constructor constructs an empty array (`Num() == 0`). This array can dynamically be expanded via assignments. The memory management, i.e., reallocation, freeing, is also managed by the class. The index for accessing the array is in the range between 0 and `Num() - 1`.

The following member functions reflect only a small portion of the total. For further information please refer to the file `HTuple.h` in `%HALCONROOT%\include\cpp`.

- `HTuple(void)`
  Default constructor. Constructs an empty tuple.
• HTuple(long l)
  Constructing an array of length 1 from a discrete number long at index position 0.

• HTuple(int l)
  Constructing an array of length 1 from a discrete number converted to the internal type long at index position 0.

• HTuple(HCoord c)
  Constructing an array of length 1 from a coordinate at index position 0.

• HTuple(double d)
  Constructing an array of length 1 from a floating number double at index position 0.

• HTuple(const char *s)
  Constructing an array of length 1 from a string char* at index position 0.

• HTuple(const HTuple &t)
  Copying a tuple.

• HTuple(int length, const HTuple &value)
  Constructing an array of the specified length with a constant value, similar to the operator tuple_gen_const.

• ~HTuple()
  Destructor.

• HTuple &operator = (const HTuple& in)
  Assignment operator.

• HTuple Sum(void) const
  Adding all elements in case they are numbers, similar to the operator tuple_sum.

• HCtrlVal &operator [] (int i)
  Setting the i—th element.

• HCtrlVal operator [] (int i) const
  Reading the i—th element.

• HTuple operator + (const HTuple &val) const
  Adding two tuples element by element, similar to the operator tuple_add. The arrays have to be of the same size.

• HTuple operator + (double &val) const
  HTuple operator + (int &val) const
  Adding a number to each element of the tuple, similar to the operator tuple_add.

• HTuple operator - (const HTuple &val) const
  Subtracting two tuples element by element, similar to the operator tuple_sub. The arrays have to be of the same size.

• HTuple operator - (double &val) const
  HTuple operator - (int &val) const
  Subtracting a number from each element of the tuple, similar to the operator tuple_sub.
• HTuple operator * (const HTuple &val) const
  Multiplying two tuples element by element, similar to the operator tuple_mult. The arrays have to be of the same size.

• HTuple operator * (double &val) const
  HTuple operator * (int &val) const
  Multiplying a number with each element of the tuple, similar to the operator tuple_mult.

• HTuple operator / (const HTuple &val) const
  Division of two tuples element by element, similar to the operator tuple_div. The arrays have to be of the same size.

• HTuple operator / (double &val) const
  HTuple operator / (int &val) const
  Division of each element of the tuple by a number, similar to the operator tuple_div.

• HTuple Concat(const HTuple &t) const
  Concatenating two tuples, similar to the operator tuple_concat.

```
#include "HalconCpp.h"
using namespace Halcon;
#include "HIOstream.h"
#if !defined(USE_IOSTREAM_H)
using namespace std;
#endif
main ()
{
    HTuple t;
    cout << t.Num() << 'n'; // The length of the tuple is 0
    t[0] = 0.815; // Assigning values to the tuple
    t[1] = 42;
    t[2] = "HAL";
    cout << t.Num() << 'n'; // The length of the tuple is 3
    cout << "HTuple = " << t << 'n'; // Using the << operator
    double d = t[0]; // Accessing the tuple, if the
    long l = t[1]; // the types of the elements
    const char *s = t[2]; // are known
    // Accessing the tuple, if the types of the elements are known
    printf("Values: %g %ld %s\n",t[0].D(),t[1].L(),t[2].S());
}
```

Figure 6.9: Sample for the use of the class HTuple.

Figure 6.9 shows a short sample how to use tuples, i.e., the class HTuple: The default constructor generates an empty tuple. By assigning values to the tuple it is automatically expanded, and the data types of the values are also stored. For accessing the tuple the normal array notation can be used. If the data type of a value is not known in advance, an explicit type conversion has to be performed, see figure 6.9.
6.2 Control Parameters

6.2.3 Classes Encapsulating Handles

The perhaps most prominent handle class is HWindow, which is described in section 6.2.3.1. Starting with version 6.1, HALCON/C++ also provides classes for handles to files or functionality like access to image acquisition devices, measuring, or shape-based matching. See section 6.2.3.2 on page 58 for an overview.

6.2.3.1 Windows

The class HWindow provides the management of HALCON windows in a very convenient way. The properties of HALCON windows can be easily changed, images, regions, and polygons can be displayed, etc. Besides those operators that can be called via HWindow (see also section 5.2.2 on page 25), HWindow provides the following member functions:

- HWindow(int Row=0, int Column=0,
          int Width=-1, int Height=-1,
          int Father = 0, const char *Mode = "",
          const char *Host = "")
  Default constructor. The constructed window is opened.

- ~HWindow(void)
  Destructor. This closes the window.

- void Click(void) const
  Waiting for a mouse click in the window.

- HDPoint2D GetMbutton(int *button) const
  HDPoint2D GetMbutton(void) const
  Waiting for a mouse click in the window. It returns the current mouse position in the window and the number of the button that was pressed, see the reference manual entry of get_mbutton.

- HDPoint2D GetMposition(int *button) const
  HDPoint2D GetMposition(void) const
  Returning the mouse position and the pressed button without waiting for a mouse click, see the reference manual entry of get_mposition.

- HCircle DrawCircle(void) const
  Waiting for the user to draw a circle in the window, see the reference manual entry of draw_circle.

- HEllipse DrawEllipse(void) const
  Waiting for the user to draw an ellipse in the window, see the reference manual entry of draw_ellipse.

- HRectangle1 DrawRectangle1(void) const
  Waiting for the user to draw a rectangle parallel to the coordinate axis in the window, see the reference manual entry of draw_rectangle1.

- HRectangle2 DrawRectangle2(void) const
  Waiting for the user to draw a rectangle with an arbitrary orientation and size in the window, see the reference manual entry of draw_rectangle2.
```cpp
#include "HalconCpp.h"
using namespace Halcon;

main ()
{
    HImage image("control_unit"); // Reading an image from a file
    HWindow w; // Opening an appropriate window
    image.Display(w); // Display the image
    w.SetLut("change2"); // Set a lookup table
    w.Click(); // Waiting for a mouse click
    w.SetLut("default"); // Set the default lookup table
    w.SetPart(100,100,200,200); // Set a part of the window
    image.Display(w);
    w.Click(); // Adapting the part to the image again
    w.SetPart(0,0,bild.Height()-1,bild.Width()-1);
    image.Display(w);
    HRegionArray regs = image.Regiongrowing(1,1,4,100);
    w.SetDraw("margin");
    w.SetColored(6);
    regs.Display(w);
    w.Click();
    image.Display(w);
    w.SetShape("rectangle1");
    regs.Display(w);
}
```

Figure 6.10: Sample program for the use of the class HWindow.

Figure 6.10 shows the typical use of some member functions of the class HWindow and the different possibilities of displaying images and regions. The window is opened after reading the image from a file. This means, the window is scaled to the size of the image. The lookup table is changed afterwards, and the program waits for a mouse click in the window. A part of the image is zoomed now, and the program waits again for a mouse click in the window. By applying a region growing algorithm from the HALCON library (Regiongrowing) regions are generated and displayed in the window. Only the margin of the regions is displayed. It is displayed in 6 different colors in the window. The example ends with another way of displaying the shape of regions. The smallest rectangle parallel to the coordinate axes surrounding each region is displayed.

### 6.2.3.2 Other Handle Classes

Starting with version 6.1, HALCON/C++ provides the so-called handle classes like HFramegrabber, HBarCode1d, or HClassBoxMlp.

Besides the default constructor, the classes typically provide additional constructors based on suitable operators as described in section 5.2.2.1 on page 26; e.g., the class HBarCode1d provides a constructor based on the operator gen_1d_bar_code_descr.
All handle classes listed above provide the methods `SetHandle()` and `GetHandle()`, which allow to access the underlying handle; furthermore, the classes provide an operator that casts an instance of the class into the corresponding handle. These methods are typically used when combining procedural and object-oriented code; for examples please refer to section 5.5 on page 34.

The reference manual provides short overview pages for these classes, listing the operators that can be called via them.

### 6.3 Auxiliary Classes

In section 6.1.1 on page 37, you already got a glimpse of additional classes provided by HALCON/C++: Instances of `HRegion` can be constructed from classes like `HDPoint2D` or `HRectangle1`. Currently, these classes are not documented in any of the manuals. We recommend to have a look at the header files in the directory `include/cpp`.

Please note that the header files in `include/cpp` include other classes, which do not appear in this manual. These classes are used by MVTec internally for testing purposes; they should not be used an application.
Chapter 7

Creating Applications With HALCON/C++

The HALCON distribution contains examples for building an application with HALCON/C++. Here is an overview of the relevant directories/files (relative to %HALCONROOT%, Windows notation of paths):

include\cpp\HalconCpp.h:
   include file; contains all user-relevant definitions of the HALCON system and the declarations necessary for the C++ interface.

bin\%HALCONARCH%\halcon.dll,
lib\%HALCONARCH%\halcon.lib:
   The HALCON library (Windows).

bin\%HALCONARCH%\halconcpp.dll,
lib\%HALCONARCH%\halconcpp.lib:
   The HALCON/C++ library (Windows).

bin\%HALCONARCH%\parhalcon.dll, parhalconcpp.dll,
lib\%HALCONARCH%\parhalcon.lib, parhalconcpp.lib:
   The corresponding libraries of Parallel HALCON (Windows).

lib/$HALCONARCH/libhalcon.so:
   The HALCON library (Linux/UNIX).

lib/$HALCONARCH/libhalconcpp.so:
   The HALCON/C++ library (Linux/UNIX).

lib/$HALCONARCH/libparhalcon.so,libparhalconcpp.so:
   The corresponding libraries of Parallel HALCON (Linux/UNIX).

include\HProto.h:
   External function declarations.
There are several example programs in the HALCON/C++ distribution (`examples/cpp/source\`). To experiment with these examples we recommend to create a private copy in your working directory.

- `example1.cpp` reads an image and demonstrates several graphics operators.
- `example2.cpp` demonstrates the direct pixel access.
- `example3.cpp` is an example for the usage of pixel iterators.
- `example4.cpp` demonstrates the edge detection with a sobel filter.
- `example5.cpp` solves a more complicated problem.
- `example6.cpp` is a very simple test program.
- `example7.cpp` demonstrates the generic pixel access.
- `example8.cpp` is an example for the usage of the tuple mode.
- `example9.cpp` introduces the XLD structure.
- `example10.cpp` demonstrates the usage of several contour structures.
- `example11.cpp` is another simple example for the usage of tuples.
- `example_errorhandling.cpp` demonstrates the C++ exception handling (see section 5.3.1 on page 32).
bottle.cpp recognizes on numbers on a beer bottle (OCR).
ean13.cpp reads an EAN13 bar code.
ecc200.cpp reads a Data Matrix (ECC200) code.
engraved.cpp recognizes engraved characters (OCR).
fuzzy_measure_pin.cpp measures distances between pins using the fuzzy measure tool.
multi_chars.cpp performs optical character verification (OCV).
pen.cpp uses shape-based matching and the variation model for print quality inspection.
xing.cpp monitors traffic using background estimation (by Kalman filtering).

Additional examples for using HALCON/C++ can be found in the subdirectories examples\mfc, examples\motif and examples\qt.

In the following, we briefly describe the relevant environment variables; see the Installation Guide, section A.2 on page 54, for more information, especially about how to set these variables. Note, that under Windows, all necessary variables are automatically set during the installation.

While a HALCON program is running, it accesses several files internally. To tell HALCON where to look for these files, the environment variable HALCONROOT has to be set. HALCONROOT points to the HALCON home directory. HALCONROOT is also used in the sample makefile.

The variable HALCONARCH describes the platform HALCON is used on. Please refer to section 1.1 on page 3 for more information.

If user-defined packages are used, the environment variable HALCONEXTENSIONS has to be set. HALCON will look for possible extensions and their corresponding help files in the directories given in HALCONEXTENSIONS.

Two things are important in connection with the example programs: The default directory for the HALCON operator read_image to look for images is \images. If the images reside in different directories, the appropriate path must be set in read_image or the default image directory must be changed, using set_system("image_dir","..."). This is also possible with the environment variable HALCONIMAGES. It has to be set before starting the program.

The second remark concerns the output terminal under Linux/UNIX. In the example programs, no host name is passed to open_window. Therefore, the window is opened on the machine that is specified in the environment variable DISPLAY. If output on a different terminal is desired, this can be done either directly in OpenWindow(...,"hostname",...) or by specifying a host name in DISPLAY.

In order to link and run applications under Linux/UNIX, you have to include the HALCON library path $HALCONROOT/lib/$HALCONARCH in the system variable LD_LIBRARY_PATH.

7.1 Creating Applications Under Windows

Your own C++ programs that use HALCON operators must include the file HalconCpp.h, which contains all user-relevant definitions of the HALCON system and the declarations necessary for the C++
interface. Do this by adding the command

```
#include "HalconCpp.h"
```
	near the top of your C++ file. In order to create an application you must link the library halconcpp.lib/.dll to your program.

The example projects show the necessary Visual C++ settings. For the examples the project should be of the WIN 32 ConsoleApplication type. Please note that the Visual C++ compiler implicitly calls “Update all dependencies” if a new file is added to a project. Since HALCON runs under Linux/UNIX as well as under Windows, the include file HalconCpp.h includes several Linux/UNIX-specific headers as well if included under Linux/UNIX. Since they don’t exist under NT, and the Visual C++ compiler is dumb enough to ignore the operating-system-specific cases in the include files, you will get a number of warning messages about missing header files. These can safely be ignored.

Please assure that the stacksize is sufficient. Some sophisticated image processing problems require up to 6 MB stacksize, so make sure to set the settings of your compiler accordingly (See your compiler manual for additional information on this topic).

**Parallel HALCON applications:** If you want to use Parallel HALCON, you have to link the libraries parhalcon.lib/.dll and parhalconcpp.lib/.dll instead of halcon.lib/.dll and halconcpp.lib/.dll in your project.

### 7.2 Creating Applications Under Linux/UNIX

Your own C++ programs that use HALCON operators must include the file HalconCpp.h, which contains all user-relevant definitions of the HALCON system and the declarations necessary for the C++ interface. Do this by adding the command

```
#include "HalconCpp.h"
```

near the top of your C++ file. Using this syntax, the compiler looks for HalconCpp.h in the current directory only. Alternatively you can tell the compiler where to find the file, giving it the -I<pathname> command line flag to denote the include file directory.

To create an application, you have to link two libraries to your program: The library libhalconcpp.so contains the various components of the HALCON/C++ interface. libhalcon.so is the HALCON library.

**Parallel HALCON applications:** If you want to use Parallel HALCON, you have to link the libraries libparhalconcpp.so and libparhalcon.so instead.

Please take a look at the example makefiles for suitable settings. If you call gmake without further arguments, the example application ean13 will be created. To create the other example applications (e.g., example2), call

```
gmake example2
```
You can use the example makefiles not only to compile and link the example programs but also your own programs (if placed in the subdirectory source). For example, to compile and link a source file called myprogram.cpp call

```
gmake myprogram
```

You can link the program to the Parallel HALCON libraries by adding PAR=1 to the make command, for example

```
gmake myprogram PAR=1
```
Chapter 8

Typical Image Processing Problems

This chapter shows the power the HALCON system offers to find solutions for image processing problems. Some typical problems are introduced together with sample solutions.

8.1 Thresholding an Image

Some of the most common sequences of HALCON operators may look like the following one:

```cpp
HByteImage Image("file_xyz");
HRegion Threshold = Image.Threshold(0,120);
HRegionArray ConnectedRegions = Threshold.Connection();
HRegionArray ResultingRegions =
    ConnectedRegions.SelectShape("area","and",10,100000);
```

This short program performs the following:

- All pixels are selected with gray values between the range 0 and 120.
- A connected component analysis is performed.
- Only regions with a size of at least 10 pixel are selected. This step can be considered as a step to remove some of the noise from the image.
8.2 Edge Detection

For the detection of edges the following sequence of HALCON/C++ operators can be applied:

```c
HByteImage Image("file_xyz");
HByteImage Sobel = Image.SobelAmp("sum_abs",3);
HRegion Max = Sobel.Threshold(30,255);
HRegion Edges = Max.Skeleton();
```

A brief explanation:

- Before applying the sobel operator it might be useful first to apply a low-pass filter to the image in order to suppress noise.
- Besides the sobel operator you can also use filters like `EdgesImage`, `PrewittAmp`, `RobinsonAmp`, `KirschAmp`, `Roberts`, `BandpassImage`, or `Laplace`.
- The threshold (in our case 30) must be selected appropriately depending on data.
- The resulting regions are thinned by a Skeleton operator. This leads to regions with a pixel width of 1.

8.3 Dynamic Threshold

Another way to detect edges is e.g. the following sequence:

```c
HByteImage Image("file_xyz");
HByteImage Mean = Image.MeanImage(11,11);
HRegion Threshold = Image.DynThreshold(Mean,5,"light");
```

Again some remarks:

- The size of the filter mask (in our case 11 × 11) is correlated with the size of the objects which have to be found in the image. In fact, the sizes are proportional.
- The dynamic threshold selects the pixels with a positive gray value difference of more than 5 (brighter) than the local environment (mask 11 × 11).

8.4 Texture Transformation

Texture transformation is useful in order to obtain specific frequency bands in an image. Thus, a texture filter detects specific structures in an image. In the following case this structure depends on the chosen filter; 16 are available for the operator TextureLaws.

```c
HByteImage Image("file_xyz");
HByteImage TT = Image.TextureLaws(Image,"ee",2,5);
HByteImage Mean = TT.MeanImage(71,71);
HRegion Reg = Mean.Threshold(30,255);
```
• The mean filter \texttt{MeanImage} is applied with a large mask size in order to smooth the “frequency” image.
• You can also apply several texture transformations and combine the results by using the operators \texttt{AddImage} and \texttt{MultImage}.

8.5 Eliminating Small Objects

The following morphological operator eliminates small objects and smoothes the contours of regions.

```cpp
... 
segmentation(Image,&Seg);
HCircle Circle(100,100,3.5);
HRegionArray Res = Seg.Opening(Circle);
```

• The term \texttt{segmentation()} is an arbitrary segmentation operator that results in an array of regions (Seg).
• The size of the mask (in this case the radius is 3.5) determines the size of the resulting objects.
• You can choose an arbitrary mask shape.

8.6 Selecting Oriented Objects

Another application of morphological operators is the selection of objects having a certain orientation:

```cpp
... 
segmentation(Image,&Seg);
HRectangle2 Rect(100,100,0.5,21,2);
HRegionArray Res = Seg.Opening(Rect);
```

• Again, \texttt{segmentation()} leads to an array of regions (Seg).
• The width and height of the rectangle determine the minimum size of the resulting regions.
• The orientation of the rectangle determines the orientation of the regions.
• Lines with the same orientation as Rect are kept.

8.7 Smoothing Contours

The last example in this user’s manual deals again with morphological operators. Often the margins of contours have to be smoothed for further processing, e.g. fitting lines to a contour. Or small holes inside a region have to be filled:
Again, segmentation() leads to an array of regions (Seg).

For smoothing the contour a circle mask is recommended.

The size of the mask determines how much the contour is smoothed.
Part III

Programming With HALCON/.NET
Chapter 9

Introducing HALCON/.NET

This chapter introduces you to HALCON/.NET. Chapter 10 on page 75 shows how to use it to create .NET applications, chapter 11 on page 95 contains additional information.

What is HALCON/.NET?

HALCON/.NET is HALCON’s interface to .NET programming languages, e.g., C# or Visual Basic .NET. It provides you with a set of .NET classes and controls.

Why use HALCON/.NET and not HALCON/COM or HALCON/C++?

HALCON/.NET is a native .NET assembly, whereas HALCON/COM is used in .NET via so-called wrappers. This means that HALCON/.NET applications are faster than their HALCON/COM counterparts.

HALCON/C++ is meant for standard, i.e., unmanaged C++ applications. If your application needs both managed and unmanaged code, you can combine HALCON/C++ and HALCON/.NET.

Being the newest HALCON interface, HALCON/.NET provides you with state-of-the-art technology, e.g., an improved IntelliSense support in Visual Studio.

Platform Independency

HALCON/.NET is highly platform-independent: It is written in C# but can be used in any .NET language. Like .NET in general, it can be used under Windows and UNIX, on 32bit- and 64-bit systems.

What’s more, you can not only use it on all these platforms, but you can run an application created on one of them on the other ones without needing to recompile. This is possible because applications written in .NET languages are stored in a platform-independent intermediate language, which is then compiled by the so-called common language runtime into platform-specific code.

HDevEngine/.NET

By combining HALCON/.NET and HDevEngine/.NET, you can execute HDevelop programs and procedures from a .NET application. For more information, please refer to part VI on page 151.
Parallel HALCON/.NET and Parallel HDevEngine/.NET

HALCON/.NET and HDevEngine/.NET are provided in two versions: Standard and Parallel. The latter are based on Parallel HALCON, i.e., they use the parallel versions of the HALCON library and of HDevEngine, respectively.
Chapter 10

Creating Applications With HALCON/.NET

This chapter shows you how to create applications with HALCON/.NET. The examples are given in C#, using Visual Studio .NET under Windows as development environment.

If programming constructs or activities differ in Visual Basic .NET or managed C++, this is noted at the first occurrence. How to create applications under UNIX using Mono is described in section 11.2 on page 96.

The provided online help is listed in section 10.4.1 on page 81.

This chapter describes how to

- add HALCON/.NET to an application (section 10.2 on page 78)
- add and customize HWindowControl for visualization (section 10.3 on page 80)
- use HALCON/.NET classes to call HALCON operators (section 10.4 on page 81)
- work with tuples (section 10.5 on page 87)
- visualize images and results (section 10.6 on page 91)
- perform error handling (section 10.7 on page 92)
- deploy an application (section 10.8 on page 93)

Many of the code examples stem from the example Matching, which is provided in C# (examples\c#), Visual Basic .NET (examples\vb.net), and managed C++ (examples\cpp.net). An overview of the provided example applications can be found in section 11.1 on page 95.

But before explaining how to create applications, we must take a brief look under the hood of .NET: at the dependency of applications on the .NET Framework.
10.1 .NET Framework, Development Environments, and Example Directory Structure

Chapter 9 on page 73 emphasized the platform-independency of .NET applications. However, applications are not completely independent: They depend on the version of the .NET Framework they have been created with, i.e., the underlying SDK, tools, etc.

10.1.1 .NET Framework Versions

Table 10.1 shows which version of Visual Studio is based on which version of the .NET Framework.

<table>
<thead>
<tr>
<th>Visual Studio .NET 2002</th>
<th>→</th>
<th>.NET Framework 1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Visual Studio .NET 2003</td>
<td>→</td>
<td>.NET Framework 1.1</td>
</tr>
<tr>
<td>Visual Studio 2005</td>
<td>→</td>
<td>.NET Framework 2.0</td>
</tr>
</tbody>
</table>

Table 10.1: Versions of the .NET Framework and of development environments.

Mono (current version: 1.2) implements most features of .NET Framework 1.1 and many of .NET Framework 2.0.

.NET Framework 1.1 is only an updated version of .NET Framework 1.0, whereas .NET Framework 2.0 added many new features, e.g., the support of managed C++.

Consequently, the HALCON/.NET assembly is provided in two variants (with identical functionality):

- based on .NET Framework 1.0: in the directory %HALCONROOT%\bin\dotnet10
- based on .NET Framework 2.0: in the directory %HALCONROOT%\bin\dotnet20

10.1.2 Example Directory Structure

Furthermore, the example applications (see section 11.1 on page 95 for an overview) are provided in a modular directory structure:

- **source**: contains shared source files
- **vs.net**: contains the project files for Visual Studio .NET 2002 and 2003
- **vs2005**: contains the project files for Visual Studio 2005
- **makefiles**: contains makefiles for compiling the applications from the command line, in particular using Mono (see section 11.2 on page 96 for more information)

Thus, to open an example in Visual Studio .NET 2002 or 2003, you open the corresponding project file in the subdirectory vs.net. To open it in Visual Studio 2005, you open the corresponding project file in the subdirectory vs2005, instead.
Table 10.2: Compatibility of applications.

<table>
<thead>
<tr>
<th>created with \ open with</th>
<th>VS .NET 2002</th>
<th>VS .NET 2003</th>
<th>VS 2005</th>
</tr>
</thead>
<tbody>
<tr>
<td>VS .NET 2002</td>
<td>×</td>
<td>minor upgrade (autom.)</td>
<td>major upgrade (autom.)</td>
</tr>
<tr>
<td>VS .NET 2003</td>
<td>–</td>
<td>×</td>
<td>major upgrade (autom.)</td>
</tr>
<tr>
<td>VS 2005</td>
<td>–</td>
<td>–</td>
<td>×</td>
</tr>
</tbody>
</table>

Note that when opening a project in Visual Studio .NET 2003, the project will be upgraded automatically (with only minor changes to the project file). Table 10.2 lists more generally whether a project created with one version of Visual Studio can be opened with another.

Please note that the provided examples reference the HALCON/.NET assembly using a local path. Therefore, when you copy an example to another location, the assembly will not be found, which is signalled in the Solution Explorer by a broken reference. In such a case, delete the reference and add the correct one as described in section 10.2.2 on page 78.

If you are copying examples to locations other than on a local disk, please note that you must adapt the .NET Framework Configuration as described in section 1.3 on page 6, otherwise you get a warning upon loading the project and a run-time error upon executing it.
10.2 Adding HALCON/.NET to an Application

You add HALCON/.NET to an application with the following steps: For the first application:

- customize Visual Studio’s toolbox (section 10.2.1)

For each application:

- add a reference to HALCON/.NET (section 10.2.2) and
- specify the namespace (section 10.2.3)

10.2.1 Customizing Visual Studio’s Toolbox

In fact, the HALCON/.NET assembly provides not only a class library but also one control: HWindowControl, which contains a HALCON graphics window for visualizing images and results.

You can add this control to Visual Studio’s toolbox by performing the following steps (note that the exact menu names are slightly different in different versions of Visual Studio!):

- Right-click on the toolbox and select Customize Toolbox. This will open a dialog displaying all available .NET Framework components in a tab.
- Click on Browse, navigate to the directory %HALCONROOT%/bin/dotnet10 (Visual Studio .NET 2002 and 2003) or %HALCONROOT%/bin/dotnet20 (Visual Studio 2005) and select halcondotnet.dll.
- Then, the icon of HWindowControl appears in the toolbox (compare figure 10.1).

Parallel HALCON applications: When developing an application with Parallel HALCON, you must select parhalcondotnet.dll instead of halcondotnet.dll. In the toolbox, the control appears with the same name but with a different icon (see figure 10.1). You can add both HALCON versions to the toolbox, but only one of them to an application.

10.2.2 Adding a Reference to HALCON/.NET

In many applications you will use at least one instance of HWindowControl to visualize results. By adding the control to the form (as described in section 10.3 on page 80), you automatically create a reference to the assembly halcondotnet.dll.

If you do not want to use HWindowControl, you add a reference as follows:

- Right-click References in the Solution Explorer and select Add Reference.
10.2 Adding HALCON/.NET to an Application

- Click on Browse, navigate to the subdirectory %HALCONROOT%\bin\dotnet10 (Visual Studio .NET 2002 and 2003) or %HALCONROOT%\bin\dotnet20 (Visual Studio 2005) and select the assembly halcondotnet.dll.

**Parallel HALCON applications:** When developing an application with Parallel HALCON, you must select parhalcondotnet.dll instead of halcondotnet.dll. If you already added a reference to the Standard HALCON version, simply delete that reference and add one to parhalcondotnet.dll.

