HALCON

a product of MVTec

Programmer's Guide

HALCON 25.11 *Progress*

MVTec Software GmbH

Building Vision for Business

All about using the programming language interfaces of HALCON, Version 25.11.0.0

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

Copyright © 2003-2025 by MVTec Software GmbH, Munich, Germany 

Protected by the following patents: US 7,751,625, US 7,953,290, US 7,953,291, US 8,260,059, US 8,379,014, US 8,830,229, US 11,328,478. Further patents pending.

AMD and AMD Athlon™ are either trademarks or registered trademarks of Advanced Micro Devices, Inc.
Arm® is a registered trademark of Arm Limited.
OpenGL® and the oval logo are either trademarks or registered trademarks of Hewlett Packard Enterprise in the United States and/or other countries worldwide.
Intel® , the Intel® logo, OpenVINO™ , the OpenVINO™ logo, and Pentium® are either trademarks or registered trademarks of Intel® Corporation or its subsidiaries.
Linux ® is a registered trademark of Linus Torvalds.
Microsoft, Windows, Microsoft .NET, Visual C++ and Visual Basic are either trademarks or registered trademarks of Microsoft Corporation.
Python® is a registered trademark of PSF.
UNIX® is a registered trademark of The Open Group.

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

More information about HALCON can be found at: https://www.halcon.com

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 HALCON with parallel programming.
- **Programming With HALCON/C++**
  This part describes HALCON's language interface to C++.
- **Programming With HALCON/.NET**
  This part describes HALCON's language interface to .NET programming languages (C#, Visual Basic.NET, etc.).
- **Programming With HALCON/Python**
  This part describes HALCON's language interface to Python.
- **Programming With HALCON/C**
  This part describes 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.

Symbols

The following symbols are used within the manual:

This symbol indicates a **tip**.

This symbol indicates an information you should **pay attention** to.

Contents

Part I

General Issues

Chapter 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 on page 13 guides you through these possibilities.

recommendation			alternative(s)
C	→	HALCON/C	HALCON/C++
C++ (unmanaged)	→	HALCON/C++	
C++ (managed)	→	HALCON/.NET	
C#	→	HALCON/.NET	
Visual Basic.NET	→	HALCON/.NET	
Python	→	HALCON/Python	

Table 1.1: Which interface to use for which programming language.

The system requirements and supported platforms are listed in the Installation Guide, section 1.4 on page 8.

Chapter 2

Parallel Programming and HALCON

This chapter explains how to use HALCON on multi-core or multi-processor hardware, concentrating on the main features: automatic parallelization (section 2.1 on page 15) and the support of parallel programming (section 2.2 on page 16).

2.1 Automatic Parallelization

If HALCON is used on multi-processor or multi-core hardware, it will automatically parallelize image processing operators. Section 2.1.1 on page 15 describes how to initialize HALCON in order to use this mechanism. Section 2.1.2 on page 15 explains the different methods which are used by HALCON operators for their automatic parallelization.

2.1.1 Initializing HALCON

In order to adapt the parallelization mechanism optimally to the actual hardware, 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 `optimize_aop` (see the corresponding entry in the HALCON Reference Manuals for further information). Note, that this operator will only work correctly if called on a multi-core or multi-processor hardware; 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 `optimize_aop`, 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. Note that this examination may take some hours, depending on your computer and the optimization parameters!

The extracted information is stored in the file `.aop_info` in the common application data folder (under Windows) or in the HALCON installation directory `$HALCONROOT` (under Linux). Please note, that on some operating systems you need special privileges to initialize HALCON successfully, otherwise the operator `optimize_aop` is not able to store the extracted information. Note that in order to **execute command line tools with administrator privileges**, you will need to select "Run as Administrator" (even if you are already logged in as administrator).

Please refer to the examples in the directory `%HALCONEXAMPLES%\hdevelop\System\Parallelization` for more information about `optimize_aop` and about other operators that allow you to query and modify the parallelization information.

2.1.2 Methods of Automatic Parallelization

For the automatic parallelization of operators, 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 four 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**
   An operator supporting this level can be parallelized by dividing its domain and distributing its parts on parallel threads.

4. **Internal data level**
   Only parts of the operator are parallelized. The actual degree of parallelization depends on the implementation of the operator. As a result, the potential speedup on multi-core systems varies among operators utilizing this parallelization method.

The description of a HALCON operator in the Reference Manuals contains an entry called 'Execution Information', which specifies its behavior when using HALCON on a multi-core or multi-processor hardware. This entry indicates whether the operator will be automatically parallelized by HALCON and by which method (tuple, channel, domain, internal data).

The parallelization method of an arbitrary operator *opname* can also be determined using `get_operator_info`:

```
get_operator_info('opname', 'parallel_method', Information)
```

2.2 Parallel Programming Using HALCON

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 HALCON. Furthermore, it points out issues that should be kept in mind when writing parallel programs that use 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 HALCON/C.

Furthermore, HALCON provides special operators to synchronize threads (see section 2.2.3 on page 17).

2.2.1 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.

   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**
   Operators marked as *local* should be called only from the thread that instantiates the corresponding objects.

3. **Single write, multiple read**
   A certain group of operators should be called simultaneously only if the different calling threads work on different data.

As this thread behavior is not recommended in general, 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 undesirable effects.

4. **Mutually exclusive**
   Some operators cannot be called simultaneously by multiple threads but may be executed in parallel to *other* HALCON operators.

5. **Exclusive**
   A group of operators is executed exclusively by HALCON, i.e., while such an operator is executed, all other threads cannot call another HALCON operator.

6. **Independent**
   A group of operators is executed independently from others, even exclusive operators.

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

2.2.2 Program Design Issues for Parallelization

The following issues have to be considered 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).

When using HALCON, please keep the following issues in mind:

- **Initialization**
  HALCON is initialized implicitly at the first operator call. During this initialization, among others, the license is checked, HALCON memory management is set up, the thread pool for internal operator parallelization is started, internal system parameters and synchronization objects are initialized. This initialization process is thread safe in a multithreading environment.

  Note that the default value of some system parameters can be changed before this initialization process. Compare the include/HGlobal.h file and some references to the corresponding parameters in this document. If used, make sure that all default values are set before calling the first operator in your multithreaded environment.

- **I/O and visualization**
  Keep in mind that operators that create or delete files work exclusively, i.e., other threads have to wait.

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

  See section 2.3 on page 18 for information about visualization issues on different operating systems.

- **Multithreading vs. automatic parallelization**
  If you explicitly balance the load on multiple processors or cores in a multithreaded program, we recommend that you either switch off the automatic parallelization mechanism to get an optimal 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 of the automatic parallelization or reduce the number of internal threads is described in section 2.5.1 on page 19.

2.2.3 Multithreading Operators

In the operator section "System ▷ Multithreading", HALCON provides operators for creating and using synchronization objects like mutexes, events, condition variables, and barriers.

With them, you can synchronize threads in a platform-independent way. Note, however, that up to now no operators for creating the threads are provided.

2.2.4 Examples

HALCON currently provides the following examples for parallel programming (paths relative to %HALCONEXAMPLES%):

HALCON/C

- c\source\example_multithreaded1.c

  two threads extract different elements on a board in parallel

HALCON/.NET

- c#\MultiThreading (C#)

  performs image acquisition, processing, and display in three threads

- hdevengine\c#\MultiThreading (C#)

  executes the same HDevelop procedure in parallel by two threads using HDevEngine

- hdevengine\c#\MultiThreadingTwoWindows (C#)

  executes different HDevelop procedures in parallel by two threads using HDevEngine

HALCON/C++

- mfc\FGMultiThreading (using MFC)

  performs image acquisition / display and processing in two threads

- hdevengine\mfc\source\exec_programs_mt_mfc.cpp

  executes HDevelop procedures for image acquisition, data code reading, and visualization in parallel using HDevEngine and MFC

- hdevengine\cpp\source\exec_procedures_mt.cpp

  executes HDevelop programs in parallel using HDevEngine

2.3 Threading Issues With Graphics

Various windowing systems have different restrictions on multithreading that have an impact on HALCON graphic operators.

2.3.1 Microsoft Windows

On Microsoft Windows, accessing a window's message queue is only possible from the thread that created the window. Furthermore, a window that has a parent window must be opened in the thread that created the parent window.

All HALCON graphic operators are automatically redirected to the correct thread. This is done by sending a special message to the window. To avoid conflicts with user code, the ID of this message is dynamically generated using the Win32 RegisterWindowMessage function. For windows not created by HALCON (this is relevant for new_extern_window and when specifying a parent window to open_window), the non-HALCON window is automatically subclassed using the Win32 SetWindowSubclass function. The only requirement for user code is that for this mechanism to work, each window must have an active message loop.

To avoid having to write user code to handle the message loop of a HALCON window, HALCON can be instructed to create all top level HALCON windows from a special thread via set_system('use_window_thread','true'),

which will then also take care of the message loop. Note that this may negatively affect performance if many windows are used simultaneously, as HALCON provides only a single thread for all windows.

The various HALCON `draw_*` operators work by actively polling the message queue. Several `draw_*` operators can be active if they target windows belonging to different threads.

2.3.2 X11

When using the X11 windowing system, HALCON will automatically call the Xlib function `XInitThreads` when the first window is opened (note that querying the available OpenGL features will open a hidden X11 window in the background).

This means that if an application using HALCON also uses Xlib functions independently of HALCON, either directly by explicitly calling Xlib functions or indirectly through some other library that uses Xlib, care must be taken that `XInitLibrary` is called first before any other Xlib function. The easiest way to ensure this is to call `XInitThreads` very early in the application's main function, before calling any other library functions.

2.4 Using HALCON with OpenMP

Using different OpenMP runtimes in a single program leads to an undefined behavior. Thus, if you have an application using HALCON and OpenMP, meaning your application uses the `halcondl` library and OpenMP, you have to link against the Intel OpenMP library `libiomp5`. The latter one is installed with HALCON. For the linking, the necessary steps depend on your platform:

Windows
You can do this in Visual Studio:

1. Open Visual Studio.
2. Go to `Project Properties` → `Linker` → `Input`.
3. Add `vcomp.lib` (`vcompd.lib` for the debug configuration) to `Ignore Specific Default Libraries`
4. Add `libiomp5md.lib` from `%HALCONROOT%\lib %HALCONARCH%` to the `Additional Dependencies`.

Linux
For `gcc` and `clang`, add

- `-fopenmp` to the compile flags
- `-L$HALCONROOT/lib/$HALCONARCH/thirdparty -liomp5` to the linker flags

2.5 Additional Information

This section contains additional information that helps you to use HALCON on multi-core or multi-processor hardware.

2.5.1 Customizing the Parallelization Mechanisms

With the help of HALCON's system parameters, which can be set and queried with the operators `set_system` and `get_system`, respectively, you can customize the behavior of the parallelization mechanisms.

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

```
get_system('processor_num', Information)
```

You can **switch off parts of the features of HALCON** with the help of the operator set_system. To switch off the automatic parallelization mechanism, call (HDevelop notation, see the Reference Manual for more information)

```
set_system('parallelize_operators','false')
```

To switch off reentrancy, call

```
set_system('reentrant','false')
```

Of course, you can switch on both behaviors again by calling 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.

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

When switching off the automatic parallelization, you might consider switching off the use of thread pools (see the parameter 'thread_pool' of set_system).

Do not switch on reentrancy if this is already the case. Otherwise, this will reset the parallelization system, which includes switching on the automatic operator parallelization. This will decrease the performance in case of manual parallelization (multithreading).

With the system parameter 'parallelize_operators', you can customize the automatic parallelization mechanisms in more detail. Please see the description of set_system for more information.

Finally, you can **influence the number of threads used for automatic parallelization** with the parameters 'thread_num' and 'tsp_thread_num' (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 on Multi-Core or Multi-Processor Hardware

All image acquisition devices supported by HALCON can be used on multi-core or multi-processor hardware. 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, and close_framegrabber) are processed exclusively in their group, i.e., they block the concurrent execution of other image acquisition operators but run in parallel with all non-exclusive operators outside of this group (see open_framegrabber). Furthermore, these operators are *local*, i.e., under Windows they should be called from the thread that instantiates the corresponding object (see section 2.2.1 on page 16).

2.5.3 Spinlocks, the HALCON Thread Pool, and Real-time Scheduling

By default, HALCON uses spinlocks for synchronization between threads to maximize performance. However, there are several situations where using spinlocks is not recommended:

1. If there are more threads running than there are CPUs to run them on, performance will be severely reduced.
2. If any threads using HALCON use any form of real-time scheduling (SCHED_FIFO, SCHED_RR, or SCHED_DEADLINE on Linux systems, REALTIME_PRIORITY_CLASS on Windows systems), HALCON can deadlock.

To turn spinlocks off, call the function HSetUseSpinLock(0) before calling the first HALCON operator. Please note that calling HSetUseSpinLock after HALCON has been initialized leads to undefined behavior.

Spinlocks and the HALCON Thread Pool

The HALCON thread pool used for the automatic operator parallelization will always use spinlocks, even if `HSetUseSpinLock` has been called to turn them off. Thus, it is important to make sure the thread pool does not use more threads than CPUs are actually available to run them on. You must be especially careful if your program uses any form of real-time scheduling, as your program can deadlock otherwise.

HALCON will normally always create the thread pool during initialization. If you do not need the thread pool, you can prevent it from being created by calling the function `HSetStartUpThreadPool(0)` before calling the first HALCON operator.

HALCON Thread Pool and CPU Affinity on Linux

On Linux systems, threads inherit the CPU affinity of the parent thread that created them. This means that if the HALCON thread pool is enabled, its threads will be limited by the CPU affinity mask active of the thread creating the pool at the time the thread pool is created. If the pool is turned off, HALCON's automatic operator parallelization will create threads on the fly. These threads will be limited to the CPU affinity mask of the thread calling the HALCON operator being parallelized.

Please note that HALCON looks at the CPU affinity mask of the process during initialization to determine how many of the CPUs present can actually be used for automatic operator parallelization only once at startup. Thus, if you change the CPU affinity subsequently, HALCON will not notice and may end up attempting to use more CPUs than are available to it. This can degrade performance or lead to deadlocks when using any form of real-time scheduling. Problems will also ensue if the thread used to initialize HALCON uses a different CPU affinity mask than its parent process, as HALCON only takes the CPU affinity mask of the process into account.

HALCON Thread Pool and CPU Affinity on Windows

On Windows systems, threads inherit the CPU affinity of the process they are created by. This means that HALCON thread pool threads will be limited by the CPU affinity mask of the process at the time they are created.

As for Linux, on Windows HALCON looks at the CPU affinity mask of the process during initialization to determine how many of the CPUs present can actually be used for automatic operator parallelization only once at startup. Thus, if you change the CPU affinity subsequently, HALCON will not notice and may end up attempting to use more CPUs than are available to it.

On a machine with more than 64 processors, Windows will split the processors into multiple processor groups of at most 64 processors. For Windows versions prior to Version 11 or Windows Server 2022, processes are limited to a single processor group unless each thread is explicitly assigned a processor affinity. Since HALCON does not modify CPU affinities, this means that HALCON is limited to using at most 64 processors for automatic operator parallelization.

Starting with Windows 11 and Windows Server 2022, processes may use all available processors by default, unless the process affinity mask has been set, in which case the process will revert to the pre-Windows 11 behavior. HALCON can thus use more than 64 processors on Windows 11 and Windows Server 2022 if the process affinity mask is not modified. If a program wants to limit which processors HALCON may run on but still use more than one processor group, it must not set the process affinity mask, but instead set a default process CPU Set using the Windows CPU Set API prior to calling the first HALCON operator.

Note that while the Windows CPU Set API is partially available in Windows 10, HALCON will only take CPU sets into account on Windows 11 or Windows Server 2022 or later, and only if no process affinity mask was set.

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.

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.4 on page 47.

You specify the monitoring mode by calling the operator `set_spy` again with a pair of parameters to be informed about all operator calls and the names and values of input control parameters. For example:

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

The monitoring mode can also be specified via the environment variable `HALCONSPY`, using a colon to separate multiple options:

```
operator=on:input_control=on
```

For detailed information on all the debugging options, see the entry for `set_spy` in the HALCON Reference Manuals.

3.1.1 HALCON Spy on Multi-Core or Multi-Processor Hardware

Please note that HALCON Spy cannot be used to debug multithreaded programs or programs using the automatic parallelization.

If you want to use HALCON Spy on a multi-core or multi-processor hardware, you must therefore first switch off the automatic parallelization as described in section 2.5.1 on page 19.

3.2 Terminating HALCON Library

In applications where DLLs are unloaded in a thread-exclusive context, e.g., with `FreeLibrary()` under Windows, the HALCON library will not terminate properly if there are still active HALCON threads.

A possible scenario in which the problem may occur is using HALCON/C++ to implement an ATL control, for example.

To overcome this problem, it is necessary to call the function `FinalizeHALCONLibrary()` before unloading the DLL. Please note that the application has to be linked against `halcon.lib`. Make sure that the call to `FinalizeHALCONLibrary()` is not from within any other DLL's `DllMain` function. Please note that once `FinalizeHALCONLibrary()` has been called, no further HALCON functions may be called. Please also note that the HALCON library is not capable to free all resources properly during termination. This might result in memory leaks when loading and unloading the HALCON library repeatedly.

3.3 Returning Floating Licenses on Windows

Applications using a HALCON floating license should return the license to the license server on exit so that it can be reused immediately. As socket communication is not possible during process termination, a process using HALCON cannot return floating licenses automatically on Windows by itself and requires the help of the license watchdog process hlwd (see Installation Guide, section 6.7 on page 33 for more details).

If the license watchdog is not available, the process using HALCON must return the floating license before termination either by calling `FinalizeHALCONLibrary()` as described in section 3.2 on page 23, or by calling the HALCON `set_system` operator with the parameter `'return_license'`. This is only necessary on Windows when using a floating license and when the license watchdog is unavailable.

3.4 Inspecting HALCON Variables in Visual Studio

HALCON includes an extension for the inspection of HALCON variables in Visual Studio (see the Installation Guide, section 1.4 on page 8, for the system requirements). A development license is required for this extension to work (see the Installation Guide, chapter 6 on page 27 for the different license types). If the license is valid for the HALCON Progress edition, only the corresponding edition of the HALCON Variable Inspect can be used, i.e., HALCON Progress Variable Inspect requires HALCON Progress. To change the HALCON Variable Inspect edition, you have to reopen Visual Studio and choose the desired HALCON Variable Inspect edition.