### 10.2.3 Specifying the Namespace

To be able to use the HALCON/.NET classes without prefixing them with their namespace, we recommend to specify this namespace at the beginning of each source file (see, e.g., the example MatchingForm.cs)) by adding the following line:

```csharp
using HalconDotNet;
```

**Visual Basic .NET applications:** The corresponding Visual Basic .NET code is (see, e.g., MatchingForm.vb):

```vbnet
Imports HalconDotNet
```

**C++ applications:** The corresponding managed C++ code is (see, e.g., MatchingForm.h):

```c++
using namespace HalconDotNet;
```
10.3 Adding and Customizing HWindowControl for the Visualization

In most application you want to visualize at least some results. Then, you start by adding HWindowControl to the form by double-clicking the corresponding icon in the toolbar (see figure 10.1 on page 78). An empty (black) window appears (see figure 10.2).

Parallel HALCON applications: If you already added the Standard HALCON version of the control, but now want to use Parallel HALCON, simply delete the reference to halcondotnet.dll in the Solution Explorer and add a reference to parhalcondotnet.dll instead (also see section 10.2.2 on page 78).

If you want to visualize images (which is typically the case), you should adapt the size of the window to the size of the images to display, otherwise the display will be significantly slower because of the necessary interpolation. The window should have the same size as the image, or half its size, quarter its size, etc. To be more exact, it is not the image size that matters, but the size of the part of the image you want to display. But in most cases you will display the full image, then the two sizes are identical.

You can modify both the window size and the image part very comfortably in the Property Window in two ways (see figure 10.2):

- If you know the desired values, enter them in the properties WindowSize and ImagePart. Note

Figure 10.2: Adapting window size and image part of HWindowControl.
that the part is specified with the values \( X, Y, \text{Width}, \) and \( \text{Height}, \) whereas the corresponding operator \texttt{SetPart} expects the four corner points.

- Alternatively, you can specify a reference image by clicking on the property LayoutBitmap. At the right side of the field, a button appears; when clicked, a file selection box opens that lets you choose a reference image file. After the selection, the dialog depicted in figure 10.2 on page 80 appears. It displays the image size, lets you choose a suitable window size, and sets the image part to the full image.

Note that you can modify the displayed part in your application at any time, e.g., to display a zoomed part of the image. See section 10.6 on page 91 for more information about actually visualizing results.

### 10.4 Using HALCON/.NET Classes to Call HALCON Operators

In HALCON/.NET, you call HALCON operators via instances of classes. The following code, e.g., grabs the first image of an image sequence and displays it in the graphics window of \texttt{HWindowControl}:

```csharp
private HWindow Window;
private HFramegrabber Framegrabber;
private HImage Img;
Window = WindowControl.HalconWindow;
Framegrabber = new HFramegrabber("File", 1, 1, 0, 0, 0, 0, "default", -1, "default", -1, "default", "board/board.seq", "default", 1, -1);
Img = Framegrabber.GrabImage();
Img.DispObj(Window);
```

The operator \texttt{GrabImage} is called via an instance of \texttt{HFramegrabber}. As an experienced HALCON user you will perhaps have identified the constructor of \texttt{HFramegrabber} as a call to the operator \texttt{OpenFramegrabber}.

Below, we take a closer look at:

- how to call operators via HALCON/.NET’s classes (section 10.4.2)
- construction, initialization, and destruction of class instances (section 10.4.3 on page 83)
- overloads of operator calls (section 10.4.4 on page 85)

But first, we give you an overview of the provided online help.

#### 10.4.1 Online Help

The main source of information about HALCON/.NET operators and classes is the HALCON/.NET Reference Manual, which is available as PDF and HTML version. Under Windows, you can open both versions via the Start Menu. Under UNIX, open ReferenceDotNet.pdf in the
directory \%HALCONROOT\%\doc\pdf\reference and table_of_contents.html in the directory \%HALCONROOT\%\doc\html\reference\dotnet, respectively.

The Reference Manual describes the functionality of each HALCON operator and its signatures, i.e., via which classes it can be called with which parameters. Furthermore, it gives an overview of the provided classes (which does not list all methods, however, only the HALCON operators).

Online help is also available in Visual Studio:

- When you type a dot (.) after the name of a class or class instance, the automatic context help (IntelliSense) lists all available methods.
- Similarly, when you type the name of a method, its signature(s) is (are) listed.
- For parameters of HALCON operators, a short description and the the so-called default value is shown. Note that HALCON operators do not have “real” default parameter values, i.e., you cannot leave out a parameter and let HALCON use a default value. Instead, the listed default value is a typical value chosen for the parameter.
- The Object Browser lists all HALCON/.NET classes with their methods, including a short description.

10.4.2 Calling HALCON Operators

Via which classes you can call a HALCON operator is listed in the reference manual. Figure 10.3 shows the corresponding part of the description of the operator GrabImage:

```csharp
static void HOperatorSet.GrabImage ( out HObject image, HTuple acqHandle )
void HImage.GrabImage ( HFramegrabber acqHandle )
HImage HFramegrabber.GrabImage ( )
```

```csharp
image (output_object) ................................................. image ~ HImage
acqHandle (input_control) ......................... framegrabber ~ HFramegrabber / HTuple (IntPtr)
```

Figure 10.3: The head and parts of the parameter section of the reference manual entry for GrabImage.

As you can see, the operator can be called via three classes: HOperatorSet, HImage, and HFramegrabber. The first variant, via HOperatorSet, is mainly used for the export of HDevelop programs (see section 11.3 on page 98).

For normal applications, we recommend to call operators via the other classes, in the example HImage and HFramegrabber as in the following code example:
10.4 Using HALCON/.NET Classes to Call HALCON Operators

```csharp
HImage Image1;
HImage Image4 = new HImage();
HFramegrabber Framegrabber =
    new HFramegrabber("File", 1, 1, 0, 0, 0, 0, "default", -1,
    "default", -1, "default", "board/board.seq", "default", -1, -1);

Image1 = Framegrabber.GrabImage();
HImage Image3 = null;
```

Note that in the call via `HFramegrabber` the grabbed image is the return value of the method, whereas the call via `HImage` has no return value and the calling class instance is modified instead. Usually, calling class instances are not modified by an operator call - with the exception of “constructor-like” operator calls as in the example above.

Some operators like `CountSeconds` are available as class methods, i.e., you can call them directly via the class and do not need an instance:

```csharp
double S1, S2;
S1 = HSystem.CountSeconds();
```

In the reference manual, these operator calls start with the keyword `static`:

```csharp
static void HOperatorSet.CountSeconds ( out HTuple seconds )
static double HSystem.CountSeconds ( )
```

Figure 10.4: The head of the reference manual entry for CountSeconds.

### 10.4.3 Declaration, Construction, Initialization, Finalization

During the lifecycle of an object, i.e., from declaration to finalization, different amounts of memory are allocated and released. The following **declaration** just declares a variable of the class `HImage` that does not yet refer to any object:

```csharp
HImage Image1;
```

In this state, you cannot use the variable to call operators; depending on the programming language, you might not even be able to use it as an output parameter (e.g., in Visual Basic 2005). However, you can assign image objects to the variable, e.g., from the return value of an operator:

```csharp
Image1 = Framegrabber.GrabImage();
```

You can also initialize a variable when declaring it:

```csharp
HImage Image2 = Framegrabber.GrabImage();
HImage Image3 = null;
```
10.4.3.1 Constructors

In contrast, the following declaration calls the “empty” constructor of the class HImage, which creates an uninitialized class instance:

```
HImage Image4 = new HImage();
```

This class instance can be used to call “constructor-like” operators like GrabImage, which initializes it with a grabbed image:

```
Image4.GrabImage(Framegrabber);
```

Besides the empty constructor, most HALCON/.NET classes provide one or more constructors that initialize the created object based on HALCON operators. For example, HImage provides a constructor based on the operator ReadImage:

```
HImage Image5 = new HImage("fuse");
```

You can check which constructors are provided via the online help:

- The reference manual pages for the classes don’t list the constructors themselves but the operators they are based on. The constructor then has the same signature as the operator (minus the output parameter that corresponds to the class, of course).
- The online help in Visual Studio lists the constructors but not the operators they are based on.

10.4.3.2 Finalizers

The main idea behind memory management in .NET is that the programmer does not worry about it and lets the garbage collector delete all objects that are not used anymore. HALCON/.NET fully complies to this philosophy by providing corresponding finalizers for all classes so that even unmanaged resources, e.g., a connection to an image acquisition device, are deleted correctly and automatically.

For most classes, the finalizer automatically calls suitable operators like CloseFramegrabber to free resources. Which operator is called is listed in the reference manual page of a class (see, e.g., the entry for HFramegrabber). This operator cannot be called via the class, as can be seen in the corresponding reference manual entry:

```
static void HOperatorSet.CloseFramegrabber ( HTuple acqHandle )
```

Figure 10.5: The head of the reference manual entry for CloseFramegrabber.

You do not even need to call such an operator if you, e.g., want to re-open the connection with different parameters, because this is done automatically.

⚠️ Please **don’t call** Close or Clear operators via HOperatorSet when using the normal classes like HFramegrabber.
10.4.3.3 Garbage Collection

As remarked above, the .NET philosophy is to let the garbage collector remove unused objects. However, because the garbage collector deletes unused objects only from time to time, the used memory increases in the meantime. There are two ways to force the removal of (unused) objects:

- Call the garbage collector manually. In the example Matching, this is done after each processing run in the timer event:

```csharp
private void Timer_Tick(object sender, System.EventArgs e)
{
    Action();
    GC.Collect();
    GC.WaitForPendingFinalizers();
}
```

**C++ applications:** The code for calling the Garbage Collector in a managed C++ application is

```csharp
GC::Collect();
GC::WaitForPendingFinalizers();
```

- Dispose of individual objects manually by calling the method `Dispose`:

```csharp
HImage Image = new HImage("fuse");
...
Image.Dispose();
```

Besides reducing memory consumption, another reason to manually dispose of objects is to free resources, e.g., close a connection to an image acquisition device or a serial interface.

Please note that HALCON operators always create a new object instance for output parameters and return values (but not in the “constructor-like” operator calls that modify the calling instance). If the variable was already initialized, its old content (and the memory allocated for it) still exists until the garbage collector removes it. If you want to remove it manually, you must call `Dispose` before assigning an object to it.

10.4.4 Operator Overloads

Some classes overload standard operators like + (addition) to call HALCON operators. The following line, e.g., adds two images by internally calling `AddImage`:

```csharp
Image5 = Image1 + Image2;
```

Please note that **operator overloads are only available in C#!**

Figure 10.6 lists the currently available operator overloads.
## Operator overloads for HImage

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>- (unary)</td>
<td>inverts an image</td>
</tr>
<tr>
<td>+ (image)</td>
<td>adds two images</td>
</tr>
<tr>
<td>- (image)</td>
<td>subtracts image2 from image1</td>
</tr>
<tr>
<td>* (image)</td>
<td>multiplies two images</td>
</tr>
<tr>
<td>+ (scalar)</td>
<td>adds a constant gray value offset</td>
</tr>
<tr>
<td>- (scalar)</td>
<td>subtracts a constant gray value offset</td>
</tr>
<tr>
<td>* (scalar)</td>
<td>scales an image by the specified factor</td>
</tr>
<tr>
<td>/ (scalar)</td>
<td>scales an image by the specified divisor</td>
</tr>
<tr>
<td>&gt;= (image)</td>
<td>segments an image using dynamic threshold</td>
</tr>
<tr>
<td>&lt;= (image)</td>
<td>segments an image using dynamic threshold</td>
</tr>
<tr>
<td>&gt;= (scalar)</td>
<td>segments an image using constant threshold</td>
</tr>
<tr>
<td>&lt;= (scalar)</td>
<td>segments an image using constant threshold</td>
</tr>
<tr>
<td>&amp; (region)</td>
<td>reduces the domain of an image</td>
</tr>
</tbody>
</table>

## Operator overloads for HRegion

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp; (region)</td>
<td>returns the intersection of regions</td>
</tr>
<tr>
<td></td>
<td>(region)</td>
</tr>
<tr>
<td>/ (region)</td>
<td>returns the difference of regions</td>
</tr>
<tr>
<td>! (unary)</td>
<td>returns the complement of the region (may be infinite)</td>
</tr>
<tr>
<td>&amp; (image)</td>
<td>returns the intersection of the region and the image domain</td>
</tr>
<tr>
<td>+ (region)</td>
<td>returns the Minkowski addition of regions</td>
</tr>
<tr>
<td>- (region)</td>
<td>returns the Minkowski subtraction of regions</td>
</tr>
<tr>
<td>+ (scalar)</td>
<td>dilates the region by the specified radius</td>
</tr>
<tr>
<td>- (scalar)</td>
<td>erodes the region by the specified radius</td>
</tr>
<tr>
<td>+ (point)</td>
<td>translates the region</td>
</tr>
<tr>
<td>* (scalar)</td>
<td>zooms the region</td>
</tr>
<tr>
<td>- (unary)</td>
<td>transposes the region</td>
</tr>
</tbody>
</table>

## Operator overloads for HFunction1D

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ (scalar)</td>
<td>adds a constant offset to the function’s Y values</td>
</tr>
<tr>
<td>- (scalar)</td>
<td>subtracts a constant offset from the function’s Y values</td>
</tr>
<tr>
<td>- (unary)</td>
<td>negates the Y values of the function</td>
</tr>
<tr>
<td>* (scalar)</td>
<td>scales the function’s Y values</td>
</tr>
<tr>
<td>/ (scalar)</td>
<td>scales the function’s Y values</td>
</tr>
<tr>
<td>* (function)</td>
<td>composes two functions (not a point-wise multiplication)</td>
</tr>
<tr>
<td>! (unary)</td>
<td>calculates the inverse of the function</td>
</tr>
</tbody>
</table>

Figure 10.6: Available operator overloads.
10.5 Working with Tuples

A strength of HALCON is that most operators automatically work with multiple input values (tuple values). For example, you can call the operator `AreaCenter` with a single or with multiple input regions; the operator automatically returns the area and center coordinates of all passed regions. Analogously, if you call `GenRectangle1` with multiple values for the rectangle coordinates, it creates multiple regions.

### 10.5.1 Calling HALCON Operators with Single or Multiple Values

You can check whether an operator also works with tuples in the reference manual. Figure 10.7, e.g., shows the relevant parts of the operators `AreaCenter` and `GenRectangle1`. As you see, the iconic classes like `HRegion` automatically handle multiple values; whether such a parameter accepts / returns...
multiple values is not visible from the signature but only in the parameter section: Here, an appended (-array) (in the example: HRegion(-array)) signals that the parameter can contain a single or multiple values.

In contrast, control parameters show by their data type whether they contain a single or multiple values: In the first case, they use basic data types like double, in the second case the HALCON/.NET class HTuple. Thus, you can call GenRectangle1 via HRegion in two ways, either by passing doubles or HTuples (here using the constructor form):

```csharp
HRegion SingleRegion = new HRegion(10.0, 10.0, 50.0, 50.0);
HRegion MultipleRegions = new HRegion(new HTuple(20.0, 30.0), new HTuple(20.0, 30.0),
            new HTuple(60.0, 70.0), new HTuple(60.0, 70.0));
```

Similarly, AreaCenter can be called in two ways:

```csharp
double Area, Row, Column;
HTuple Areas, Rows, Columns;

Area = SingleRegion.AreaCenter(out Row, out Column);
Areas = MultipleRegions.AreaCenter(out Rows, out Columns);
```

Below, we provide additional information about iconic tuples (section 10.5.2) and control tuples (section 10.5.3).

### 10.5.2 Iconic Tuples

The iconic classes HImage, HRegion, and HXLD can contain single or multiple objects. To process all elements of a tuple you first query the number of elements with with the operator CountObj

```csharp
int NumRegions = MultipleRegions.CountObj();
```

and then access elements either with the HALCON operator SelectObj or (when using C#) with the operator []:

```csharp
for (int i=1; i<=NumRegions; i++)
{  
    HRegion Region = MultipleRegions[i];
    ...
}
```

Note that in iconic tuples **element indices start with 1!**

You can create or extend iconic tuples with the HALCON operator ConcatObj:

```csharp
HRegion ThreeRegions = SingleRegion.ConcatObj(MultipleRegions);
```

### 10.5.3 Control Tuples and the Class HTuple

For control tuples, HALCON/.NET provides the class HTuple. Instances of HTuple can contain elements of the types double, int, and string. They can also contain a mixture of element types.
10.5.3.1 Accessing Tuple Elements

To process all elements of a tuple, you first query its length via the property Length:

```csharp
int TupleLength = Areas.Length;
```

You can access tuple elements with the operator []:

```csharp
for (int i=0; i<TupleLength; i++)
{
    double Element = Areas[i];
    ...
}
```

Note that you get an exception if you try to read a non-existing tuple element or if you try to assign an element to a variable with a different type without cast.

10.5.3.2 Creating Tuples

The class `HTuple` provides many different constructors (see the Visual Studio’s Object Browser for a list). The following line creates an int tuple with a single value:

```csharp
HTuple Tuple1 = new HTuple(1);
```

In contrast, the following line creates a double tuple:

```csharp
HTuple Tuple2 = new HTuple(1.0);
```

You can also pass multiple values to a constructor. Note that when mixing `double` and `int` values as in the following line, a `double` tuple is created:

```csharp
HTuple Tuple3 = new HTuple(1.0, 2);
```

In contrast, when the list of values also contains a `string`, a **mixed type** tuple is created, in which the second value is stored as an `int`:

```csharp
HTuple Tuple4 = new HTuple(1.0, 2, "s");
```

The **type of a tuple or of a tuple element** can be queried via its property `Type`:

```csharp
HTupleType TupleType = Tuple4.Type;
HTupleType TupleElementType = Tuple4[1].Type;
```

You can **concatenate tuples** very simply by passing them in a constructor:

```csharp
HTuple Tuple5 = new HTuple(Tuple2, Tuple3);
```

You can also append elements to a tuple by writing into a non-existing element:

```csharp
Tuple3[2] = 3;
```
10.5.3.3 Casts, Ambiguities, Unexpected Results

The class **HTuple** provides many implicit cast methods so that you can intuitively use the basic data types in most places. For example, the line

```csharp
double Element = Areas[i];
```

automatically casts the element, which is in fact an instance of the class **HTupleElement**, into a double.

Similarly, basic types are automatically casted into instances of **HTuple**. The drawback of the casts is that the compiler often cannot decide whether you want to use the simple or the tuple version of an operator and issues a corresponding error. For example, if you used the following line, the values can either be casted from int to double or to **HTuple**:

```csharp
// HRegion SingleRegion = new HRegion(10, 10, 50, 50);
```

You can **resolve the ambiguity** very simply by appending `.0` to the first parameter:

```csharp
HRegion SingleRegion = new HRegion(10.0, 10.0, 50.0, 50.0);
```

The example **Matching** contains two other cases of ambiguities, both arising because basic-type and **HTuple** parameters are mixed in the same call. In the first, the ambiguity is solved by explicitly casting the double parameters into instances of **HTuple**:

```csharp
private double Row, Column;
HTuple RowCheck, ColumnCheck, AngleCheck, Score;
HHomMat2D Matrix = new HHomMat2D();
Matrix.VectorAngleToRigid(new HTuple(Row), new HTuple(Column), new HTuple(0.0),
RowCheck, ColumnCheck, AngleCheck);
```

In the second case, the instances of **HTuple** (which only contain single values) are explicitly casted into doubles by using the property **D**, which returns the value of the first element as a double (actually, it is a shortcut for `tuple[0].D`):

```csharp
private double RectPhi, RectLength1, RectLength2;
HTuple Rect1RowCheck, Rect1ColCheck;
Rectangle1.GenRectangle2(Rect1RowCheck.D, Rect1ColCheck.D,
RectPhi + AngleCheck.D,
RectLength1, RectLength2);
```

With similar properties, you can cast tuple elements into the other basic types. Note, however, that you get an exception if you try to cast an element into a “wrong” type.

In contrast to input parameters, output parameters are not automatically casted. Sometimes, this leads to unexpected results. In the following code, e.g., doubles are used for the output parameters and the return value in a call to **AreaCenter** with a tuple of regions:
HRegion MultipleRegions = new HRegion(new HTuple(20.0, 30.0), new HTuple(20.0, 30.0),
     new HTuple(60.0, 70.0), new HTuple(60.0, 70.0));
double Area, Row, Column;
HTuple Areas, Rows, Columns;

Area = MultipleRegions.AreaCenter(out Row, out Column);

Consequently, only the area and the center of the first region are returned. The same happens if you
assign the return value to an HTuple, but still pass doubles for the output parameters:

Areas = MultipleRegions.AreaCenter(out Row, out Column);

In contrast, if you pass HTuples for the output parameters and assign the return value to a double, the
operator returns the center coordinates of all regions but only the area of the first region:

Area = MultipleRegions.AreaCenter(out Rows, out Columns);

10.5.3.4 HALCON Operators and Operator Overloads

HALCON provides many operators for processing tuples. In the reference manual, these operators can
be found in the chapter “Tuple”. An overview of these operators is given in the HDevelop User’s Guide in
chapter 8 on page 147. Note that instead of the operator name, the name of the corresponding HDevelop
function is used, which omits the Tuple and uses lowercase characters, e.g., rad instead of TupleRad.

For the basic arithmetic operations, HALCON/.NET provides operator overloads. For example, the
operator + automatically calls the HALCON operator TupleAdd.

10.6 Visualization

In most applications you will use an instance of HWindowControl for the display of images of results.
How to configure this control is described in section 10.3 on page 80. The actual display operators,
however, do not use the control but the HALCON graphics window (class HWindow) encapsulated inside.
You can access the graphics window via the property HalconWindow of HWindowControl:

private HWindow Window;

private void Form1_Load(object sender, System.EventArgs e)
{
    Window = WindowControl.HalconWindow;
}

In the code above, the variable for the instance of HWindow was declared globally and initialized in the
event Load of the form. If you (also) want to run your application under UNIX, you currently cannot use
this event (see the restrictions of Mono listed in section 11.2.1 on page 96). Instead, you can perform the
initialization in the event HInitWindow of HWindowControl:
private void WindowControl_HInitWindow(object sender, System.EventArgs e)

You can configure the display parameters like pen color or line width with the operators in the reference manual chapter “Graphics › Parameters”:

    Window.SetDraw("margin");
    Window.SetLineWidth(3);

Images and other iconic objects are displayed with the operator DispObj, which can be called via the object to display with the window as parameter or vice versa:

    Img.DispObj(Window);

More display operators, e.g., to display lines or circles, can be found in the reference manual chapter “Graphics › Output”.

Instead of (or in addition to) using HWindowControl, you can also open a HALCON graphics windows directly with the operator OpenWindow:

    HWindow ZoomWindow = new HWindow(0, 0, width, height, 0, "visible", "");

In the code above, the window was opened “free-floating” on the display. You can also open it within another GUI element by passing its handle in the parameter fatherWindow.

Before displaying anything in the graphics window, you should set the image part to display with the operator SetPart. In the example code below, the opened window is used to display a zoomed part of the image:

    ZoomWindow.SetPart(row1, col1, row1+height-1, col1+width-1);

More information about visualization in general can be found in the Solution Guide I, chapter 17 on page 239. Note that in this manual, the HDevelop version of the display operators is used, i.e., with the prefix dev_, e.g., dev_open_window instead of OpenWindow.

## 10.7 Error Handling

The .NET programming languages each offer a mechanism for error handling. In C# and managed C++, you use try...catch blocks. Within this standard mechanism, HALCON/.NET offers its special exceptions:

- **HOperatorException** is raised when an error occurs within a HALCON operator
- **HTupleAccessException** is raised when an error occurs upon accessing a HALCON tuple

The following code shows how to catch the error that occurs when the operator ReadImage is called with a wrong image file name. Then, a message box is shown that displays the error number in the caption and the HALCON error message:
10.8 Deploying an Application

By default, .NET applications use local assemblies. For HALCON/.NET applications, this means that the HALCON/.NET assembly halcondotnet.dll is automatically copied to the directory of the application’s executable (e.g., bin\Release). To deploy an application on another computer, you therefore simply copy the content of this directory. Because of .NET’s platform independency, this computer can also run under a different operating system than the development computer.

Of course, the .NET Framework and HALCON must be installed on the destination computer as well, and the environment variables PATH and HALCONARCH must be set correctly (see the Installation Guide, section A.2 on page 54).

Note that you can also install the HALCON/.NET assembly in the so-called global assembly cache. Then, it is not necessary to copy it with each application. See the .NET Framework documentation for details about this method.
Chapter 11

Additional Information

This chapter provides additional information for developing applications with HALCON/.NET:

- **Section 11.1** gives an overview of the available example applications.
- **Section 11.2** explains how to use HALCON/.NET applications under UNIX using Mono.
- **Section 11.3** on page 98 shows how to use HDevelop programs or procedures in your .NET application.
- **Section 11.4** on page 100 contains miscellaneous information.

### 11.1 Provided Examples

The following lists briefly describe the provided example applications.

#### C#

- **examples\c#\Matching** (Visual Studio .NET 2002/2003, Visual Studio 2005, Mono)
  Locate an IC on a board and measure pin distances using shape-based matching (**HShapeModel**) and 1D measuring (**HMeasure**)

- **examples\c#\MultiThreading** (Visual Studio .NET 2002/2003, Visual Studio 2005, Mono)
  Use Parallel HALCON/.NET with multiple threads for image acquisition, processing (2D data code reading, **HDataCode2D**), and display

- **examples\hdevengine\c#\ExecProgram** (Visual Studio .NET 2002/2003, Visual Studio 2005)
  Execute an HDevelop program for fin detection using HDevEngine

- **examples\hdevengine\c#\ExecExtProc** (Visual Studio .NET 2002/2003, Visual Studio 2005, Mono)
  Execute an external HDevelop procedure for fin detection using HDevEngine
• examples\hdevengine\c#\ErrorHandling (Visual Studio .NET 2002/2003, Visual Studio 2005, Mono)

Handle HDevEngine exceptions

**Visual Basic .NET**

• examples\vb.net\Matching (Visual Studio .NET 2002/2003, Visual Studio 2005)

Locate an IC on a board and measure pin distances using shape-based matching (HShapeModel) and 1D measuring (HMeasure)

• examples\hdevengine\vb.net\ExecProgram (Visual Studio .NET 2002/2003, Visual Studio 2005)

Execute an HDevelop program for fin detection using HDevEngine

• examples\hdevengine\vb.net\ExecExtProc (Visual Studio .NET 2002/2003, Visual Studio 2005)

Execute an external HDevelop procedure for fin detection using HDevEngine

• examples\hdevengine\vb.net\ErrorHandling (Visual Studio .NET 2002/2003, Visual Studio 2005)

Handle HDevEngine exceptions

**C++**

• examples\cpp.net\Matching (Visual Studio 2005)

Locate an IC on a board and measure pin distances using shape-based matching (HShapeModel) and 1D measuring (HMeasure)

Please note that the files of this example are not located in subdirectories as described in section 10.1.2 on page 76 but all reside directly in the example directory.

## 11.2 HALCON/.NET Applications under UNIX Using Mono

Chapter 10 on page 75 describes in detail how to develop HALCON/.NET applications in general. This section contains additional information for the case that you want to create applications under UNIX using Mono.

### 11.2.1 Restrictions

Please note the following restrictions when developing or using HALCON/.NET applications via Mono:

• Mono supports Windows Forms but does not claim to implement the full functionality (yet). This has to be kept in mind when developing applications under Windows and compiling or deploying them under UNIX.


- **HWindowControl** is not yet initialized in the event Load of a form, due to a different initialization order of X Window widgets. Please place initialization and similar code in the event handler of **HWindowControl**'s event HInitWindow (see e.g. the example Matching):

```c
private HWindow Window;
private HFramegrabber Framegrabber;
private HImage Img;

private void WindowControl_HInitWindow(object sender, System.EventArgs e)
{
    Window = WindowControl.HalconWindow;
    Framegrabber = new HFramegrabber("File", 1, 1, 0, 0, 0, 0, "default",
            -1, "default", -1, "default",
            "board/board.seq", "default", 1, -1);
    Img = Framegrabber.GrabImage();
    Img.DispObj(Window);
}
```

- When using HDevEngine/.NET under UNIX, you cannot provide an implementation for HDevelop’s display operators like `dev_display` (see section 22.2.3 on page 175).