The extension is registered by default for your Visual Studio installation while installing HALCON. In case this option was deactivated for the installation, you can register the extension manually by double-clicking one of the following files, depending on your version of Visual Studio:

- For Visual Studio 2019 and before (32-bit), use
  `%HALCONROOT%/misc/HALCON2511ProgressVariableInspectVisualStudio2019AndEarlier.vsix`
- For Visual Studio 2022 and later (64-bit), use
  `%HALCONROOT%/misc/HALCON2511ProgressVariableInspectVisualStudio2022AndLater.vsix`

If multiple versions of Visual Studio are installed on your machine, the installer will let you select the version(s) you wish to add the extension to. Once installed, the extension should appear in the side pane of Visual Studio. If it does not, select `Tools` → `HALCON 25.11 Progress Variable Inspect`. Please note that the environment variable `%HALCONROOT%` must be set for the extension to work.

The extension is provided only for the inspection of iconic objects and tuples in the following languages:

Interface	Language
HALCON/C++	C++
HALCON/.NET	C#, Visual Basic.NET

Table 3.1: Interface and language.

Restrictions When Using HALCON Variable Inspect

- HALCON Variable Inspect cannot inspect objects located in GPU memory. You can only inspect objects if they are stored in CPU memory.
- HALCON Variable Inspect does not support remote debugging. You can only inspect processes that are running on the same machine.
- At the moment, HALCON Variable Inspect does not work with C++/CLI applications. The only supported combinations are native C++ with HALCON/C++, C# with HALCON/.NET, and Visual Basic.NET with HALCON/.NET.
- Note that Visual Studio Express does not support extensions.
- HALCON Variable Inspect cannot be used to inspect images, with fewer channels than the default number of channels for `HImage`.

Using HALCON Variable Inspect

While debugging HALCON programs, the extension allows you to inspect the current values of initialized control and iconic variables. You can either set breakpoints at the desired places or single-step through your program.

The inspection window is divided into two tabs `Locals` and `Watch`, see figure 3.1 on page 27.

The `Locals` tab provides a list of all local HALCON variables from the current scope of the program. The variable list is automatically updated if the scope of the program changes.

The `Watch` tab contains only selected HALCON variables that have been added by the user to the "Watch" list. It also allows the inspection of global HALCON variables. Variables can only be added to the "Watch" list if the program has reached a break point. In addition, local variables can only be added if they are currently in scope.

There are several ways to add a HALCON variable to the "Watch" list:

- Mark a variable in the program code. Then drag the selected variable from the program code to the "Watch" list.
- Right-click the variable in the program code and select `Add to HALCON Watch` from the context menu.
- Right-click a variable in the "Locals" list and select `Add to Watch` from the context menu.

Note that the fewer variables are listed in the inspection window the better the performance. Therefore, moving only the desired variables to the "Watch" list and inspecting the variables in the `Watch` tab improves the performance.

By default, the variables in the inspection window are listed in the same order as in the `Locals` window of Visual Studio. The variables can be sorted in descending or ascending order by clicking on the header of the respective columns in the inspection window. To switch back to the default order, click the header of the column "#".

The table Table 3.2 on page 26 describes which information is displayed for the variables of the "Locals" and "Watch" list of the inspection window.

The `Watch` tab of the inspection window gives some additional information about the scope, see Table 3.3 on page 26.

To select a variable for the inspection, click the corresponding entry in the variable list of the inspection window. If the debug information and HALCON data of the variable has already been retrieved, i.e., if `Status` has a green checkmark, the value of the variable is displayed. Otherwise, the retrieval of the HALCON data is started and the value of the variable is displayed if the data could be retrieved. To enforce a complete new retrieval of the data, even for variables whose data have already been obtained, double-click the variable. The inspection window is automatically updated if the content of the currently displayed variable changes.

To deselect a variable, press Ctrl and click the respective line of the variable. Alternatively, right-click the line of the variable and select `Deselect all` from the context menu. In some cases, it is desired to display the data of a variable again. This can be achieved by deselecting and selecting the variable again or by double-clicking it.

If a member of a variable cannot be resolved, e.g., if the variable is related to a class, it is not sufficient to inspect only the member of this variable. Instead, mark the complete expression before right-clicking and inspecting the member variable.

Please note that native 64-bit data types, e.g., int8 images, and image data with a size > 0.5 GB are not supported.

Columns	Description
#	This column contains no values. It can be used to restore the default order of the variable list by clicking on the header of this column.
Name	Name of the local HALCON variable.
Type	Type of the local HALCON variable. Only iconic variables and tuples are supported in the extension.
Status	Current status that indicates if the debug information of the variable could be retrieved. In case it has been successfully retrieved, a green checkmark is displayed. If the information could not be retrieved, a red crossmark is displayed.
Status Message	Status message for the variable. If the HALCON data of a variable could not be retrieved, the status message delivers more detailed information about the cause. If the HALCON data could be retrieved, additional information about the data is displayed.

Table 3.2: Columns of the variable lists in the inspection window.

Column	Description
Scope	Scope of the HALCON variable. For local HALCON variables the scope "Local" is returned and for global HALCON variables the scope "Global". If the scope could not be determined, i.e., because the variable is not in the current scope or uninitialized, "Unknown" is returned.
Inside Scope	Current status if the HALCON variable is in the current scope of the program. If the variable is in the current scope, "True" is returned, else "False".

Table 3.3: Additional columns of the variable list in the `Watch` tab.

Inspecting Control Variables

The following information is displayed for control variables (tuples): Name of the control variable, type, and length. The actual values are displayed as a list from 0 to length−1.

Inspecting Iconic Variables

Iconic variables are displayed graphically in the inspection window. Apart from the visualization of the content, the following information is displayed:

- Name of the iconic variable, type, number of channels (for images), and the dimensions (for images).
- Gray value (`v`), row (`r`), and column (`c`) at the mouse position

The inspection window has a toolbar with the following functionalities:

- 1 Clear inspection window and free the retrieved data
- 2 Fit image into graphics window
- 3 Only display last iconic object
- 4 Switch interpretation of string values between system locale and UTF-8. Typically, this setting should match the used HALCON/C++ interface encoding as specified with `HalconCpp::SetHcppInterfaceStringEncodingIsUtf8`; see also section 5.2.1 on page 36.
- 5 Switch layout between vertical and horizontal

Iconic variables may be stacked: For example, image data may be overlayed with region data by first inspecting the image variable and then the region variable.

Getting Help

You can open the operator reference of a specific operator right from the context menu of Visual Studio. Move the mouse cursor over a method name, right-click it and select `HALCON Help`.

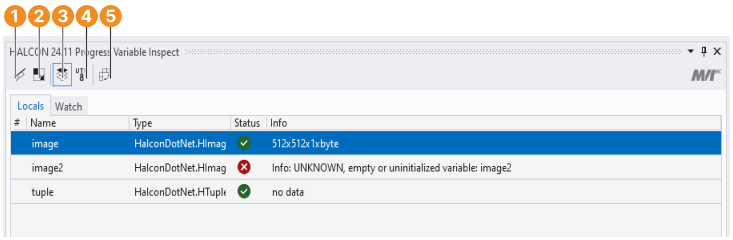


Figure 3.1: Inspecting iconic variables.

Persistent Settings

The following settings are persistent between sessions:

- Only display last iconic object, figure 3.1 on page 27 (3).
- Window layout (horizontal/vertical), figure 3.1 on page 27 (4).
- Selected tab (Locals/Watch), figure 3.1 on page 27.

Removing HALCON Variable Inspect From Visual Studio

To remove the extension from Visual Studio follow these steps:

1. Open Visual Studio.
2. Select `Extensions` → `Manage Extensions`.
3. Look for "HALCON 25.11 Progress Variable Inspect" in the list of installed extensions and select `Uninstall`.

3.5 Handling Licensing Errors

When running HALCON with a runtime license and a dongle, HALCON checks regularly whether the dongle is still available. This check has no measurable impact on the performance of HALCON.

In order to be notified if the dongle was removed and a license error is imminent, applications can register a callback function with HALCON that will be called when HALCON detects that the dongle is no longer available. HALCON operators will fail with licensing errors after about two to four minutes after this callback fires. If the dongle is reinserted before this time, HALCON will continue to operate normally.

Please note that if you do not register a callback and the dongle becomes unavailable during runtime, HALCON operators will fail with a license error and will therefore not provide meaningful output parameters. Depending on your application, this might even lead to crashes. It is highly suggested to handle errors from operators correctly or register the callback to notify the user and/or shut down the application in a controlled way.

To register your own callback function, use the following code in your application:

```
void __stdcall MyLicenseRecheckFailedCallback(void *context, Herror error)
{
  <Add your application-specific callback code here.>
}
```

Somewhere in your application startup code, add the following:

```
HSetLicenseRecheckFailedCallback(MyLicenseRecheckFailedCallback,
                                 &MyLicenseRecheckFailedContext);
```

`MyLicenseRecheckFailedContext` is a user-defined structure that you can use to pass extra information to your callback. If you do not need this, you can pass a `NULL` pointer instead.

To unregister a callback, simply call HSetLicenseRecheckFailedCallback with a NULL pointer as argument.

Note that callbacks are not chained – registering a callback will overwrite any previously registered callback.

For example, in your application, you might implement a proper licensing error handling like this:

```
Herror error;
Htuple param, value;

create_tuple_s(&param, "version");
set_check("~give_error");
error = T_get_system(param, &value);
destroy_tuple(value);
destroy_tuple(param);

if ((error >= H_ERR_LIC_NO_LICENSE) && (error <= H_ERR_LAST_LIC_ERROR))
{
  /* Handle licensing error here. */
}
```

Figure 3.2: Check for licensing errors in C.

```
try
{
  HalconCpp::HTuple value = HalconCpp::HSystem::GetSystem("version");
}
catch (HalconCpp::HException &exception)
{
  if (   (exception.ErrorCode() >= H_ERR_LIC_NO_LICENSE)
      && (exception.ErrorCode() <= H_ERR_LAST_LIC_ERROR))
  {
    // Handle licensing error here.
  }
}
```

Figure 3.3: Check for licensing errors in C++.

To check whether a license is available at all, you can use any operator at the beginning of your application and check the result.

3.6 Graphical Applications

This section gives some general hints for developing graphical applications.

If you are using graphical user interface (GUI) frameworks, each dialog is managed by a user interface (UI) thread. The UI thread is responsible for updating the GUI and the execution of callbacks. This means that the GUI is blocked as long as the UI thread executes the callback. Therefore, callbacks should require as little time as possible.

If you are using HDevEngine, we recommend that you divide your programming tasks into at least two categories: image processing and visualization.

For image processing tasks that are initiated by a user action you should start a new thread that handles those tasks. As soon as the image processing is finished, the UI thread should fetch and display the results. For more information, please refer to our multithreading example program:

```
%HALCONEXAMPLES%/c#/MultiThreading
```

Part II

Programming With HALCON/C++

Chapter 4

Introducing HALCON/C++

HALCON/C++ is HALCON's interface 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:

- In section 4.1 on page 31, we start with a first example program.
- Chapter 5 on page 33 then takes a closer look at the basics of the HALCON/C++ interface,
- while chapter 6 on page 43 gives an overview of the classes `HImage`, etc.
- Chapter 7 on page 49 shows how to create applications based on HALCON/C++.
- Chapter 8 on page 55 presents typical image processing problems and shows how to solve them using HALCON/C++.

4.1 A First Example

This section demonstrates how to create a simple HALCON application with C++. For a more comprehensive description, see chapter 7 on page 49.

The task is to read an image and compute the number of connected regions in it, as illustrated in figure 4.1 on page 31

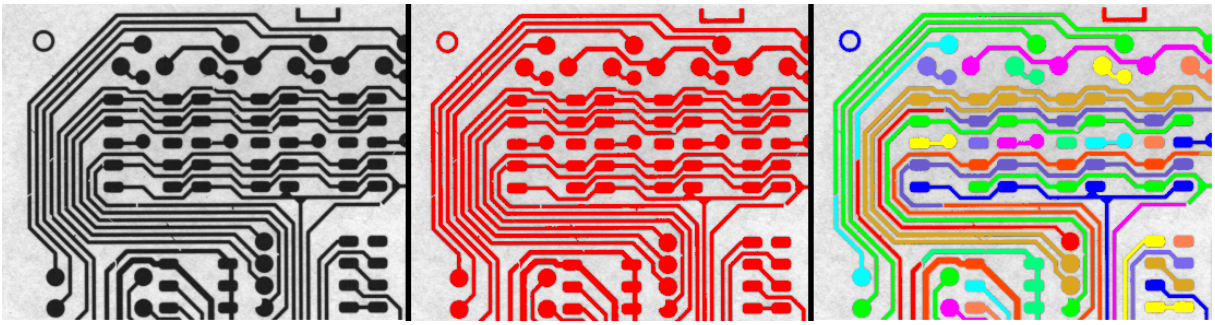


Figure 4.1: Left: Input image of a printed circuit board. Middle: Regions found by `threshold`, colored red. Right: Connected regions, a result of `connection`.

1. Install HALCON 25.11
2. Install a C++11 or newer toolchain on your system.
3. Setup your C++ environment of choice.

4. Run the following commands in a shell:

```
mkdir region_example
cd region_example
```

5. Create a file named `main.cpp` and change the content to:

```
#include <iostream>

#include <HalconCpp.h>

using HalconCpp::HImage;
using HalconCpp::HRegion;

int main()
{
    HImage img{"pcb"};

    HRegion region = img.Threshold(0, 122);
    Hlong numRegions = region.Connection().CountObj();

    std::cout << "Number of Regions: " << numRegions << '\n';
}
```

6. Compile the program.
   For details see section 7.5 on page 51 for Windows and section 7.6 on page 53 for Linux.
7. To run the application, type the following command in the same shell:

```
./region_example
```

As a result, you should see the following output `'Number of Regions:  43'`.

Chapter 5

Basics of the HALCON/C++ Interface

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.:

```
HObject original_image, smoothed_image;
ReadImage(&original_image, "monkey");
MeanImage(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:

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

This simple example already shows that the two approaches result in clearly different code: 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 using the object-oriented approach. Note, however, that HDevelop can export programs only as procedural C++ code. Section 5.5 on page 41 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 43 describes the provided classes in more detail.

5.1 HalconCpp Namespace

All functions and classes of HALCON/C++ use the namespace `HalconCpp` to prevent potential name conflicts with other C++ libraries.

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

- **Specifically**, by prefixing each class name or operator call with the namespace:

  ```
  HalconCpp::HObject original_image, smoothed_image;
  HalconCpp::ReadImage(&original_image, "monkey");
  ```

- **Locally**, by placing the directive `using namespace HalconCpp;` at the beginning of a block, e.g., at the beginning of a function:

```
int main(int argc, char* argv[])
{
  using namespace HalconCpp;

  HObject original_image, smoothed_image;
  ReadImage(&original_image, "monkey");
```

In this case, you can use HALCON's classes and functions without prefix inside this block.

- **Globally**, by placing the directive directly after including `HalconCpp.h`. In this case, you do not need the prefix in your whole application:

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

- **Universally**, which can be done both at local and global scope:

```
using HalconCpp::HObject;
using HalconCpp::ReadImage;