### 11.2.2 Deploying HALCON/.NET Applications Created under Windows

Because of HALCON/.NET’s platform independency, you can copy an application created under Windows to a UNIX computer and simply start it there - provided that Mono and HALCON are installed on the destination computer (see section 10.8 on page 93 for more information).

### 11.2.3 Compiling HALCON/.NET Applications with Mono

Most of the HALCON/.NET examples provide a set of makefiles in the subdirectory `makefiles` to let you compile them under UNIX (see section 11.1 on page 95 for a list of the examples that support UNIX). To start the compilation, simply type

```
gmake
```

The executable is placed in the subdirectory `makefiles/bin`.

**Parallel HALCON applications:** To create a Parallel HALCON/.NET application, type

```
gmake PARALLEL=1
```

In some cases, Mono may not find the native HALCON library `libhalcon.so`, which should be resolved via the environment variable `LD_LIBRARY_PATH` and issue a corresponding error. You can create configuration files for the HALCON/.NET (and HDevEngine/.NET) assembly, which explicitly specify the path to the HALCON library (see figure 11.1 for an example), by calling

```
gmake config
```
If you want to create a configuration file for only one of the assemblies, use the make commands `config_halcon` and `config_engine`.

### 11.2.4 Using Other GUI Libraries

In principle, you could also use other GUI libraries instead of Windows Forms, e.g., Gtk#. Note, however, that `HWindowControl` is a Windows Forms element and thus can no longer be used. Instead, you can open HALCON graphics windows directly with the operator `OpenWindow`. If you want to place a graphics window inside another element, pass the element’s native window handle in the parameter `fatherWindow`.

Please note that HALCON/.NET has not been tested with other GUI libraries.

### 11.3 Using HDevelop Programs

You can use HDevelop programs or procedures in two ways in your .NET application:

- by executing them directly via HDevEngine (see part VI on page 151 for detailed information) or
- by exporting them into C# or Visual Basic .NET code via the menu item File ⊖ Export (see the HDevelop User’s Guide, section 6.2.1.11 on page 40) and integrating the code in your application.

The latter method is described in this section.

### 11.3.1 Using the Template Aplication

In most cases you will manually integrate the exported code into your application. To quickly test the exported code, you can integrate it into the so-called template project in the subdirectory `HDevelopTemplate` (available for C# and Visual Basic .NET) as follows:

- Move or copy the exported source code file into subdirectory `source` of the template application.
- Open the solution file, right-click in the Solution Explorer, and select the menu item Add Existing Item. Navigate to the source code file, but don’t click Open but on the arrow on the right side of this button and select Link File (see figure 11.2).
- When you run the application, the form depicted in figure 11.3 appears. Click Run to start the exported HDevelop program.
- If you did not add the exported code correctly, the error message depicted in figure 11.4 appears. In Visual Basic .NET, different error messages appear.
11.3.2 Combining the Exported Code with the HALCON/.NET Classes

The exported code does not use the classes like HImage described in the previous chapter. Instead, all operators are called via the special class HOperatorSet. Iconic parameters are passed via the class HObject (which is the base class of HImage, HRegion, and HXLD), control parameters via the class HTuple.
You can combine the exported code easily with “normal” HALCON/.NET code because iconic classes provide constructors that initialize them with instances of `HObject`. Furthermore, iconic classes can be passed to methods that expect an `HObject`.

## 11.4 Miscellaneous

### 11.4.1 HALCON/.NET and Remote Access

For performance reasons, HALCON/.NET suppresses unmanaged code security when making calls into the native HALCON library. Should your machine vision application run in an environment which allows remote access, you might wish to explicitly check permissions for code calling into your application or library.
Part IV

Programming With HALCON/COM
Chapter 12

Introduction

This chapter provides a short overview of the main aspects of component-based software development and shows how this can be achieved with Microsoft COM (Component Object Model). It is not intended to discuss all aspects of COM in detail in this manual; please refer to the corresponding literature.

12.1 The Microsoft Component Object Model (COM)

Component-based software engineering has been discussed intensively in the past years. Concrete standards evolving from this discussion include DCE, CORBA, Microsoft COM, and lately the Microsoft Common Language Runtime, which is part of the .NET framework. The main features of components are:

1. Reusability at binary level independent of the used language
2. Object orientation
3. Robust versioning
4. Location Transparency

Microsoft COM is the base for many commonly used software technologies, e.g., ActiveX, OLE and DirectX. There are numerous tools for developing COM components, e.g., Visual Basic and Visual C++ (and their .NET versions), or Borland Delphi.

12.1.1 COM and .NET

As already mentioned, Microsoft provides a second approach to component-based programming with the Common Language Runtime of the .NET framework. The two approaches are independent, i.e., a COM component is not related to a .NET component and vice versa. However, the two can communicate with each other; furthermore, you can create COM components within Visual Studio .NET just as within Visual Studio.
12.1.2 A Quick Look at Some Programming Aspects

12.1.2.1 Interfaces

An important aspect of COM is the usage of component interfaces. Compared to the C++ terminology, an interface corresponds to the declaration of a class, thus showing its functionality and hiding the (internal) implementation details. This encapsulation allows to use a COM component efficiently, without the need to know anything about its implementation.

In contrast to a C++ class, a COM component can have multiple interfaces. This is useful, e.g., for the versioning: Once defined, an interface stays exactly the way it is. Newer versions of the component then define enhanced or extended functionality by new interfaces, while allowing existing software to remain unchanged, because it can still use the older interfaces. Furthermore, you can partition a component’s functionality into several well-defined portions via multiple interfaces.

There are a couple of standard COM interfaces every component has by default, e.g., IUnknown and IDispatch. They are not described in this manual; please refer to appropriate literature.

12.1.2.2 Objects, Methods and Data Members

Just as with C++ classes there are interfaces in COM and those interfaces are made up from some methods. Several objects of a class correspond to several instances of a component, each with its own internal state. However, there are some differences with data members, as their handling within COM depends on the tool the component is used by. COM follows a concept of properties, which are special methods allowing tools like Visual Basic to treat them just like data members. With C++ they still are only methods, thus there is no way to modify and retrieve data members directly. From this point of view COM class data members can be compared with C++ private class members.

12.1.2.3 Inheritance and Polymorphism

As COM is a binary standard, inheritance must also take place at binary level which makes the process slightly uncomfortable. Without going into detail one can say that the only thing of interest here is that there are two methods of reusing existent components: containment and aggregation. Both techniques are commonly used by C++ programmers as well: Containment corresponds to C++ classes instantiated as a member object in other classes, whereas aggregation roughly corresponds to inheritance.

The main difference between containment and aggregation is the way the interface(s) of the contained/aggregated component(s) are handled: The methods and properties of a contained component are hidden to the outside so that only the containing component can use them. Any method that should be visible from outside has to be re-defined in the outer component. In contrast to this, the interface(s) of an aggregated component are merged with the interfaces of the aggregating one, thus automatically making their methods visible to the outside.

The object-oriented feature of polymorphism is also achieved through interfaces: Different COM classes exposing the same interface can be understood as showing polymorphic behavior, as they act differently responding to the same methods.
12.1.2.4 Early and Late Binding

Binding is the process of resolving the call addresses of a component’s exposed methods. This can be done at compilation time (early binding) or at runtime (late binding). Thus, when using a component you can take a specification of the contained methods and integrate it directly into the application. Therefore, the specification must be available in a format that fits the used programming language (e.g., when using C++ as client language one would typically include the header files containing the method declarations).

Alternatively, you can let the methods be bound at runtime; in this case, no language-dependent information is needed for compiling and the calling mechanism is totally different. Without going into details we want to point out that runtime-bound methods calls are slower than their early-bound counterparts. The HALCON/COM interface supports both early and late binding so that special needs of different applications can be satisfied.

12.2 HALCON and COM

With the HALCON/COM interface you can use the full functionality of the HALCON library within COM components (and therefore also ActiveX controls) developed, e.g., using Microsoft Visual Basic or Borland Delphi. In fact, there are two HALCON/COM interfaces, one to the Standard HALCON library and one to Parallel HALCON, which automatically exploits multi-processor or multi-core hardware and supports parallel programming. Please refer to section 2.2 on page 7 for more information about Parallel HALCON; how to use the corresponding COM interface is described in section 14.4 on page 122.

Note that currently the HALCON Extension Package Interface cannot be used to create extension packages for COM.
Chapter 13

The HALCON/COM Interface

Let’s have a closer look at the HALCON/COM interface. This language interface (provided in form of the DLL halconx.dll) contains the full functionality of HALCON partitioned into several classes. Each class has one or more interfaces, which are built up from methods and properties. The HALCON/COM interface uses inheritance and thus contains derived classes and also an abstract base class. Since COM is not meant to supply a standardized inheritance mechanism, HALCON/COM makes extensive use of interfaces to simulate inheritance (we will discuss this topic in more detail afterwards).

Naming Conventions:

- Classes are capitalized and begin with an “H”. They always end with an upper-case “X”; for example: HFramegrabberX.
-Interfaces are also capitalized and end with “X”, but begin with an “I”; for example: IHObjectX.
- Methods, properties and parameters are capitalized; for example: GrabImage, FileName.

Since COM is only available for the Windows operating systems family, all file paths and environment variables in this manual are printed in the Windows convention, e.g.,

```
%HALCONROOT%\examples\vb\Manual
```

to denote a subdirectory containing an example package within the HALCON base directory referenced by the environment variable HALCONROOT.

13.1 More about Classes

Classes are the essential items to deal with when writing HALCON/COM applications. There are quite a lot of them and there are different types of classes, thus we will have a closer look on how classes are used and what attributes they have. The classes of the HALCON/COM interface are partly related in a way which could be described as inheritance. Therefore, it is important to get an impression of how inheritance is achieved within HALCON/COM and how to use it.
13.1.1 Different Types of Classes

There are two major categories of classes:

1. classes instantiating objects that have an internal state and
2. classes instantiating objects with no internal state.

The first category is used to implement special data structures (like images, files, image acquisition devices, etc.) and the operators belonging to this data, whereas the second category is only used to group operators belonging together. If several objects of a class belonging to the first category are instantiated, they are all different from the HALCON point of view, whereas the second category does not have this quality. For example, if we consider several objects instantiated from the image class HImageX, they all denote something different: they represent different HALCON images. In contrast, if we have an object of a class like HSystemX that represents the internal state of the HALCON kernel, it does not matter how many of those objects will be instantiated, because all of them denote the same (there is only one HALCON kernel to be represented). Those classes can be understood as group classes, denoting that they supply a bunch of methods that all have some semantic peculiarities in common.

In contrast to this, methods of the first category classes may share a common semantics, but work on different data. For example, if an object of the class HImageX is instantiated, its methods always work exactly on one specific HALCON image: the one that is represented by the object (to be precise an HImageX object may also represent an array of images). Different HImageX objects represent different images and therefore work on different data.

Besides the first two groups, we can categorize classes in another way well known in the object oriented world:

1. abstract classes and
2. non-abstract classes.

A class is called abstract, if it can not be instantiated. Thus, an abstract class must be the base class for other classes, if it should be of any use. The idea behind this concept is that abstract classes provide a semantic base for their derived classes without being concrete enough for instantiation. A look on real world terms reveals many analogous cases: there is, e.g., the class animal which can be seen as an abstract base class for classes like fish, bird and horse. Any existing being can not be only an animal, it is always either a fish, a bird or a horse (or whatever else). There is only one such abstract class in the HALCON/COM interface: HObjectX. It represents a HALCON iconic object, such as an image, a region, or an XLD. The derived classes (HImageX, HRegionX and so on) then specify the exact type of the HALCON iconic object (see also the class overview in figure 13.1).

13.1.2 Classes for Special Purposes

There are some HALCON/COM classes that have special jobs. Although they do fit into the above systematics, they are worth mentioning, because they have to be used in a specific way or show some specific semantics.
13.1 More about Classes

13.1.2.1 HWindowXCtrl

This class is a HALCON window in the form of an ActiveX control. Its advantage against the ordinary HALCON window (HWindowX) is the possibility to exist inside an ActiveX container. An ActiveX container is a part of a window which can contain GUI elements, such as buttons, sliders and so on. When developing complex image processing applications, it is often necessary to integrate all user interaction components (input and output) seamlessly into one surface. The HALCON window class HWindowX does not provide this flexibility, since it always appears as a top-level window. In order to behave like an ordinary HWindowX window, HWindowXCtrl uses the COM technique of aggregation. Thus, any newly created HWindowXCtrl automatically instantiates an HWindowX object which is bound to the control.

13.1.2.2 HOperatorSetX

This is basically a group class for all existing HALCON operators. HOperatorSetX is meant to provide access to a procedural way of HALCON programming. The reason for that is the fact that it is easier for some non-object-oriented tools like HDevelop to generate COM code automatically when using a
The HALCON/COM Interface

procedural technique. The specificity about \texttt{HOperatorSetX} is that all its methods require instances of \texttt{HUntypedObjectX} (see below) for all iconic input and output parameters. Furthermore, all control parameters (input and output) are of the type \texttt{VARIANT} (see also section 13.5 on page 113).

Hand-written COM code should not use the \texttt{HOperatorSetX/HUntypedObjectX} concept, since it weakens the otherwise strong-typed object-oriented approach; it becomes relevant only in cases, where automatically generated code is involved.

13.1.2.3 \texttt{HUntypedObjectX}

The class \texttt{HUntypedObjectX} is derived from \texttt{HObjectX} just like \texttt{HImageX}, \texttt{HRegionX} and so on. Its purpose is to get an instantiable form of the abstract base class. The class does not have any members, as it just consists of a polymorphic data type without any special meaning. As it is this weak-typed, it can not be used together with the strong-typed methods of the other classes, except by using the method \texttt{Cast()} which allows arbitrary type conversions between all classes derived from \texttt{HObjectX} (explained later on). \texttt{HUntypedObjectX} is meant to be a generic data type used by the special class \texttt{HOperatorSetX}.

13.1.2.4 The Method \texttt{Cast()}

All classes derived from \texttt{HObjectX} supply the method \texttt{Cast()} which is used for type conversions between the concerned classes. This method breaks up the well-defined data type scheme evolving from the class hierarchy, so it should not be used except when there is no other possibility! By using the method \texttt{Cast()}, an image object can be turned into a region, an XLD can become an \texttt{HUntypedObjectX} object and so on. The reason for that is, as mentioned above, the need to convert back and forth to the \texttt{HUntypedObjectX} objects used by the \texttt{HOperatorSetX} methods. For example, if an automatically generated code fraction produces an object variable of type \texttt{HUntypedObjectX} which shall be used as an image in some handwritten code, it can be forced to become an \texttt{HImageX} object. Of course, it must be assured that the variable really \textit{does} contain an image (it is the programmer’s task to take care for that).

The following short Visual Basic example reads an image via \texttt{HOperatorSetX} and then casts it into the class \texttt{HImageX}. Note that in order to apply the method \texttt{Cast()}, the variable \texttt{Image} must be a “living” COM object, thus it is declared as \texttt{New HImageX}. See section 13.7 on page 114 for more information about using HALCON/COM within Visual Basic.

```vbnet
Dim Op As New HOperatorSetX
Dim Untyped As HUntypedObjectX
Dim Image As New HImageX

Call Op.ReadImage(Untyped, "monkey")
Call Image.Cast(Untyped)
```
13.2 Object Construction and Destruction

13.2.1 Construction

A HALCON/COM object can be in different states. It can be *instantiated* and it can be *initialized*. An instantiated object does not necessarily need to be initialized, but an initialized object is always instantiated. Instantiation means using an appropriate technique to produce a “living” COM object (this technique differs according to what client language is used). Initializing means to give the object a well-defined internal state.

There is another state an object can have: neither instantiated nor initialized. Since not being instantiated means not existing, what exactly does this condition mean? The answer depends somewhat on the client language used, but is rather easy to understand, if we realize that COM object variables actually are *references* to an existing (or not existing) object. Thus, if such a variable is declared, there is not necessarily already an object created it refers to. As we like to mix up the terms “reference to an object” and “object” in this manual, we may speak of an uninstatiated object in that case. For example, if an HImageX object is created (not only the referring variable is declared), it is usable in terms of COM (since it is instantiated), but it does *not* yet contain any HALCON image. To be a valid HALCON/COM object, it still must be initialized. This can be done in two different ways:

- The object can initialize itself or
- the object can be initialized by another object’s method.

In the first case, a so-called constructor is executed. The term “constructor” is somewhat misleading here, since it has a slightly different meaning than, e.g., a C++ constructor. In C++ terms, a constructor is a special method (always named exactly as the class) which performs the object construction *automatically* without the need to be called explicitly. In the HALCON/COM, case a constructor is an ordinary method which initializes the object’s internal state and must be called explicitly.

For that reason, a HALCON/COM class can have many (differently named) constructors. One of the constructors of the class HImageX, e.g., is ReadImage, which initializes the object by reading an image file from the hard disk. Another way to initialize an object is to get it as result after calling another object’s method. For example, an uninitialized HImageX object becomes initialized when it is returned from a call to MeanImage (which does a convolution on an image and produces another image).

Note that the mechanism is indeed slightly more complicated and again depends on the client language. In fact, an object variable needs *not* refer to a living COM object, when being used as a return value of a method call. Even more, if it *does*, the referred object gets destroyed before the new object is assigned to the variable.

Actually, there is yet another way to initialize a HALCON/COM object: by using the method Cast() mentioned before. This method takes another object of a related class, i.e., derived from HObjectX, as parameter, which in turn has to be initialized. Then, the object the calling method belongs to gets initialized with the internal state of the other object. Because no copying or duplication takes place, the other object gets *uninitialized*.

Note that not all HALCON/COM objects can have this initialized state! An object can only be initialized, if it *has* an internal state. Thus, all the group classes described in section 13.1.1 on page 108 have no constructor methods.
13.2.2 Destruction

We have seen that creating a valid HALCON/COM object is a two-step process: First, the COM object needs to be instantiated, then it must be initialized. This implies also a two-step destruction process: To destruct an object it must be uninitialized before being destroyed (which is the opposite of being instantiated). The latter is done by COM and the client environment, the first must be performed by some automatic destruction process. To pick up the HImageX example again, the automatic destructor must free the image data using the HALCON memory management. The automatic destructors of classes like HShapeModelX or HFramegrabberX internally apply operators like ClearShapeModel or CloseFramegrabber, respectively. These operators cannot be called via instances of the class (only via HOperatorSetX); the same holds for operators like ClearAllShapeModels. Instead of using these operators, you can destroy instances and then initialize anew.

Please note that you must not use operators like ClearShapeModel, ClearAllShapeModels, or CloseFramegrabber via HOperatorSetX together with instances of the corresponding handle classes (HShapeModelX, HFramegrabberX, etc.)!

13.3 Interfaces and Inheritance

As said before, COM classes are built from interfaces which in turn contain methods and properties. Each HALCON/COM class has got one default interface named according to the class. For example, the class HImageX contains the default interface IHimageX which provides all the relevant methods. Dealing with interfaces is strongly related to the client environment used, so we will not have a too close look at this topic now. For example, Visual Basic tries to completely hide the whole interface topic from the user; in that case, an interface and the related class are the same most of the time.

Actually, an interface behaves similar to a C++ pointer or reference to an object and that is exactly what Visual Basic tries to hide. As said before, interfaces are used for inheritance simulation within HALCON/COM. The way this works is simple: We have seen that HObjectX is an abstract (and thus not instantiable) class. On the other hand, the derived classes HImageX, HRegionX, etc. have to supply their own functionality plus the inherited HObjectX functionality. This is achieved with the help of interfaces: Currently there is no COM class named HObjectX (thus no HObjectX object can be initialized), only an interface named IHObjectX. This interface appears in all “derived” classes together with their default interfaces. HImageX, e.g., has two interfaces (in fact it has a hidden third one):

1. IHImageX (the default interface) and
2. IHObjectX (the inherited interface).

This allows to satisfy every method which expects a parameter HObjectX (actually it expects a reference to that class in form of the interface IHObjectX) with any derived class object, as such an object always has the interface IHObjectX, too. This corresponds to a certain object-oriented rule which allows an automatic cast from the derived class to the base class (not the other way).

How intuitive this feature can be used, again depends on the client language/tool: Visual Basic 5.0, for example, only regards the default interface an object supplies and treats that as if it were the class itself. That implies that only the methods contained in the default interface seem to belong to a class. To also use the “inherited” methods (contained in the IHObjectX interface), thus an explicit cast to the base class is necessary. This is a widely known Visual Basic weakness and may get improved by Microsoft in forthcoming versions.
13.4 Methods and Properties

There is not much to say about methods, since they can be used quite intuitively. The only interesting aspect here is the fact that different classes can have methods with the same name, but with a different parameter configuration. These methods perform the same action (as their identical names let expect), but “show a different point of view”. For example, the operator GrabImage is part of the class HImageX as well as of the class HFramegrabberX. In the first case, the method is a constructor for an image object and takes an HFramegrabberX object as parameter (which denotes the image acquisition device from which to retrieve the data). In the second case, the method takes the HImageX object into which the image should be grabbed as parameter.

Properties are a special COM feature and can be treated like data members by tools like Visual Basic. There are two kinds of them: put and get properties. A put property allows the user to change the internal state of an object, whereas the get property only allows to read out the internal state. Usually (but not always), both types are present acting like an ordinary variable class member in C++. The properties in the HALCON/COM interface are just for convenience purposes, since they always map to an existing method. The class HWindowX, e.g. has quite a lot of properties: Width and Height define the window’s extent, Draw sets or gets the current drawing mode, and so on. All of these properties are mapped to their corresponding methods. Reading out the draw mode, e.g., results in a call to the operator GetDraw, whereas Width/Height both are mapped to the operator GetWindowExtents when retrieving values, and SetWindowExtents for changing values, respectively.

13.5 A Closer Look at Data Types

We have seen that a lot of the data types used within the HALCON/COM interface are actually the classes themselves. But there are also more basic, “everyday” types being used. A widely used data type in the COM world (and thus in Visual Basic) is the type VARIANT. All users of the HALCON/C++ interface will know the data type HTuple which is polymorphic (i.e., it can be one of several different “subtypes”) and supplies array functionality (i.e., it can hold several data items at once). VARIANTS behave analogously and are the COM equivalent for HTuples. Exactly like an HTuple a VARIANT is able to hold data of different basic types plus the information what kind of data it contains. Also a combination of different or equal types is possible just like with HTuples.

The main difference is that a VARIANT is no class. This implies the complete absence of any methods. Thus, all the additional functionality of the very powerful class HTuple must be accessed in another way. For this reason, there is an class HTupleX, which groups methods for tuple operations like vector addition, string concatenation, and so on.

Another important and widely used COM data type is BSTR. This standard COM flavor of character strings is not directly compatible with standard C-like string implementations, mainly because it uses wide chars. This means, that auxiliary functions must be used to access or modify BSTRs when using C/C++. Again, this is no problem with Visual Basic, where it is the default string data type. Additionally, there are integral data types like long and double as well. They are used in situations where array- or multitype-parameters are not allowed or make no sense.
13.6 Error Handling

The HALCON/COM interface uses the standard COM error handling technique where every method call passes both a numerical and a textual representation of the error to the calling framework. It is then up to the caller to react to this information. Since low error numbers are reserved for COM, HALCON/COM uses a very high offset to its own error codes. To get the correct HALCON error code, this offset must be subtracted from the received code. The offset is a (read only) property of the class HSystemX. There are two offsets: one for the HALCON/COM interface and one for HALCON itself. Those properties are named:

- HSystemX.ErrorBaseCOM
- HSystemX.ErrorBaseHalcon

In order to get the correct HALCON error code, an HSystemX object must be instantiated (one is enough for the whole application anyway, since HSystemX objects have no “identity”). Then, the value of the respective property of HSystemX must be subtracted from the returned error code to get the correct HALCON error code. For an example how to deal with error codes see section 13.7.2.

13.7 HALCON/COM and Visual Basic

So far, the important basics of the HALCON/COM interface have been explained. Now let’s have a look at what things behave like when using Visual Basic.

13.7.1 Object Instantiation

There are many different ways to instantiate COM objects in Visual Basic. We will discuss only one of them, because it has certain advantages over all the others. We have seen in the sections before that a distinction should be made between the instantiation and the initialization of objects. Even more important, we should also distinguish objects from object reference variables. An object reference variable is set up by its declaration with the keyword Dim:

Dim image1 As HImageX

This statement does not yet create a COM object, it just declares a variable able to reference an HImageX object. If we want to declare a reference variable and immediately create an object it refers to, we should write

Dim image1 As New HImageX

Now, a “new” HImageX object is created and the variable 'image1' refers it. Note that the declaration of variables is not obligatory with Visual Basic, but should be done anyway! Undeclared variables get declared automatically when referenced and that can lead to errors which are very hard to track down! It is a good idea to place the statement 'Option Explicit' on top of every Visual Basic module, because then variable declaration is forced.
We now have a valid COM object, to which the declared variable refers. To initialize this object, we could call a constructor method:

```vba
Dim image1 As New HImageX
Call image1.ReadImage('some_file_name')
```

Note the keyword `Call`! It’s necessary in Visual Basic if the called method doesn’t return a value. The other way of initialization would be using another object’s method:

```vba
Dim image1 As New HImageX
Dim region1 As HRegionX
Call image1.ReadImage('some_file_name')
Set region1 = image1.Threshold(128, 255)
```

There are two important things here. First, the keyword `Set`, which replaces the `Call` keyword when the called method returns another COM object (in this case a region). Secondly, the second variable declaration omits the keyword `New` because the corresponding variable does not need to instantiate an object at declaration time. Instead, this is achieved within the operator `Threshold`, which creates a new COM object itself and passes a reference to this object as its return value.

HALCON/COM objects get destroyed as soon as no variable references them anymore. For local variables, this is the case when they go out of scope (e.g., when a subroutine is left). For global variables, or if an explicit destruction is desired, this has to be done by the user:

```vba
Dim image1 As New HImageX
Dim region1 As HRegionX
Call image1.ReadImage('some_file_name')
Set region1 = image1.Threshold(128, 255)
Set image1 = Nothing
Set region1 = Nothing
```

Here, both variables are assigned the special Visual Basic keyword `Nothing` which denotes that they do not reference their related COM objects anymore. These COM objects thus are not referenced at all which leads to their immediate destruction.

There is, of course, a lot more to say about Visual Basic/HALCON programming. Some further aspects might become clear in the example session described in chapter 14 on page 117.

### 13.7.2 Error Handling

When using Visual Basic, errors can be trapped by an error handler. If no custom error handler is present, Visual Basic itself supplies a default one, which shows a message box containing the textual error description. Error handlers in Visual Basic can be set with the keyword `On Error`. To trap an error in a portion of code, the appropriate construct could look like this:

```vba
On Error GoTo 0
```
Dim LastErrorCode As Long
Dim SysObject As New HSystemX

On Error Goto myErrorHandler

<some code>

myErrorHandler:
    ' do something with the error information, for example:
    Debug.Print "Error occurred: " + Err.Description
    LastErrorCode = Err.Number - SysObject.ErrorBaseHalcon
    Resume Next

If an error occurs in <some code>, an immediate jump to the label myErrorHandler is made, where an arbitrary error processing mechanism can be placed. The scheme used in the example tries to model a traditional, “procedural” error recovery strategy, where every function call returns an error code, which has to be checked before program execution can continue. When an error occurs, the error handling code at the corresponding label takes over, writes a status message (the textual error representation) to the Visual Basic window 'Immediate' and stores the error code in a global integer variable. The global Visual Basic object Err is the source of information in this case. Afterwards, control is returned to the line following the statement which produced the error via the Visual Basic command Resume Next. The next line of code then would be responsible for checking the error code stored in LastErrorCode.