HObject original_image, smoothed_image;
ReadImage(&original_image, "monkey");
```

Which method is most suitable depends on your application, more specifically, what other libraries it contains and whether there are any name collisions.

Please note that the namespace is not mentioned in the operator descriptions in the reference manual 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 operator reference manual. As an example, figure 5.1 shows parts of the entry for the operator `MeanImage`.

void `MeanImage` (const HObject& Image, HObject* ImageMean, const HTuple& MaskWidth, const HTuple& MaskHeight)

HImage `HImage::MeanImage` (Hlong MaskWidth, Hlong MaskHeight) const

`Image` (input_object) ... (multichannel-)image(-array) ⇝ *HObject* (byte / int2 / uint2 / int4 / int8 / real / vector_field)

`ImageMean` (output_object) ... (multichannel-)image(-array) ⇝ *HObject* (byte / int2 / uint2 / int4 / int8 / real / vector_field)

`MaskWidth` (input_control) ..... extent.x ⇝ *HTuple (Hlong)*

`MaskHeight` (input_control) ..... extent.y ⇝ *HTuple (Hlong)*

Figure 5.1: The head and parts of the parameter section of the reference manual entry for `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\halconcpp\HOperatorSet.h`.

Below, we

- take a closer look at the parameters of an operator call (section 5.2.1 on page 35)
- describe how to call operators via classes (section 5.2.2 on page 37) or via special constructors (section 5.2.3 on page 37) or destructors (section 5.2.4 on page 38)
- explain another special HALCON concept, the *tuple mode* (section 5.2.5 on page 38)

void `FindBarCode` (const HObject& Image, HObject* SymbolRegions, const HTuple& BarCodeHandle, const HTuple& CodeType, HTuple* DecodedDataStrings)

HRegion `HBarCode::FindBarCode` (const HImage& Image, const HTuple& CodeType, HTuple* DecodedDataStrings) const

HRegion `HBarCode::FindBarCode` (const HImage& Image, const HString& CodeType, HString* DecodedDataStrings) const

HRegion `HBarCode::FindBarCode` (const HImage& Image, const char* CodeType, HString* DecodedDataStrings) const

HRegion `HBarCode::FindBarCode` (const HImage& Image, const wchar_t* CodeType, HString* DecodedDataStrings) const

HRegion `HImage::FindBarCode` (const HBarCode& BarCodeHandle, const HTuple& CodeType, HTuple* DecodedDataStrings) const

HRegion `HImage::FindBarCode` (const HBarCode& BarCodeHandle, const HString& CodeType, HString* DecodedDataStrings) const

HRegion `HImage::FindBarCode` (const HBarCode& BarCodeHandle, const char* CodeType, HString* DecodedDataStrings) const

HRegion `HImage::FindBarCode` (const HBarCode& BarCodeHandle, const wchar_t* CodeType, HString* DecodedDataStrings) const

`Image` (input_object) ..... singlechannelimage ⇝ *HObject* (byte)

`SymbolRegions` (output_object) ..... region(-array) ⇝ *HObject*

`BarCodeHandle` (input_control) ..... barcode ⇝ *HTuple (HHandle)*

`CodeType` (input_control) ..... string(-array) ⇝ *HTuple (HString)*

`DecodedDataStrings` (output_control) ..... string(-array) ⇝ *HTuple (HString)*

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

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 values such as integers, floating-point numbers, strings, or handles.

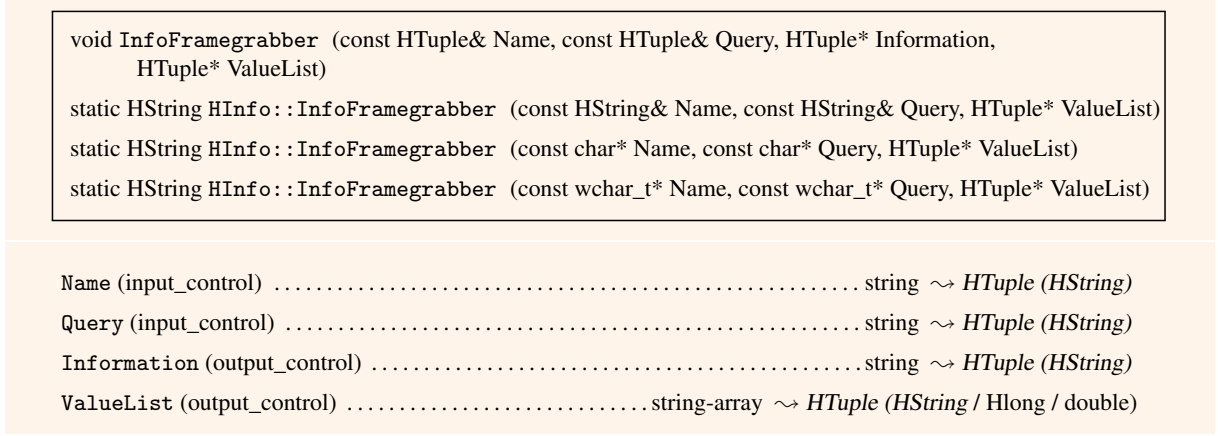
A special form of control parameters are the so-called *handles*. A well-known representative of this type is the *window handle*, which provides access to an opened HALCON window, e.g., to display an image in it. Besides, handles are used when operators share complex data, e.g., the operators for shape-based 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 45.

Both iconic and control parameters can appear as input and output parameters of a HALCON operator. For example, the operator `MeanImage` 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 parentheses if you call an operator via a class; this mechanism is described in more detail in section 5.2.2 on page 37.

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., `Hlong MaskWidth` in figure 5.1) or via a constant reference (e.g., `const HObject& Image`). 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:

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

In contrast to input parameters, output parameters are always modified, thus they must be passed *by reference*.



void InfoFramegrabber (const HTuple& Name, const HTuple& Query, HTuple* Information, HTuple* ValueList)

static HString HInfo::InfoFramegrabber (const HString& Name, const HString& Query, HTuple* ValueList)

static HString HInfo::InfoFramegrabber (const char* Name, const char* Query, HTuple* ValueList)

static HString HInfo::InfoFramegrabber (const wchar_t* Name, const wchar_t* Query, HTuple* ValueList)

Name (input_control) ..... string ⇝ *HTuple (HString)*

Query (input_control) ..... string ⇝ *HTuple (HString)*

Information (output_control) ..... string ⇝ *HTuple (HString)*

ValueList (output_control) ..... string-array ⇝ *HTuple (HString* / Hlong / double)

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

Note that operators expect a pointer to **an already existing variable or class instance**! For example, when calling the operator `FindBarCode` as in the following lines of code, variables of the class `HTuple` are declared before passing the corresponding pointers using the operator `&`.

```
HImage   image("barcode/ean13/ean1301");
HBarCode barcode(HTuple(), HTuple());
HString  result;

HRegion code_region = barcode.FindBarCode(image, "EAN-13", &result);
```

The above example shows another interesting aspect of output parameters: When calling operators via classes, one output parameter may become the return value (see section 5.2.2 on page 37 for more details); in the example, `FindBarCode` returns the bar code region.

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 5.2.5 on page 38 for more information.

String Parameters

Regardless of the encoding of the HALCON library (`set_system('filename_encoding', ...)`) the HALCON/C++ interface expects raw char pointer strings that are passed to HALCON operators and to `HTuple` or `HString` instances to be UTF-8 encoded.

Output strings are always of type `HString` with automatic memory management. These strings are by default also UTF-8 encoded. The encoding of the HALCON/C++ interface (interface encoding) can be changed to local-8-bit encoding via a call of `HalconCpp::SetHcppInterfaceStringEncodingIsUtf8(false)`. The current interface encoding can be requested via `HalconCpp::IsHcppInterfaceStringEncodingUtf8()`. It is not recommended to switch the interface encoding back and forth. The setting should be adjusted only once at the very beginning of the program (before the first HALCON operator or assignment), because `HTuple` instances can not store in which encoding the contained strings are present, i.e, for all write and read accesses, the same encoding must be set. Furthermore, the interface encoding is set globally and is therefore not suitable for multithreading programs: Changing the setting in one thread has an effect on other threads.

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:

```
HString     sInfo, sValue;

InfoFramegrabber(FGName, "info_boards", &sInfo, &sValue);
```

Note that it is also not necessary to allocate memory for multiple output string parameters returned as `HTuple`:

```
HImage   image("barcode/ean13/ean1301");
HBarCode barcode(HTuple(), HTuple());
HString  result;


HRegion code_region = barcode.FindBarCode(image, "EAN-13", &result);
```

```
HRegion code_region = image.FindBarCode(barcode, "EAN-13", &result);
```

```
HObject image;
HTuple  barcode;
HObject code_region;
HTuple  result;

ReadImage(&image, "barcode/ean13/ean1301");
CreateBarCodeModel(HTuple(), HTuple(), &barcode);
FindBarCode(image, &code_region, barcode, "EAN-13", &result);
```

Figure 5.4: Using `FindBarCode` via `HBarCode`, via `HImage`, or in the procedural approach.

```
HTuple      tInfo, tValues;

InfoFramegrabber(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, `FindBarCode` can be called via objects of the class `HImage` or `HBarCode` (see figure 5.2 on page 35). In both cases, the corresponding input parameter (`Image` or `BarCodeHandle`, respectively) does not appear within the parentheses 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 `SymbolRegions`) also disappears from within the parentheses and becomes the return value instead of the error code (more about error handling can be found in section 5.3 on page 40).

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

5.2.3 Constructors and Halcon Operators

As can be seen in figure 5.4 on page 37, the HALCON/C++ parameter classes provide additional constructors, which are based on suitable HALCON operators. The constructors for `HImage` and `HBarCode` used in the example are based on `ReadImage` and `CreateBarCodeModel`, respectively.

As a rule of thumb: If a class appears only as an output parameter in an operator, there automatically exists a constructor based on this operator. Thus, instances of `HBarCode` can be constructed based on `CreateBarCodeModel` as shown in figure 5.4 on page 37, instances of `HShapeModel` based on `CreateShapeModel`, instances of `HFramegrabber` based on `OpenFramegrabber` and so on. Note that for classes where many such operators exist (e.g., `HImage`), only a subset of commonly used operators with unambiguous parameter list are actually used as constructor.

In addition, all classes have empty constructors to create an uninitialized object. For example, you can create an instance of `HBarCode` with the default constructor and then initialize it using `CreateBarCodeModel` as follows:

```
HBarCode barcode;
barcode.CreateBarCodeModel(HTuple(), HTuple());
```

If the instance was already initialized, the corresponding data structures are automatically destroyed before constructing and initializing them anew (see also section 5.2.4). The handle classes are described in more detail in section 6.2.3.2 on page 46.

```
HImage image;                  // still uninitialized
image.ReadImage("clip");
```

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 HALCON operator reference and the corresponding header files in `%HALCONROOT%\include\cpp`.

- **Images:**
  The class `HImage` provides constructors based on the operators `ReadImage`, `GenImage1`, and `GenImageConst`.
- **Regions:**
  The class `HRegion` provides constructors based on operators like `GenRectangle2` or `GenCircle`.
- **Windows:**
  The class `HWindow` provides a constructor based on the operator `OpenWindow`.

  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 46.

5.2.4 Destructors and Halcon Operators

All HALCON/C++ classes provide default destructors which automatically free the corresponding memory.

The default destructors of classes encapsulating handles, e.g., `HShapeModel` or `HFramegrabber`, work similar to members like `ClearShapeModel` or `CloseFramegrabber`, respectively.

There is no need to call these operators as you can initialize instances anew as described in section 5.2.3.

Basically, we differentiate between destroying a handle and destroying the underlying data structure. The data structure can be destroyed in two ways: Automatically as soon as the last reference to the data structure has been deleted. Explicitly by calling an operator, e.g., CloseWindow. Explicit destruction invalidates references, but access is secure.

5.2.5 The Tuple Mode

As already mentioned in section 5.2.1 on page 35, 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 `CharThreshold`: 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.

If you call `CharThreshold` 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.

Note that the class `HTuple` can also contain arrays (tuples) of control parameters of mixed type; please refer to section 6.2.1 on page 44 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.

In the object-oriented 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 us take a look at two examples which are both realized in the object-oriented and in the procedural approach. The examples highlight some interesting points:

void CharThreshold (const HObject& Image, const HObject& HistoRegion, HObject* Characters, const HTuple& Sigma, const HTuple& Percent, HTuple* Threshold)

HRegion HImage::CharThreshold (const HRegion& HistoRegion, double Sigma, const HTuple& Percent, HTuple* Threshold) const

HRegion HImage::CharThreshold (const HRegion& HistoRegion, double Sigma, double Percent, Hlong* Threshold) const

Image (input_object) ..... singlechannelimage(-array) ⇝ *HObject* (byte)
HistoRegion (input_object) ..... region ⇝ *HObject*
Characters (output_object) ..... region(-array) ⇝ *HObject*
Sigma (input_control) ..... number ⇝ *HTuple (double)*
Percent (input_control) ..... number ⇝ *HTuple (double* / Hlong)
Threshold (output_control) ..... integer(-array) ⇝ *HTuple (Hlong)*

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

- **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 `CountObj()`. In the procedural approach, objects must be selected explicitly using the operator `SelectObj`; the number of objects can be queried via `CountObj`.

  Note that object indexes start with 1 (as used by `SelectObj`).

- **Polymorphism of `HObject`:**
  The class `HObject` is used for all types of iconic objects. What is more, image objects can be used for parameters expecting a region, as in the call to `CharThreshold` in the examples; in this case, the *domain* of the image, i.e., the region in which the pixels are "valid", is extracted automatically. The object-oriented approach supports an implicit cast from `HImage` to `HRegion`.

The first example shows how `CharThreshold` is applied in simple mode, i.e., to a single image:

```
// object-oriented approach
  HImage  image("alpha1");
  HRegion region;
  Hlong   threshold;

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

```
// procedural approach
  HObject image;
  HObject region;
  HTuple  threshold;

  ReadImage(&image, "alpha1");
  CharThreshold(image, image, &region, 2, 95, &threshold);
  DispObj(image, window);
  DispObj(region, window);
  cout << "Threshold for 'alpha1': " << threshold.ToString();
```

The second example shows how `CharThreshold` is applied in tuple mode, i.e., to two images at once:

```
// object-oriented approach
  HImage  images;
  HRegion regions;
  HTuple  thresholds;

  images.GenEmptyObj();
  for (int i = 1; i <= 2; i++)
  {
    images = images.ConcatObj(HImage(HTuple("alpha") + i));
  }

  regions = images.CharThreshold(images.GetDomain()[1], 2, 95, &thresholds);

  for (int i = 1; i <= images.CountObj(); i++)
  {
    images[i].DispImage(window);
    regions[i].DispRegion(window);
    cout << "Threshold for 'alpha" << i << "': " << thresholds[i - 1].L();
    window.Click();
```

```
// procedural approach
  HObject images, image;
  HObject regions, region;
  HTuple  num;
  HTuple  thresholds;

  GenEmptyObj(&images);

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

  CharThreshold(images, image, &regions, 2, 95, &thresholds);
  CountObj(images, &num);

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

5.3 Error Handling

Error handling is fully based on exceptions using `try` ... `catch` blocks.

The following code shows how to catch and evaluate errors that might occur when reading an image from file. The call to `ReadImage` is encapsulated by a `try` block; the error code of the exception is evaluated in a corresponding `catch` block. For more information on HALCON error codes please refer to the Extension Package Programmer's Manual, appendix A on page 105.

```
try
{
  image.ReadImage(filename);
}
catch (HException &except)
{
  if (except.ErrorCode() == H_ERR_FNF)
  {
    // Handle file not found error
  }
  else
  {
    // Pass on unexpected error to caller
    throw except;
  }
}
```


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 resources automatically in their destructor (see also section 5.2.4 on page 38). Furthermore, when constructing instances anew, e.g., by calling `CreateBarCodeModel` via an already initialized instance as mentioned in section 5.2.3 on page 37, the already allocated memory is automatically released before reusing the instance. Thus, there is no need to call the operator `ClearObj` in HALCON/C++; what is more, if you do use it HALCON will complain about already released memory. To explicitly release the resources before the instance gets out of scope, you can call the method `Clear()` of the instance.

5.5 How to Combine Procedural and Object-Oriented Code

As already noted, we recommend using 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.

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**

  ```
  HObject p_image;
  ReadImage(&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 specific handle classes**

  ```
  HTuple  p_barcode;
  CreateBarCodeModel(HTuple(), HTuple(), &p_barcode);
  HBarCode o_barcode(p_barcode.H());
  o_code_region = o_barcode.FindBarCode(o_image, "EAN-13", &result);
  ```

Note that instances of `HImage` can be used in procedural code where `HObject` is expected.

As already remarked in section 5.2.4 on page 38, you must not use operators like `ClearShapeModel`, `ClearAllShapeModels`, or `CloseFramegrabber` together with instances of the corresponding handle classes!

5.6 I/O Streams

HALCON/C++ provides iostream operators by default. Note that it may be necessary to enable the namespace `std`:

```
using namespace std;
```

If you want to use the older iostream interface (i.e., <`iostream.h`> instead of <`iostream`>), the following line has to be added (otherwise, there may be conflicts with the HALCON include files):

```
#define HCPP_NO_USE_IOSTREAM
```

5.7 Windows API Collisions

`FindText`, `CreateMutex`, `CreateEvent`, and `DeleteFile` are also functions of the Windows API. There are defines on `FindTextW`, `CreateMutexW`, `CreateEventW`, and `DeleteFileW` if `UNICODE` is defined, otherwise there are defines on `FindTextA`, `CreateMutexA`, `CreateMutexA`, and `DeleteFileA`. These defines are undefined in `HalconCpp.h`. If you want to use the corresponding Windows API calls, you must use `FindTextA`, `FindTextW`, `CreateMutexA`, `CreateMutexW`, `CreateEventA`, `CreateEventW`, `DeleteFileA`, or `DeleteFileW` directly.

Chapter 6

The HALCON Parameter Classes

C++

The HALCON operator reference contains a complete list of the generic classes and member functions of HALCON/C++. This chapter contains a summary of additional convenience members.

In addition, HALCON/C++ contains many **operator overloads**, which are consistent with HALCON/.NET. See section 11.1.4 on page 71 for a list of the overloaded operators.

6.1 Iconic Objects

The base class of the iconic parameter classes in HALCON/C++ is the class `HObject` which manages entries in the database, i.e., the copying or releasing of objects. The class `HObject` can contain all types of iconic objects. This has the advantage that important methods like `DispObj()` can be applied to all iconic objects in the same manner.

Three classes are derived from the root class `HObject`:

- Class `HRegion` for handling regions.
- Class `HImage` for handling images.
- Class `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 `HRegion` represents a region in HALCON/C++. Besides those operators that can be called via `HRegion` (see also section 5.2.2 on page 37), `HRegion` provides the following member functions:

- `HTuple HRegion::Area()`
  Area of the region, i.e., number of pixels, see reference manual entry of `AreaCenter`.
- `HTuple HRegion::Row()`
  Center row of the region.
- `HTuple HRegion::Column()`
  Center column of the region.

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.

- `HTuple HImage::Width()`
  Return the width of the image, see reference manual entry of `GetImageSize`.
- `HTuple HImage::Height()`
  Return the height of the image, see reference manual entry of `GetImageSize`.

6.1.3 XLD Objects

XLD is the abbreviation for e**X**tended **L**ine **D**escription. 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.

HALCON/C++ provides both a base class `HXLD` and a set of specialized classes derived from `HXLD`, e.g., `HXLDCont` for contours or `HXLDPoly` for polygons.

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 37).

6.2 Control Parameters

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

- integer numbers (`Hlong`),
- floating point numbers (`double`), and
- strings (`HString`).
- handles (`HHandle`).

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 represented by a distinct control data type. They are "magic" values that must not be changed and can differ from execution to execution and version to version. Once all instances of the handle are overwritten or cleared, the handle and its content will be destroyed and will become invalid. For handles there exist corresponding classes, which are described in section 6.2.3 on page 45.

With the class `HTuple`, HALCON/C++ provides a container class for control parameters. `HTuple` may contain arrays of control parameters of mixed type.

6.2.1 Tuples

The class `HTuple` implements an array of dynamic length. The default constructor constructs an empty array (`Length() == 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 `Length()` $- 1$.

In order to use instances of an `HTuple` in different threads concurrently, the method `HTuple::Clone()` has to be used (see below).

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\halconcpp`.

- `HTuple &HTuple::Append(const HTuple& tuple)`
  Append data to existing tuple.

- `void HTuple::Clear()`
  Clear all data inside the tuple.
- `HTuple HTuple::Clone()`
  Create a detached copy duplication the tuple data.
- `Hlong HTuple::Length()`
  Return the number of elements of the tuple.
- `HTupleType HTuple::Type()`
  Return the data type of the tuple (pure data types or mixed tuple).
- `HString HTuple::ToString()`
  Return a simple string representation of the tuple contents.
- `Hlong* HTuple::LArr()`
  `double* HTuple::DArr()`
  `char** HTuple::SArr()`
  `Hcpar* HTuple::PArr()`
  `Hphandle* HTuple::HArr()`
  Access tuple data.

String Encoding

The class `HTuple` always stores raw char pointer strings in the interface encoding. This is important especially when accessing the internal raw pointer via the methods `HTuple::SArr()` and `HTupleElement::C()`.

Under Windows, it is possible to initialize an `HTuple` or an `HTupleElement` with a wide character string (`wchar_t*`, UTF-16). This wide character string is converted into the current interface encoding (default: UTF-8). If the interface encoding is not UTF-8, the conversion can involve transcoding errors, i.e., the stored string may not contain all characters of the input wide character string. The class `HTuple` does not allow accessing the string in a different encoding or as wide character string.

6.2.2 Strings

The class `HString` can be used when transcoding or access to a wide character raw pointer is needed. Like `HTuple`, the class can be initialized with a `wchar_t` pointer string (Windows only) or with a raw char pointer string with the current interface encoding. `HString` allows to create a string with a specific encoding via `HString::FromUtf8(const char*)` or `HString::FromLocal8bit(const char*)`.

Furthermore, this class allows storing the same string in UTF-8, local-8-bit encoding, and wide character string at the same time. Thus it is save to use the required raw pointer, as long as the `HString` instance is not modified or destroyed. The string remains the owner of the string memory for all representations.

The following methods for accessing raw pointers are provided:

- `HString::TextA()` and `HString::Text()`
  Return the string as char pointer in the interface encoding.

  When the char pointer string is needed in a different encoding, the methods `HString::ToUtf8()` and `HString::ToLocal8bit()` can be used.

  The methods `HString::LengthA()` and `HString::Length()` return the length in bytes of the char pointer string returned by `HString::TextA()` and `HString::Text()`, respectively.
- `HString::TextW()` (Windows only)
  Returns the string as wide character pointer.

  The method `HString::LengthW()` returns the length of the string returned by `HString::TextW()` in words of the datatype `wchar_t`.

6.2.3 Classes Encapsulating Handles

The perhaps most prominent handle class is `HWindow`, which is described in section 6.2.3.1. 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 46 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 37), HWindow provides the following member functions:

- void HWindow::Click()
  Wait for a mouse click in the window.
- void HWindow::CloseWindow()
  Close the window.

6.2.3.2 Other Handle Classes

HALCON/C++ provides the so-called handle classes like HFramegrabber, HBarCode, or HClassMlp. These are based on the class HHandle.

Besides the default constructor, the classes typically provide additional constructors based on suitable operators as described in section 5.2.3 on page 37; e.g., the class HBarCode provides a constructor based on the operator CreateBarCodeModel.

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

6.3 Vectors

HALCON/C++ provides the class HVector for the use of HALCON vectors in C++ programs. A HALCON vector is a container that can hold an arbitrary number of elements of the identical data type (i.e., tuple, iconic object, or vector) and dimension. The type of a vector, i.e., its dimension and the type of its elements is defined when initializing the vector instance and cannot be changed during its lifetime. A vector with one dimension may be a vector of tuples or a vector of iconic objects. A two-dimensional vector may be a vector of vectors of tuples or a vector of vectors of iconic objects, and so on.

Two classes are derived from the root class HVector:

- Class HObjectVector for handling vectors of iconic objects
- Class HTupleVector for handling vectors of tuples

In the following some basic information on how to use vectors in HALCON/C++ is given, e.g., how to construct vectors and how to access and set vector elements. For a complete list of the available functionality please refer to the corresponding header file HVector.h in %HALCONROOT%\include\halconcpp.

Construction of Vectors

As already mentioned above, a distinction is made between vectors of iconic objects (HObjectVector) and vectors of tuples (HTupleVector). The type of a vector must be defined at its construction as in the following lines:

```
// Create a one-dimensional vector of iconic objects
HObjectVector    vectorObj(1);

// Create a one-dimensional vector of tuples
HTupleVector     vectorTup(1);
```

Note that the type of the vector cannot be changed within a program after its construction. Thus, a tuple cannot be assigned to a vector of iconic objects and vice versa.

To create a two-dimensional vector, i.e., a vector of vectors of iconic objects or a vector of vector of tuples you may use the following line:

```
// Create a two-dimensional vector of iconic objects
HObjectVector    vectorObjMulti(2);

// Create a two-dimensional vector of tuples
HTupleVector     vectorTupMulti(2);
```

You can also create a multi-dimensional vector with more than two dimensions by specifying the desired dimension in brackets. However, the dimension of a vector is part of its type and has to remain constant within the program and cannot be changed.

Note that the vectors created by these calls are still empty. How to set the elements of vectors and how to access them is described below.

Accessing and Setting Vector Elements

Like with the construction of vectors, the call for accessing and setting vector elements differs depending on the vector type. A single element of a vector of iconic objects may be accessed by using the method `O()` whereas the elements of a vector of tuples may be accessed with `T()`.

```
// Access a vector element of a one-dimensional HObjectVector
vectorObj[elem_index].O();

// Access a vector element of a one-dimensional HTupleVector
vectorTup[elem_index].T();
```

The vector element to be accessed is addressed by the specified index in square brackets. If a subelement of a multi-dimensional vector is to be accessed, you have to use the indices of the corresponding subvector and its subelement instead.

```
// Access a subelement of a two-dimensional HObjectVector
vectorObjMulti[vec_index][elem_index].O();
```

The left index `vec_index` defines the index of the subvector and `elem_index` defines the desired element of the specified subvector.

If a vector element is to be set, the expression for accessing a vector element is used as reference to the `HObject` or `HTuple` element to be set. The right side of the assignment specifies the value which is assigned to the vector element.

```
// Set a vector element of a one-dimensional HObjectVector
vectorObj[0].O() = HImage("Image");

// Set a vector element of a one-dimensional HTupleVector
HTuple tuple;
tuple[0] = 1.0;
tuple[1] = 2.5;
vectorTup[0].T() = tuple;
```

In the example code above the `Image` is copied and set as the first vector element of `vectorObj`. The tuple is also copied and set as first vector element of `vectorTup`.

Setting a subelement of a multi-dimensional vector can be done with the same call. However, instead of a single index, multiple indices must be specified for the corresponding vector element and its subelement, which is to be set.

```
// Set a subelement in a two-dimensional HObjectVector
vectorObjMulti[0][1].O() = HImage("Image");
```

It is also allowed to write to a non-existing vector element. Then, the vector is automatically filled with empty elements if necessary.

Destruction of a Vector

If a vector is not needed anymore for further processing, its contents can be cleared with the following call, explicitly:

```
vectorTup.Clear();
```

Additional Information

In addition to the described functionalities `HObjectVector` and `HTupleVector` provide some more functionality for the use of HALCON vectors in HALCON/CPP, e.g., inserting or removing vector elements, or concatenation of vectors. Please refer to the corresponding header file `HVector.h` in `%HALCONROOT%\include\halconcpp` for more information.

Chapter 7

Creating Applications With HALCON/C++

The HALCON distribution contains examples for creating an application with HALCON/C++. The following sections show

- the relevant directories and files (section 7.1 on page 49)
- the list of provided example applications (section 7.2 on page 50)
- the relevant environment variables (section 7.3 on page 50)
- how to store source files with non-ASCII characters (section 7.4 on page 51)
- how to create an executable under Windows (section 7.5 on page 51)
- how to create an executable under Linux (section 7.6 on page 53)

7.1 Relevant Directories and Files

Here is an overview of the relevant directories and files (relative to `%HALCONROOT%`, Windows notation of paths):

- `include`
  include directory; contains, e.g., `Halcon.h`, which is referenced by `HalconCpp.h`
- `include\halconcpp\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`
  HALCON library (Windows)
- `bin\%HALCONARCH%\halconcpp.dll`, `lib\%HALCONARCH%\halconcpp.lib`
  HALCON/C++ library (Windows)
- `bin\%HALCONARCH%\halconxl.dll`, `halconcppxl.dll`, `lib\%HALCONARCH%\halconxl.lib`, `halconcppxl.lib`
  Corresponding libraries of HALCON XL (Windows)
- `lib/$HALCONARCH/libhalcon.so`
  HALCON library (Linux)
- `lib/$HALCONARCH/libhalconcpp.so`
  HALCON/C++ library (Linux)
- `lib/$HALCONARCH/libhalconxl.so, libhalconcppxl.so`
  Corresponding libraries of HALCON XL (Linux)

- include\HProto.h
  External function declarations
- %HALCONEXAMPLES%\cpp\console\CMakeLists.txt
  Example CMake file, which can be used to compile the example programs
- %HALCONEXAMPLES%\cpp\console\README.md
  Information about building the examples using CMake
- %HALCONEXAMPLES%\cpp\console\source\
  Directory containing the source files of the example programs
- %HALCONEXAMPLES%\images\
  Images used by the example programs
- help\operators_*
  Files necessary for online information
- doc\pdf\
  Various manuals (in subdirectories)

7.2 Example Programs

You can find several example programs in the HALCON/C++ distribution (%HALCONEXAMPLES%\cpp\source\console\). To experiment with these examples, we recommend creating a private copy in your working directory.

error_handling.cpp	Demonstrates the C++ exception handling (see section 5.3 on page 40)
ia_callback.cpp	Shows the usage of the HALCON image acquisition callback functionality
matching.cpp	Locates a chip on a board and measures the pins
serialized_item.cpp	Shows how to use the serialization of HALCON objects and tuples in the C++ interface

You can find additional examples for using HALCON/C++ in the subdirectories mfc, motif, and qt of %HALCONEXAMPLES%.

7.3 Relevant Environment Variables

In the following, we briefly describe the relevant environment variables. For more information, especially about how to set these variables, see the Installation Guide, section A.4 on page 47. 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.

The variable HALCONARCH describes the platform HALCON is used on. Please refer to the Installation Guide, section 1.4 on page 8, for more information.

The variable HALCONEXAMPLES indicates where the provided examples are installed.

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.

Keep the following in mind in connection with the example programs:

- Default location for images
  The default directory for the HALCON operator ReadImage to look for images is %HALCONEXAMPLES%\images. If the images reside in different directories, the appropriate path must be set in ReadImage or the default image directory must be changed, using SetSystem("image_dir","..."). This is also possible with the environment variable HALCONIMAGES. The latter has to be set before starting the program.

- Output terminal under Linux
  In the example programs, no host name is passed to `OpenWindow`. 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`.

7.4 Writing a Program With Non-ASCII Characters

There are some common traps and pitfalls when programs containing string constants with non-ASCII characters are executed. The following ways to store the source files have different implications:

- Store source file in local 8-bit encoding
  Only native characters can be stored. String constants that are created by the compiler are also encoded in local 8-bit encoding, i.e., the execution character set is local 8-bit. If strings in local 8-bit encoding are passed to HALCON C or HALCON/C++, either change the interface default encoding (by calling `SetHcInterfaceStringEncodingIsUtf8(false)` or `SetHcppInterfaceStringEncodingIsUtf8(false)`, respectively) or create the string that is passed to HALCON with a suitable conversion function, e.g., `HString::FromLocal8Bit()`. When using HALCON/C++ as an alternative, the string constant can be created as wide character string with "L...".
- Store source file in UTF-8 with or without BOM signature
  The big advantage is that any Unicode characters can be stored. To store files without BOM signature in Visual Studio, choose `Unicode (UTF-8 without signature) - Codepage 65001` in the `Save As` or `Advanced Save Options` dialog. Without BOM signature, the files are not automatically detected as UTF-8 files, i.e. they are interpreted as local 8-bit files. Files with BOM signature marker are correctly read as UTF-8 files. However, the compiler converts the string constants automatically into local 8-bit encoding, i.e., although the source character set is UTF-8, the execution character set is still local 8-bit. Hence, when passing these strings to HALCON, all points mentioned above are valid. When using Visual Studio 2015 it is possible to set the compiler option `/utf-8`. You can add this option in `Project Property Page > Configuration Properties > C/C++ > All Options > Additional Options`. If this option is set, both the source character set and the execution character set are UTF-8 (which works also without the BOM signature). In this case, all strings are UTF-8 and can directly be passed to HALCON.

7.5 Creating an Executable 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 following command near the top of your C++ file:

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

To create an application, link the library `halconcpp.lib` to your program.

The example projects show the necessary Visual C++ project settings. Basically, you need to specify the correct include path in the compiler settings, and the correct library path and libraries in the linker settings:

```
Compiler:
  Include Directories:     $(HALCONROOT)\include,$(HALCONROOT)\include\halconcpp
Linker:
  Library Directories:     $(HALCONROOT)\lib\$(HALCONARCH)
  Additional Dependencies: halconcpp.lib
```

HALCON XL applications:
Please note that you should **use HALCON XL only when you need its features**.

If you want to use HALCON XL, link the library `halconcppxl.lib` instead.

Compiling and Linking an Example Program Under Windows

The following example shows a very basic C++ program called `main.cpp`, which reads an image and then prints its width and height:

```
#include "HalconCpp.h"

#include <iostream>

int main(int argc, char **argv)
{
  HalconCpp::HObject hobj;
  HalconCpp::HTuple width, height;
  HalconCpp::ReadImage(&hobj, "printer_chip/printer_chip_01");
  HalconCpp::GetImageSize(hobj, &width, &height);

  std::cout << "Image - width: " << width.I() << ", height: " << height.I() << '\n';
  return 0;
}
```

To compile and link the program under Windows, run:

```
cl main.cpp /I %HALCONROOT%\include /I %HALCONROOT%\include\halconcpp
  /link %HALCONROOT%\lib\%HALCONARCH%\halconcpp.lib
```

Building HALCON Examples With CMake Under Windows

You can find examples that showcase various HALCON use cases and how they are implemented in C++ in the directories below `%HALCONEXAMPLES%\cpp`. You can build them with CMake by using the provided `CMakeLists.txt` file.

To build example programs using CMake, do the following:

1. If required, download CMake (version 3.7.1 or later) from the CMake website and install it.
2. Create a build directory and then run `cmake` to configure the build and create the application:

   ```
   mkdir build
   cd build
   cmake %HALCONEXAMPLES%\cpp\console
   cmake --build .
   ```

   To configure, CMake needs to know the location of the HALCON installation, the location of the example files, and which HALCON architecture to use:

   - Specify the location of the HALCON installation via the `HALCON_DIR` CMake option, or via the `%HALCONROOT%` environment variable if the option is not set.
   - Specify the location of the HALCON example files via the `%HALCONEXAMPLES%` environment variable.
   - Specify the HALCON architecture with the `HALCON_ARCHITECTURE` CMake option, or via the `%HALCONARCH%` environment variable. If neither the option nor the environment variable are set, CMake will try to guess the architecture based on the host build system.

   For general information on how to use CMake, see the CMake documentation.

   HALCON XL applications: By default, the examples will be built using the normal version of HALCON. If you want to build using HALCON XL, set the option `HALCON_XL` to `ON` or `1` in CMake during the configuration step. For this, use the following syntax:

   ```
   cmake -DHALCON_XL=1 %HALCONEXAMPLES%\cpp\console
   ```

3. Optionally, you can use the `-G` option to specify the generator for a new build tree.
   For more information about CMake generators, see the CMake documentation.

You can find corresponding `CMakeLists.txt` files in the other subdirectories as well (`mfc\FGMultiThreading`, `mfc\Matching`, `mfc\MatchingExtWin`, `qt\Matching`).

7.6 Creating an Executable Under Linux

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 following command near the top of your C++ file:

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

To specify the include path for the compiler on the command line, use the following syntax:

```
-I$HALCONROOT/include -I$HALCONROOT/include/halconcpp
```

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. The library `libhalcon.so` is the HALCON library. To specify the library path and the libraries for the linker, use the following syntax:

```
-L$HALCONROOT/lib/$HALCONARCH -lhalconcpp -lhalcon -lpthread
```

On some systems, you also have to link the libraries `libdl.so` and `librt.so` (by using the additional options `-ldl` and `-lrt`, respectively).

HALCON XL applications:
Please note that you should **use HALCON XL only when you need its features**.

If you want to use HALCON XL, link the libraries `libhalconcppxl.so` and `libhalconxl.so` instead.

To link and run applications under Linux, ensure that the system variable `LD_LIBRARY_PATH` contains the HALCON library path `$HALCONROOT/lib/$HALCONARCH`.

Compiling and Linking an Example Program Under Linux

The following example shows a very basic C++ program called `main.cpp`, which reads an image and then prints its width and height:

```
#include "HalconCpp.h"

#include <iostream>

int main(int argc, char **argv)
{
  HalconCpp::HObject hobj;
  HalconCpp::HTuple width, height;
  HalconCpp::ReadImage(&hobj, "printer_chip/printer_chip_01");
  HalconCpp::GetImageSize(hobj, &width, &height);

  std::cout << "Image - width: " << width.I() << ", height: " << height.I() << '\n';
  return 0;
}
```

To compile and link the program under Linux, run:

```
g++ -o example main.cpp  -I$HALCONROOT/include -I$HALCONROOT/include/halconcpp  \
  -L$HALCONROOT/lib/$HALCONARCH -lhalconcpp -lhalcon -lpthread
```

Building HALCON Examples With CMake Under Linux

You can find examples that showcase various HALCON use cases and how they are implemented in C++ in the directories below `$HALCONEXAMPLES/cpp`. You can build them with CMake by using the provided `CMakeLists.txt` file.

To build example programs using CMake, do the following:

1. If required, download CMake (version 3.7.1 or later) from the CMake website and install it.
2. Create a build directory and then run `cmake` to configure the build and create the application:

   ```
   mkdir build
   cd build
   cmake $HALCONEXAMPLES/cpp/console
   cmake --build .
   ```

   To configure, CMake needs to know the location of the HALCON installation, the location of the example files, and which HALCON architecture to use:

   - Specify the location of the HALCON installation via the `HALCON_DIR` CMake option, or via the `$HALCONROOT` environment variable if the option is not set.
   - Specify the location of the HALCON example files via the `$HALCONEXAMPLES` environment variable.
   - Specify the HALCON architecture with the `HALCON_ARCHITECTURE` CMake option, or via the `$HALCONARCH` environment variable. If neither the option nor the environment variable are set, CMake will try to guess the architecture based on the host build system.

   For general information on how to use CMake, see the CMake documentation.

   HALCON XL applications: By default, the examples will be built using the normal version of HALCON. If you want to build using HALCON XL, set the option `HALCON_XL` to `ON` or `1` in CMake during configuration.

3. Optionally, you can use the `-G` option to specify the generator for a new build tree.
   For more information about CMake generators, see the CMake documentation.

You can find corresponding `CMakeLists.txt` files in the other subdirectories as well (`mfc/FGMultiThreading`, `mfc/Matching`, `mfc/MatchingExtWin`, `motif/Matching`, `qt/Matching`).

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:

```
HImage  Image("file_xyz");
HRegion Threshold        = Image.Threshold(0,120);
HRegion ConnectedRegions = Threshold.Connection();
HRegion 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. It is also possible to use the equivalent call:

  ```
  HRegion Threshold = (Image <= 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:

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

Some notes:

- 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 the following sequence:

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

Note the following:

- 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`.

```
HImage  Image("file_xyz");
HImage  TT   = Image.TextureLaws("ee",2,5);
HImage  Mean = TT.MeanImage(71,71);
HRegion Reg  = Mean.Threshold(30,255);
```

- The mean filter `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 `AddImage` and `MultImage`.

8.5 Eliminating Small Objects

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

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

- The term `segmentation()` is an arbitrary segmentation step 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.

Part III

Programming With HALCON/.NET

Chapter 9

Introducing HALCON/.NET

This chapter introduces you to HALCON/.NET. Chapter 10 on page 61 shows how to use it to create .NET applications, chapter 12 on page 83 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.

Platform Independence

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 Linux, on 32-bit and 64-bit systems. In addition, successful experiments with the .NET Core implementation of HALCON/.NET were performed on Arm-based platforms.

Moreover, not only can you use it on all these platforms, but you can also run an application created on one of them on the others without having to recompile it. This is possible because applications written in .NET languages are stored in a platform-independent intermediate language (IL), which is then compiled by the so-called common language runtime (CLR) into platform-specific code.

.NET Core and .NET Framework

Basically, a .NET application is developed for one or more implementations of .NET like:

- .NET Core (and later versions starting with .NET 5)
- .NET Framework
- Mono

All .NET implementations listed above are supported. The interface for all of these implementations is the same as well as their documented behavior. For users they mostly differ in terms of tooling and platform support. Most of the relevant differences are documented in chapter 10 on page 61.

NuGet packages are provided for .NET Core, and assemblies for .NET Framework.

HDevEngine/.NET

By using the HDevEngine/.NET language bindings, you can execute HDevelop programs and procedures from a .NET application. For more information, please refer to part VI on page 137.

HALCON/.NET XL and HDevEngine/.NET XL

Packages and assemblies are provided for both HALCON and HALCON XL. The packages for .NET Core are available with XL as suffix, for example `MVTec.HalconDotNet` and `MVTec.HalconDotNetXL`. The assemblies for .NET Framework are available with xl as suffix, for example `halcondotnet.dll` and `halcondotnetxl.dll`.

9.1 A First Example

This section demonstrates how to create a simple HALCON application with .NET Core. For a more comprehensive description, see section 10.1 on page 61.

The task is to read an image and compute the number of connected regions in it, as illustrated in figure 9.1 on page 60

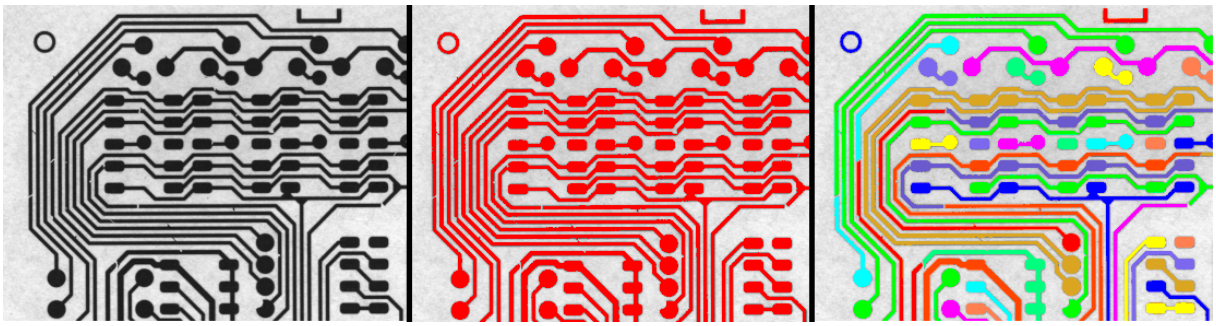


Figure 9.1: Left: Input image of a printed circuit board. Middle: Regions found by `threshold`, colored red. Right: Connected regions, a result of `connection`.

1. Install HALCON 25.11
2. Install the .NET Core SDK for your system.
3. Run the following commands in a shell:

```
dotnet new console -n region_example
cd region_example
dotnet add package MVTec.HalconDotNet -v 25110
```

4. Change the content of `Program.cs` to:

```
using System;

using HalconDotNet;

namespace region_example
{
  class Program
  {
    static void Main(string[] args)
    {
      HImage img = new HImage("pcb");

      HRegion region = img.Threshold(0d, 122d);
      int numRegions = region.Connection().CountObj();

      Console.WriteLine("Number of Regions: " + numRegions);
    }
  }
}
```

5. To run the application, type the following command in the same shell:

```
dotnet run
```

As a result, you should see the following output `'Number of Regions:  43'`.

Chapter 10

Creating Applications With HALCON/.NET

10.1 Creating Applications With HALCON/.NET

For .NET Framework, the examples are given in C#, using Visual Studio 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.

For .NET Core, the examples are given in C# using the .NET Core SDK, usable on both Windows and Linux. At the moment we do not officially support using the .NET Core interface with other languages, C# has been our focus for testing and documentation.

How to create applications under Linux using Mono is described in section 12.2 on page 85. Many of the code examples stem from the example `Matching`, which is provided in C# (`%HALCONEXAMPLES%\c#`), Visual Basic.NET (`%HALCONEXAMPLES%\vb.net`), and managed C++(`%HALCONEXAMPLES%\cpp.net`). An overview of the provided example applications can be found in section 12.1 on page 83.

But before explaining how to create applications, we must take a brief look under the hood of .NET, particularly at the dependency of applications on the .NET Framework.

10.2 .NET Development Environments

Chapter 10 on page 61 emphasized the platform-independence of .NET applications. However, applications still depend on their target .NET runtime environment and for GUI applications, the corresponding implementations for, e.g., Windows Forms or WPF.

There are two main environments, .NET Framework 2.0-4.8 and .NET Core 3.1, with the next versions being named .NET 5 and .NET 6. .NET 6 delivers the final parts of the .NET unification plan that started with .NET 5. Generally, these dependencies are backwards compatible within a branch, e.g., an application targeting .NET Framework 2.0 should run with all newer .NET Framework versions. It should still be possible to open a project file for Visual Studio 2013 with Visual Studio 2019 as well. In addition, the NuGet packages have been tested in applications targeting .NET 5 and .NET 6 and no compatibility or portability issues were found.

Table 10.1 on page 62 indicates the minimum requirements and capabilities of the different HALCON/.NET interface variants.

The examples provided generally target the oldest supported environment. For example, Windows Forms applications target .NET Framework 2.0 and are shipped with Visual Studio 2013 versions that open in all newer versions of Visual Studio.

The non-WPF C# examples also come with makefiles to support building via command line both with .NET Framework under Windows or Mono under Linux. Source files are placed in a separate, shared source directory next to the project or makefile directories. Some dedicated .NET Core examples are provided, which are suitable for building via command line using the .NET Core SDK.

Minimum Development Environment	HALCON/.NET Interface	Minimum Runtime Environment	Windows Forms	WPF	Linux Support
Visual Studio 2013	`%HALCONROOT%\bin\dotnet20`	.NET Framework 2.0	Yes	No	Mono
Visual Studio 2013 [a]	`%HALCONROOT%\bin\dotnet35`	.NET Framework 3.5	Yes	Yes	Mono (no WPF)
.NET Core 3.1 SDK (Windows)[b]	MVTec.HalconDotNet (NuGet Package) MVTec.HalconDotNet-Windows (NuGet Package)	.NET Standard 2.0 .NET Core 3.1	No Yes	No Yes	Yes No
.NET Core 3.1 SDK (Linux)	MVTec.HalconDotNet (NuGet Package)	.NET Standard 2.0	No	No	Yes

[a] WPF support was still rudimentary in Visual Studio 2008 with limited designer support.

[b] If you want to use Visual Studio you will need at least version 2019.

Table 10.1: Properties of different HALCON/.NET interface variants.

10.3 Adding HALCON/.NET to an Application

10.3.1 Adding a Package Reference HALCON/.NET to a .NET Core Application

To leverage the cross-platform nature of .NET Core, the interface is offered in two package variants. Both of them contain the complete HALCON/.NET and HDevEngine/.NET language bindings.

MVTec.HalconDotNet

This package targets .NET Standard 2.0 and can be used on all architectures supported by HALCON.

MVTec.HalconDotNet-Windows

This package targets .NET Core 3.1 and additionally includes Windows Forms and WPF controls for integrating HALCON windows into GUI applications. This package can only be used on Windows since other .NET Core 3.1 implementations do not offer Windows Forms or WPF support.

Use the following command to add package references to your project:

```
dotnet add package <PACKAGE_NAME> -v 25110
```

While NuGet uses SemVer to version packages, HALCON versions are not SemVer compatible. To avoid unnecessary conflicts, .NET Core packages receive a SemVer-compatible version by combining the major, minor, and revision into a new version, e.g.:

- HALCON Steady 25.05.1 → 25051.0.0
- HALCON Progress 25.05.0 → 25050.0.0
- Current HALCON version: 25.11.0 → 25110.0.0

HALCON/.NET is not a self-contained package but an interface to the native HALCON library. Therefore, the package version used by your project has to match the installed HALCON version exactly.

Even for maintenance releases, which are backwards compatible at application level, mixing of binaries from different releases is not recommended because the internal communication between HALCON/.NET and the native HALCON library is **not** guaranteed to be always compatible. Therefore, the revision of the HALCON release is also part of the SemVer major version. The SemVer minor and patch version are reserved for fully backwards-compatible intermediate releases of the package itself. See section 10.5.0.3 on page 65 for updating dependencies.

To use the packages offline, download them from the NuGet website. Consider adding a `nuget.config` file to your project. This helps to source it reliably from a file system.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <packageSources>
     <add key="LocalHalconPackages" value="path/to/package" />
  </packageSources>
</configuration>
```

When developing an application with HALCON XL, select the following XL packages:

- MVTec.HalconDotNetXL
- MVTec.HalconDotNetXL-Windows

For upgrading HALCON/.NET in relation to a HALCON upgrade, see section 10.5.0.3 on page 65.

Manage Nuget Packages

When working in Visual Studio, you can also add the `nuget package reference` to the project, by using the `nuget package manager` from the `solution explorer`. When using the command line, make sure you choose the matching package version for your installed HALCON version.

10.3.2 Creating a WPF Application With Visualization in Visual Studio

The previous section describes, how to create a console application. To create a WPF Application in Visual Studio, that uses a HALCON window for visualization, follow these steps:

1. Use the common Visual Studio mechanism for creating a WPF application.
2. Add a package reference as described in the previous section. The needed package is MVTec.HalconDotNet-Windows.
3. Edit the XAML file.
   (a) Add the HALCON namespace reference.
   ```
   xmlns:ha="http://schemas.mvtec.com/halcondotnet
   ```
   (b) Add the `HSmartWindowControlWPF` element.
   ```
   ha:HSmartWindowControlWPF Name="xxx"  Margin="xxx"
   ```

After this change, the XAML file should look something like this:

```
<Window x:Class="WpfApp2.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:ha="http://schemas.mvtec.com/halcondotnet"
        xmlns:local="clr-namespace:WpfApp2"
        mc:Ignorable="d"
        Title="MainWindow" Height="513" Width="862">
<Grid Margin="0,0,-433,-146">
<Button Content="Button" HorizontalAlignment="Left" Height="5"
Margin="178,121,0,0" VerticalAlignment="Top" Width="7"/>
<Button Name="MyButton" Content="My Button&#xD;&#xA;&#xD;&#xA;"
HorizontalAlignment="Left" Height="84" Margin="156,121,0,0"
VerticalAlignment="Top" Width="151" Click="Button_Click"/>
<ha:HSmartWindowControlWPF Name="MyControl"  Margin="399,55,313,55" />
</Grid>
</Window>
```

10.3.3 Adding HALCON/.NET to a .NET Framework Application

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

For the first application, customize Visual Studio's toolbox (see section 10.3.3.1 on page 64).

For each application, add a reference to HALCON/.NET (section 10.3.3.2 on page 64) and specify the namespace (section 11.0.1 on page 67).

10.3.3.1 Customizing Visual Studio's Toolbox

The HALCON/.NET assembly provides not only a class library but also one control: `HSmartWindowControl` (or `HSmartWindowControlWPF`), which contains a HALCON graphics window for visualizing images and results.

The older control `HWindowControl` is still available for backwards compatibility but it is strongly advised against using this control for new projects (see section 11.6 on page 79 for information about the differences).

You can add these controls to Visual Studio's toolbox by performing the following steps. Note that the exact menu names slightly differ in different versions of Visual Studio:

1. Right-click the toolbox and select `Choose Items` (`Customize Toolbox` in previous versions of Visual Studio). This will open a dialog displaying all available .NET Framework components in a tab.
2. Click `Browse`, navigate to the directory `%HALCONROOT%\bin\dotnet20` or `%HALCONROOT%\bin\dotnet35` and select `halcondotnet.dll`.
3. The icon of `HSmartWindowControl` and the older `HWindowControl` appear in the toolbox.

HALCON XL applications:
Please note that you should **use HALCON XL only when you need its features**.

When developing an application with HALCON XL, select `halcondotnetxl.dll` instead of `halcondotnet.dll`. In the toolbox, the control appears with the same name but with a different icon. You can add both HALCON versions to the toolbox but only one of them to an application.

10.3.3.2 Adding a Reference to HALCON/.NET

In many applications, you will use at least one instance of `HSmartWindowControl` to visualize results. By adding the control to the form (as described in section 11.7 on page 80), you automatically create a reference to the assembly `halcondotnet.dll`.

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

1. Right-click `References` in the Solution Explorer and select `Add Reference`.
2. Click `Browse`, navigate to the subdirectory `%HALCONROOT%\bin\dotnet20` or `%HALCONROOT%\bin\dotnet35` and select the assembly `halcondotnet.dll`.

HALCON XL applications:
Please note that you should **use HALCON XL only when you need its features**.

When developing an application with HALCON XL, select `halcondotnetxl.dll` instead of `halcondotnet.dll`. If you already added a reference to the HALCON version, simply delete this reference and add one to `halcondotnetxl.dll`.

10.4 Deploying an Application

.NET applications typically include all .NET package and assembly dependencies in their build output, hence deployment is simply a copy operation. Depending on whether the build type is self-contained or not, the appropriate .NET runtime must be installed on the destination machine as well.

However, native DLLs are resolved at runtime from the system environment. Therefore, a matching HALCON version and license must be installed on the destination computer, and the environment variables `HALCONROOT` and `HALCONARCH` must be set correctly (see the Installation Guide,section A.4 on page 47).

10.5 Using a Newer HALCON/.NET Release

10.5.0.3 .NET Core

.NET Core projects resolve a specific version the moment you add the package. If you want to update to a newer HALCON version, for example from 25.05. to 25.11.0, you must change the version specified in your project.

Note that 20.11.2 is a breaking change compared to 20.11.1, which can also be seen in the HALCON SemVer major version change from 20111 to 20112.

For example from 25.05. to 25.11.0, change your project `.csproj` from:

```
<PackageReference Include="MVTec.HalconDotNet" Version="2505.0.0" />
```

to:

```
<PackageReference Include="MVTec.HalconDotNet" Version="25110.0.0" />
```

10.5.0.4 .NET Framework

`halcondotnet.dll` is not a self-contained library but an interface to the native HALCON library. Therefore, the library version used by your project has to match the installed HALCON version exactly.

Even for maintenance releases, which are backwards compatible at application level, mixing of binaries from different releases is not recommended because the internal communication between HALCON/.NET and the native HALCON library is **not** guaranteed to be always compatible.

Applications that use HALCON/.NET have local copies of the corresponding assemblies. After installing a newer release of HALCON, these applications would therefore still use their old HALCON assemblies. To use the new `halcondotnet.dll` re-build your project, Visual Studio-> Solution -> Rebuild Solution.

If you want to replace the `halcondotnet.dll` without recompiling the application, you need to both manually replace the `halcondotnet.dll` with the newly installed one, and update the application's `app.config`, to inform the executable of the new `halcondotnet.dll`. Do this by copying either `bin\dotnet20\app.config` or `bin\dotnet35\app.config` into the directory containing the `halcondotnet.dll`, and rename it to `<application_name>.exe.config`.

Chapter 11

HALCON/.NET Interface

11.0.1 Specifying the Namespace

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

```
using HalconDotNet;
```

Visual Basic.NET applications: The corresponding Visual Basic.NET code is (for example, see `MatchingForm.vb`):

```
Imports HalconDotNet
```

11.1 Using HALCON/.NET Classes

In HALCON/.NET, you call HALCON operators via instances of classes. The following code grabs the first image of an image sequence and displays it in the graphics window of `HSmartWindowControl`:

```
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.GetImagePointer1(out ImgType, out imgWidth, out imgHeight);
window.SetPart(0, 0, imgHeight - 1, imgWidth - 1);
img.DispObj(window);
```

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

Below, we take a closer look at:

- how to call operators via HALCON/.NET's classes (section 11.1.2 on page 68)
- construction, initialization, and destruction of class instances (section 11.1.3 on page 69)
- overloads of operator calls (section 11.1.4 on page 71)

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

11.1.1 Online Help

The main source of information about HALCON/.NET operators and classes is the reference manual, which is available as HTML and PDF version (note that the latter is only provided in HDevelop syntax). Under Windows, you can open both versions via the Start Menu. Under Linux, open `index.html` in the directory `$HALCONROOT/doc/html/reference/operators`, and `reference_hdevelop.pdf` in the directory `$HALCONROOT/doc/pdf/reference`, respectively. You can access them also via HDevelop's Help Browser.

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 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.