We have seen that there are two types of errors: HALCON-related errors and COM-interface-related ones. Since the COM-interface errors have smaller error numbers than the HALCON error codes, the above mechanism would lead to negative numbers. In this case, the produced error code would have to be subtracted from SysObject.ErrorBaseCOM to get the correct (interface-related) error code.
Chapter 14

Example Visual Basic Session

In this chapter you will learn how to develop HALCON applications quickly using Microsoft Visual Basic and the HALCON/COM interface. There will be simple steps describing what to do. The result will be a very small example vision application equipped with a graphical user interface. As an additional source of information you are strongly encouraged to have a look at the other examples which are supplied as Visual Basic sources together with HALCON.

The program developed in this chapter is also available ready to go in the directory examples\vb\Manual together with other examples. However, it is recommended to follow the steps below and program it yourself, as you will get a better impression of how Visual Basic program development works and gain a lot of additional information with the single steps.

Please note, that to use HALCON/COM inside Visual Basic you need Windows NT 4.0 with Service Pack 4 or Windows 2000 or Windows XP; the example projects are created for Visual Basic 6.0.

If you want to use HALCON/COM inside other programming languages, please have a look at the examples in the subdirectories examples\delphi and examples\mfc; they show how to use HALCON/COM within Borland Delphi, or together with Microsoft MFC in Visual C++. Note that the examples using C# or Visual Basic .NET are based on HALCON/.NET.

14.1 First Step: The GUI

Go ahead and

1. Launch Visual Basic. A dialog named New Project should appear allowing you to select the type of project you want. Switch to New in the tab list, select Standard EXE and click Open.

2. Select Project from the menu bar and click Components. A dialog box shows up listing the components installed on your system. Switch to Controls in the tab list and place a check next to the item Halcon/COM library.

3. Press F2. The object browser should appear. See if you can find HImageX and browse through some of the corresponding methods. Clicking on a method shows its parameterization as well as
a short help text about what it will do in the status area at the bottom of the object browser. Close the object browser.

4. Have a look at the dialog template ('form') showing in the lower half of the screen; it should be titled Form1. In the upper half you will discover an area titled Properties - Form1. Here you can set and retrieve the active GUI object’s (in this case the form’s) properties. Click on Form1 right beside Caption and change the string to HalconX example. You should see the effect of your action immediately in the caption text of the below form.

5. Grab the form and resize it to a suitable extent.

6. Have a look at the tool bar to the left: Here you can find all the control elements you can place inside your form. They are represented as small icons. Move the mouse cursor over the different icons to see the bubble help revealing their names. You should find an icon showing the HALCON symbol named HWindowXCtrl1. You guessed it! That is our ActiveX control HALCON window.

7. Activate the HWindowXCtrl1 icon. Draw a rectangular region inside the form | make sure it is approximately square. When releasing the mouse button the square area should turn black.
8. Switch to the CommandButton icon (looking like a brick) in the left tool bar. Draw a button inside the form beside or below the HALCON window. Change the button’s caption text to Next >> in the properties box.

9. Now switch to Label in the tool bar and draw a longish rectangular area at the bottom of the form. If you encounter placement difficulties due to lack of form space, you can always resize the form to fit your needs.

10. Resize the form so that it fits around the before created items. Now you have the entire GUI for your application ready to go and your screen should look similar to figure 14.1.

### 14.2 Second Step: Functionality

Now you have the finished GUI, you should go ahead and make the application do something:

1. Right-click somewhere inside the form and select View Code. Another window will pop up over the form with two combo boxes at its top border. Select Form in the left combo box. You will see the code to be executed when the form is created.

2. Insert a line into the subroutine:

   ```vbnet
   Private Sub Form_Load()
   Label1.Caption = "Click Next to start"
   End Sub
   ```

   You just changed the text the label at the bottom will show when the application is launched.

3. Next we will declare some important variables: Switch back to (General) in the left combo box above the source code window and insert in the following lines at the top:

   ```vbnet
   Dim Monkey As New HImageX
   Dim Window As HWindowX
   ```

   Some online selection boxes for the desired object type will assist you. We have just created two objects: an HImageX and an HWindowX. The reason for the keyword New in the first line is that we want the HImageX object to be instantiated (i.e., memory being allocated for it). This is not necessary for the HWindowX, since it is already instantiated; it is the ActiveX control we have drawn inside the form.

4. The object 'Monkey' is instantiated as we know (although it is not yet initialized with an image), but the variable 'Window' still refers to nowhere. Insert another line into the subroutine Form_Load():

   ```vbnet
   Private Sub Form_Load()
   Set Window = HWindowXCtrl1.HalconWindow
   Label1.Caption = "Click Next to start"
   End Sub
   ```

   Now, the variable 'Window' refers to the HWindowX part of our ActiveX control.
5. Switch to Command1. Another subroutine appears, which you complete like this:

```vbnet
Private Sub Command1_Click()
    Call Monkey.ReadImage ("monkey")
    Call Window.DispObj(Monkey)
End Sub
```

6. Start your application by pressing F5 and see what happens!

While typing, you will notice a very convenient Visual Basic feature: Since it knows the methods of a class, it allows you to select one from a list, if you wish to do so (see figure 14.2). You will also get assistance in supplying the parameter values for a method call in the right order and with the right types (see figure 14.3); if no type is shown, a VARIANT is required.

```vbnet
Private Sub Command1_Click()
    Call Monkey.ReadImage ("monkey")
    Call Window.Display()
End Sub
```

Figure 14.2: Visual Basic helping you to select a method.

Figure 14.3: Visual Basic helping you with the correct parameters.

### 14.3 Final Step: More Functionality

What we have now is a very basic application which can’t do very much but it needs only 10 lines of code! Below, we will extend the functionality, turning our application into a small image processing demo:

1. Extend the variable declaration section at the beginning of your listing so it looks like this:

```vbnet
Dim Monkey As New HImageX
Dim Window As HWindowX
Dim Region As HRegionX
Dim Eyes As HRegionX
Dim State As Integer
```
Although these declarations are not necessary (Visual Basic declares variables automatically), it is nevertheless a good idea to do so.

2. Select the subroutine `Command1_Click()` and modify it like this:

```vba
Private Sub Command1_Click()
    If State = 3 Then
        End
    End If

    If State = 2 Then
        Set Eyes = Region.SelectShape("area", "and", 500, 50000)
        Set Eyes = Eyes.SelectShape("anisometry", "and", 1, 1.7)
        Call Window.DispObj(Monkey)
        Call Window.DispObj(Eyes)
        Label1.Caption = "click Finish to terminate"
        Command1.Caption = "Finish"
        State = 3
    End If

    If State = 1 Then
        Set Region = Monkey.Threshold(128, 256)
        Set Region = Region.Connection()
        Call Window.SetColored(12)
        Call Window.DispObj(Region)
        Label1.Caption = "Next, the ape's eyes will be selected"
        State = 2
    End If

    If State = 0 Then
        Call Monkey.ReadImage("monkey")
        Call Window.DispObj(Monkey)
        Label1.Caption = "Next, the image will be segmented into several regions"
        State = 1
    End If
End Sub
```

3. Run your little program and enjoy a guided tour through a very common image processing example.
14.4 Using Parallel HALCON

In the example described in the previous sections, the COM interface of Standard HALCON was used. From HALCON 6.0 on, there exists a second version, Parallel HALCON, which in addition to the image processing functionality automatically exploits multi-processor or multi-core hardware and supports parallel programming.

Please note that you should use Parallel HALCON only when you need its features, e.g., because your application uses HALCON operators that can be automatically parallelized on a multi-processor or multi-core system or because you want to create multithreaded programs in which more than one thread uses HALCON operators. Otherwise, we recommend to use Standard HALCON to save computing overhead. Please refer to section 2.2 on page 7 to check whether your application can profit from Parallel HALCON.

In order to use the COM interface of Parallel HALCON in your Visual Basic application (or in other environments like Borland Delphi, Visual C++, or .NET), all you need to do is to register the corresponding DLL parhalconx.dll, e.g., via the dialog Start > Run together with the Windows Explorer: In the latter, “open” the directory bin\x86-win32 of the folder where you installed HALCON. Now, type regsvr32 the dialog Run and then drag and drop parhalconx.dll from the Explorer into the dialog, where it automatically appears with the full path. To execute the command, click OK.

Now, Parallel HALCON is automatically used whenever you add HALCON/COM to the Components of a Visual Basic project. Moreover, it is also used automatically in all projects and executables that were created before you registered parhalconx.dll. The reason for this is that, from the point of view of a COM application, the two DLLs halconx.dll and parhalconx.dll are identical, therefore one can replace the other directly. To check which one is currently registered, open the dialog Components via the menu Project and select Halcon/COM library; below the list box, the corresponding DLL is displayed as shown in figure 14.4.

Figure 14.4: The dialog Components shows whether HALCON/COM or Parallel HALCON/COM is registered.
14.5 Other Examples

There are some more pre-coded examples, so you can discover how things work with HALCON/COM. These examples can be found under the following directory:

```
%HALCONROOT%\examples\%
```

The following list shows all the supplied examples and explains their topics in short. To experiment with these examples we recommend to create a private copy in your working directory.

1. `vb\Applications\FA\`  
   An example showing how to use correlation-based pattern matching.

2. `vb\Applications\Monitoring\`  
   An example showing how to use a background estimator for traffic monitoring.

3. `vb\Online\Barcode\`  
   An online example showing how to read a bar code.

4. `vb\Online\Measure\`  
   An interactive example showing how to use the measure tool.

5. `vb\Online\Movement\`  
   An example showing how to discover movement by using the difference of images.

6. `vb\Segmentation\`  
   An example illustrating the possibilities for interactive image processing applications.

7. `vb\Tools\Calibration\`  
   An example showing how to calibrate a camera.

8. `vb\Tools\Matching\`  
   An example showing how to use shape-based matching.

9. `vb\Tools\Measure\`  
   An example showing how to measure the pins of an IC.

10. `vb\Manual\`  
    The example described in this chapter.
Part V

Programming With HALCON/C
Chapter 15

Introducing HALCON/C

HALCON/C is the interface of the image analysis system HALCON to the programming language C. Together with the HALCON library, it allows to use the image processing power of HALCON inside C programs.

This part is organized as follows: We start with a first example program to show you how programming with HALCON/C looks like. Chapter 16 on page 129 introduces the four different parameter classes of HALCON operators. We will explain the use of HALCON tuples (section 16.2.2 on page 133) for supplying operators with tuples of control parameters in great detail: Using tuples, the two calls to `select_shape` in our example program could be combined into only one call. Chapter 17 on page 139 is dedicated to the return values of HALCON operators. Chapter 18 on page 141 gives an overview over all the include files and C libraries necessary for compiling C programs and shows how to create a stand-alone application. Finally, chapter 19 on page 145 contains example solutions for some common problems in image processing (like edge detection).

15.1 A First Example

Figure 15.1 depicts the example C program together with the input image and the results. The goal is to find the eyes of the monkey by segmentation. The segmentation result is shown in figure 15.1 on the upper right side.

The program is quite self-explanatory. We will describe the basic principles nevertheless: First, all image pixels with gray values greater than 128 are selected. Then all connected components of the region formed by these pixels are calculated. The corresponding HALCON operator calculates a region tuple, and thus splits the image in different regions (objects). From these, the mandrill’s eyes are selected by their area and shape.

This example shows how easy it is to integrate HALCON operators in any C program. Their use is very intuitive: Users don’t have to think about the basic data structures and algorithms involved. And since all HALCON operators are hardware independent, users don’t even have to care about things like different I/O devices. HALCON has its own memory management and provides a sophisticated runtime environment.
```c
#include "HalconC.h"

main()
{
    Hobject mandrill,thresh,conn,area,eyes; /* required objects */
    long WindowHandle;

    open_window(0,0,512,512,"visible","",&WindowHandle); /* open window */
    read_image(&mandrill,"monkey"); /* read input image ("monkey") */
    disp_image(mandrill,WindowHandle); /* display input image */
    get_mbutton(WindowHandle,_,_,_); /* wait for mouse click */

    /* Select image region with pixels in [128,255] */
    threshold(mandrill,&thresh,128.0,255.0);
    connection(thresh,&conn); /* compute connected components */

    /* select regions with an area of at least 500 pixels */
    select_shape(conn,&area,"area","and",500.0,90000.0);

    /* select the eyes in these regions by using the anisometry feature */
    select_shape(area,&eyes,"anisometry","and",1.0,1.7);
    disp_region(eyes,WindowHandle); /* display result */
    get_mbutton(WindowHandle,_,_,_); /* wait for mouse click */
    close_window(WindowHandle); /* close window */

    /* delete image objects from the Halcon database */
    clear_obj(mandrill); clear_obj(thresh); clear_obj(conn);
    clear_obj(area); clear_obj(eyes);
}
```

Figure 15.1: Example program with input image (upper left) and segmentation results (upper right).
Chapter 16

The HALCON Parameter Classes

HALCON distinguishes four different classes of operator parameters:

- Input image objects
- Output image objects
- Input control parameters
- Output control parameters

Input parameters are passed by value, output parameters are passed by reference (using the &-operator). An exception to this rule are output control parameters of type char*. Here, the caller has to provide the memory and only a pointer to that memory is passed to the operator.

As a rule of thumb, all HALCON operators can also be called using tuples of parameters instead of single values. Take the operator connection from our example program in the previous chapter: It calculates a tuple of output image objects (the connected components). Of course there are several HALCON operators that cannot be called with tuples for some or all parameters. Whether this is the case for specific operators is described in detail in the HALCON reference manual. Unfortunately, C doesn’t provide a generic list constructor (e.g., like the one in PROLOG). Therefore, the use of tuples of control parameters is a little elaborate. Using tuples of image objects on the other hand is in no way different from using single values.

HALCON/C provides the data structure Htuple for tuples of control parameters (see section 16.2.2 on page 133 for details) and the data structure Hobject for image objects (single objects as well as object tuples — see section 16.1).

16.1 Image objects

Image processing isn’t possible without actual images. By using image objects, HALCON provides an abstract data model that covers a lot more than simple image arrays.
Basically, there are two different types of image objects:

- Images
- Regions

A region consists of a set of coordinate values in the image plane. Regions do not need to be connected and may include “holes.” They may even be larger than the image format. Internally, regions are stored in the so-called runlength encoding.

Images consist of at least one image array and a region, the so-called *domain*. The domain denotes the pixels that are “defined” (i.e., HALCON operators working on gray values will only access pixels in this region). But HALCON supports multi-channel images, too: Images may consist of an (almost) arbitrary number of channels. An image coordinate therefore isn’t necessarily represented by a single gray value, but by a vector of up to \( n \) gray values (if the coordinate lies within the image region). This may be visualized as a “stack” of image arrays instead of a single array. RGB- or voxel-images may be represented this way.

HALCON provides operators for region transformations (among them a large number of morphological operators) as well as operators for gray value transformations. Segmentation operators are the transition from images (gray values) to regions.

HALCON/C provides the data type \( \texttt{Hobject} \) for image objects (both images and regions). In fact, \( \texttt{Hobject} \) is a surrogate of the HALCON database containing all image objects. Input image objects are passed to the HALCON operators *by value* as usual, output image objects are passed *by reference*, using the \&-operator. Variables of type \( \texttt{Hobject} \) may be a single image object as well as tuples of image objects. Single objects are treated as tuples with length one.

Of course, users can access specific objects in an object tuple, too. To do so, it is necessary to extract the specific object key (converted to integer) first, using the operators \texttt{obj_to_integer} or \texttt{copy_obj}. The number of objects in a tuple can be queried with \texttt{count_obj}. To convert the keys (returned from \texttt{obj_to_integer}) back to image objects again, the operator \texttt{integer_to_obj} has to be used. It may be noted that \texttt{integer_to_obj} duplicates the image objects (Don’t worry, this doesn’t mean necessarily that the corresponding gray value arrays are duplicated too. As long as there is only read-access, a duplication of the references is sufficient). Therefore, all extracted objects have to be deleted explicitly from the HALCON database, using \texttt{clear_obj}. Figure 16.1 contains an excerpt from a C program to clarify that approach.

Some HALCON operators like \texttt{difference} allow the use of the following specific image objects as input parameters:

- \texttt{NO_OBJECTS}: An empty tuple of image objects.
- \texttt{EMPTY_REGION}: An image object with empty region (area = 0).
- \texttt{FULL_REGION}: An image object with maximal region.

These objects may be returned by HALCON operators, too.
16.2 Control parameters

HALCON/C supports the following data types as types for control parameters of HALCON operators:

- integers,
- floating point numbers,
- character arrays (strings)

As already mentioned in the introduction to this chapter, using control parameter tuples in C isn’t as elegant as using image object tuples. To circumvent the missing generic lists in C, it was necessary to introduce two different working modes into HALCON/C: The **simple mode** and the **tuple mode**. If a tuple is necessary for at least one control parameter, the tuple mode has to be used for operator calls. In tuple mode, *all* control parameters of an operator must be passed as type Htuple (*Mixing of the two modes is not possible*). The tuple mode also has to be used if the number or type of values that an operator calculates isn’t known beforehand.

Mentioning the control parameter types — How is the default type of control parameters determined for a given operator? Basically there are three ways:

```
... count_obj(&num);
/* variant 1: object key -> control parameter */
create_tuple(&Index, 1); set_i(Index, 1, 0);
create_tuple(&Num, 1); set_i(Num, num, 0);
T_obj_to_integer(objects, Index, Num, &Tsurrogates);
for (i = 0; i < num; i++)
{
    surrogate = get_i(Tsurrogates, i);
    /* process single object */
}
/* variant 2: copying objects individually */
for (i = 1; i <= num; i++)
{
    copy_obj(objects, &obj, i, 1);
    /* process single object */
}
... Figure 16.1: Accessing the i-th image object in a tuple of image objects.
```
1. The operator description in the HALCON reference manual,
2. the HALCON system operator \texttt{get\_param\_info} and
3. the description of the HALCON interface in the file \texttt{HProto.h}.

Sometimes the manuals mention more than one possible type. If only integers and floating point numbers are allowed for a parameter, values have to be passed as parameters of type \texttt{double}. For all other combinations of types, the tuple mode has to be used.

HALCON operators, that are called in tuple mode are distinguished from simple mode calls by a preceding \(T\). That means,

\[
\texttt{select\_shape}
\]

is a call of the HALCON operator \texttt{select\_shape} (as described in the HALCON reference manual) in simple mode, whereas

\[
T\texttt{\_select\_shape}
\]

is a call of the same operator in tuple mode.

\section*{16.2.1 The Simple Mode}

In the so-called \textit{simple mode}, all operators described in the HALCON reference manual can be used in a very intuitive way in your own C programs. All control parameters are variables (or constants) of the data types

- \texttt{long} for integers (HALCON type \texttt{INT\_PAR}),
- \texttt{double} for floating point numbers (\texttt{DOUBLE\_PAR}) or
- \texttt{char*} for character arrays (strings, \texttt{STRING\_PAR}).

\texttt{long} and \texttt{double} input control parameters are passed by \texttt{value} as usual, the corresponding output control parameters are passed by \texttt{reference}, using the \&-operator. String parameters are pointers to \texttt{char} in both cases. \textit{Please note, that the memory for output control parameters (esp. strings) has to be provided by the caller!} Output parameter values that are of no further interest can be denoted by the anonymous variables

- “\_” or “\_i” for \texttt{long}-parameters,
- “\_d” for \texttt{double}-parameters and
- “\_s” for \texttt{char*}-parameters

As an example for the use of anonymous variables see the small introductory program in \textit{figure 15.1} on page 128: We coded \texttt{get\_mbutton(WindowHandle,_,_,_,_)}, because the actual button pressed is of no further interest.

Examples for HALCON operator calls in simple mode can be found in the C programs in \textit{figure 16.1} and \textit{figure 15.1} on page 128.
16.2.2 The Tuple Mode

We mentioned already that control parameter tuples for HALCON operators need special treatment. In this chapter we will give the details on how to construct and use those tuples. The HALCON reference manual describes a large number of operators that don’t operate on single control values but on tuples of values. Using those operators, it is easy to write very compact and efficient programs, because often it is possible to combine multiple similar operator calls into a single call.

Unfortunately, C provides no generic tuple or list constructor. In contrast, HALCON allows tuples with mixed types as control parameter values (e.g., integers mixed with floating point numbers).

Therefore, in addition to the very intuitive simple mode there is another mode in HALCON/C: the tuple mode. Using this mode is a little more elaborate. If at least one of the control parameters of a HALCON operator is passed as a tuple, the tuple mode has to be used for all control parameters (Mixing of both modes isn’t possible). Furthermore, the tuple mode also has to be used if the number or type of the calculated values aren’t known beforehand.

Syntactically, tuple mode is distinguished from simple mode by a T_ preceding the operator name. For example, calling disp_circle in tuple mode is done by

\[ T_{\text{disp\_circle}}(...) \].

To ease the usage of the tuple mode, HALCON/C provides the abstract data type Htuple for control parameter tuples. Objects of type Htuple may be constructed using values of the types

- long for integers (HALCON type INT_PAR),
- double for floating point numbers (DOUBLE_PAR) or
- char* for character arrays (strings, STRING_PAR)

in arbitrary combination. Control parameter tuples must be created, deleted, and manipulated using the appropriate HALCON/C procedures only (overview in figures 16.2 and 16.3).

The rules for parameter passing are valid in tuple mode, too: Input control parameters (type Htuple) are passed by value as usual, output control parameters are passed by reference, using the &-operator. Output parameters that are of no further interest can be denoted by the anonymous variable “_t” instead of a “dummy” tuple.

Let’s summarize the five most important steps when calling a HALCON operator in tuple mode:

1. **step** First, memory must be allocated for all tuples of input control parameters, using create_tuple. Memory for output control parameter tuples is allocated by HALCON/C (a call of create_tuple isn’t necessary).

2. **step** Now, the input control parameter tuples are constructed, using the appropriate procedures set_*., set_s, which inserts a string into a tuple allocates the needed memory by itself and then copies the string.

3. **step** Then, the HALCON operator is actually called. The operator name is (as already explained) preceded by a T_ to denote tuple mode.
4. step Further processing of the output parameter tuples takes place, using the procedures `length_tuple`, `get_type` and `get_*`. When processing strings (using `get_s`), please note that the allocated memory is freed automatically upon deleting the tuple with `destroy_tuple`. If the string has to be processed even after the deletion of the tuple, the whole string must be copied first. The maximal string length (incl. termination character “\0”) in HALCON is `MAX_STRING` (1024 in HALCON version 8.0).

5. step Finally the memory allocated by all the tuples (input and output) has to be freed again. This is done with `destroy_tuple`. If you still need the values of the tuple variables, remember to copy them first. Now, the whole series can start again — using different or the same tuple variables.

```c
void create_tuple(tuple,length) or macro CT(tuple,length)
    Htuple *tuple;
    long length;
    /* creates a tuple that can hold 'length' entries */

void destroy_tuple(tuple) or macro DT(tuple)
    Htuple tuple;
    /* deletes a tuple (if the tuple contains string entries, */
    /* the memory allocated by the strings is freed, too) */

long length_tuple(tuple) or macro LT(tuple)
    Htuple tuple;
    /* returns the length of a tuple (number of entries) */

void set_i(tuple,val,index) or macro SI(tuple,val,index)
    Htuple tuple;
    long val;
    long index;
    /* inserts an integer with value 'val' into a tuple at */
    /* position 'index' ('index' in [0,length_tuple(tuple) - 1]) */

void set_d(tuple,val,index) or macro SD(tuple,val,index)
    Htuple tuple;
    double val;
    long index;
    /* inserts a double with value 'val' into a tuple at */
    /* position 'index' ('index' in [0,length_tuple(tuple) - 1]) */

void set_s(tuple,val,index) or macro SS(tuple,val,index)
    Htuple tuple;
    char *val;
    long index;
    /* inserts a copy of string 'val' into a tuple at */
    /* position 'index' ('index' in [0,length_tuple(tuple) - 1]). */
    /* The memory necessary for the string is allocated by set_s. */
```

Figure 16.2: HALCON/C Htuple procedures (part one).
Before we end this chapter with a short example program, we will explain an alternative **generic calling mechanism** for HALCON operators in tuple mode. This mechanism is intended for the use in interpreters or graphical user interfaces:

```
T_call_halcon(ProcName)
```

calls the HALCON operator ProcName in tuple mode. To do so, the operator parameters have to be set first, using

```
set_in_opar, set_out_opar, set_in_tpar and set_out_tpar
```

Accessing these parameters is still possible with the ordinary tuple procedures. Figure 16.4 summarizes the procedures of the generic HALCON/C calling interface.

But now to the mentioned example program (see figure 16.5 or the file example3.c): The aim is to obtain information about the current HALCON system state. The operator `get_system('?',Values)`
void set_in_opar(obj,par) or macro IO(obj,par)
    Hobject obj;
    int par;
    /* defines 'obj' as input image object parameter no. 'par' */
    /* (inside the input image object parameter parameter class) */

void set_out_opar(obj,par) or macro OO(obj,par)
    Hobject *obj;
    int par;
    /* defines 'obj' as output image object parameter no. 'par' */
    /* (inside the output image object parameter parameter class) */

void set_in_tpar(tuple,par) or macro IT(tuple,par)
    Htuple tuple;
    int par;
    /* defines 'tuple' as input control parameter no. 'par' */
    /* (inside the input control parameter parameter class) */

void set_out_tpar(tuple,par) or macro OT(tuple,par)
    Htuple *tuple;
    int par;
    /* defines 'tuple' as output control parameter no. 'par' */
    /* (inside the output control parameter parameter class) */

Herror T_call_halcon(ProcName) or macro TC(ProcName)
    char *ProcName;
    /* calls the Halcon operator 'ProcName' using tuple mode; */
    /* input and output parameters of 'ProcName' must be declared */
    /* using set_in_*par and set_out_*par first */

Figure 16.4: Generic calling mechanism for the HALCON/C tuple mode.

(here in HDevelop syntax) returns all system flags with their current values. Since in our case neither number nor type of the output parameters is known beforehand, we have to use tuple mode for the actual operator call in HALCON/C. The rest of the program should be self explanatory.
#include "HalconC.h"

main ()
{
    Htuple In,SysFlags,Info;  /* tuple variables */
    long i,num;

    printf("system information:\n");
    create_tuple(&In,1);  /* prepare first query */
    set_s(In,"?",0);  /* only value of 'In': '?' */
    T_get_system(In,&SysFlags);  /* first query */
    destroy_tuple(In);  /* free parameter */
    num = length_tuple(SysFlags);  /* number of system flags */
    for (i=0; i<num; i++)
    {
        /* determine the value of the i-th system flag: */
        create_tuple(&In,1);  /* prepare query */
        set_s(In,get_s(SysFlags,i),0);  /* insert i-th system flag */
        printf("%s ",get_s(SysFlags,i));  /* print name */
        T_get_system(In,&Info);  /* get corresponding info */
        destroy_tuple(In);  /* free parameter */
        switch(get_type(Info,0))  /* print the value according to the flag's type: */
        {
            case INT_PAR:  printf("(int): %ld\n",get_i(info,0));
                break;
            case DOUBLE_PAR:  printf("(double): %f\n",get_d(info,0));
                break;
            case STRING_PAR:  printf("(string): %s\n",get_s(info,0));
                break;
        }
        destroy_tuple(Info);  /* free parameter */
    }  /* for(i=... */
}

Figure 16.5: Tuple mode example program: Printing the current HALCON system state.
The HALCON Parameter Classes
Chapter 17

Return Values of HALCON Operators

HALCON operator return values (type Herror) can be divided into two categories:

- Messages (H_MSG_*) and
- Errors (H_ERR_*).

According to its procedural concept, HALCON distinguishes four kinds of messages:

- **H_MSG_TRUE**: The operator finished without error and returns the boolean value true.
- **H_MSG_FALSE**: The operator finished without error and returns the boolean value false.
- **H_MSG_VOID**: The operator finished without error, but doesn’t return a value.
- **H_MSG_FAIL**: the operator finished without error (!) and returns “operation failed”. This could mean that the operator doesn’t consider itself relevant for the operation or that a specific event didn’t happen.

Nearly all HALCON operators return **H_MSG_TRUE**, if no error occurs.

Errors in HALCON operators usually result in an exception, i.e., a program abort with the appropriate error message in HALCON/C (default exception handling). However, users can disable this mechanism (with a few exceptions, like errors in Htuple operators), using

```c
set_check("~give_error");
```

to provide their own error handling routines. In that case, the operator **get_error_text** is very useful: This operator returns the plain text message for any given error number. Finally, the operator

```c
set_check("give_error");
```

enables the HALCON error handling again. Several examples showing the handling of error messages can be seen in the file example5.c.
The HALCON distribution contains examples for building an application with HALCON/C. Here is an overview of HALCON/C (Windows notation of paths):

`include\HalconC.h`: include file; contains all user-relevant definitions of the HALCON system and the declarations necessary for the C interface.

`bin\%HALCONARCH\%halcon.dll`,
`lib\%HALCONARCH\%halcon.lib`:
   The HALCON library (Windows).

`bin\%HALCONARCH\%halconc.dll`,
`lib\%HALCONARCH\%halconc.lib`:
   The HALCON/C library (Windows).

`bin\%HALCONARCH\%parhalcon.dll, parhalconc.dll`,
`lib\%HALCONARCH\%parhalcon.lib, parhalconc.lib`:
   The corresponding libraries of Parallel HALCON (Windows).

`lib/$HALCONARCH/libhalcon.so`:
   The HALCON library (Linux/UNIX).

`lib/$HALCONARCH/libhalconc.so`:
   The HALCON/C library (Linux/UNIX).

`lib/$HALCONARCH/libparhalcon.so,libparhalconc.so`:
   The corresponding libraries of Parallel HALCON (Linux/UNIX).

`include\HProto.h`:
   External function declarations.
There are several example programs in the HALCON/C distribution (`examples\c\source\`). To experiment with these examples we recommend to create a private copy in your working directory.

- **example1.c** reads an image and demonstrates several graphics operators.
- **example2.c** introduces several image processing operators.
- **example3.c** is an example for the usage of the tuple mode.
- **example4.c** shows more (basic) image processing operators like the sobel filter for edge detection, region growing, thresholding, histograms, the skeleton operator, and the usage of different color lookup tables.
- **example5.c** describes the HALCON messages and error handling.
- **example6.c** demonstrates the generic calling interface for the tuple mode (`T_call_halcon`).
- **example7.c** describes the handling of RGB images.
- **example8.c** demonstrates the creation of an image from user memory.
- **example9.c** describes some additional handling of RGB images.

A special case is the example program **example_multithreaded1.c**. It demonstrates the use of Parallel HALCON in a multithreaded application. Please note, that this example must be linked to the libraries...
of Parallel HALCON as described in the following sections. Of course, it does not make sense to run on a single-processor or single-core computer.

In the following, we briefly describe the relevant environment variables; see the Installation Guide, section A.2 on page 54, for more information, especially about how to set these variables. Note, that under Windows, all necessary variables are automatically set during the installation.

While a HALCON program is running, it accesses several files internally. To tell HALCON where to look for these files, the environment variable HALCONROOT has to be set. HALCONROOT points to the HALCON home directory; it is also used in the sample makefile.

The variable HALCONARCH describes the platform HALCON is used on. Please refer to section 1.1 on page 3 for more information.

If user-defined packages are used, the environment variable HALCONEXTENSIONS has to be set. HALCON will look for possible extensions and their corresponding help files in the directories given in HALCONEXTENSIONS.

Two things are important in connection with the example programs: The default directory for the HALCON operator read_image to look for images is %HALCONROOT%\images. If the images reside in different directories, the appropriate path must be set in read_image or the default image directory must be changed, using set_system("image_dir","..."). This is also possible with the environment variable HALCONIMAGES. It has to be set before starting the program.

The second remark concerns the output terminal under Linux/UNIX. In the example programs, no host name is passed to open_window. Therefore, the window is opened on the machine that is specified in the environment variable DISPLAY. If output on a different terminal is desired, this can be done either directly in open_window(...,"hostname",...) or by specifying a host name in DISPLAY.

In order to link and run applications under Linux/UNIX, you have to include the HALCON library path $HALCONROOT/lib/$HALCONARCH in the system variable LD_LIBRARY_PATH.

### 18.1 Creating Applications Under Windows

Your own C programs that use HALCON operators must include the file HalconC.h, which contains all user-relevant definitions of the HALCON system and the declarations necessary for the C interface. Do this by adding the command

```
#include "HalconC.h"
```

near the top of your C file. In order to create an application you must link the library halconc.lib/.dll to your program.

The example projects show the necessary Visual C++ settings. For the examples the project should be of the WIN 32 ConsoleApplication type. Please note that the Visual C++ compiler implicitly calls “Update all dependencies” if a new file is added to a project. Since HALCON runs under Linux/UNIX as well as under Windows, the include file HalconC.h includes several Linux/UNIX-specific headers as well if included under Linux/UNIX. Since they don’t exist under Windows, and the Visual C++ compiler is dumb enough to ignore the operating-system-specific cases in the include files, you will get a number of warning messages about missing header files. These can safely be ignored.
Please assure that the stacksize is sufficient. Some sophisticated image processing problems require up to 6 MB stacksize, so make sure to set the settings of your compiler accordingly (See your compiler manual for additional information on this topic).

**Parallel HALCON applications:** If you want to use Parallel HALCON, you have to link the libraries parhalcon.lib/.dll and parhalconc.lib/.dll instead of halcon.lib/.dll and halconc.lib/.dll in your project.

### 18.2 Creating Applications Under Linux/UNIX

Your own C programs that use HALCON operators must include the file `HalconC.h`, which contains all user-relevant definitions of the HALCON system and the declarations necessary for the C interface. Do this by adding the command

```c
#include "HalconC.h"
```

near the top of your C file. Using this syntax, the compiler looks for `HalconC.h` in the current directory only. Alternatively you can tell the compiler where to find the file, giving it the `-I<pathname>` command line flag to denote the include file directory.

To create an application, you have to link two libraries to your program: The library `libhalconc.so` contains the various components of the HALCON/C interface. `libhalcon.so` is the HALCON library.

**Parallel HALCON applications:** If you want to use Parallel HALCON, you have to link the libraries `libparhalconc.so` and `libparhalcon.so` instead.

Please take a look at the example makefiles for suitable settings. If you call `gmake` without further arguments, the example application `example1` will be created. To create the other example applications (e.g., `example2`), call

```
make example2
```

You can use the example makefiles not only to compile and link the example programs but also your own programs (called e.g. `test.c`) by calling

```
make test
```

You can link the program to the Parallel HALCON libraries by adding `PAR=1` to the make command, for example

```
make test PAR=1
```
Chapter 19

Typical Image Processing Problems

This final chapter shows the possibilities of HALCON and HALCON/C on the basis of several simple image processing problems.

19.1 Thresholding

One of the most common HALCON operators is the following:

```c
read_image(&Image,"File_xyz");
threshold(Image,&Thres,0.0,120.0);
connection(Thres,&Conn);
select_shape(Conn,&Result,"area","and",10.0,100000.0);
```

Step-by-step explanation of the code:

- First, all image pixels with gray values between 0 and 120 (channel 1) are selected.
- The remaining image regions are split into connected components.
- By suppressing regions that are too small, noise is eliminated.

19.2 Detecting Edges

The following HALCON/C sequence is suitable for edge detection:

```c
read_image(&Image,"File_xyz");
sobel_amp(Image,&Sobel,"sum_abs",3);
threshold(Sobel,&Max,30.0,255.0);
skeleton(Max,&Edges);
```
Some remarks about the code:

- Before filtering edges with the sobel operator, a low pass filter may be useful to suppress noise.
- Apart from the sobel operator, filters like `edges_image`, `roberts`, `bandpass_image` or `laplace` are suitable for edge detection, too.
- The threshold (30.0, in this case) has to be selected depending on the actual images (or depending on the quality of the edges found in the image).
- Before any further processing, the edges are reduced to the width of a single pixel, using `skeleton`.

### 19.3 Dynamic Threshold

Among other things, the following code is suitable for edge detection, too:

```plaintext
read_image(&Image,"File_xyz");
mean_image(Image,&Lp,11,11);
dyn_threshold(Image,Lp,&Thres,5.0,"light");
```

- The size of the filter mask (11 x 11, in this case) depends directly on the size of the expected objects (both sizes are directly proportional to each other).
- In this example, the dynamic threshold operator selects all pixels that are at least 5 gray values brighter than their surrounding (11 x 11) pixels.

### 19.4 Simple Texture Transformations

Texture transformations are used to enhance specific image structures. The behavior of the transformation depends on the filters used (HALCON provides 16 different texture filters).

```plaintext
read_image(&Image,"File_xyz");
Filter = "ee";
texture_laws(Image,&TT,Filter,2,5);
mean_image(TT,&Lp,31,31);
threshold(Lp,&Seg,30.0,255.0);
```

- `mean_image` has to be called with a large mask to achieve a sufficient generalization.
- It is also possible to calculate several different texture transformations and to combine them later, using `add_image`, `mult_image` or a similar operator.
19.5 Eliminating Small Objects

The following morphological operation eliminates small image objects and smoothes the boundaries of the remaining objects:

```c
... segmentation(Image,&Seg);
gen_circle(&Mask,100.0,100.0,3.5);
opening(Seg,Mask,&Res);
```

- The size of the circular mask (3.5, in this case) determines the smallest size of the remaining objects.
- It is possible to use any kind of mask for object elimination (not only circular masks).
- `segmentation(...)` is used to denote a segmentation operator that calculates a tuple of image objects (`Seg`).

19.6 Selecting Specific Orientations

Yet another application example of morphological operations is the selection of image objects with specific orientations:

```c
... segmentation(Image,&Seg);
gen_rectangle2(&Mask,100.0,100.0,0.5,21.0,2.0);
opening(Seg,Mask,&Res);
```

- The rectangle’s shape and size (length and width) determine the smallest size of the remaining objects.
- The rectangle’s orientation determines the orientation of the remaining regions (In this case, the main axis and the horizontal axis form an angle of 0.5 rad).
- Lines with an orientation different from the mask’s (i.e., the rectangle’s) orientation are suppressed.
- `segmentation(...)` is used to denote a segmentation operator that calculates a tuple of image objects (`Seg`).

19.7 Smoothing Region Boundaries

The third (and final) application example of morphological operations covers another common image processing problem — the smoothing of region boundaries and closing of small holes in the regions:

```c
... segmentation(Image,&Seg);
gen_circle(&Mask,100.0,100.0,3.5);
closing(Seg,Mask,&Res);
```
• For the smoothing of region boundaries, circular masks are suited best.
• The mask size determines the degree of the smoothing.
• `segmentation(...)` is used to denote a segmentation operator that calculates a tuple of image objects (Seg).
Part VI

Using HDevEngine
Chapter 20

Introducing HDevEngine

As the name suggests, HDevEngine is the “engine” of HDevelop. This chapter briefly introduces you to its basic concepts. Chapter 21 on page 153 explains how to use it in C++ applications, chapter 22 on page 169 how to use it in .NET applications (C#, Visual Basic .NET, etc.), and chapter 23 on page 179 how to use it in COM applications (Visual Basic 6.0). Additional information that is independent of the used programming language can be found in chapter 24 on page 193.

What Can You Do With HDevEngine?

With HDevEngine, you can execute complete HDevelop programs or individual procedures from a C++ application or an application that can integrate .NET or COM objects, e.g., C#, Visual Basic .NET, or Visual Basic 6.0. Thus, you can use HDevelop not only for prototyping, but also to completely develop and run the machine vision part of your application.

Because HDevEngine acts as an interpreter, you can modify the HDevelop program or procedure without needing to compile and link the application (if you don’t change the procedure’s signature), as would be necessary if you export the program or procedure and integrate the code manually. This means that you can easily update the machine vision part of an application by replacing individual HDevelop files.

What HDevEngine Does Not Do

Note that HDevEngine does not implement the complete functionality of HDevelop, only what is necessary to execute programs and procedures. In particular, it does not implement the display of variables and results in the graphics window, i.e., the internal operators like dev_display. However, you can “redirect” these operators to your own implementation. Thus, you can decide which visualization is important and where and how it is to take place.

External or Local Procedures?

You can execute both local and external HDevelop procedures with HDevEngine. The main difference is that external procedures are loaded individually, while local ones are loaded by loading the complete HDevelop program.

In the following, we use the term “procedure” as a synonym to “external procedure”.

**What is HDevEngine?**

HDevEngine is provided as a C++ and COM class library and a .NET assembly. Like HDevelop, it internally uses HALCON/C. Please note that the interfaces of the C++ classes also require you to use HALCON/C. In contrast, the .NET and COM classes act as a wrapper around the C++ classes and allow you to use HALCON/.NET and HALCON/COM, respectively.

HDevEngine consists of the following classes:

- **HDevEngine (C++, .NET), HDevEngineX (COM)**
  This is the main class of HDevEngine. With instances of it you load and execute HDevelop programs.

- **HDevProcedureCall (C++, .NET), HDevProcedureCallX (COM)**
  With this class you pass input parameters to HDevelop procedures and retrieve their output parameters.

  As noted above, HDevEngine does not implement internal HDevelop operators like dev_display. All HDevEngine variants provide a class or interface to provide your own implementation for those operators that are useful in your application. HDevEngine/.NET also provides two convenience classes that provide a default implementation of the operators.

- **HDevEngineException (C++, .NET)**
  Instances of this class are “thrown” if an exception occurs inside HDevEngine, e.g., because the application tried to load a non-existing program or because of an error inside an operator in the executed program or procedure.

  Note that in the COM version of HDevEngine, the standard error handling mechanism is used instead of a separate class.

**Parallel HDevEngine**

Like HALCON, the two versions of HDevEngine are provided in two variants: Standard and Parallel. The latter are based on Parallel HALCON, i.e., they use the parallel versions of the HALCON library and of HALCON/C, HALCON/.NET, and HALCON/COM, respectively.

Please note that the main purpose of Parallel HDevEngine is to allow to use the automatic operator parallelization in Parallel HALCON.

In addition, you can use Parallel HDevEngine in multithreaded applications, because it is thread-safe. However, **Parallel HDevEngine is not reentrant**. This means that you can not execute different procedures (or the same procedure or program) in different threads simultaneously. Tips about using Parallel HDevEngine in multithreaded applications can be found in section 24.2.2 on page 198.
Chapter 21

HDevEngine in C++ Applications

This chapter explains how to use HDevEngine in C++ applications. Section 21.1 quickly summarizes some basic information, e.g., how to compile and link such applications. Section 21.2 then explains how to use HDevEngine by presenting example applications for

- executing an HDevelop program (section 21.2.1 on page 155),
- executing an (external) HDevelop procedure (section 21.2.2 on page 157),
- implementing display operators (section 21.2.3 on page 160), and
- error handling (section 21.2.4 on page 165).

An overview about the classes of HDevEngine and their methods can be found in section 24.1 on page 193.

21.1 Basics

An important point to keep in mind when using the C++ version of HDevEngine is that it is based on HALCON/C and not, as one would expect, on HALCON/C++. The reason for this surprising fact is that HDevelop is also based on HALCON/C.

If you are normally using HALCON/C++, the following aspects of HALCON/C take some time getting used to:

- You must take care of the memory management yourself, i.e., destroy iconic objects using clear_obj (see section 16.1 on page 129) and create and destroy tuples using create_tuple and destroy_tuple, respectively (see section 16.2.2 on page 133). The only exception are tuple variables that are passed to HDevEngine as output parameters of a procedure call: They are “created” by HDevEngine, you must only destroy them.
- You must set and access tuple elements explicitly using set_* and get_* (see section 16.2.2 on page 133).
• In HALCON/C, **operators exist in two variants**, one for the so-called simple mode (only single parameter values) and one for the tuple mode (see section 16.2 on page 131).

You create executable HDevEngine applications in a way similar to normal HALCON/C applications. Chapter 18 on page 141 describes this in detail; here, we summarize the most important points and include the extensions for HDevEngine:

• In your application, you include the main header file HalconC.h and HDevEngine's header file HDevEngine.h:

```cpp
#include "HalconC.h"
#include "HDevEngine.h"
```

• To compile the application, use the following include paths on Windows systems

```cmd
/I "$\text{(HALCONROOT)}\text{\include}" /I "$\text{(HALCONROOT)}\text{\include\hdevengine}"
```

and on UNIX systems

```cmd
-I$(HALCONROOT)/include -I$(HALCONROOT)/include/hdevengine
```

• Link the following libraries on Windows systems

```cmd
/libpath:"$(HALCONROOT)/lib/$(HALCONARCH)" hdevengine.lib halconc.lib
```

and on UNIX systems

```cmd
-L$(HALCONROOT)/lib/$(HALCONARCH) -lhdevengine -lhalconc -lhalcon
```

**Parallel HDevEngine applications:** If you want to use Parallel HDevEngine, link the following libraries on Windows systems

```cmd
/libpath:"$(HALCONROOT)/lib/$(HALCONARCH)" parhdevengine.lib parhalconc.lib
```

and on UNIX systems

```cmd
-L$(HALCONROOT)/lib/$(HALCONARCH) -lparhdevengine -lparhalconc -lparhalcon
```

### 21.2 Examples

This section explains how to employ HDevEngine based on example applications, which reside in the subdirectory examples\hdevengine\cpp of the folder where you installed HALCON. Like the examples for HALCON/C described in chapter 18 on page 141, they are provided as Visual Studio projects for Windows systems and with makefiles for UNIX systems. The only difference in the directory structure is that the source code files are in a separate subdirectory **source**.
21.2.1 Executing an HDevelop Program

In this section, we explain how to load and execute an HDevelop program with HDevEngine. The code fragments stem from the example application exec_program (source file exec_program.cpp), which checks the boundary of a plastic part for fins. Figure 21.1 shows a screenshot of the application.

First, we include the main header files of HALCON/C and of HDevEngine:

```c
#include "HalconC.h"
#include "HDevEngine.h"
```

In the main procedure, we create an instance of the main HDevEngine class HDevEngine and store a pointer to it in a global variable, so that other procedures in the application can use it. Then, we call a procedure that contains the “action” of the application. Afterwards, we destroy the instance of HDevEngine by calling its destructor.

```c
HDevEngine *MyEngine;

int main(int argc, char *argv[]) {
    MyEngine = new HDevEngine;
    DetectFin();
    delete MyEngine;

    return 0;
}
```

Let’s take a closer look at the “action” procedure. First, we store the path to the HDevelop program and the external procedure path in string variables, with a suitable syntax for the used platform:
void DetectFin(void)
{
    char ProgramPathStr[1024], ProcPathStr[1024];

#ifdef _WIN32
    sprintf(ProgramPathStr,"%s\examples\hdevengine\hdevelop\" "fin_detection.dev",getenv("HALCONROOT"));
    sprintf(ProcPathStr,"%s\examples\hdevengine\procedures;", getenv("HALCONROOT"));
#else
    sprintf(ProgramPathStr,"%s/examples/hdevengine/hdevelop/" "fin_detection.dev",getenv("HALCONROOT"));
    sprintf(ProcPathStr,"%s/examples/hdevengine/procedures;", getenv("HALCONROOT"));
#endif

If the HDevelop program calls external procedures, you must set the external procedure path with the method SetProcedurePath:

    MyEngine->SetProcedurePath(ProcPathStr);

Now, we load the HDevelop program with the method LoadProgram. The call is encapsulated in a try...catch-block to handle exceptions occurring in the HDevEngine method, e.g., because the file name was not specified correctly. A detailed description of error handling can be found in section 21.2.4 on page 165.

    try
    {
        MyEngine->LoadProgram(ProgramPathStr);
    }
    catch (HDevEngineException &hdev_exception)
    {
        PrintErrorMessage(hdev_exception.Message());
        exit(0);
    }

If the program could be loaded successfully, we execute it with the method ExecuteProgram:

    try
    {
        MyEngine->ExecuteProgram();
    }
    catch (HDevEngineException &hdev_exception)
    {
        PrintErrorMessage(hdev_exception.Message());
        exit(0);
    }

That’s all you need to do to execute an HDevelop program. How to display the results is described in section 21.2.3 on page 160.
21.2.2 Executing an External HDevelop Procedure

In this section, we explain how to load and execute an external HDevelop procedure with HDevEngine. The code fragments in the following stem from the example application exec_extproc (source file exec_extproc.cpp), which, like the example described in the previous section, checks the boundary of a plastic part for fins. Figure 21.2 shows a screenshot of the application.

In contrast to the previous example, the result display is programmed explicitly in HALCON/C instead of relying on the internal display operators. How to provide your own implementation of the internal display operators is described in section 21.2.3 on page 160.

As when executing an HDevelop program, we include the main header files of HALCON/C and of HDevEngine and create an instance of the main HDevEngine class HDevEngine in the main procedure and store a pointer to it in a global variable:

```c
#include "HalconC.h"
#include "HDevEngine.h"

HDevEngine *MyEngine;

int main(int argc, char *argv[]) {
    MyEngine = new HDevEngine;
    DetectFin();
    delete MyEngine;

    return 0;
}
```

For the display of results, we declare a global variable for the window handle:

```c
Hlong WinID;
```
In the “action” routine, we declare variables for the procedure call and then set the external procedure path with the method SetProcedurePath:

```cpp
void DetectFin(void)
{
    Hobject Image, FinRegion;
    Htuple FinArea;
    HDevProcedureCall *proc_call;
    Hlong FGHandle;
    char FinAreaStr[64];
    char ProcPathStr[1024], ImageSeqString[1024];

    #ifdef _WIN32
        sprintf(ProcPathStr, "%s\examples\hdevengine\procedures;",
                 getenv("HALCONROOT"));
        sprintf(ImageSeqString, "fin");
    #else
        sprintf(ProcPathStr, "%s/examples/hdevengine/procedures: ",
                 getenv("HALCONROOT"));
        sprintf(ImageSeqString, "fin");
    #endif

    MyEngine->SetProcedurePath(ProcPathStr);
}
```

Now, we load the HDevelop procedure with the method LoadExternalProcedure. The call is encapsulated in a try...catch-block to handle exceptions occurring in the HDevEngine method, e.g., because the file name or the procedure path was not specified correctly. A detailed description of error handling can be found in section 21.2.4 on page 165.

```cpp
try
{
    MyEngine->LoadExternalProcedure("detect_fin");
}
catch (HDevEngineException &hdev_exception)
{
    PrintErrorMessage(hdev_exception.Message());
    exit(0);
}
```

Executing a procedure consists of multiple steps. First, we create an instance of the class HDevProcedureCall, specifying the name of the procedure (which at this point should be already loaded):

```cpp
proc_call = new HDevProcedureCall("detect_fin");
```
Before executing it, we open and initialize the graphics window in which the results are to be displayed and load an example image sequence:

```c
open_window(100,100,384,288,0,visible,"",&WinID);
set_part(WinID,0,0,575,767);
set_draw(WinID,"margin");
set_line_width(WinID,4);
open_framegrabber("File",1,1,0,0,0,default,-1,default,-1,default,
    ImageSeqString,default,-1,-1,&FGHandle);
```

Each image should now be processed by the procedure, which has the following signature, i.e., it expects an image as (iconic) input parameter and returns the detected fin region and its area as iconic and control output parameter, respectively:

```c
procedure detect_fin (Image: FinRegion: : FinArea)
```

We pass the image as input object by storing it in the instance of HDevProcedureCall with the method SetInputObject. Which parameter to set is specified via its index; there is also a method to specify it via its name (see section 24.1.2 on page 195):

```c
for (long i=0; i<3; i++)
{
    grab_image(&Image,FGHandle);
    disp_obj(Image,WinID);
    proc_call->SetInputObject(1,Image);
}
```

Now, we execute the procedure by passing the instance of HDevProcedureCall to the HDevEngine with the method ExecuteProcedure.

```c
try
{
    MyEngine->ExecuteProcedure(proc_call);
}
```

If an exception occurs, we must remember to free the memory of the image object:

```c
catch (HDevEngineException &hdev_exception)
{
    PrintErrorMessage(hdev_exception.Message());
    clear_obj(Image);
    continue;
}
```

If the procedure was executed successfully, we can access its results, i.e., the fin region and its area, with the methods GetOutputObject and GetOutputCtrl of the class HDevProcedureCall; again, you can specify the parameter via its index or name (see section 24.1.2 on page 195).

```c
proc_call->GetOutputObject(1,&FinRegion);
proc_call->GetOutputCtrl(1,&FinArea);
```
Now, we display the results in the graphics window. Note how we access the area by selecting the first element of the returned tuple with `get_i`:

```cpp
disp_obj(Image,WinID);
set_color(WinID,"red");
disp_obj(FinRegion,WinID);
set_color(WinID,"white");
sprintf(FinAreaStr,"Fin Area: %ld",get_i(FinArea,0));
set_tposition(WinID,150,20);
write_string(WinID,FinAreaStr);
```

After the image is processed, we destroy the iconic objects and the tuple using `clear_obj` and `destroy_tuple`:

```cpp
clear_obj(Image);
clear_obj(FinRegion);
destroy_tuple(FinArea);
}
```

At the end of the application, we destroy the instance of `HDevProcedureCall` and close the image acquisition device and the graphics window:

```cpp
delete proc_call;
close_framegrabber(FGHandle);
close_window(WinID);
}
```

### 21.2.3 Display

In this section, we explain how to provide your own implementation of HDevelop’s internal display operators. The code fragments in the following stem from the example application `exec_program` (source file `exec_program.cpp`), which was already discussed in section 21.2.1 on page 155.

In fact, HDevEngine does not provide an implementation of the internal display operators but provides the class `HDevOperatorImpl`, which contains empty virtual methods for all those operators that you can implement yourself. The methods are called like the object-oriented version of the operators, e.g., `DevDisplay` for `dev_display` and have the same parameters (see section 24.1.3 on page 196 for the definition of the class).

The first step towards the implementation is to derive a child of this class and to specify all methods that you want to implement. The example program implements the operators `dev_close_window`, `dev_set_window`, `dev_display`, `dev_set_draw`, `dev_set_color`, `dev_set_part`, `dev_set_line_width`, and `dev_open_window`:
class MyHDevOperatorImpl: public HDevOperatorImpl
{
public:
  virtual int DevCloseWindow(void);
  virtual int DevSetWindow(Htuple win_id);
  virtual int DevDisplay(Hobject obj);
  virtual int DevSetDraw(Htuple draw);
  virtual int DevSetColor(Htuple color);
  virtual int DevSetPart(Htuple row1, Htuple col1,
                          Htuple row2, Htuple col2);
  virtual int DevSetLineWidth(Htuple width);
  virtual int DevOpenWindow(Htuple row, Htuple col, Htuple width,
                            Htuple height, Htuple background, Htuple *win_id);
};

In addition to these methods, the class contains a tuple variable to store multiple window handles, which
is created and destroyed in the constructor and destructor of the class, respectively. In the executed
HDevelop program, two graphics windows are used, one for the main display and one for zooming into
the image (see figure 21.1 on page 155).

MyHDevOperatorImpl(void);
~MyHDevOperatorImpl(void);

protected:
  Htuple   WinIDs;

With the method SetHDevOperatorImpl, you pass an instance of your version of HDevOperatorImpl
to HDevEngine, which then calls its methods when the corresponding operator is used in the HDevelop
program or procedure.

int main(int argc, char *argv[])
{
  MyEngine = new HDevEngine;

  MyEngine->SetHDevOperatorImpl(new MyHDevOperatorImpl);

  DetectFin();
  ...