11.1.2 Calling HALCON Operators

Via which classes you can call a HALCON operator is listed in the reference manual. Figure 11.1 on page 68 shows the corresponding part of the description of the operator `GrabImage`:

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

void `HImage.GrabImage` (`HFramegrabber` acqHandle)

`HImage` `HFramegrabber.GrabImage` ()

`image` (output_object) ..... image ⇝ *`HImage`* (byte / uint2)

`acqHandle` (input_control) ..... framegrabber ⇝ *`HFramegrabber`* / *`HTuple`* *(IntPtr)*

Figure 11.1: 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 12.3 on page 86).

For normal applications, we recommend calling operators via the other classes, in the example `HImage` and `HFramegrabber` as in the following code example:

```
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:

```
double            s1, s2;

s1 = HSystem.CountSeconds();
```

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

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

Figure 11.2: The head of the reference manual entry for `CountSeconds`.

11.1.3 From Declaration to 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:

```
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:

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

You can also **initialize** a variable when declaring it:

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

Note that you can check the initialization state of a class instance with the method `IsInitialized`.

11.1.3.1 Constructors

In contrast, the following declaration calls the "empty" constructor of the class `HImage`, which creates an unitialized 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.

11.1.3.2 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 (for example, see 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)

void `HFramegrabber.CloseFramegrabber` ()

Figure 11.3: 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.

Do not call `Close` or `Clear` operators via `HOperatorSet` when using the normal classes like `HFramegrabber`.

11.1.3.3 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. Even more important is that, to the garbage collector, HALCON's iconic variables (images, regions, ...) seem to be rather "small", because they only contain a reference to the (in many cases rather large) iconic objects in the database. Thus, the garbage collector may not free such variables even if they are not used anymore.

Therefore, you might need to force the removal of (unused) objects. There are two ways to do this:

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

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

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

```
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.

`HTuple` instances that contain handles also need to be disposed if the referenced resource is to be released in a deterministic way.

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.

11.1.4 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`:

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

Note that **operator overloads are not available in Visual Basic.NET.**

The following tables list the currently available operator overloads.

Operator overloads for `HImage`

-	(unary)	inverts an image
+	(image)	adds two images
-	(image)	subtracts image2 from image1
*	(image)	multiplies two images
+	(scalar)	adds a constant gray value offset
-	(scalar)	subtracts a constant gray value offset
*	(scalar)	scales an image by the specified factor
/	(scalar)	scales an image by the specified divisor
>=	(image)	segments an image using dynamic threshold
<=	(image)	segments an image using dynamic threshold
>=	(scalar)	segments an image using constant threshold
<=	(scalar)	segments an image using constant threshold
&	(region)	reduces the domain of an image

Operator overloads for `HRegion`

&	(region)	returns the intersection of regions
|	(region)	returns the union of regions
/	(region)	returns the difference of regions
!	(unary)	returns the complement of the region (may be infinite)
&	(image)	returns the intersection of the region and the image domain
+	(region)	returns the Minkowski addition of regions
-	(region)	returns the Minkowski subtraction of regions
+	(scalar)	dilates the region by the specified radius
-	(scalar)	erodes the region by the specified radius
+	(point)	translates the region
*	(scalar)	zooms the region
-	(unary)	transposes the region

Operator overloads for `HFunction1D`

+	(scalar)	adds a constant offset to the function's Y values
-	(scalar)	subtracts a constant offset from the function's Y values
-	(unary)	negates the Y values of the function
*	(scalar)	scales the function's Y values
/	(scalar)	scales the function's Y values
*	(function)	composes two functions (not a point-wise multiplication)
!	(unary)	calculates the inverse of the function

11.2 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.

The following sections provide more detailed information about

- how to find out whether an operator can be called in tuple mode (section 11.2.1 on page 72)
- tuples of iconic objects (section 11.2.2 on page 73)
- tuple of control values (section 11.2.3 on page 73)

11.2.1 Calling HALCON Operators with Single or Multiple Values

You can check whether an operator also works with tuples in the reference manual. Below, e.g., we show 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.

static void `HOperatorSet.AreaCenter` (`HObject` regions, out `HTuple` area, out `HTuple` row, out `HTuple` column)

`HTuple HRegion.AreaCenter` (out `HTuple` row, out `HTuple` column)

`int HRegion.AreaCenter` (out double row, out double column)

`regions` (input_object) ..... region(-array) ⇝ *`HRegion`*

`area` (output_control) ..... integer(-array) ⇝ *`HTuple` (int / long)*

`row` (output_control) ..... point.y(-array) ⇝ *`HTuple` (double)*

`column` (output_control) ..... point.x(-array) ⇝ *`HTuple` (double)*

static void `HOperatorSet.GenRectangle1` (out `HObject` rectangle, `HTuple` row1, `HTuple` column1, `HTuple` row2, `HTuple` column2)

public `HRegion` (`HTuple` row1, `HTuple` column1, `HTuple` row2, `HTuple` column2)

public `HRegion` (double row1, double column1, double row2, double column2)

void `HRegion.GenRectangle1` (`HTuple` row1, `HTuple` column1, `HTuple` row2, `HTuple` column2)

void `HRegion.GenRectangle1` (double row1, double column1, double row2, double column2)

Parameter Broadcasting

`rectangle` (output_object) ..... region(-array) ⇝ *`HRegion`*

`row1` (input_control) ..... rectangle.origin.y(-array) ⇝ *`HTuple` (double / int / long)*

`column1` (input_control) ..... rectangle.origin.x(-array) ⇝ *`HTuple` (double / int / long)*

`row2` (input_control) ..... rectangle.corner.y(-array) ⇝ *`HTuple` (double / int / long)*

`column2` (input_control) ..... rectangle.corner.x(-array) ⇝ *`HTuple` (double / int / long)*

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):

```
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:

```
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 11.2.2 on page 73) and control tuples (section 11.2.3 on page 73).

11.2.2 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 the operator `CountObj`

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

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

```
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`:

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

11.2.3 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`, `string`, and `HHandle`. They can also contain a mixture of element types.

The following sections describe

- how to access tuple elements (section 11.2.3.1 on page 73)
- how to create tuples (section 11.2.3.2 on page 74)
- the automatic cast methods and how to resolve ambiguities caused by the casts (section 11.2.3.3 on page 74)
- HALCON operators for processing tuples (section 11.2.3.4 on page 75)
- proved overloads for arithmetic operations (section 11.2.3.5 on page 76)

11.2.3.1 Accessing Tuple Elements

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

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

You can access tuple elements with the operator `[]`:

```
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.

11.2.3.2 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:

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

In contrast, the following line creates a `double` tuple:

```
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:

```
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`:

```
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`:

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

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

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

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

```
Tuple3[2] = 3;
```

11.2.3.3 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

```
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:

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

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

```
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:

```
private double    row, column;
HTuple            angleCheck;
HHomMat2D         matrix = new HHomMat2D();
matrix.VectorAngleToRigid(new HTuple(row), new HTuple(column), new HTuple(0.0),
      result.GetGenericShapeModelResult(0, "row"),
      result.GetGenericShapeModelResult(0, "column"),
      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):

```
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);
```

11.2.3.4 HALCON Operators for Processing Tuples

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 HDevelopUser’s Guide in chapter 8 on page

247. 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`.

11.2.3.5 Operator Overloads

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

11.3 Working With Vectors

HALCON/.NET provides the class `HVector` for the use of variables of type 'vector' of the HDevelop language, i.e., containers that can hold an arbitrary number of elements of the identical data type (i.e., tuple, iconic object, or vector) and dimension. The type of a vector, i.e., its dimension and the type of its elements is defined when initializing the vector instance and cannot be changed during its lifetime. A one-dimensional vector may be a vector of tuples or a vector of iconic objects. A two-dimensional vector may be a vector of vectors of tuples or a vector of vector of iconic objects, and so on. Instances of vectors can be created from the following derived classes:

- Class `HObjectVector` for handling vectors of iconic objects
- Class `HTupleVector` for handling vectors of tuples

In the following some basic information about the use of vectors in HALCON/.NET is given when using C#.

Construction of Vectors

As already mentioned, an instance of a vector can only be created from `HObjectVector` or `HTupleVector`.

```
// Create a vector of iconic objects
HObjectVector   vectorObj;

// Create a vector of tuples
HTupleVector    vectorTup;
```

The vector type cannot be changed after its construction. Thus, a tuple cannot be assigned to a vector of iconic objects and vice versa.

You may also create a multi-dimensional vector, i.e., a vector of vectors and so on, by specifying the number of dimensions in brackets. However, the dimension of a vector has to remain constant within the program and cannot be changed. The following code line describes how to create a vector of two dimensions, i.e., a vector of vectors of tuples.

```
HObjectVector vectorObjMulti = new HObjectVector(2);
```

Note that the vectors created by these calls are still empty. How to access and set vector elements is described below.

Accessing and Setting Vector Elements

When setting or accessing an element of a vector you have to differ between vectors of type `HObjectVector` and `HTupleVector`. Elements of vectors of iconic objects can be accessed with `.O` whereas elements of vectors of tuples can be accessed with `.T`.

```
// Access an element of a one-dimensional HObjectVector
vectorObj[0].O;

// Access an element of a one-dimensional HTupleVector
vectorTup[0].T;
```

The specified index in square brackets defines the element to be accessed. If a subelement of a multi-dimensional vector is to be accessed, you have to use the indices of the corresponding subvector and its subelement instead.

```
// Access a subelement of a two-dimensional HObjectVector
vectorObjMulti[0][1].O;
```

When setting a vector element the expression for accessing a vector element is needed as reference to the `HObject` or `HTuple`, respectively. The value to be set must be specified on the right side of the assignment.

```
// Set an element of a one-dimensional HObjectVector
HImage image = new HImage("Image1");
vectorObj[0].O = image;

// Set an element of a HTupleVector
HTuple tuple = new HTuple(1,2);
vectorTup[1].T = tuple;
```

In the example code the image is copied and set as the first element of `vectorObj`. The tuple is also copied but set as the second vector element of `vectorTup`.

Setting a subelement of a multi-dimensional vector can be done analogously to setting an element in a one-dimensional vector when using the corresponding indices of the subvector(s) and its subelement.

If a non-existing vector element is set or accessed, the vector is automatically enlarged and filled with empty elements if necessary.

Destruction of Vectors

If a `HObjectVector` or `HTupleVector` is not needed anymore for further processing its contents should be cleared with `.Dispose()`.

```
vectorObj.Dispose();
```

Additional Information

In addition to accessing and setting the elements of vectors, `HObjectVector` and `HTupleVector` provide further functionalities for the use of HALCON vectors such as inserting and removing vector elements or concatenation of vectors. For more details on the provided vector functionality you may use the automatic context help in Visual Studio, IntelliSense.

11.4 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 code in the caption and the HALCON error message:

```
HImage Image;

try
{
  Image = new HImage("unknown");
}
catch (HOperatorException exception)
{
  MessageBox.Show(exception.Message, "HALCON error # " + exception.GetErrorCode());
}
```

All HALCON error codes and their corresponding error messages are summarized in the Extension Package Programmer's Manual, appendix A on page 105.

The `HSmartWindowControl` and `HSmartWindowControlWPF` provide the event `HErrorNotify`. This event allows the user to react to errors that take place internally within the control, but can have external causes, like for example an unplugged dongle or a missing license file.

11.5 Visualization

Applications can use an instance of `HSmartWindowControl` to display results. However, this is only available for .NET Framework and the MVTec.HalconDotNet-Windows .NET Core packages. How to configure this control is described in section 11.7 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 `HSmartWindowControl`:

```
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.

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 `HSmartWindowControl`, 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 21 on page 223. 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`.

11.6 Window Controls for Visualization

Note that at the time of writing Visual Studio Designer support for .NET Core is not mature yet. There are some workarounds and a preview of the .NET Core Windows Forms Designer. Also, you have the option to manually edit the relevant files. The following documentation is explained using Visual Studio and .NET Framework, the same application functionality is possible in .NET Core.

Depending on which graphical subsystem a project is based on, a specific window control is available in Visual Studio's toolbox.

- `HSmartWindowControl` (Windows Forms), and
- `HSmartWindowControlWPF` (WPF).

See section 12.2.4 on page 86 for information about other GUI libraries.

The smart window control provides several advantages over `HWindowControl`:

- It is used like any other control (e.g., it can be embedded in `TabControls` or `ScrollViewers`, or overlaid with other controls).
- Predefined mouse interaction is provided (moving of the window contents and zooming using the mouse wheel). The view can be reset by double-clicking the window.
- The control automatically rescales without flickering.

In contrast to `HSmartWindowControlWPF`, a callback is required for `HSmartWindowControl` in order to enable zooming using the mouse wheel:

```
private void WindowControl_Load(object sender, EventArgs e)
{
  this.MouseWheel += my_MouseWheel;
}
```

In addition, you need to transform the mouse coordinates, so that they are relative to the upper left corner of the `HSmartWindowControl`.

```
private void my_MouseWheel(object sender, MouseEventArgs e)
{
    Point pt = windowControl.Location;
    MouseEventArgs newe = new MouseEventArgs(e.Button, e.Clicks,
                                             e.X - pt.X, e.Y - pt.Y, e.Delta);
    windowControl.HSmartWindowControl_MouseWheel(sender, newe);
}
```

Using the smart window control, the following events are triggered and can be reacted to:

- `Click`
- `GotFocus`, `LostFocus`
- `MouseEnter`, `MouseLeave`, `MouseHover`
- `Resize`, `SizeChanged`
- `KeyDown`, `KeyUp`, `KeyPress`

Not all operators can be used with the smart window control. The following operators are *not* supported:

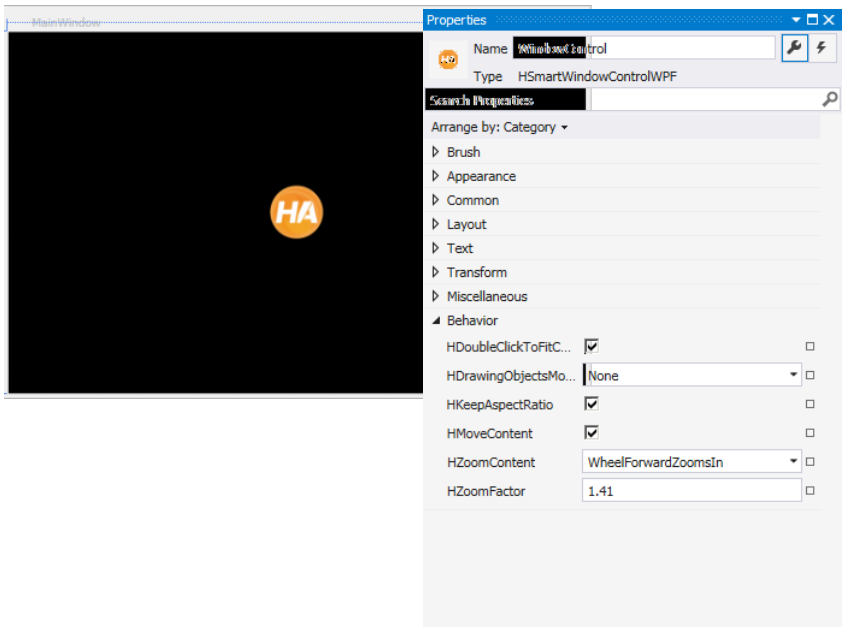


Figure 11.4: Adapting the properties of `HSmartWindowControlWPF`.

- `draw_nurbs`, `draw_nurbs_mod`, `draw_nurbs_interp`, `draw_nurbs_interp_mod`
- `drag_region1`, `drag_region2`, `drag_region3`

Instead, use drawing objects.

In the following sections, the term `HSmartWindowControl` will be used for simplicity. Read it as `HSmartWindowControlWPF` if your project is based on WPF. The same applies to the older `HWindowControl` correspondingly.

11.7 Customizing `HSmartWindowControl` for the Visualization

In most applications you want to visualize at least some results. Then, you start by adding `HSmartWindowControl` to the form by double-clicking the corresponding icon in the toolbar. An empty (black) window appears (see figure 11.4 on page 80).

HALCON XL applications:
Please note that you should **use HALCON XL only when you need its features**.

If you already added the HALCON version of the control, but now want to use HALCON XL, simply delete the reference to `halcondotnet.dll` in the Solution Explorer and add a reference to `halcondotnetxl.dll` instead.

If you want to fit an image automatically without the need to double click, you can use the operator `SetPart` to adjust the part to the size of the last image displayed:

```
img.DispObj(window);
```

The properties specific to this control are listed below in alphabetical order. They can be adapted in the Properties window (see figure 11.4 on page 80). Note that some properties are only available for `HSmartWindowControlWPF` elements. All properties of the `HSmartWindowControlWPF` support data binding.

Do not modify the “Brush” properties in Visual Studio. They are disabled for `HSmartWindowControlWPF` to prevent undesired side-effects.

`HColor`
: Specifies the color of HRegion and HXLDCont objects.

HColored
: Specifies the colors for displaying multiple HRegion or HXLDCont objects in different colors.

HDisableAutoResize
: If set to true, images are not automatically scaled when they are displayed.

HDisplayCurrentObject
: Displays the assigned HImage or HObject.

HDoubleClickToFitContent
: If set to true (the default), double clicking resizes the content of the HSmartWindowControl to fit the size of the control. If the HKeepAspectRatio is also set to true, the contents are rescaled so that the aspect ratio is maintained.

HDraw
: Specifies the fill mode of HRegion objects. If HDraw is set to 'fill', output regions are filled, if set to 'margin', only contours are displayed.

HDrawingObjectsModifier
: Specifies the modifier key to interact with drawing objects. If a modifier key is set, the user can only interact with drawing objects while keeping the modifier key pressed. This is especially useful when interacting with XLD drawing objects. By default, it is set to None. Other possible values are Alt, Ctrl, or Shift.

HFont
: Specifies the font for displaying messages in the HSmartWindowControlWPF.

HImagePart
: Specifies the image part of the corresponding HALCON window. Note that the part is specified with the values X, Y, Width, and Height, whereas the corresponding operator SetPart expects the four corner points. 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 11.5 on page 78 for more information about actually visualizing results.

HKeepAspectRatio
: If set to true (the default), the content of the HSmartWindowControl keeps its aspect ratio when the control is resized or zoomed. The aspect ratio is the quotient Width/Height set at design time with HImagePart.

HLineStyle
: Specifies the contour pattern of HRegion and HXLDCont objects.

HLineWidth
: Specifies the contour thickness of HRegion and HXLDCont objects.

HMoveContent
: If set to true (the default), the contents of the HALCON window can be dragged using the mouse.

HZoomContent
: Specifies the behavior of the mouse wheel. If set to WheelForwardZoomsIn (the default), the contents of the HALCON window is zoomed in and out when moving the mouse wheel forwards and backwards, respectively. Setting the property to WheelBackwardZoomsIn reverses the behavior. If set to Off, zooming using the mouse wheel is disabled.

HZoomFactor
: Specifies the step size when zooming with the mouse wheel. The default is 1.41. Values must be greater than 1 and less or equal to 100. A higher value leads to faster zooming.

The HSmartWindowControlWPF can be used to implement the Model-View-Viewmodel (MVVM) pattern, as all properties of the control support data binding. By adding objects to the Items collection or setting the ItemsSource property you can specify what the control should display, also in pure XAML code. The following example illustrates how to show an image, color a region in "magenta", and display a text message:

```
<ha:HSmartWindowControlWPF HDraw="fill">
  <!--Iconic items can be added using HIconicDisplayObjectWPFs-->
  <ha:HIconicDisplayObjectWPF IconicObject="{Binding DisplayImage}"/>
  <!--Also with individual drawing properties-->
  <ha:HIconicDisplayObjectWPF IconicObject="{Binding DisplayRegion}"
  HDraw="margin" HColor="magenta"/>
  <!--Messages can be displayed using HMessageDisplayObjectWPFs-->
  <ha:HMessageDisplayObjectWPF HMessageText="{Binding ImageName}"/>
</ha:HSmartWindowControlWPF>
```

Note that `DisplayImage`, `DisplayRegion` and `ImageName` have to exist in the DataContext of the `HSmartWindowControlWPF`.

Chapter 12

Additional Information

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

- Section 12.1 on page 83 gives an overview of the available example applications.
- Section 12.2 on page 85 explains how to use HALCON/.NET applications under Linux using Mono.
- Section 12.3 on page 86 shows how to use HDevelop programs or procedures in your .NET application.
- Section 12.4 on page 87 contains information to keep in mind if your environment allows remote access.

12.1 Provided Examples

The following sections briefly describe the provided example applications for

- C# (section 12.1.1 on page 83)
- Visual Basic.NET (section 12.1.2 on page 84)
- (managed) C++ (section 12.1.3 on page 84)

All paths are relative to `%HALCONEXAMPLES%`.

12.1.1 C#

- `c#\IACallback` (Visual Studio 2013 or higher)

  Image acquisition example to demonstrate how to use image acquisition callbacks.

- `c#\DrawingObjects` (Visual Studio 2013 or higher)

  Use event-based interaction with drawing objects within a WinForms dialog.

- `c#\DrawingObjectsWPF` (Visual Studio 2013 or higher)

  Use event-based interaction with drawing objects within a WPF application and integrate exported C# Code.

- `c#\Matching` (Visual Studio 2013 or higher, Mono)

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

- `c#\MatchingAvalonia` (Visual Studio 2019 or higher, .NET Core)

  Matching example to show how a HALCON window can be used as part of an Avalonia application.

- `c#\MatchingWPF` (Visual Studio 2013)

  Matching example to demonstrate the use of HALCON in a WPF application using Visual Studio 2013 or higher.

- c#\MultiThreading (Visual Studio 2013 or higher, Mono)

  Use HALCON/.NET with multiple threads for image acquisition, processing (2D data code reading, HDataCode2D), and image display.

- c#\SerializedItem (Visual Studio 2013 or higher, Mono)

  Use serialization of HALCON objects and tuples in the C# interface.

- hdevengine\c#\ExecProgram (Visual Studio 2013 or higher, Mono)

  Execute an HDevelop program for fin detection using HDevEngine

- hdevengine\c#\ExecExtProc (Visual Studio 2013 or higher, Mono)

  Execute an external HDevelop procedure for fin detection using HDevEngine

- hdevengine\c#\ExecProcedures (Visual Studio 2013 or higher, Mono)

  Execute local and external HDevelop procedures for fin detection using HDevEngine

- hdevengine\c#\ErrorHandling (Visual Studio 2013 or higher, Mono)

  Handle HDevEngine exceptions

- hdevengine\c#\MultiThreading (Visual Studio 2013 or higher, Mono)

  Executing an HDevelop procedure in parallel by two threads using HDevEngine/.NET

- hdevengine\c#\MultiThreadingTwoWindows (Visual Studio 2013 or higher, Mono)

  Executing different HDevelop procedures in parallel by two threads using HDevEngine/.NET

12.1.2 Visual Basic.NET

- vb.net\MatchingWPF (Visual Studio 2013 or higher)

  Matching example to demonstrate the use of HALCON in a WPF application using Visual Studio 2013 or higher.

- hdevengine\vb.net\ExecProgram (Visual Studio 2013 or higher)

  Execute an HDevelop program for fin detection using HDevEngine

- hdevengine\vb.net\ExecExtProc (Visual Studio 2013 or higher)

  Execute an external HDevelop procedure for fin detection using HDevEngine

- hdevengine\vb.net\ExecProcedures (Visual Studio 2013 or higher)

  Execute local and external HDevelop procedures for fin detection using HDevEngine

- hdevengine\vb.net\ErrorHandling (Visual Studio 2013 or higher)

  Handle HDevEngine exceptions

12.1.3 C++

- cpp.net\Matching (Visual Studio 2013 or higher)

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

- cpp.net\Interoperate (Visual Studio 2013 or higher)

  Demonstrate the use of a HALCON/C++ DLL from within a HALCON/.NET application using Visual Studio 2013 or higher.

12.2 HALCON/.NET Applications under Linux Using Mono

Chapter 10 on page 61 describes in detail how to develop HALCON/.NET applications in general. If you want to create applications under Linux using Mono, see the following sections for additional information:

- restrictions (section 12.2.1 on page 85)
- how to deploy applications created under Windows (section 12.2.2 on page 85)
- how to compile an application with Mono (section 12.2.3 on page 85)
- other GUI libraries (section 12.2.4 on page 86)

If you want to target Linux and do not need Windows Forms, consider using .NET Core.

12.2.1 Restrictions

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

- Mono only supports Windows Forms 2.0.
- `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 `HSmartWindowControl`'s (or `HWindowControl`'s) event `HInitWindow`.

```
private void hWindowControl1_HInitWindow(object sender, System.EventArgs e)
{
  window = hWindowControl1.HalconWindow;
  window.SetDraw("margin");
  window.SetColor("cyan");
}
```

12.2.2 Deploying HALCON/.NET Applications Created under Windows

Because of HALCON/.NET's platform independence, you can copy an application created under Windows to a Linux computer and simply start it there – provided that Mono and HALCON are installed on the destination computer (see section 10.4 on page 64 for more information).

12.2.3 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 Linux (see section 12.1 on page 83 for a list of the examples that support Linux). To start the compilation, simply type

```
gmake
```

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

HALCON XL applications:
Please note that you should **use HALCON XL only when you need its features**.

To create a HALCON/.NET XL application, type

```
gmake XL=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 that explicitly specify the path to the HALCON library (see figure 12.1 on page 86 for an example) by calling

```
<configuration>
  <dllmap dll="halcon"
          target="/opt/halcon/lib/x64-linux/libhalcon.so"/>
</configuration>
```

Figure 12.1: Example for a configuration file with HALCON being installed in the directory `/opt/halcon`.

```
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`.

Note that you can also use xbuild (Mono's implementation of msbuild) with the project files.

12.2.4 Using Other GUI Libraries

In principle, you can also use other GUI libraries instead of Windows Forms or WPF, e.g., Gtk#. However, `HSmartWindowControl` or `HSmartWindowControlWPF` are Windows Forms or WPF elements, respectively, 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`.

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

12.3 Using HDevelop Programs

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

- Execute them directly via HDevEngine (see part VI on page 137 for detailed information).
- Export them into C# or Visual Basic.NET code via the menu item `File` ▷ `Export` (see the HDevelopUser's Guide, section 6.6 on page 69) and integrate the code in your application.

The latter method is described in this section.

12.3.1 Using the Template Application

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 (available for C# and Visual Basic.NET) in the subdirectory `HDevelopTemplate` (or `HDevelopTemplateWPF`, depending on your preferred GUI platform) as follows:

1. Move or copy the exported source code file into subdirectory `source` of the template application.
2. Open the solution file, right-click the current project in the Solution Explorer, and select the menu item `Add Existing Item`. Navigate to the source code file and select `Add As Link`, which is accessed via the arrow on the right side of the `Add` button (see figure 12.2 on page 86).

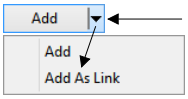


Figure 12.2: Linking existing items to an application.

3. When you run the application, the form depicted in figure 12.3 on page 87 appears. Click `Run` to start the exported HDevelop program.
4. If you did not add the exported code correctly, the error message depicted in figure 12.4 on page 87 appears. In Visual Basic.NET, different error messages appear.

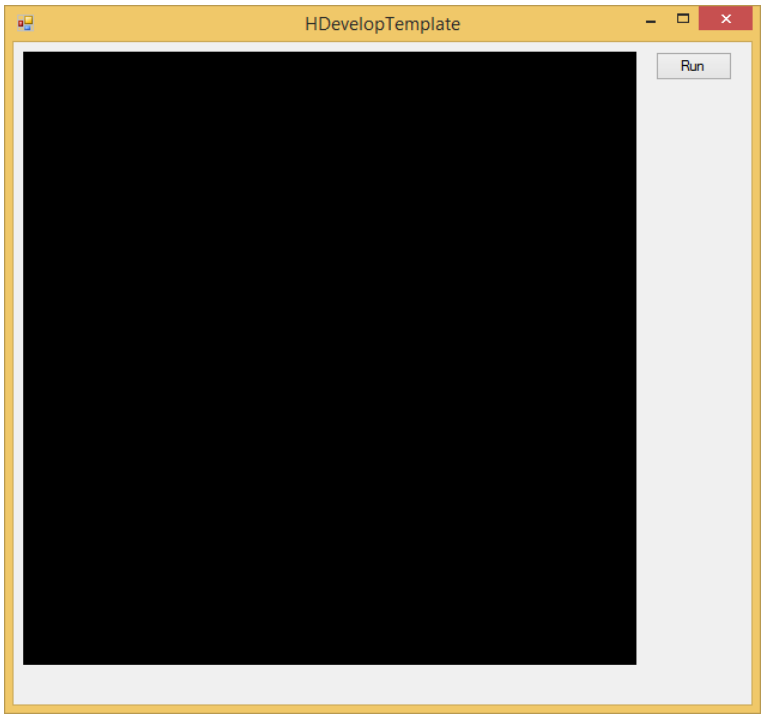


Figure 12.3: The template form for exported code.

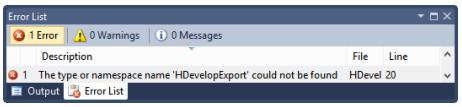


Figure 12.4: Error message upon running a template without exported code.

12.3.2 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`.

12.4 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 that allows remote access, you might wish to explicitly check permissions for code calling within your application or library.

Part IV

Programming With HALCON/Python

Chapter 13

Introducing HALCON/Python

This chapter introduces you to HALCON/Python. Chapter 14 on page 93 shows how to use it to create Python applications, chapter 15 on page 95 contains detailed information.

What is HALCON/Python?

HALCON/Python is a set of native Python language bindings for HALCON. This includes interfaces for operators, HDevEngine and interoperability for third-party libraries like NumPy.

The major design goals for HALCON/Python are simplicity and rapid prototyping.

Platform Independence

HALCON/Python is officially supported for CPython, the reference implementation of the Python programming language. We test the interface on x64-win64and x64-linux. In addition, successful experiments were performed on ARM with CPython.

Other Python implementations should work as long as they are appropriate implementations of the Python standard.

What's more, not only can you use it on all these platforms, but you can run an application created on one of them on the other ones without having to recompile it. This is possible because applications written in Python are interpreted at runtime, instead of being compiled ahead of time.

HDevEngine/Python

By using the HDevEngine/Python language bindings, you can execute HDevelop programs and procedures from a Python application. For more information, please refer to part VI on page 137.

13.1 A First Example

This section demonstrates how to create a simple HALCON application with Python. For a more comprehensive description, see chapter 14 on page 93.

The task is to read an image and compute the number of connected regions in it, as illustrated in figure 13.1 on page 92

1. Install HALCON 25.11
2. Install Python 3.8 or newer on your system.
3. Setup your Python environment of choice, e.g., using `python -m venv path_to_new_virtual_environment`
4. Run the following commands in a shell:

```
mkdir region_example
cd region_example
pip install mvtec-halcon==25110
```

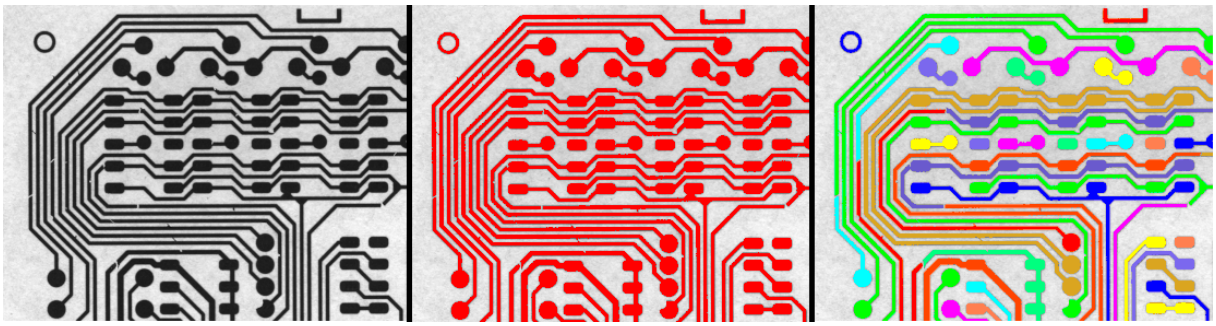


Figure 13.1: Left: Input image of a printed circuit board. Middle: Regions found by `threshold`, colored red. Right: Connected regions, a result of `connection`.

5. Create a file named `program.py` and change the content to:

```
import halcon as ha

if __name__ == '__main__':
    img = ha.read_image('pcb')

    region = ha.threshold(img, 0, 122)
    num_regions = ha.count_obj(ha.connection(region))

    print(f'Number of Regions: {num_regions}')
```

6. To run the application, type the following command in the same shell:

```
python program.py
```

As a result, you should see the following output `'Number of Regions:  43'`.

Chapter 14

Creating Applications With HALCON/Python

14.1 Python Development Environments

The examples are written in Python 3.8. They work at least on the following platforms when using CPython as interpreter and runtime:

1. x64-win64
2. x64-linux

Newer Python versions should work without problems. It is possible that other platforms and alternative Python implementations are functional, of course native HALCON binaries also need to be available on that platform.

14.2 Adding HALCON/Python to an Application

HALCON/Python packages are pip compatible.

Use the following command to add HALCON/Python to your project:

```
pip install mvtec-halcon==25110
```

While PEP 440 and by extension PyPI does not require SemVer, HALCON/Python and other language interface packages follow a SemVer approach to avoid unnecessary conflicts and tooling issues. However, HALCON versions are not SemVer compatible, so Python packages receive a SemVer-compatible version by combining the major, minor, and revision into a new version, e.g.:

- 25.05.1 → 25051.0.0
- 25.05.0 → 25050.0.0
- 25.11.0 → 25110.0.0

HALCON/Python is not a self-contained package but an interface to the native HALCON library. Therefore, the package version used by your project has to match the installed HALCON version exactly.

Even for maintenance releases, which are backwards compatible at application level, mixing of binaries from different releases is not recommended because the internal communication between HALCON/Python and the native HALCON library is **not** guaranteed to be always compatible. Therefore, the revision of the HALCON release is also part of the SemVer major version. The SemVer minor and patch version are reserved for fully backwards-compatible intermediate releases of the package itself. See section 14.4 on page 94 for updating dependencies.

To use the packages offline, download them from the PyPI website.

14.3 Deploying an Application

There is no standard way for assembling Python applications into ready-to-use applications. Approaches range from just copying files to embedding the source code, interpreter and runtime via tools like cx_Freeze.

Note that native DLLs are resolved at runtime from the system environment. Therefore, a matching HALCON version and license must be installed on the destination computer, and the environment variables set correctly (see the Installation Guide,section A.4 on page 47).

14.4 Using a Newer HALCON/Python Release

You can check which HALCON/Python version is installed in your Python environment by running `pip freeze`. If you want to update to a newer HALCON version you must install the new version in your Python environment.

For example, to update to 25.11.0, run this command in a shell inside your project directory:

```
pip install -Iv mvtec-halcon==25110
```

Chapter 15

HALCON/Python Interface

15.1 Module Import

To strike a good balance between ease of use, readability, and traceability we recommend importing the HALCON/Python module like this:

```
import halcon as ha
```

15.2 Using HALCON Operators From HALCON/Python

In HALCON/Python, you call HALCON operators directly as non-member functions, exposed directly through the top level `halcon` module.

The following code grabs the first image of an image sequence using a framegrabber and displays the image in a window:

```
framegrabber = ha.open_framegrabber(
    name='File',
    horizontal_resolution=1,
    vertical_resolution=1,
    image_width=0,
    image_height=0,
    start_row=0,
    start_column=0,
    field='default',
    bits_per_channel=-1,
    color_space='default',
    generic=-1,
    external_trigger='default',
    camera_type='board/board.seq',
    device='default',
    port=1,
    line_in=-1
)

img = ha.grab_image(framegrabber)
width, height = ha.get_image_size_s(img)

window = ha.open_window(
    row=0,
    column=0,
    width=width,
    height=height,
    father_window=0,
    mode='visible',
    machine=''
)
ha.disp_obj(img, window)
```

See the subsequent sections for a breakdown of the example.

15.3 Operators Are Standalone Functions

Notice how all operator calls are standalone functions.

```
ha.open_framegrabber(...)
ha.grab_image(...)
ha.get_image_size_s(...)
ha.open_window(...)
ha.disp_obj(...)
```

15.4 Inputs Are Parameters, Outputs Are Return Values

Another important aspect of HALCON/Python is the split of inputs and outputs. Inputs are function parameters, and outputs are return values.

```
img = ha.grab_image(framegrabber)
width, height = ha.get_image_size_s(img)
```

Single outputs are returned directly, and multiple outputs are returned as Python tuple. This allows clearer reasoning about data transformations, aided by ergonomic features such as destructuring. This can be seen with `get_image_size_s`, which returns two values, both of which can be given names directly.