Now, we take a closer look at the implementation of the display operators in the example. It tries to
mimic the behavior in HDevelop: Multiple graphics windows can be open, with one being “active”. The
handle of this active window is stored in a global variable:

Hlong   WinID = -1;

If the variable contains a valid handle, the methods for the internal display operators simply call the
 corresponding non-internal display operator, e.g., a call to dev_display in the HDevelop program is
 “redirected” in DevDisplay to disp_obj, with the iconic object to display and the handle of the active
window as parameters:
```c++
int MyHDevOperatorImpl::DevDisplay(Hobject obj)
{
    if (WinID < 0)
        return H_MSG_TRUE;
    disp_obj(obj,WinID);
    return H_MSG_TRUE;
}
```

Similarly, `dev_set_draw` is redirected in `DevSetDraw` to `set_draw`; note how the first element of the input tuple is accessed using `get_s` to pass it to the (simple-mode version) of `set_draw`:

```c++
int MyHDevOperatorImpl::DevSetDraw(Htuple draw)
{
    if (WinID < 0)
        return H_MSG_TRUE;
    set_draw(WinID,get_s(draw,0));
    return H_MSG_TRUE;
}
```

The implementation of `dev_set_part` (DevSetPart) shows how to access integer elements of input tuples using `get_i`:

```c++
int MyHDevOperatorImpl::DevSetPart(Htuple row1, Htuple col1,
                                        Htuple row2, Htuple col2)
{
    if (WinID < 0)
        return H_MSG_TRUE;
    set_part(WinID,get_i(row1,0),get_i(col1,0),get_i(row2,0),get_i(col2,0));
    return H_MSG_TRUE;
}
```

As you can see, these operators can be implemented quite easily. The implementation of the operators for handling graphics windows is more complicated. As noted above, the child class of `HDevOperatorImpl` contains a tuple in which the window handles are stored. In the constructor of the class, this tuple is created using `create_tuple` and initialized with a single value, the handle of the active window:

```c++
MyHDevOperatorImpl::MyHDevOperatorImpl(void)
{
    create_tuple(&WinIDs,1);
    set_i(WinIDs,WinID,0);
}
```

In the destructor of the class, the tuple is destroyed using `destroy_tuple`:
dev_open_window is implemented in DevOpenWindow as follows: First, we open the window with open_window and store the returned handle in the variable for the active window handle. To return the handle to the HDevelop program, we create a tuple and set its (first) value using set_i. Finally, we append the handle to the tuple containing all window handles using (the tuple-mode version) of tuple_concat:

```c
int MyHDevOperatorImpl::DevOpenWindow(Htuple row, Htuple col, Htuple width, Htuple height, Htuple background, Htuple *win_id)
{
    Hlong Width, Height;
    Width = get_i(width,0);
    Height = get_i(height,0);
    create_tuple(win_id, 1);
    open_window(get_i(row,0),get_i(col,0),Width,Height,0,"visible","",&WinID);
    set_i(*win_id,WinID,0);
    T_tuple_concat(WinIDs,*win_id,&WinIDs);
    return H_MSG_TRUE;
}
```

dev_close_window is implemented analogously in DevCloseWindow: If there is an active window, it is closed using close_window. Then, its handle is removed from the tuple containing all window handles using tuple_remove. The handle of the active window is always stored at the end of the tuple, so that we do not need to “search” for it. Finally, the handle of the new active window is stored in the corresponding global variable. In between, you can see another example of HALCON/C memory management: For the call to (the tuple-mode version of) tuple_remove, we must first create a tuple for the index of the element to remove and then destroy it afterwards.

```c
int MyHDevOperatorImpl::DevCloseWindow(void)
{
    if (WinID >= 0)
    {
        Htuple Index;
        close_window(WinID);
        create_tuple(&Index,1);
        set_i(Index,WinIDs.length-1,0);
        T_tuple_remove(WinIDs,Index,&WinIDs);
        destroy_tuple(Index);
        WinID = get_i(WinIDs,WinIDs.length-1);
    }
    return H_MSG_TRUE;
}
```

The third operator for handling graphics windows, dev_set_window, is implemented as follows in DevSetWindow: As noted above, the handle of the active window is expected to be at the end of the
tuple containing all window handles. Therefore, we first search for the specified window handle in the tuple using \texttt{tuple\_find}. If the search was successful, we remove the handle from the tuple using \texttt{tuple\_remove} and then append it at the end using \texttt{tuple\_concat}. Finally, we store it in the corresponding global variable:

```c
int MyHDevOperatorImpl::DevSetWindow(Htuple win_id)
{
    Htuple Index;
    T_tuple_find(WinIDs,win_id,&Index);
    if (get_i(Index,0) == -1)
    {
        return H_MSG_FAIL;
    }
    else
    {
        // move handle to the end of the tuple = top of the stack
        T_tuple_remove(WinIDs,Index,&WinIDs);
        T_tuple_concat(WinIDs,win_id,&WinIDs);
        WinID = get_i(win_id,0);
    }
    return H_MSG_TRUE;
}
```

Note that this operator is not used in the executed HDevelop program, it is only implemented so that you can use it in your own applications.
21.2.4 Error Handling

In this section, we take a closer look at exceptions in HDevEngine. The code fragments in the following stem from the example application error_handling (source file error_handling.cpp), which provokes different types of exceptions and “catches” them.

HDevEngine “throws” exceptions in form of the class HDevEngineException, which contains the type (category) of the exception, a message describing the exception, and, depending on the exception type, information like the name of the executed procedure or the HALCON error number (see section 24.1.4 on page 197 for the declaration of the class).

In the example application, the following procedure displays all the information contained in HDevEngineException in a window (only relevant code shown):

```c
void PrintErrorMessage(HDevEngineException Exception)
{
    Hlong ErrorWinID;
    char *CategoryText[4], ProgLineNrText[1024], HalconErrNrText[20];

    CategoryText[0] = (char*)"Exception";
    CategoryText[1] = (char*)"ExceptionInpNotInit";
    CategoryText[2] = (char*)"ExceptionCall";
    CategoryText[3] = (char*)"ExceptionFile";
    sprintf(ProgLineNrText,"%i",Exception.ProgLineNr());
    sprintf(HalconErrNrText,"%i",Exception.HalconErrNr());
    open_window(100,100,850,150,0,"visible","",&ErrorWinID);

    write_string(ErrorWinID," Error category: <");
    write_string(ErrorWinID,CategoryText[Exception.Category()]);
    write_string(ErrorWinID,">");
    new_line(ErrorWinID);
    write_string(ErrorWinID," Message: <");
    write_string(ErrorWinID,Exception.Message());
    write_string(ErrorWinID,">");
    new_line(ErrorWinID);
    write_string(ErrorWinID," Error in procedure / program: <");
    write_string(ErrorWinID,Exception.ExecProcedureName());
    new_line(ErrorWinID,">, program line: <");
    write_string(ErrorWinID,Exception.ProgLineName());
    new_line(ErrorWinID,">, line number <");
    write_string(ErrorWinID,ProgLineNrText);
    new_line(ErrorWinID,">");
    write_string(ErrorWinID," HALCON error number: <");
    write_string(ErrorWinID,HalconErrNrText);
    write_string(ErrorWinID,">");
    
 ...
}
```

This procedure is called when an exception occurs; note that the example applications described in the previous sections use a simplified version of the procedure that only displays the exception message.
Figure 21.3 displays an exception that occurred because the application tried to load a non-existing HDevelop program (category ExceptionFile). As you can see, only the message contains useful information in this case.

The next exception occurs when executing a procedure in which an input parameter is not initialized (category ExceptionInpNotInit):

procedure detect_fin_with_error_inpninit (Image: FinRegion: : FinArea)
    bin_threshold (NotExistingImage, Dark)
    ...

Figure 21.4 displays the content of the exception, which now contains very detailed information about where the error occurred and why.

The final exception is provoked by executing a procedure in which the call to the operator closing_circle fails because the third parameter is not valid (category ExceptionCall):
Figure 21.5 shows the content of the exception if an error occurred in a HALCON operator call.

```plaintext
procedure detect_fin_with_error_call (Image: FinRegion: : FinArea)
    bin_threshold (Image, Dark)
    difference (Image, Dark, Background)
    dev_set_color ('blue')
    dev_display (Background)
    closing_circle (Background, ClosedBackground, -1)
...
```

Figure 21.5 shows the content of the exception.
Chapter 22

HDevEngine in .NET Applications

This chapter explains how to use HDevEngine in C# and Visual Basic .NET applications. Section 22.1 quickly summarizes some basic information about creating HDevEngine applications with Visual Studio .NET and Visual Studio 2005. Section 22.2 then explains how to use HDevEngine/.NET by presenting example applications for

- executing an HDevelop program (section 22.2.1),
- executing an (external) HDevelop procedure (section 22.2.2 on page 172), and
- display operators (section 22.2.3 on page 175), and
- error handling (section 22.2.4 on page 176).

22.1 Basics

To use HDevEngine in Visual Studio .NET, you must

- add a reference to the HALCON/.NET assembly halcondotnet.dll, either by adding an instance of HWindowControl to the form or by adding the reference directly via the Solution Explorer (also see section 10.2.2 on page 78):

- add a reference to the HDevEngine/.NET assembly hdevenginedotnet.dll via the Solution Explorer

- specify the namespace with the following line (also see section 10.2.3 on page 79):

  ```
  using HalconDotNet;
  ```


A short reference of the .NET classes for the HDevEngine can be found in section 24.1 on page 193.
22.2 Examples

This section explains how to employ HDevEngine/.NET based on example applications for C# and Visual Basic .NET, which reside in the subdirectories examples\hdevengine\csharp and examples\hdevengine\vb.net of the folder where you installed HALCON. In the following, we describe only the C# examples; the Visual Basic .NET versions are identical except for the standard differences between the two languages. Furthermore, in contrast to the C# versions, the Visual Basic .NET versions do not contain support for UNIX.

22.2.1 Executing an HDevelop Program

In this section, we explain how to load and execute an HDevelop program with HDevEngine. The code fragments stem from the example application ExecProgram, which checks the boundary of a plastic part for fins. Figure 22.1 shows a screenshot of the application; it contains two buttons to load and execute the HDevelop program.

First, we create a global instance of the main HDevEngine class HDevEngine.

```csharp
private HDevEngine MyEngine = new HDevEngine();
```

Upon loading the form, we store the path to the HDevelop program and set the external procedure path with the method SetProcedurePath:

```csharp
String ProgramPathString;

private void ExecProgramForm_Load(object sender, System.EventArgs e)
{
    string halconRoot = Environment.GetEnvironmentVariable("HALCONROOT");

    MyEngine.SetProcedurePath(halconRoot + @"\examples\hdevengine\procedures");
    ProgramPathString = halconRoot + @"\examples\hdevengine\hdevelop\fin_detection.dev";
}
```

Note that the latter is only necessary if the HDevelop program calls external procedures.
When you click the button to load the HDevelop program, the method `LoadProgram` is called. Exceptions occurring in the HDevEngine method, e.g., because the file name was not specified correctly, are handled with the standard C# error handling mechanism:

```csharp
private void LoadBtn_Click(object sender, System.EventArgs e)
{
    try
    {
        MyEngine.LoadProgram(ProgramPathString);
    }
    catch (HDevEngineException Ex)
    {
        MessageBox.Show(Ex.Message, "HDevEngine Exception");
        return;
    }
}
```

More information on error handling can be found in section 22.2.4 on page 176.

When you click the button to execute the program, the method `ExecuteProgram` is called:

```csharp
private void ExecuteBtn_Click(object sender, System.EventArgs e)
{
    try
    {
        MyEngine.ExecuteProgram();
    }
    catch (HDevEngineException Ex)
    {
        MessageBox.Show(Ex.Message, "HDevEngine Exception");
        return;
    }
}
```

That’s all you need to do to execute an HDevelop program. How to display the results is described in section 22.2.3 on page 175.
### 22.2.2 Executing an External HDevelop Procedure

In this section, we explain how to load and execute an external HDevelop procedure with HDevEngine. The code fragments in the following stem from the example application ExecExtProc, which, like the example described in the previous section, checks the boundary of a plastic part for fins. Figure 22.2 shows a screenshot of the application; it contains two buttons to load and execute the HDevelop procedure.

In contrast to the previous example, the result display is programmed explicitly instead of relying on the internal display operators.

As when executing an HDevelop program, we create a global instance of the main HDevEngine class HDevEngine and set the external procedure path with the method SetProcedurePath upon loading the form:

```csharp
private HDevEngine MyEngine = new HDevEngine();

private void ExecExtProcForm_Load(object sender, System.EventArgs e)
{
    string halconRoot = Environment.GetEnvironmentVariable("HALCONROOT");

    MyEngine.SetProcedurePath(halconRoot + @"\examples\hdevengine\procedures");
}
```

In contrast to the C++ version of this example application, we want to display the results not in a free-floating graphics window, but within the form, i.e., inside an instance of HWindowControl (also see section 10.3 on page 80 and section 10.6 on page 91). For calling the HALCON operators, we declare a global variable of the class HWindow for the underlying HALCON window; upon loading the form, we set this variable to the HALCON window in the HWindowControl and initialize the window:
private HWindow Window;

private void WindowControl_HInitWindow(object sender, System.EventArgs e)
{
    Window = WindowControl.HalconWindow;
    Window.SetDraw("margin");
    Window.SetLineWidth(4);
}

When you click the button Load, the HDevelop procedure is loaded with the method CreateProcedureCall, which returns an instance of the class HDevProcedureCall. Exceptions occurring in the HDevEngine method, e.g., because the file name or the procedure path was not specified correctly, are handled with the standard C# error handling mechanism. More information on error handling can be found in section 22.2.4 on page 176.

private void LoadBtn_Click(object sender, System.EventArgs e)
{
    try
    {
        ProcCall = MyEngine.CreateProcedureCall("detect_fin");
    }
    catch (HDevEngineException Ex)
    {
        MessageBox.Show(Ex.Message, "HDevEngine Exception");
        return;
    }
}

Executing a procedure consists of multiple steps. First, we load an example image sequence:

private void ExecuteBtn_Click(object sender, System.EventArgs e)
{
    HFramegrabber Framegrabber =
    new HFramegrabber("File", 1, 1, 0, 0, 0, 0, "default", -1,
                        "default", -1, "default", "fin.seq", "default", -1, -1);

    Each image should now be processed by the procedure, which has the following signature, i.e., it expects
    an image as (iconic) input parameter and returns the detected fin region and its area as iconic and control
    output parameter, respectively:

    procedure detect_fin (Image: FinRegion: : FinArea)
We pass the image as an input object by storing it in the instance of HDevProcedureCall with the method SetInputObject. Which parameter to set is specified via its name (as an alternative, you can specify it via its index):

```csharp
HImage Image = new HImage();
HRegion FinRegion;
HTuple FinArea;

for (int i=0; i<=2; i++)
{
    Image.GrabImage(Framegrabber);
    Image.DispObj(Window);
    try
    {
        ProcCall.SetInputObject("Image", Image);
    }
    catch (HDevEngineException Ex)
    {
        MessageBox.Show(Ex.Message, "HDevEngine Exception");
        return;
    }
}
```

Now, we execute the procedure by passing the instance of HDevProcedureCall to HDevEngine with the method ExecuteProcedure.

```csharp
MyEngine.ExecuteProcedure(ProcCall);
```

If the procedure was executed successfully, we can access its results, i.e., the fin region and its area, with the methods GetOutputRegion and GetOutputTuple of the class HDevProcedureCall; again, you can specify the parameter via its name or index. Note that you can get iconic output objects either as instances of the corresponding class (here, HRegion) or as instance of HObject by using GetOutputObject.

```csharp
FinRegion = ProcCall.GetOutputRegion("FinRegion");
FinArea = ProcCall.GetOutputTuple("FinArea");
}
```

Finally, we display the results in the graphics window:

```csharp
Image.DispObj(Window);
Window.SetColor("red");
Window.DispObj(FinRegion);
Window.SetColor("white");
Window.SetTposition(150, 20);
Window.WriteString("FinArea: " + FinArea.D);
```
22.2.3 Display

In contrast to the C++ (and COM) version of HDevEngine, HDevEngine/.NET already provides convenience implementations of HDevelop’s internal display operators in form of two classes:

- **HDevOpFixedWindowImpl** directs all display operators to a single graphics window (passed in the constructor), even if the HDevelop program or procedure uses multiple windows.
- **HDevOpMultiWindowImpl** can handle multiple graphics windows. You can pass an arbitrary number of graphics windows in the constructor; if the HDevelop program or procedure uses more than them, HDevEngine opens additional free-floating windows.

The example program ExecProgram uses HDevOpMultiWindowImpl. To use this class (or HDevOpFixedWindowImpl), you pass an instance of it to HDevEngine with the method SetHDevOperators:

```csharp
private void WindowControl_HInitWindow(object sender, System.EventArgs e)
{
    Window = WindowControl.HalconWindow;

    MyEngine.SetHDevOperators(new HDevOpMultiWindowImpl(Window));
}
```

If your application has special display requirements that are not satisfied by the two classes, you can provide your own implementation of the display operators similar to the C++ version of HDevelop (see section 21.2.3 on page 160) by creating a class implementing the interface IHDevOperators and overloading its methods DevOpenWindow, DevDisplay, etc.

Please note that currently you cannot use any form of display operator implementation with Mono (see section 11.2.1 on page 96).
22.2.4 Error Handling

In this section, we take a closer look at exceptions in HDevEngine. The code fragments in the following stem from the example application ErrorHandling, which provokes and catches different types of exceptions when you press some buttons. Figure 22.3 shows a screenshot of the application.

![Figure 22.3: Provoking exceptions in HDevEngine.](image)

HDevEngine throws exceptions as instances of the class HDevEngineException, which contains the type (category) of the exception, a message describing the exception, and, depending on the exception type, information like the name of the executed procedure or the HALCON error number (also see section 24.1.4 on page 197).

In the example application, the following procedure displays all the information contained in HDevEngineException in a message box:

```csharp
private void DisplayException(HDevEngineException Ex)
{
;

    string Title = "HDevEngine Exception (Category: " + Ex.Category.ToString() + ");

    MessageBox.Show(FullMessage, Title);
}
```

This procedure is called when an exception occurs; note that the example applications described in the previous sections only display the exception message.

```csharp
try
{
    MyEngine.LoadProgram(ProgramPathString);
}
catch (HDevEngineException Ex)
{
    DisplayException(Ex);
    return;
}
```

Figure 22.4 displays an exception that occurred because the application tried to load a non-existing HDDevelop program (category File). As you can see, only the message contains useful information in this case.
22.2 Examples

Figure 22.4: Content of the exception if an HDevelop program could not be loaded.

The next exception occurs when executing a procedure in which an input parameter is not initialized (category Input):

```haskell
procedure detect_fin_with_error_in_not_inited (Image: FinRegion: : FinArea)
  bin_threshold (NotExistingImage, Dark)
...
```

Figure 22.5 displays the content of the exception, which now contains very detailed information about where the error occurred and why.

Figure 22.5: Content of the exception if an input parameter was not initialized.

The final exception is provoked by executing a procedure in which the call to the operator `closing_circle` fails because the third parameter is not valid (category Operator):

```haskell
procedure detect_fin_with_error_call (Image: FinRegion: : FinArea)
  bin_threshold (Image, Dark)
  difference (Image, Dark, Background)
  dev_set_color ('blue')
  dev_display (Background)
  closing_circle (Background, ClosedBackground, -1)
...
```

Figure 22.6 shows the content of the exception.

Figure 22.6: Content of the exception if an error occurred in a HALCON operator call.
Chapter 23

HDevEngine in COM Applications

This chapter explains how to use HDevEngine in Visual Basic applications. Section 23.1 quickly summarizes some basic information about creating HDevEngine applications with Visual Basic 6.0. Section 23.2 then explains how to use HDevEngine by presenting example applications for

- executing an HDevelop program (section 23.2.1),
- executing an (external) HDevelop procedure (section 23.2.2 on page 182),
- implementing display operators (section 23.2.3 on page 185), and
- error handling (section 23.2.4 on page 190).

23.1 Basics

To use HDevEngine in Visual Basic 6.0, you must add the Halcon/COM library to your project via the menu item Project ⊲ Components (see also section 14.1 on page 117) and furthermore add the HDevEngine/COM library to the project’s references via the menu item Project ⊲ References.

Parallel HDevEngine applications: If you want to use Parallel HDevEngine, you must register the corresponding DLL parhdevenginex.d11 as described in section 14.4 on page 122 for the COM interface of Parallel HALCON itself.

A short reference of the COM classes for the HDevEngine can be found in section 24.1 on page 193.

23.2 Examples

This section explains how to employ the COM HDevEngine based on example applications for Visual Basic 6.0, which reside in the subdirectories examples\hdevengine\vb of the folder where you installed HALCON.
23.2.1 Executing an HDevelop Program

In this section, we explain how to load and execute an HDevelop program with HDevEngine. The code fragments stem from the example application ExecProgram, which checks the boundary of a plastic part for fins. Figure 23.1 shows a screenshot of the application; it contains two buttons to load and execute the HDevelop program.

First, we create a global instance of the main HDevEngine class HDevEngineX.

```vba
Dim MyEngine As New HDevEngineX
```

Upon loading the form, we store the path to the HDevelop program and the external procedure path in string variables:

```vba
Dim ProgramPathString As String
Dim ProcPathString As String

Private Sub Form_Load()

    ProgramPathString = Environ("HALCONROOT") & \
        "\examples\hdevengine\hdevelop\fin_detection.dev"

    ProcPathString = Environ("HALCONROOT") & \
        "\examples\hdevengine\procedures"

End Sub
```

If the HDevelop program calls external procedures, you must set the external procedure path with the method SetProcedurePath:

```vba
Call MyEngine.SetProcedurePath(ProcPathString)
```

When you click the button to load the HDevelop program, the method LoadProgram is called. Exceptions occurring in the HDevEngine method, e.g., because the file name was not specified correctly, are handled with the standard Visual Basic error handling mechanism. More information on error handling can be found in section 23.2.4 on page 190.
Private Sub LoadBtn_Click()

On Error GoTo errHandler

    Call MyEngine.LoadProgram(ProgramPathString)

    Exit Sub

errHandler:
    Call MsgBox(Err.Description)
End Sub

When you click the button to execute the program, the method ExecuteProgram is called:

Private Sub ExecuteBtn_Click()

On Error GoTo errHandler

    Call MyEngine.ExecuteProgram

    Exit Sub

errHandler:
    Call MsgBox(Err.Description)
End Sub

That’s all you need to do to execute an HDevelop program. How to display the results is described in section 23.2.3 on page 185.
23.2.2 Executing an External HDevelop Procedure

In this section, we explain how to load and execute an external HDevelop procedure with HDevEngine. The code fragments in the following stem from the example application ExecExtProc, which, like the example described in the previous section, checks the boundary of a plastic part for fins. Figure 23.2 shows a screenshot of the application; it contains two buttons to load and execute the HDevelop procedure.

In contrast to the previous example, the result display is programmed explicitly instead of relying on the internal display operators. How to provide your own implementation of the internal display operators is described in section 23.2.3 on page 185.

As when executing an HDevelop program, we create a global instance of the main HDevEngine class HDevEngineX and set the external procedure path with the method SetProcedurePath upon loading the form:

```vbnet
Dim ProcPathString As String
Dim MyEngine As New HDevEngineX
Private Sub Form_Load()
    ProcPathString = Environ("HALCONROOT") & _
                    "\examples\hdevengine\procedures"
    Call MyEngine.SetProcedurePath(ProcPathString)
End Sub
```

In contrast to the C++ version of this example application, we want to display the results not in a free-floating graphics window, but within the form, i.e., inside an instance of HWindowXCtrl (see section 13.1.2.1 on page 109). For calling the HALCON operators, we declare a global variable of the class HWindowX for the underlying HALCON window; upon loading the form, we set this variable to the HALCON window in the HWindowXCtrl and initialize the window:
Dim Window As HWindowX

Private Sub Form_Load()
    Set Window = HXCtrl.HalconWindow
    Call Window.SetPart(0, 0, 575, 767)
    Call Window.SetDraw("margin")
    Call Window.SetLineWidth(4)
End Sub

Then you click the button to load the HDevelop procedure with the method LoadExternalProcedure. Exceptions occurring in the HDevEngine method, e.g., because the file name or the procedure path was not specified correctly, are handled with the standard Visual Basic error handling mechanism. More information on error handling can be found in section 23.2.4 on page 190.

Dim ProcName As String

Private Sub LoadBtn_Click()
    On Error GoTo errHandler
    ProcName = "detect_fin"
    Call MyEngine.LoadExternalProcedure(ProcName)
    Exit Sub
errHandler:
    Call MsgBox(Err.Description)
End Sub

Executing a procedure consists of multiple steps. First, we create an instance of the class HDevProcedureCallX, specifying the name of the procedure (which at this point should be already loaded):

Private Sub ExecuteBtn_Click()
    On Error GoTo errHandler
    Dim ProcCall As New HDevProcedureCallX
    ProcCall.ProcedureName = ProcName
End Sub
Before executing it, we load an example image sequence:

```vba
Dim FGHandle As New HFramegrabberX
Call FGHandle.OpenFramegrabber("File", 1, 1, 0, 0, 0, 0, "default", -1, _
    "default", -1, "default", "fin.seq", _
    "default", -1, -1)
```

Each image should now be processed by the procedure, which has the following signature, i.e., it expects an image as (iconic) input parameter and returns the detected fin region and its area as iconic and control output parameter, respectively:

```vba
procedure detect_fin (Image: FinRegion: : FinArea)
```

We pass the image as an input object by storing it in the instance of `HDevProcedureCallX` with the method `SetInputObjectByIndex`. Which parameter to set is specified via its index (there is also a method to specify it via its name):

```vba
Dim Image As New HImageX
Dim i As Integer
For i = 0 To 2
    Set Image = FGHandle.GrabImage()
    Call Image.DispObj(Window)
    Call ProcCall.SetInputObjectByIndex(1, Image)
```

Now, we execute the procedure by passing the instance of `HDevProcedureCallX` to `HDevEngine` with the method `ExecuteProcedure`.

```vba
Call MyEngine.ExecuteProcedure(ProcCall)
```

If the procedure was executed successfully, we can access its results, i.e., the fin region and its area, with the methods `GetOutputObjectByIndex` and `GetOutputCtrlByIndex` of the class `HDevProcedureCallX`; again, you can specify the parameter via its index or name. Note that output objects are returned as instances of `HUntypedObjectX` (and not as `HRegionX` in this case), while you can pass input objects as `HImageX`.

```vba
Dim FinRegion As HUntypedObjectX
Dim FGHandle As New HFramegrabberX
Call ProcCall.GetOutputObjectByIndex(1, FinRegion)
Call ProcCall.GetOutputCtrlByIndex(1, FinArea)
```
Finally, we display the results in the graphics window:

```vbs
Call Image.DispObj(Window)
Call Window.SetColor("red")
Call Window.DispObj(FinRegion)
Call Window.SetColor("white")
Call Window.SetTposition(150, 20)
Call Window.WriteString("FinArea: " & FinArea)
Next
Exit Sub
```

```vbs
erHandler:
    Call MsgBox(Err.Description)
End Sub
```

### 23.2.3 Display

In this section, we explain how to provide your own implementation of HDevelop’s internal display operators. The code fragments in the following stem from the example application ExecProgram, which was already discussed in section 23.2.1 on page 180.

In fact, HDevEngine does not provide an implementation of the internal display operators but provides the class HDevOperatorImplX, which contains event handlers for all those operators that you can implement yourself. The handlers are called like the object-oriented version of the operators, e.g., DevDisplay for dev_display, and have the same parameters.

The first step towards the implementation is to declare a global instance of the class HDevOperatorImplX. Upon loading the form, we pass this instance to HDevEngine, which then raises the events when the corresponding operator is used in the HDevelop program or procedure.

```vbs
Dim WithEvents MyHDevOperatorImpl As HDevOperatorImplX
Private Sub Form_Load()
    Set MyHDevOperatorImpl = New HDevOperatorImplX
    MyEngine.HDevOperatorImpl = MyHDevOperatorImpl
End Sub
```

To create an event handler for a display operator, select the instance of HDevOperatorImplX in the upper left combo box of the form’s code window and then select the event handler in the combo box to the right (see figure 23.3).

Compared to the previous example, we want to realize a more complex display: We want to

- display the image and the results in a graphics window that is “embedded” within the form, i.e, inside an instance of HWindowXCtrl (see section 13.1.2.1 on page 109) and
- zoom onto the detected fin in a second, free-floating graphics window (see figure 23.1 on page 180).

To switch between multiple windows, we store their window handles in a tuple and provide a second global variable containing the handle of the active window. However, using window handles instead of
instances of the class **HWindowX** means that we need an instance of the class **HOperatorSetX** for calling HALCON operators (see section 13.1.2.2 on page 109). As another consequence, the tuple of window handles is declared as a **Variant**:

```vbnet
Dim MyOperatorSet As New HOperatorSetX
Dim WinIDs As Variant
Dim WinID As Variant
```

Furthermore, we provide a variable for the window handle of the HALCON window inside the **HWindowXCtrl**:

```vbnet
Dim EmbeddedWinID As Variant
```

We initialize all these variables upon loading the form. The handle of the active window is set to the handle of the “embedded” window. The tuple is filled with three elements: -1, -1, and the handle of the embedded window.

```vbnet
Private Sub Form_Load()

    EmbeddedWinID = HXCtrl.HalconWindow.HalconID
    WinID = EmbeddedWinID

    Call MyOperatorSet.TupleGenConst(3, -1, WinIDs)
    WinIDs(2) = EmbeddedWinID

End Sub
```

To understand this initialization, we briefly describe how the tuple of window handles is used, the details follow below: The last element of the tuple contains the handle of the active window (-1 if there is none). When a new window is opened with **dev_open_window**, its handle is appended at the end of the tuple; when **dev_close_window** is called, the last tuple element is removed. If we would use only one element -1 and then close the last window, the **Variant** would change from an array to a single value and we would get an error when trying to append a value again. To circumvent this, we use two elements -1.

Now, we show how the management of the windows is realized. **dev_open_window** is implemented in **DevOpenWindow** as follows: If there is no active window yet, i.e., the variable storing the handle of
the active window is equal to −1, we use the (already opened) “embedded” graphics window and adapt its size to the one specified in the operator call. Otherwise, we open a new window with OpenWindow, passing most of the parameters of dev_open_window and store the returned handle in the variable for the active window handle. We also copy the active window handle into the output parameter of the operator. The window background is set with the operator SetWindowAttr.

Then, we update the tuple of window handles by appending the handle of the active window using TupleConcat. Note that in contrast to, e.g., C++ or HDevelop you cannot use the same variable as input and output parameter, therefore a temporary tuple variable is necessary:

```vba
Private Sub MyHDevOperatorImpl_DevOpenWindow(ByVal vRow As Variant, _
    ByVal vCol As Variant, _
    ByVal vWidth As Variant, _
    ByVal vHeight As Variant, _
    ByVal vBackground As Variant, _
    vWindowHandle As Variant)
    Dim TmpWinIDs As Variant
    If WinID = -1 Then
        WinID = HXCtrl.HalconWindow.HalconID
        HXCtrl.Width = vWidth
        HXCtrl.Height = vHeight
    Else
        Call MyOperatorSet.SetWindowAttr("background_color", vBackground)
        Call MyOperatorSet.OpenWindow(vRow, vCol, vWidth, vHeight, 0, _
            "visible", "", WinID)
    End If
    vWindowHandle = WinID
    Call MyOperatorSet.TupleConcat(WinIDs, vWindowHandle, TmpWinIDs)
    WinIDs = TmpWinIDs
End Sub
```

dev_close_window is implemented analogously in DevCloseWindow: If there is an active window, it is closed using CloseWindow – if it is not the “embedded” window, in which case it is only cleared using ClearWindow. Then, its handle is removed from the tuple containing all window handles using TupleRemove. The handle of the active window is always stored at the end of the tuple, so that we do not need to “search” for it. Again, we need a temporary tuple variable because we cannot use the same variable as input and output parameter. Finally, the handle of the new active window is stored in the corresponding global variable.
Private Sub MyHDevOperatorImpl_DevCloseWindow()

    Dim TmpWinIDs As Variant

    If WinID <> -1 Then
        If (WinID = EmbeddedWinID) Then
            Call MyOperatorSet.ClearWindow(WinID)
        Else
            Call MyOperatorSet.CloseWindow(WinID)
        End If
        Call MyOperatorSet.TupleRemove(WinIDs, UBound(WinIDs), TmpWinIDs)
        WinIDs = TmpWinIDs
        WinID = WinIDs(UBound(WinIDs))
    End If
End Sub

The third operator for handling graphics windows, `dev_set_window`, is implemented as follows in DevSetWindow: As noted above, the handle of the active window is expected to be at the end of the tuple containing all window handles. Therefore, we first search for the specified window handle in the tuple using `TupleFind`. If the search was successful, we remove the handle from the tuple using `TupleRemove` and then append it at the end using `TupleConcat`. Finally, we store it in the corresponding global variable:

Private Sub MyHDevOperatorImpl_DevSetWindow(ByVal vWindowId As Variant)

    Dim Index As Variant
    Dim TmpWinIDs As Variant

    If WinID <> -1 Then
        Call MyOperatorSet.TupleFind(WinIDs, vWindowId, Index)
        If Index <> -1 Then
            Call MyOperatorSet.TupleRemove(WinIDs, Index, TmpWinIDs)
            Call MyOperatorSet.TupleConcat(TmpWinIDs, vWindowId, WinIDs)
            WinID = vWindowId
        Else
            Call MsgBox("DevSetWindow: window handle does not exist!")
        End If
    End If
End Sub

Note that this operator is not used in the executed HDevelop program, it is only implemented so that you can use it in your own applications.

The actual display operators can be implemented quite easily. If there is an active window, the methods simply call the corresponding non-internal display operator, e.g., a call to `dev_display` in the HDevelop program is “redirected” in DevDisplay to `DispObj`, with the iconic object to display and the handle of the active window as parameters:
Private Sub MyHDevOperatorImpl_DevDisplay(ByVal hObj As _
   HDevEngineXLib.IHUntypedObjectX)

   If WinID <> -1 Then
       Call MyOperatorSet.DispObj(hObj, WinID)
   End If

End Sub

Operators with input control parameters are implemented analogously. For example, `dev_set_color` is “redirected” in `DevSetColor` to `SetColor`:

Private Sub MyHDevOperatorImpl_DevSetColor(ByVal vColor As Variant)

   If WinID <> -1 Then
       Call MyOperatorSet.SetColor(WinID, vColor)
   End If

End Sub
23.2.4 Error Handling

In this section, we take a closer look at exceptions in HDevEngine. The code fragments in the following stem from the example application ErrorHandling, which provokes different types of exceptions when you press the buttons and “catches” them. Figure 23.4 shows a screenshot of the application.

Note that in contrast to the C++ version, the COM version of HDevEngine does not provide a special class for exceptions. Instead, it uses the standard Visual Basic error handling mechanism and displays the information contained in the C++ class HDevEngineException (see section 21.2.4 on page 165) in a message box.

Figure 23.5 displays an exception that occurred because the application tried to load a non-existing HDevelop program. As you can see, only the message contains useful information in this case.

```vbnet
Private Sub LoadProgBtn_Click()
    On Error GoTo errHandler
    Dim ProgramPathString As String
    ProgramPathString = Environ("HALCONROOT") & _
        "\examples\hdeven\hdevelop\fin_detectionn.dev"
    Call MyEngine.LoadProgram(ProgramPathString)
    Exit Sub
errHandler:
    Call MsgBox(Err.Description)
End Sub
```

The next exception occurs when executing a procedure in which an input parameter is not initialized:

```vbnet
procedure detect_fin_with_error_inpnotinit (Image: FinRegion: : FinArea)
    bin_threshold (NotExistingImage, Dark)
...
```

Figure 23.6 displays the content of the exception, which now contains very detailed information about where the error occurred and why.

The final exception is provoked by executing a procedure in which the call to the operator closing_circle fails because the third parameter is not valid:
23.2 Examples

Figure 23.5: Content of the exception if an HDevelop program could not be loaded.

Figure 23.6: Content of the exception if an input parameter was not initialized.

```
procedure detect_fin_with_error_call (Image: FinRegion: : FinArea)
  bin_threshold (Image, Dark)
  difference (Image, Dark, Background)
  dev_set_color ('blue')
  dev_display (Background)
  closing_circle (Background, ClosedBackground, -1)
...
```

Figure 23.7 shows the content of the exception.

Figure 23.7: Content of the exception if an error occurred in a HALCON operator call.
Chapter 24

General Information

This chapter contains an overview about the classes of HDevEngine and their methods (section 24.1) and miscellaneous application tips (section 24.2.1 on page 197).

24.1 Overview of the Classes

24.1.1 HDevEngine (C++, .NET), HDevEngineX (COM)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetProcedurePath</td>
<td>Set path for external procedures.</td>
</tr>
<tr>
<td>LoadProgram</td>
<td>Load the specified HDevelop program and all the (local and external) procedures it uses.</td>
</tr>
<tr>
<td>ExecuteProgram</td>
<td>Execute the specified HDevelop program.</td>
</tr>
<tr>
<td>Reset</td>
<td>Delete the currently loaded program from HDevEngine’s memory, including its local procedures but not any external procedures, and reset the callstack (also see section 24.2.1 on page 197).</td>
</tr>
<tr>
<td>LoadExternalProcedure</td>
<td>Load the specified external procedure and all procedures it uses. Please refer to section 24.2.1 on page 197 if you want to load two procedures with the same name.</td>
</tr>
<tr>
<td>CreateProcedureCall (.NET)</td>
<td>Create a procedure call for the specified external procedure (and load it and all procedures it uses).</td>
</tr>
<tr>
<td>ExecuteProcedure</td>
<td>Execute the specified (local or external) procedure.</td>
</tr>
<tr>
<td>DeleteProcedure</td>
<td>Delete the specified (local or external) procedure from HDevEngine’s memory (also see section 24.2.1 on page 197).</td>
</tr>
</tbody>
</table>

(continued on next page)
(continued class overview HDevEngine (C++, .NET), HDevEngineX (COM))

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetProcedureNames (C++, .NET)</td>
<td>Query the names of all currently loaded (local and external) procedures.</td>
</tr>
<tr>
<td>ProcedureNames (COM)</td>
<td></td>
</tr>
<tr>
<td>SetHDevOperatorImpl (C++)</td>
<td>Set implementation for HDevelop’s internal operators.</td>
</tr>
<tr>
<td>SetHDevOperators (.NET)</td>
<td></td>
</tr>
<tr>
<td>HDevOperatorImpl (COM)</td>
<td></td>
</tr>
<tr>
<td>GetVariableImage, GetVariableRegion, GetVariableXld (.NET)</td>
<td>Access iconic variable as <code>HImage/ HRegion/ HXLD</code> (by name).</td>
</tr>
<tr>
<td>GetIconicVarObject (C++)</td>
<td>Access iconic variable as <code>HObject</code> (C++, .NET) or <code>HObjectX</code> (COM) by name.</td>
</tr>
<tr>
<td>GetVariableObject (.NET)</td>
<td></td>
</tr>
<tr>
<td>GetIconicVarObjectByName (COM)</td>
<td></td>
</tr>
<tr>
<td>GetIconicVarObject (C++)</td>
<td>Access iconic variable as <code>HObject</code> (C++) or <code>HObjectX</code> (COM) by index.</td>
</tr>
<tr>
<td>GetIconicVarObjectByIndex (COM)</td>
<td></td>
</tr>
<tr>
<td>GetCtrlVarValue (C++)</td>
<td>Access control variable by name.</td>
</tr>
<tr>
<td>GetVariableTuple (.NET)</td>
<td></td>
</tr>
<tr>
<td>GetCtrlVarValueByIndex (COM)</td>
<td>Access control value by index.</td>
</tr>
<tr>
<td>GetCtrlVarValue (C++)</td>
<td></td>
</tr>
<tr>
<td>GetCtrlVarValueByIndex (COM)</td>
<td></td>
</tr>
<tr>
<td>GetIconicVarName (C++)</td>
<td>Access names of iconic variables.</td>
</tr>
<tr>
<td>GetVariableObjectNames (.NET)</td>
<td></td>
</tr>
<tr>
<td>IconicVarName (COM)</td>
<td></td>
</tr>
<tr>
<td>GetCtrlVarName (C++)</td>
<td>Access names of control variables.</td>
</tr>
<tr>
<td>GetVariableTupleNames (.NET)</td>
<td></td>
</tr>
<tr>
<td>CtrlVarName (COM)</td>
<td></td>
</tr>
<tr>
<td>GetIconicVarCount (C++)</td>
<td>Access number of iconic variables.</td>
</tr>
<tr>
<td>IconicVarCount (COM)</td>
<td></td>
</tr>
<tr>
<td>GetCtrlVarCount (C++)</td>
<td>Access number of control variables.</td>
</tr>
<tr>
<td>CtrlVarCount (COM)</td>
<td></td>
</tr>
</tbody>
</table>
### 24.1.2 HDevProcedureCall (C++, .NET), HDevProcedureCallX (COM)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetInputObject (C++, .NET)</td>
<td>Set input iconic parameter (specified via its name).</td>
</tr>
<tr>
<td>SetInputObjectByName (COM)</td>
<td></td>
</tr>
<tr>
<td>SetInputObject (C++, .NET)</td>
<td>Set input iconic parameter (specified via its index).</td>
</tr>
<tr>
<td>SetInputObjectByIndex (COM)</td>
<td></td>
</tr>
<tr>
<td>SetInputCtrl (C++)</td>
<td>Set input control parameter (specified via its name).</td>
</tr>
<tr>
<td>SetInputTuple (.NET)</td>
<td></td>
</tr>
<tr>
<td>SetInputCtrlByName (COM)</td>
<td></td>
</tr>
<tr>
<td>SetInputCtrl (C++)</td>
<td>Set input control parameter (specified via its index).</td>
</tr>
<tr>
<td>SetInputTuple (.NET)</td>
<td></td>
</tr>
<tr>
<td>SetInputCtrlByIndex (COM)</td>
<td></td>
</tr>
<tr>
<td>GetOutputImage</td>
<td>Get output iconic parameter as HImage/ HRegion/ HXLD (by name or index).</td>
</tr>
<tr>
<td>GetOutputRegion</td>
<td></td>
</tr>
<tr>
<td>GetOutputXld (.NET)</td>
<td></td>
</tr>
<tr>
<td>GetOutputObject (C++, .NET)</td>
<td>Get output iconic parameter as HObject (C++, .NET) or HObjectX (COM) (specified via its name).</td>
</tr>
<tr>
<td>GetOutputObjectByName (COM)</td>
<td></td>
</tr>
<tr>
<td>GetOutputObject (C++, .NET)</td>
<td>Get output iconic parameter as HObject (C++, .NET) or HObjectX (COM) (specified via its index).</td>
</tr>
<tr>
<td>GetOutputObjectByIndex (COM)</td>
<td></td>
</tr>
<tr>
<td>GetOutputCtrl (C++)</td>
<td>Get output control parameter (specified via its name).</td>
</tr>
<tr>
<td>GetOutputTuple (.NET)</td>
<td></td>
</tr>
<tr>
<td>GetOutputCtrlByName (COM)</td>
<td></td>
</tr>
<tr>
<td>GetOutputCtrl (C++)</td>
<td>Get output control parameter (specified via its index).</td>
</tr>
<tr>
<td>GetOutputTuple (.NET)</td>
<td></td>
</tr>
<tr>
<td>GetOutputCtrlByIndex (COM)</td>
<td></td>
</tr>
<tr>
<td>GetProcedureName (C++)</td>
<td>Get name of the procedure.</td>
</tr>
<tr>
<td>Name (.NET)</td>
<td></td>
</tr>
<tr>
<td>ProcedureName (COM)</td>
<td></td>
</tr>
<tr>
<td>GetShortDescription (C++)</td>
<td>Get short description of the procedure.</td>
</tr>
<tr>
<td>Description (.NET)</td>
<td></td>
</tr>
<tr>
<td>ShortDescription (COM)</td>
<td></td>
</tr>
<tr>
<td>GetInpObjParamName (C++)</td>
<td>Get names of input iconic parameters.</td>
</tr>
<tr>
<td>InputObjectNames (.NET)</td>
<td></td>
</tr>
<tr>
<td>InputObjParamName (COM)</td>
<td></td>
</tr>
<tr>
<td>GetOutObjParamName (C++)</td>
<td>Get names of output iconic parameters.</td>
</tr>
<tr>
<td>OutputObjectNames (.NET)</td>
<td></td>
</tr>
<tr>
<td>OutputObjParamName (COM)</td>
<td></td>
</tr>
</tbody>
</table>
(continued class overview HDevProcedureCall (C++, .NET), HDevProcedureCallX (COM))

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetInpCtrlParamName (C++)</td>
<td>Get names of input control parameters.</td>
</tr>
<tr>
<td>InputTupleNames (.NET)</td>
<td></td>
</tr>
<tr>
<td>InputCtrlParamName (COM)</td>
<td></td>
</tr>
<tr>
<td>GetOutCtrlParamName (C++)</td>
<td>Get names of output control parameters.</td>
</tr>
<tr>
<td>OutputTupleNames (.NET)</td>
<td></td>
</tr>
<tr>
<td>OutputCtrlParamName (COM)</td>
<td></td>
</tr>
<tr>
<td>GetInpObjNumber (C++)</td>
<td>Get number of input iconic parameters.</td>
</tr>
<tr>
<td>NumInputObjects (COM)</td>
<td></td>
</tr>
<tr>
<td>GetOutObjNumber (C++)</td>
<td>Get number of output iconic parameters.</td>
</tr>
<tr>
<td>NumOutputObjects (COM)</td>
<td></td>
</tr>
<tr>
<td>GetInpCtrlNumber (C++)</td>
<td>Get number of input control parameters.</td>
</tr>
<tr>
<td>NumInputCtrls (COM)</td>
<td></td>
</tr>
<tr>
<td>GetOutCtrlNumber (C++)</td>
<td>Get number of output control parameters.</td>
</tr>
<tr>
<td>OutputCtrlParamName (COM)</td>
<td></td>
</tr>
</tbody>
</table>

24.1.3 HDevOperatorImpl, IHDevOperators, HDevOperatorImplX

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DevClearWindow</td>
<td>Clear the active graphics window.</td>
</tr>
<tr>
<td>DevCloseWindow</td>
<td>Close the active graphics window.</td>
</tr>
<tr>
<td>DevDisplay</td>
<td>Displays iconic objects in the active graphics window.</td>
</tr>
<tr>
<td>DevOpenWindow</td>
<td>Open a graphics window.</td>
</tr>
<tr>
<td>DevSetColor</td>
<td>Set output color.</td>
</tr>
<tr>
<td>DevSetColored</td>
<td>Set multiple output colors.</td>
</tr>
<tr>
<td>DevSetDraw</td>
<td>Define the region fill mode.</td>
</tr>
<tr>
<td>DevSetLineWidth</td>
<td>Define the line width for region contour output.</td>
</tr>
<tr>
<td>DevSetLut</td>
<td>Set look-up table (LUT).</td>
</tr>
<tr>
<td>DevSetPaint</td>
<td>Define the gray value output mode.</td>
</tr>
<tr>
<td>DevSetPart</td>
<td>Modify the displayed image part.</td>
</tr>
<tr>
<td>DevSetShape</td>
<td>Define the region output shape.</td>
</tr>
<tr>
<td>DevSetWindow</td>
<td>Activate a graphics window.</td>
</tr>
<tr>
<td>DevSetWindowExtents</td>
<td>Change position and size of the active graphics window.</td>
</tr>
</tbody>
</table>
24.1.4 HDevEngineException (C++, .NET)

This class is not available in the COM version of HDevEngine.

<table>
<thead>
<tr>
<th>Message</th>
<th>Error text.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Category</td>
<td>Category of exception.</td>
</tr>
<tr>
<td>ExecProcedureName (C++) ProcedureName (.NET)</td>
<td>Name of executed procedure.</td>
</tr>
<tr>
<td>ProgLineNr (C++) LineNumber (.NET)</td>
<td>Number of executed procedure or operator program line.</td>
</tr>
<tr>
<td>ProgLineName (C++) LineText (.NET)</td>
<td>Name of executed procedure or operator program line.</td>
</tr>
<tr>
<td>HalconErrNr (C++) HalconError (.NET)</td>
<td>HALCON error number.</td>
</tr>
</tbody>
</table>

The class uses the enumeration type ExceptionCategory, which provides the following values:

<table>
<thead>
<tr>
<th>Exception (C++) Generic (.NET)</th>
<th>Generic exception.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExceptionInpNotInit (C++) Input (.NET)</td>
<td>Input parameters not initialized.</td>
</tr>
<tr>
<td>ExceptionCall (C++) Operator (.NET)</td>
<td>Error in HALCON or HDevelop operator call.</td>
</tr>
<tr>
<td>ExceptionFile (C++) File (.NET)</td>
<td>Error opening or reading HDevelop file.</td>
</tr>
</tbody>
</table>

24.2 Tips and Tricks

24.2.1 Loading and Deleting Programs and Procedures

In most applications there is no need to delete loaded HDevelop programs or procedures explicitly using `Reset` and `DeleteProcedure`, respectively. Reasons to do so are:

- to free memory
- to be able to load a procedure that has the same name as an already loaded one. In this case you must first delete the loaded program / procedure. If you fail to do so, `LoadProgram/LoadExternalProcedure` does not load the second procedure.

Please note that `LoadExternalProcedure` loads a procedure including all procedures it uses, but `DeleteProcedure` deletes only the specified procedure. To delete the automatically loaded procedures, you can query their names using `GetProcedureNames` and then delete them (if you are sure that they are not used by another loaded procedure!).
24.2.2 Parallel HDevEngine

As already noted in the introduction, Parallel HDevEngine is thread-safe but not reentrant. This means that you cannot execute multiple HDevelop programs or procedures in parallel using multiple threads – the threads will execute the programs / procedures one after the other.

Please note that this is true even if you create multiple instances of the base class HDevEngine or HDevEngineX: Behind the scenes only one instance is created. Thus, all methods of these classes act globally, i.e., programs and procedures are loaded and deleted globally. To avoid the case that a thread deletes a program or procedure that is used by another thread, we therefore recommend to use these methods globally (or in the main thread) and not locally within a thread.

In contrast, we recommend to create instances of HDevProcedureCall or HDevProcedureCallX locally within the threads that execute the procedure. With this, you avoid so-called races, e.g., the case that a thread sets its input parameters, but before it can execute the procedure another thread sets “its” input parameters.
# Index

<table>
<thead>
<tr>
<th>.NET, 73, 103, 122</th>
<th>examples, 96</th>
</tr>
</thead>
<tbody>
<tr>
<td>applications, 4, 73, 75</td>
<td>applications, 95</td>
</tr>
<tr>
<td>outside local disk, 6</td>
<td>applications, 75</td>
</tr>
<tr>
<td>with Parallel HALCON, 78–80, 97</td>
<td>examples, 96</td>
</tr>
<tr>
<td>.NET Framework, 76</td>
<td>Call, 115</td>
</tr>
<tr>
<td>configuration, 6</td>
<td>Cast(), 110, 111</td>
</tr>
<tr>
<td>., 24</td>
<td>Category (HDevEngine), 197</td>
</tr>
<tr>
<td>abstract class (COM), 107</td>
<td>Category (HDevEngine/.NET), 197</td>
</tr>
<tr>
<td>ActiveX, 103</td>
<td>channel, 44</td>
</tr>
<tr>
<td>ActiveX container, 109</td>
<td>char_threshold (C++), 28–30</td>
</tr>
<tr>
<td>ActiveX control, 109</td>
<td>check_par_hw_potential, 8, 9</td>
</tr>
<tr>
<td>adapt_template, 10</td>
<td>classes (C++)</td>
</tr>
<tr>
<td>add_image, 146</td>
<td>auxiliary, 59</td>
</tr>
<tr>
<td>add_image (C++), 45</td>
<td>handle classes, 27, 28, 58</td>
</tr>
<tr>
<td>AddImage (.NET), 85</td>
<td>iconic classes, 35, 37</td>
</tr>
<tr>
<td>AddImage (C++), 69</td>
<td>namespace, 21</td>
</tr>
<tr>
<td>aggregation (COM), 104</td>
<td>classes (COM), 107</td>
</tr>
<tr>
<td>area_center (C++), 40</td>
<td>abstract class, 108, 110</td>
</tr>
<tr>
<td>AreaCenter (.NET), 87, 88, 90</td>
<td>aggregation, 109</td>
</tr>
<tr>
<td>AreaCenter (C++), 30</td>
<td>base class, 110</td>
</tr>
<tr>
<td>bandpass_image, 146</td>
<td>default interface, 112</td>
</tr>
<tr>
<td>BandpassImage (C++), 68</td>
<td>definition, 104</td>
</tr>
<tr>
<td>base class, 107</td>
<td>derived class, 108, 112</td>
</tr>
<tr>
<td>binary reusability, 103</td>
<td>group class, 108</td>
</tr>
<tr>
<td>binding (COM), 105</td>
<td>inheritance, 112</td>
</tr>
<tr>
<td>BSTR, 113</td>
<td>initialization, 111</td>
</tr>
<tr>
<td>C, 4, 127, 153</td>
<td>instantiation, 111</td>
</tr>
<tr>
<td>applications, 141</td>
<td>internal state, 108</td>
</tr>
<tr>
<td>with Parallel HALCON, 144</td>
<td>special classes, 108</td>
</tr>
<tr>
<td>C++, 4, 19</td>
<td>type conversion, 110</td>
</tr>
<tr>
<td>applications, 61</td>
<td>clear_all_shape_models (C++), 28, 36</td>
</tr>
<tr>
<td>via HALCON/COM, 117</td>
<td>clear_obj, 130, 153, 160</td>
</tr>
<tr>
<td>with Parallel HALCON, 64</td>
<td>clear_obj (C++), 34</td>
</tr>
<tr>
<td>C++ (managed), 73</td>
<td>clear_shape_model (C++), 28, 36</td>
</tr>
<tr>
<td>applications, 79, 85</td>
<td>ClearAllShapeModels (COM), 112</td>
</tr>
<tr>
<td></td>
<td>ClearShapeModel (COM), 112</td>
</tr>
<tr>
<td></td>
<td>ClearWindow (COM), 187</td>
</tr>
<tr>
<td></td>
<td>close_all_framedgrabbers, 14</td>
</tr>
</tbody>
</table>
close_framegrabber, 14
close_framegrabber (C++), 28, 34, 36
close_window, 163
close_window (C++), 27, 28
CloseFramegrabber (.NET), 84
CloseFramegrabber (COM), 112
CloseWindow (C++), 27, 28
CloseWindow (COM), 187
code generation, 110
COM, 103
  applications with Parallel HALCON, 122
combine_roads_xld, 10
CommandButton, 119
compactness (C++), 40
complement (C++), 39, 42
component, 103
concat_obj (C++), 30, 52
concat_ocr_trainf, 10
ConcatObj (.NET), 88
Connection (C++), 30
connection, 129
constructors (.NET), 84
constructors (C++), 26, 35
constructors (COM), 111, 113
containment (COM), 104
contlength (C++), 40
control parameters, 22, 53
copy_obj, 130
count_obj, 130
count_obj (C++), 30, 52
CountObj (.NET), 88
CountSeconds (.NET), 83
create_shape_model (C++), 27
CreateProcedureCall (HDevEngine/.NET), 173, 193
CtrlVarCount (HDevEngine/COM), 194
CtrlVarName (HDevEngine/COM), 194
D, 90
data members (COM), 113
data type (COM), 110
data types
  Hobject (C), 129, 130
  Htuple (C), 129
DeleteProcedure (HDevEngine), 193, 197
DeleteProcedure (HDevEngine/.NET), 193
DeleteProcedure (HDevEngine/COM), 193
Delphi
  applications, 117
Delphi (Borland), 4, 103, 122
Delphi .NET (Borland), 4
Description (HDevEngine/.NET), 195
destructors (C++), 27
dev_close_window, 160, 163, 186, 187
dev_display, 97, 160, 161, 188
dev_open_window, 160, 163, 186, 187
dev_set_color, 160, 189
dev_set_draw, 160, 162
dev_set_line_width, 160
dev_set_part, 160, 162
dev_set_window, 160, 163, 188
DevClearWindow (HDevEngine), 196
DevClearWindow (HDevEngine/COM), 196
DevCloseWindow (HDevEngine), 163, 196
DevCloseWindow (HDevEngine/COM), 187, 196
DevDisplay (HDevEngine), 160, 161, 196
DevDisplay (HDevEngine/.NET), 175
DevDisplay (HDevEngine/COM), 185, 188, 196
DevOpenWindow (HDevEngine), 163, 196
DevOpenWindow (HDevEngine/.NET), 175
DevOpenWindow (HDevEngine/COM), 186, 196
DevSetColor (HDevEngine), 196
DevSetColor (HDevEngine/COM), 189, 196
DevSetColored (HDevEngine), 196
DevSetColored (HDevEngine/COM), 196
DevSetDraw (HDevEngine), 162, 196
DevSetDraw (HDevEngine/COM), 196
DevSetLineWidth (HDevEngine), 196
DevSetLineWidth (HDevEngine/COM), 196
DevSetLut (HDevEngine), 196
DevSetLut (HDevEngine/COM), 196
DevSetPaint (HDevEngine), 196
DevSetPaint (HDevEngine/COM), 196
DevSetPart (HDevEngine), 162, 196
DevSetPart (HDevEngine/COM), 196
DevSetShape (HDevEngine), 196
DevSetShape (HDevEngine/COM), 196
DevSetWindow (HDevEngine), 163, 196
DevSetWindow (HDevEngine/COM), 188, 196
DevSetWindowExtents (HDevEngine), 196
### DevSetWindowExtents (HDevEngine/COM), 196

### Difference (C++), 43
difference, 130
difference (C++), 39, 43
dilation_circle (C++), 38, 42
DilationCircle (C++), 30
Dim, 114
directories
directories
bin, 4
lib, 4
DirectX, 103
disp_circle, 133
disp_image, 11
disp_obj, 161
DISPLAY, 63, 143
DispObj (.NET), 92
DispObj (COM), 188
Dispose, 85, 85
domain (area of definition), 30, 44
double, 113
draw_circle, 10
draw_circle (C++), 57
draw_ellipse (C++), 57
draw_rectangle1 (C++), 57
draw_rectangle2 (C++), 57
dyn_threshold (C++), 46
early binding (COM), 105
eccentricity (C++), 40
directions_image, 146
EdgesImage (C++), 68
eccentricity (C++), 39
encapsulation, 104
evironment variables
evironment variables
DISPLAY, 63, 143
HALCONARCH, 4, 5, 8, 61–63, 93, 141–143
HALCONEXTENSIONS, 63, 143
HALCONIMAGES, 63, 143
HALCONROOT, 63, 143
LD_LIBRARY_PATH, 63, 97, 143
erosion_circle (C++), 38, 42
error handling (COM), 114
error handling (HALCON)
error codes
- H_MSG_FAIL, 139
- H_MSG_FALSE, 139
- H_MSG_TRUE, 139
- H_MSG_VOID, 139
error offset (COM), 114
ErrorBaseCOM, 114
ErrorBaseHalCon, 114
Exception (HDevEngine), 197
ExceptionCall (HDevEngine), 166, 197
ExceptionCategory (HDevEngine), 197
ExceptionCategory (HDevEngine/.NET), 197
ExceptionFile (HDevEngine), 166, 197
ExceptionInpNotInit (HDevEngine), 166, 197
ExecProcedureName (HDevEngine), 197
ExecuteProcedure (HDevEngine), 159, 193
ExecuteProcedure (HDevEngine/.NET), 174, 193
ExecuteProcedure (HDevEngine/COM), 184, 193
ExecuteProgram (HDevEngine), 156, 193
ExecuteProgram (HDevEngine/.NET), 171, 193
ExecuteProgram (HDevEngine/COM), 181, 193
File (HDevEngine/.NET), 176, 197
FillUp (C++), 40
finalizers (.NET), 84
Find1dBarCode (C++), 23–25
form (COM), 118
fread_string, 10
fwrite_string, 10
garbage collector, 84
Gen1dBarCodeDescr (C++), 27, 34
gen_1d_bar_code_descr (C++), 25–27, 58
gen_circle (C++), 27
gen_empty_obj (C++), 30, 52
gen_image1 (C++), 26, 45
gen_image1_extern (C++), 26
gen_image_const (C++), 26, 44
gen_rectangle2 (C++), 27
Generic (HDevEngine/.NET), 197
GenRectangle1 (.NET), 87, 88
GenRectangle2 (C++), 27
get properties (COM), 113
ger_get_error_text, 139
ger_get_error_text (C++), 34
get_grayval (C++), 45
get_image_pointer1 (C++), 45
get_mbutton, 132
get_mbutton (C++), 57
get_mposition (C++), 57
get_param_info, 132
get_system, 13, 135
GetCtrlVarCount (HDevEngine), 194
GetCtrlVarName (HDevEngine), 194
GetCtrlVarValue (HDevEngine), 194
GetCtrlVarValueByIndex (HDevEngine/COM), 194
GetCtrlVarValueByName (HDevEngine/COM), 194
GetDomain (C++), 30
GetDraw (COM), 113
GetHandle, 35, 59
GetIconicVarCount (HDevEngine), 194
GetIconicVarName (HDevEngine), 194
GetIconicVarObject (HDevEngine), 194
GetIconicVarObjectByIndex (HDevEngine/COM), 194
GetIconicVarObjectByName (HDevEngine/COM), 194
GetInpCtrlNumber (HDevEngine), 196
GetInpCtrlParamName (HDevEngine), 196
GetInpObjNumber (HDevEngine), 196
GetInpObjParamName (HDevEngine), 195
GetOutCtrlNumber (HDevEngine), 196
GetOutCtrlParamName (HDevEngine), 196
GetOutObjNumber (HDevEngine), 196
GetOutObjParamName (HDevEngine), 195
GetOutCtrl1 (HDevEngine), 159, 195
GetOutputCtrlByIndex (HDevEngine/COM), 184, 195
GetOutputCtrl1ByIndex (HDevEngine/COM), 184, 195
GetOutputImage (HDevEngine/.NET), 195
GetOutputObject (HDevEngine), 159, 195
GetOutputObject (HDevEngine/.NET), 174, 195
GetOutputObjectByIndex (HDevEngine/COM), 184, 195
GetOutputObjectByName (HDevEngine/COM), 195
GetOutputRegion (HDevEngine/.NET), 174, 195
GetOutputTuple (HDevEngine/.NET), 174, 195
GetOutputXld (HDevEngine/.NET), 195
GetProcedureName (HDevEngine), 195
GetProcedureNames (HDevEngine), 194, 197
GetProcedureNames (HDevEngine/.NET), 194
GetShortDescription (HDevEngine), 195
GetVariableImage (HDevEngine/.NET), 194
GetVariableObject, 194
GetVariableObjectNames, 194
GetVariableRegion (HDevEngine/.NET), 194
GetVariableTuple (HDevEngine/.NET), 194
GetVariableTupleNames, 194
GetVariableXld (HDevEngine/.NET), 194
GetWindowExtents (COM), 113
give_bg_esti, 10
global assembly cache, 93
GrabImage (.NET), 81, 82, 84
GrabImage (C++), 26
GrabImage (COM), 113
graphics window (HDevelop), 78, 159, 160, 172, 174, 182, 185
graphics windows, 161, 162
H_MSG_FAIL, 139
H_MSG_FALSE, 139
H_MSG_TRUE, 139
H_MSG_VOID, 139
HALCON
directories
  bin, 4
  lib, 4
error handling
  error codes, 139
extension package
  Parallel HALCON, 14
language interfaces
  HALCON/.NET, 73
  HALCON/C, 127, 153
  HALCON/C++, 19
  HALCON/COM, 103
memory management (.NET), 84
parallel programming, 7
HALCON Spy, 15
Parallel HALCON, 14
HALCON window, 109
HALCON/.NET, 73
HALCON/C, 127, 153
HALCON/C++, 19
HALCON/COM, 103
HALCONARCH, 4, 5, 8, 61–63, 93, 141–143
HalconErrNr (HDevEngine), 197
HalconError (HDevEngine/.NET), 197
HALCONEXTENSIONS, 63, 143
HALCONIMAGES, 63, 143
HALCONROOT, 63, 143
HalconWindow, 91
HalconX, 107
handle classes, 27, 28, 58
cast operator, 25, 35
handle classes (C++)
cast operator, 59
GetHandle, 35, 59
SetHandle, 35, 59
handles, 23, 25, 53
window, 23, 157, 186
HBarCode1d (C++), 25–27, 58
HByteImage (C++), 50–52
HClassBoxMlp, 58
HCtrlVal
D(), 54
I(), 54
L(), 54
S(), 54
HCtrlVal (C++), 53, 54
HDataCode2D (.NET), 95
HDevelop, 109
graphics window, 78, 159, 160, 172, 174, 182, 185
HDevEngine
applications
with Parallel HALCON, 154
C++ applications, 153
examples, 154
memory management, 153, 159, 160, 163
tuple handling, 153, 160, 162, 163
C# applications, 169
examples, 170
display in graphics window
.NET, 172
C++, 157, 160
COM, 182, 185
exception handling
.NET, 176
C++, 165
COM, 190
execute local procedure, 151
execute procedure, 151
.NET, 174
C++, 159
COM, 184
execute program
.NET, 171
C++, 156
COM, 181
load local procedure, 151
load procedure
.NET, 173
C++, 158
COM, 183
load program, 151
.NET, 171
C++, 156
COM, 180
memory management (C++), 153, 159, 160, 163
set implementation of display operators
C++, 161
COM, 185
set procedure path
.NET, 170, 172
C++, 156, 158
COM, 180, 182
tuple handling (C++), 153, 160, 162, 163
Visual Basic .NET applications, 169
examples, 170
Visual Basic applications, 179
examples, 179
HDevEngine (HDevEngine), 152, 155, 157, 193, 198
HDevEngine (HDevEngine/.NET), 152, 170, 172, 175, 193
HDevEngine() (constructor), 155, 157
~HDevEngine() (destructor), 155
HDevEngine/.NET, 169
HDevEngine/COM, 179
applications
with Parallel HALCON, 179
<table>
<thead>
<tr>
<th>HDevEngine_Exception (HDevEngine)</th>
<th>152, 165, 190, 197</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDevEngine_Exception (HDevEngine/.NET)</td>
<td>152, 176, 197</td>
</tr>
<tr>
<td>HDevEngineX (HDevEngine/COM)</td>
<td>152, 180, 182, 193, 198</td>
</tr>
<tr>
<td>HDevOperatorImpl (HDevEngine)</td>
<td>152, 160–162, 198</td>
</tr>
<tr>
<td>HDevOperatorImpl (HDevEngine/COM)</td>
<td>194</td>
</tr>
<tr>
<td>HDevOperatorImplX (HDevEngine/COM)</td>
<td>152, 185, 196</td>
</tr>
<tr>
<td>HDevOpFixedWindowImpl (HDevEngine/.NET)</td>
<td>152, 175, 175</td>
</tr>
<tr>
<td>HDevOpMultiWindowImpl (HDevEngine/.NET)</td>
<td>152, 175, 175</td>
</tr>
<tr>
<td>HDevProcedureCall (HDevEngine)</td>
<td>152, 158–160, 195, 198</td>
</tr>
<tr>
<td>HDevProcedureCallX (HDevEngine/.NET)</td>
<td>152, 173, 174, 195</td>
</tr>
<tr>
<td>HDevProcedureCallX (HDevEngine/COM)</td>
<td>152, 183, 184, 195, 198</td>
</tr>
<tr>
<td>Height</td>
<td>81</td>
</tr>
<tr>
<td>HFramegrabber (.NET)</td>
<td>81–84</td>
</tr>
<tr>
<td>HFramegrabber (C++)</td>
<td>27, 28, 34, 58</td>
</tr>
<tr>
<td>HFVunction1D (.NET)</td>
<td>86</td>
</tr>
<tr>
<td>HImage</td>
<td>Display(), 45</td>
</tr>
<tr>
<td>GetPixVal(), 45</td>
<td></td>
</tr>
<tr>
<td>Height(), 45</td>
<td></td>
</tr>
<tr>
<td>operator !=, 46</td>
<td></td>
</tr>
<tr>
<td>operator *, 45, 46</td>
<td></td>
</tr>
<tr>
<td>operator +, 45, 46</td>
<td></td>
</tr>
<tr>
<td>operator -, 45, 46</td>
<td></td>
</tr>
<tr>
<td>operator /, 45</td>
<td></td>
</tr>
<tr>
<td>operator &lt;=, 46</td>
<td></td>
</tr>
<tr>
<td>operator ==, 46</td>
<td></td>
</tr>
<tr>
<td>operator =, 46</td>
<td></td>
</tr>
<tr>
<td>operator &gt;=, 46</td>
<td></td>
</tr>
<tr>
<td>operator &amp;, 45</td>
<td></td>
</tr>
<tr>
<td>PixType(), 45</td>
<td></td>
</tr>
<tr>
<td>SetPixVal(), 45</td>
<td></td>
</tr>
<tr>
<td>Width(), 45</td>
<td></td>
</tr>
<tr>
<td>HImage (.NET)</td>
<td>82–84, 86, 88, 99, 194, 195</td>
</tr>
<tr>
<td>HImage (C++)</td>
<td>19, 21, 25–27, 29, 34, 35, 37, 44, 46, 47, 50, 52</td>
</tr>
<tr>
<td>HImageArray</td>
<td>Append(), 49</td>
</tr>
<tr>
<td>Num(), 49</td>
<td></td>
</tr>
<tr>
<td>HImageArray (C++), 29, 31, 49</td>
<td></td>
</tr>
<tr>
<td>HImageX, 108, 110, 111, 119</td>
<td></td>
</tr>
<tr>
<td>HInitWindow, 91, 97</td>
<td></td>
</tr>
<tr>
<td>HInt2Image (C++), 50</td>
<td></td>
</tr>
<tr>
<td>HMeasure (.NET), 95, 96</td>
<td></td>
</tr>
<tr>
<td>HObject (.NET), 99, 100, 174, 194, 195</td>
<td></td>
</tr>
<tr>
<td>HObject (C++), 37, 52, 194</td>
<td></td>
</tr>
<tr>
<td>Hobject, 129, 130</td>
<td></td>
</tr>
<tr>
<td>Hobject (C++), 29, 30, 34, 35, 37, 52</td>
<td></td>
</tr>
<tr>
<td>HobjectX, 108, 110</td>
<td></td>
</tr>
<tr>
<td>HOperatorException (.NET), 92</td>
<td></td>
</tr>
<tr>
<td>HOperatorSet (.NET), 82, 84, 99</td>
<td></td>
</tr>
<tr>
<td>HOperatorSetX, 109, 110</td>
<td></td>
</tr>
<tr>
<td>HPixVal</td>
<td>HByte(), 48</td>
</tr>
<tr>
<td>HPixVal (C++), 47</td>
<td></td>
</tr>
<tr>
<td>HRegion</td>
<td>Anisometry(), 40</td>
</tr>
<tr>
<td>Area(), 40</td>
<td></td>
</tr>
<tr>
<td>Bulkiness(), 40</td>
<td></td>
</tr>
<tr>
<td>Display(), 38</td>
<td></td>
</tr>
<tr>
<td>Ia(), 40</td>
<td></td>
</tr>
<tr>
<td>Ib(), 40</td>
<td></td>
</tr>
<tr>
<td>In(), 40</td>
<td></td>
</tr>
<tr>
<td>IsEmpty(), 40</td>
<td></td>
</tr>
<tr>
<td>M02(), 40</td>
<td></td>
</tr>
<tr>
<td>M11(), 40</td>
<td></td>
</tr>
<tr>
<td>M20(), 40</td>
<td></td>
</tr>
<tr>
<td>operator !, 39</td>
<td></td>
</tr>
<tr>
<td>operator *, 38</td>
<td></td>
</tr>
<tr>
<td>operator ++, 38</td>
<td></td>
</tr>
<tr>
<td>operator +=, 38, 39</td>
<td></td>
</tr>
<tr>
<td>operator +, 38, 39</td>
<td></td>
</tr>
<tr>
<td>operator -, 39</td>
<td></td>
</tr>
<tr>
<td>operator -=, 39</td>
<td></td>
</tr>
<tr>
<td>operator =, 39</td>
<td></td>
</tr>
<tr>
<td>operator /=, 39</td>
<td></td>
</tr>
<tr>
<td>operator /, 39</td>
<td></td>
</tr>
<tr>
<td>operator &lt;=, 39</td>
<td></td>
</tr>
<tr>
<td>operator &lt;&lt;=, 38</td>
<td></td>
</tr>
<tr>
<td>operator &lt;=, 38, 39</td>
<td></td>
</tr>
<tr>
<td>operator &lt;&lt;=, 38</td>
<td></td>
</tr>
<tr>
<td>operator &lt;, 39</td>
<td></td>
</tr>
<tr>
<td>operator ==, 39</td>
<td></td>
</tr>
<tr>
<td>operator &gt;=, 39</td>
<td></td>
</tr>
<tr>
<td>operator &gt;&gt;, 38</td>
<td></td>
</tr>
<tr>
<td>operator &gt;&gt;=, 38</td>
<td></td>
</tr>
</tbody>
</table>
Index 205

operator >, 39
operator &=, 39
operator & 39
operator |=, 39
operator |, 39
Phi(), 39
Ra(), 39
Rb(), 39
StructureFactor(), 40
X(), 40
Y(), 40
HRegion (.NET), 86–88, 99, 174, 194, 195
HRegion (C++), 19, 27, 29, 34, 35, 37, 40, 41, 43, 44, 59
HRegionArray
  Append(), 42
  Display(), 42
  Num(), 42
  operator +, 42
  operator −, 42
  operator /, 43
  operator <<, 42
  operator >>, 42
  operator & 42
  operator |, 43
HRegionArray (C++), 29, 30, 40, 43
HRegionX, 108, 110
HShapeModel (.NET), 95, 96
HShapeModel (C++), 27, 28
HSystemX, 114
HTuple, 113
  operator *, 56
  operator +, 55
  operator −, 55
  operator /, 56
  Sum(), 55
HTuple (.NET), 88, 88, 89–91, 99
HTuple (C++), 23, 29, 30, 34–36, 53, 54, 56
Htuple, 129
HTupleAccessException (.NET), 92
HTupleElement (.NET), 90
HUntypedObjectX, 110
HWindow
  Click(), 57
HWindow (.NET), 91, 172
HWindow (C++), 19, 27, 36, 57, 58
HWindowControl (.NET), 75, 78, 80, 81, 91, 92, 97, 98, 169, 172
HWindowX, 109
HWindowXCtrl, 109, 182, 185
HXLD (.NET), 88, 99, 194, 195
HXLD (C++), 27, 37, 52
HXLDArray (C++), 52
HXLDCont (C++), 27, 52
HXLDPoly (C++), 52

iconic classes, 35, 37
iconic parameters, 22
IconicVarCount (HDevEngine/COM), 194
IconicVarName (HDevEngine/COM), 194
IDispatch, 104
IHDvOperators (HDevEngine/.NET), 152, 175, 196
IHImageX, 112
IHObjectX, 112
image, 44
  channel (image matrix), 44
  domain (area of definition), 30, 44
  region of interest (ROI), 44
image acquisition device
  Parallel HALCON, 14
ImagePart, 80
info_framegrabber, 14
InfoFramegrabber (C++), 24
inheritance (COM), 104, 107
inheritance simulation (COM), 107
Input (HDevEngine/.NET), 177, 197
input object, 159, 174, 184
InputCtrlParamName (HDevEngine/COM), 196
InputObjectNames (HDevEngine/.NET), 195
InputObjParamName (HDevEngine/COM), 195
InputTupleNames (HDevEngine/.NET), 196
integer_to_obj, 130
interfaces (COM), 104, 107, 112
Intersection (C++), 43
intersection (C++), 39, 42
invert_image (C++), 45
iostream, 36
IUnknown, 104

KirschAmp (C++), 68
Label, 119
language independence, 103
language interfaces
   HALCON/.NET, 73
   HALCON/C, 127, 153
   HALCON/C++, 19
   HALCON/COM, 103
Laplace (C++), 68
laplace, 146
late binding (COM), 105
LayoutBitmap, 81
LD_LIBRARY_PATH, 63, 97, 143
Length, 89
library, 107
LineNumber (HDevEngine/.NET), 197
LineText (HDevEngine/.NET), 197
LoadExternalProcedure (HDevEngine), 158, 193, 197
LoadExternalProcedure (HDevEngine/COM), 183, 193
LoadProgram (HDevEngine), 156, 193, 197
LoadProgram (HDevEngine/.NET), 171, 193
LoadProgram (HDevEngine/COM), 180, 193
location transparency, 103
long, 113
main thread, 10
mean_image, 146
mean_image (C++), 22, 23
MeanImage (C++), 23, 24, 69
MeanImage (COM), 111
meaning of objects (COM), 108
memory management (.NET), 84
memory management (COM), 112
Message (HDevEngine), 197
Message (HDevEngine/.NET), 197
methods
   operators, 25
   methods (COM), 107, 113
minkowski_add1 (C++), 39, 42
minkowski_sub1 (C++), 39, 42
moments_region_2nd (C++), 40
Mono, 5
mult_image, 146
mult_image (C++), 45
MultImage (C++), 69
Name (HDevEngine/.NET), 195
namespace Halcon (C++, 21
naming convention (COM), 107
New, 115, 119
newExternWindow (C++), 27
Nothing, 115
NumInputCtrls (HDevEngine/COM), 196
NumInputObjects (HDevEngine/COM), 196
NumOutputObjects (HDevEngine/COM), 196
obj_to_integer, 130
object browser (COM), 117
object copying (COM), 111
object destruction (COM), 112
object duplication (COM), 111
object initialization (COM), 119
object instantiation (COM), 119
object orientation, 103
object-oriented, 21
OLE, 103
On Error, 115
open_framegrabber, 14
open_framegrabber (C++), 27, 34
open_window, 10, 11, 143, 163
open_window (C++), 27, 63
OpenFramegrabber (.NET), 81
OpenWindow (.NET), 92, 98
OpenWindow (C++), 27, 28
OpenWindow (COM), 187
operating systems
   UNIX, 144
Operator (HDevEngine/.NET), 177, 197
OutputCtrlParamName (HDevEngine/COM), 196
OutputObjectNames (HDevEngine/.NET), 195
OutputObjParamName (HDevEngine/COM), 195
OutputTupleNames (HDevEngine/.NET), 196
Parallel HALCON
   applications
      HALCON/.NET, 78–80, 97
      HALCON/C, 144
      HALCON/C++, 64
      HALCON/COM, 122
      HDevEngine, 154
      HDevEngine/COM, 179
extension package, 14
HALCON Spy, 14
image acquisition device, 14
initialization, 8
programming, 9
switch off, 13
Parallel HDevEngine, 152
  in multithreaded applications, 198
Parallel HDevEngine applications, 169
parallel programming, 9
  applications
    Standard HALCON, 7
  reentrancy, 7, 9
  thread-safety, 7, 9
parameters
  _, 24
  control, 22, 53
  handles, 23, 25, 53
    window, 23, 157, 186
  iconic, 22
  simple mode, 28
    tuple mode, 22, 24, 25, 28
pixel type, 44
polymorphism (COM), 104
pouring, 10
PrewittAmp (C++), 68
procedural, 21
ProcedureName (HDevEngine/.NET), 197
ProcedureName (HDevEngine/COM), 195
ProcedureNames (HDevEngine/COM), 194
ProgLineName (HDevEngine), 197
ProgLineNr (HDevEngine), 197
properties (COM), 107, 113
put properties (COM), 113
read_image, 10, 143
read_image (C++), 25, 26, 44, 63
ReadImage (.NET), 84, 92
ReadImage (C++), 26, 33
ReadImage (COM), 111
reduce_domain (C++), 45
reentrancy, 7, 9
region of interest (ROI), 44
RegionGrowing (C++), 58
remote access, 100
Reset (HDevEngine), 193, 197
Reset (HDevEngine/.NET), 193
Reset (HDevEngine/COM), 193
reset_obj_db, 11
Resume Next, 116
Roberts (C++), 68
roberts, 146
RobinsonAmp (C++), 68
scale_image (C++), 46
select_obj (C++), 30, 52
select_shape, 127, 132
SelectObj (.NET), 88
semantics of objects (COM), 108
Set, 115
set_check (C++), 34
set_draw, 162
set_grayval (C++), 45
set_spy, 15, 16
set_system, 13, 143
set_system(image_dir,...) (C++), 63
SetColor (COM), 189
SetHandle, 35, 59
SetHDevOperatorImpl (HDevEngine), 161, 194
SetHDevOperators (HDevEngine/.NET), 175, 194
SetInputCtrl (HDevEngine), 195
SetInputCtrlByName (HDevEngine/COM), 195
SetInputCtrlByName (HDevEngine/COM), 195
SetInputObject (HDevEngine), 159, 195
SetInputObject (HDevEngine/.NET), 174, 195
SetInputObjectByName (HDevEngine/COM), 195
SetInputObjectByName (HDevEngine/COM), 195
SetInputTuple (HDevEngine/.NET), 195
SetPart (.NET), 81, 92
SetProcedurePath (HDevEngine), 156, 158, 193
SetProcedurePath (HDevEngine/.NET), 170, 172, 193
SetProcedurePath (HDevEngine/COM), 180, 182, 193
SetWindowAttr (COM), 187
SetWindowExtents, 113
ShortDescription (HDevEngine/COM), 195
simple mode, 28
Skeleton (C++), 68
skeleton, 146
smallest_rectangle1 (C++), 40
SSE2, 3
strong-typed (COM), 110
sub_image (C++), 45
test_equal_region (C++), 39
test_region_point (C++), 40
TextureLaws (C++), 68
thread-safety, 7, 9
Threshold (COM), 115
threshold (C++), 46
top-level window, 109
transpose_region (C++), 39
tuple mode, 22, 24, 25, 28
tuple_add (C++), 55
tuple_concat, 163, 164
tuple_concat (C++), 56
tuple_div (C++), 56
tuple_find, 164
tuple_gen_const (C++), 55
tuple_mult (C++), 56
tuple_remove, 163, 164
tuple_sub (C++), 55
tuple_sum (C++), 55
TupleAdd (.NET), 91
TupleConcat (COM), 187, 188
TupleFind (COM), 188
TupleRad (.NET), 91
TupleRemove (COM), 187, 188
Type, 89

Union1 (C++), 43
union2 (C++), 39, 43
UNIX, 144
update_bg_esti, 10

variable (COM), 111
variable declaration (COM), 119, 120
VARIANT, 110, 113
vector_to_rigid (C++), 35
versioning, 103
Visual Basic, 4, 103, 103, 110, 114
    applications, 117
Visual Basic .NET, 4, 73
    applications, 75, 79
examples, 96
Visual C++, 103, 122

weak-typed (COM), 110
Width, 81
window handle, 23, 157, 186
window handles, 161, 185
WindowSize, 80
worker threads, 11
write_image, 10

X, 81
XLD, 52

Y, 81