15.5 HALCON Tuples Are Represented With Native Python Types

Both when passing values into operators as input parameters, and when receiving values as return values from operators, what is represented with HALCON Tuples in HDevelop is a native Python value in HALCON/Python.

Don't confuse HALCON Tuples with Python tuples, they are unconnected concepts.

For example, the return value for `get_image_size_s` is truly of type `int` and not something that represents the HALCON version of integers.

```
img = ha.read_image('pcb')
width, height = ha.get_image_size_s(img)

assert width == 1109
assert type(width) == int

assert height == 871
assert type(height) == int
```

In HALCON/Python, `HTuple` maps to either one of the following types or to a possibly mixed list of them:

- `int`
- `float`
- `str`
- The class `HHandle`

Python's `float` maps to HALCON's real type. While Python's `int` is of arbitrary precision, calling operators with `int` values outside what a signed integer of platform size can represent, e.g.: `ptrdiff_t` in C, will result in an exception.

15.6 HHandle

`HHandle` is a class that represents control handle values.

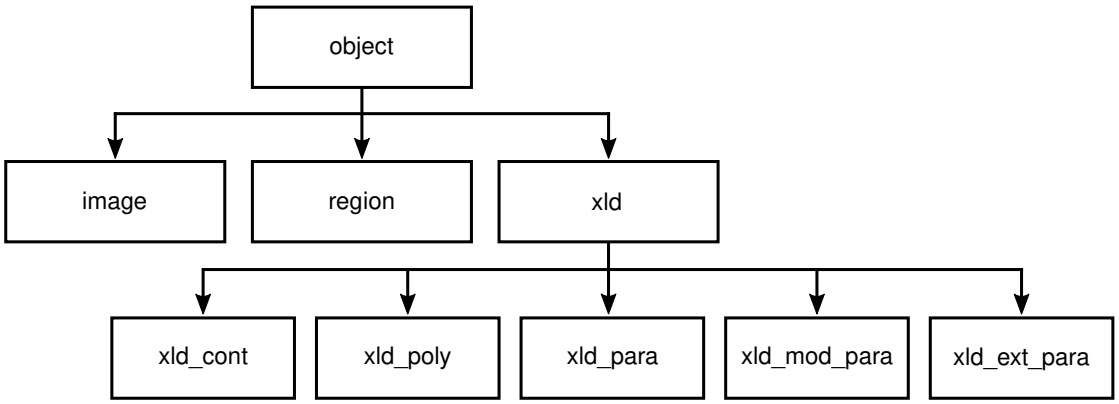
```
camera_model = ha.create_camera_setup_model(num_cameras=4)
assert isinstance(camera_model, ha.HHandle)
```

Here, `create_camera_setup_model` returns a `HHandle` that represents a camera model. This `HHandle` can later be passed to other operators that work with camera models. Other than equality comparison, `HHandle` does not implement any further functionality by itself.

15.7 HObject

`HObject` is a class that represents iconic objects.

In HALCON, iconic objects model a hierarchy:



HALCON/Python does not model this object hierarchy with a class hierarchy, rather it only provides a single class `HObject`.

Iconic objects in HALCON can represent collections of objects, such as a sequence of images, or multiple regions etc. HALCON/Python mirrors this behavior:

```
img = ha.read_image('pcb')
img_seq = ha.concat_obj(img, ha.read_image('fuse'))

width, height = ha.get_image_size(img_seq)
assert width == [1109, 768]
assert height == [871, 576]
```

Here, `img_seq` represents two images, and follow-up operators like `get_image_size` work in a batch fashion accordingly.

`HObject` behaves similarly to lists in Python:

```
assert len(img_seq) == 2

for single_img in img_seq:
    assert ha.get_image_size_s(single_img) > (600, 400)

assert ha.get_image_size_s(img_seq[1]) == (768, 576)
assert ha.get_image_size_s(img_seq[-2]) == (1109, 871)
assert ha.get_image_size(img_seq[:4]) == ([1109, 768], [871, 576])
assert ha.get_image_size(img_seq[1:4]) == ([768], [576])
```

This includes support for `len`, iteration, positive and negative indexing, and efficient slicing.

15.8 HDict

HALCON HDicts can be converted to Python dicts and vice versa with the following functions:

- `as_python_dict_s` – Converts HALCON dict (HDict) to Python dict.
- `as_python_dict` – Converts HALCON dict (HDict) to Python dict.
- `from_python_dict` – Converts Python dict to a HALCON dict (HDict).

A HDict can always be represented as a Python dict. However, not all Python dicts can be represented as HDict. Consult the documentation of `from_python_dict` for more details and examples.

HALCON/Python can convert both single values and lists of values to their HTuple representation, that is, the Python values `345` and `[345]` share the same HTuple representation. As a consequence, once Python values have been converted to their HALCON counterparts, this information is lost. As a consequence, roundtrip converting a Python dict through an HDict and back is a lossy operation. `as_python_dict` represents all HTuples as lists. In addition, `as_python_dict_s` can be used if the user expects that all HTuple values are single values. `as_python_dict_s` represents all HTuples as single values and not as lists. `as_python_dict_s` raises an `HError` if any HTuple has more than one value.

An example usage of the conversion hooks can be found in section 15.9 on page 98.

15.9 HALCON/Python with NumPy

HALCON images can be converted to NumPy arrays and vice versa with the following functions:

- `himage_from_numpy_array` – Converts NumPy array to HALCON object.
- `himage_as_numpy_array` – Converts single HALCON image to NumPy array.

The functions are only usable if the third-party library NumPy package has been installed. Usage example:

```
import halcon as ha
import numpy as np


if __name__ == '__main__':
    # Single-channel image.
    img = ha.read_image('pcb')

    numpy_array = ha.himage_as_numpy_array(img)
    assert (numpy_array[0][:4] == [225, 216, 224, 230]).all()

    img_roundtrip = ha.himage_from_numpy_array(numpy_array)
    assert ha.compare_obj(img, img_roundtrip, 0)

    # Multi-channel image: Supported
    _ = ha.himage_as_numpy_array(ha.read_image('patras'))

    # Image list: Not supported
    img_list = ha.HObject([ha.read_image('pcb'), ha.read_image('pcb')])
    try:
        _ = ha.himage_as_numpy_array(img_list)
        assert False
    except Exception as exc:
        assert isinstance(exc, ha.HTupleConversionError)
```

The dict conversion functions (see section 15.8 on page 98) have customization points that can be used in conjunction with the NumPy conversion functions. For example:

```
import halcon as ha
import numpy as np


if __name__ == '__main__':
    numpy_image = np.array([[5, 6, 7], [8, 2, 6], [4, 4, 5]], dtype=np.int32)
    py_dict = {'val': [4, 6], 'img': numpy_image}
    hdict = ha.from_python_dict(py_dict, unknown_hook=ha.himage_from_numpy_array)
    py_dict_roundtrip = ha.as_python_dict(hdict, hobject_hook=ha.himage_as_numpy_array)

    assert isinstance(py_dict_roundtrip['img'], np.ndarray)
```

15.10 Output Values

Outputs come in three variants:

- Always as single value, e.g.: `read_image`
  In this case, the Python version of the operator will return a single value of the list's element type.
- Always as list, e.g.: `quat_normalize`
  In this case, the Python version of the operator will always return a list.
- Maybe as single value or maybe as list, e.g.: `get_image_size`
  In this case, the Python version of the operator will also return a list. However, in many cases it will be known in advance, e.g.: from the number of input parameters, that only a single value is expected. For convenience, a second variant of the operator with suffix `_s` is provided. This will always return a single value, or raise an exception when the operator does not return a single value.

The difference between operators that always return a list and those that sometimes return a list is rather just conceptual. The advantage lies mainly in the convenience of access for the common case of working with single values.

```
img = ha.read_image('pcb')

width, height = ha.get_image_size_s(img)
assert width == 1109
assert height == 871

width, height = ha.get_image_size(img)
assert width == [1109]
assert height == [871]

img_seq = ha.concat_obj(img, ha.read_image('fuse'))
width, height = ha.get_image_size(img_seq)
assert width == [1109, 768]
assert height == [871, 576]
```

Asking for the size of a single image, and getting a list with a single element is not particularly useful. `_s` versions express the expectation that there should always be exactly one answer, neither zero nor many, explicitly and conveniently.

For information about which operators have such versions, see the HALCON operator reference.

15.11 Error Handling

Just like Python, HALCON/Python handles errors via exceptions. As shown in the example below, HALCON error constants are defined at the module level.

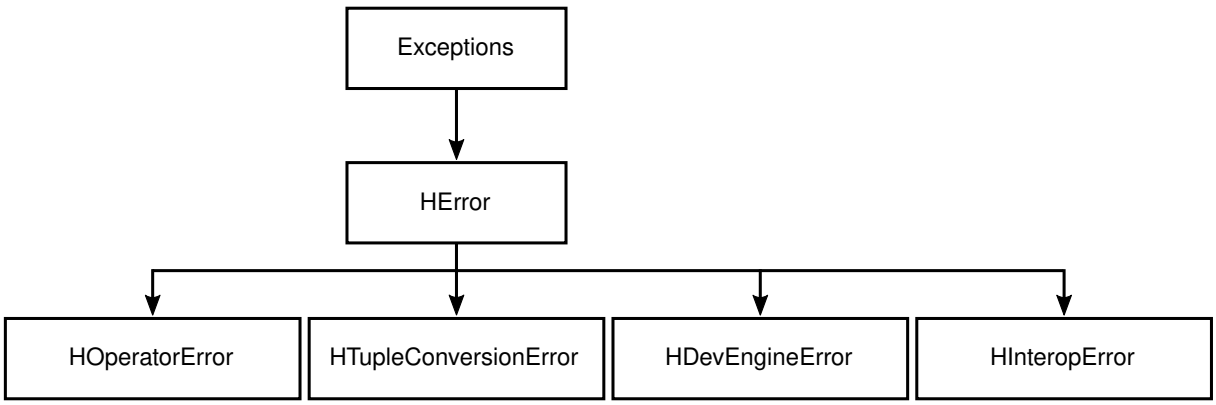
```
def load_custom_pcb_img_or_fallback(custom_pcb_filename):
    try:
        return ha.read_image(custom_pcb_filename)
    except ha.HOperatorError as err:
        if err.error_code != ha.errors.H_ERR_FNF:  # HALCON error code File not found.
            raise err

        return ha.read_image('pcb')

    img = load_custom_pcb_img_or_fallback('does-not-exist')
```

In this example, we want to either load a custom PCB image or fallback to the default PCB image if the image cannot be found. If another error occurs, we want to propagate the original error further up the call stack.

The custom exceptions used by HALCON/Python form an inheritance hierarchy:



It starts with the standard Python `Exception`, from which `HError` inherits, followed by four more specific exceptions that inherit from `HError`. They occur in different situations. Operators and other functions may raise more than one type of exception.

```
class CustomClass(object):
    pass

img = ha.read_image(CustomClass())
```

HALCON/Python does not know how to pass the custom user defined class to the HALCON operator `read_image`. For any instance where a conversion to or from a native HALCON tuple fails, an instance of `HTupleConversionError` is raised.

- `HError` is the base exception all other exceptions inherit from.
- `HTupleConversionError` is for errors that occur during conversion to or from a native HALCON tuple.
- `HDevEngineError` is for errors that occur while using the HDevEngine.
- `HInteropError` is for errors that are conceptually related to functionality while interfacing with third-party libraries, such as NumPy.

All custom HALCON/Python exceptions implement `__str__`.

15.12 Garbage Collection

HALCON/Python objects that have some natively backed resources implement cleanup via finalizers, implementing `__del__`.

This means no additional care needs to be taken to ensure that native resources such as memory are freed after a variable is no longer reachable.

For example, CPython uses ref counting to implement garbage collection, which results in moderately deterministic behavior.

15.13 Named Parameters

Python offers users the choice to specify the name of function parameters on the call site. As seen with the call to `open_framegrabber`, this can help improve readability of code.

```
framegrabber = ha.open_framegrabber(
    name='File',
    horizontal_resolution=1,
    vertical_resolution=1,
    image_width=0,
    image_height=0,
    start_row=0,
    start_column=0,
    field='default',
    bits_per_channel=-1,
    color_space='default',
    generic=-1,
    external_trigger='default',
    camera_type='board/board.seq',
    device='default',
    port=1,
    line_in=-1
)
```

Compare this to a version without named parameters:

```
framegrabber = ha.open_framegrabber('File', 1, 1, 0, 0, 0, 0, 'default', -1,
  'default', -1, 'default', 'board/board.seq', 'default', 1, -1
)
```

For example, what does the 0 at position 7 mean?

We suggest using named parameters where it makes it easier for readers to understand what is going on.

Note that HALCON does not offer 100% backwards compatibility for operator parameter names. Should a parameter change, this will be noted in the compatibility notes, and can be quickly addressed with some string replacements.

15.14 Operator and Parameter Capitalization

Following Python's PEP8 standard for code style, functions and parameter in HALCON/Python are snake case.

15.15 UTF-8 in HALCON/Python

By default, all strings passed to HALCON from Python are encoded as UTF-8 bytes, and all string bytes passed from HALCON to HALCON/Python are decoded as UTF-8 strings. This is the only supported encoding.

Configuring HALCON via `set_system('filename_encoding', 'locale')` to use a local encoding is not supported.

When used in a legacy mode, some operators may return strings that are invalid UTF-8, even when 'filename_encoding' is set to 'utf8'. A workaround for such situations exists in the form of:

```
try:
    # Replace invalid UTF-8 when encoding \Halcon strings for use in Python.
    ha.ffi.enable_utf8_error_replace()

    # Call problematic operator ...
finally:
    # Restore default behavior of raising a UnicodeError.
    ha.ffi.disable_utf8_error_replace()
```

Disabling the safe default mode is only recommended as a workaround and should only be done where needed, for as long as needed.

15.16 HALCON/Python With HALCON XL

It's possible to use HALCON/Python with HALCON XL. To do so, set the environment variable HALCON_PYTHON_XL to the string true before the halcon module is imported. This can be done directly in Python:

```
import os
os.environ['HALCON_PYTHON_XL'] = 'true'

import halcon as ha
```

Alternatively, it's possible to set the environment variable through the usual system specific settings. If possible we recommend using the suggested method, to minimize external dependencies.

15.17 Global HALCON Functions in HALCON/Python

The following functions can be used to change global HALCON behavior. They must be called before any HALCON usage to prevent undefined behavior. It is safe to call these functions multiple times. The only exceptions are `HCancelDraw` and `HSetMemoryAllocatorType` which can be called at any time.

- `HUseSpinLock`
  See section 2.5.3 on page 20.

- `HDoLicenseError`
  Controls the display license error messages.
- `HStartUpThreadPool`
  See section 2.5.3 on page 20.
- `HCancelDraw`
  Cancels draw operators explicitly.
- `HSetMemoryAllocatorType`
  Sets the allocator used by HALCON. Can be used to avoid mimalloc initialization. Supported values are 'system' and 'mimalloc'.

Example usage:

```
import halcon as ha
ha.ffi.HUseSpinLock(False)
ha.ffi.HSetMemoryAllocatorType('system')

if __name__ == '__main__':
    img = ha.read_image('pcb')
    assert ha.get_system_s('memory_allocator') == 'system'

    # ha.ffi.HUseSpinLock(False) This is not allowed
    # ha.ffi.HUseSpinLock(True) This is not allowed

    # Allowed because HSetMemoryAllocatorType is exempt.
    ha.ffi.HSetMemoryAllocatorType('mimalloc')
    # The same can be achieved with set_system.
    ha.set_system('memory_allocator', 'mimalloc')
```

Part V

Programming With HALCON/C

Chapter 16

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 17 on page 109 introduces the different parameter classes of HALCON operators. We will explain the use of HALCON tuples (section 17.2.4 on page 112) 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. We will further explain the use of HALCON vectors in section 17.3 on page 115. Chapter 18 on page 123 is dedicated to the return values of HALCON operators. Chapter 19 on page 125 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 20 on page 131 contains example solutions for some common problems in image processing (like edge detection).

16.1 A First Example

This section demonstrates how to create a simple HALCON application with C. For a more comprehensive description, see chapter 19 on page 125.

The task is to read an image and compute the number of connected regions in it, as illustrated in figure 16.1 on page 107

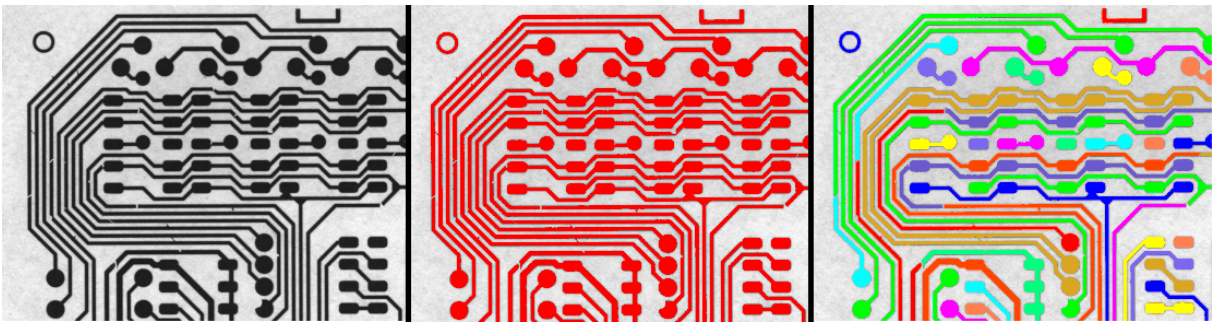


Figure 16.1: Left: Input image of a printed circuit board. Middle: Regions found by `threshold`, colored red. Right: Connected regions, a result of `connection`.

1. Install HALCON 25.11
2. Install a C99 or newer toolchain on your system.
3. Setup your C environment of choice.
4. Run the following commands in a shell:

```
mkdir region_example
cd region_example
```

5. Create a file named `main.c` and change the content to:

```
#include <stdio.h>
#include <inttypes.h>

#include <HalconC.h>

int main()
{
    Hobject img;
    read_image(&img, "pcb");

    Hobject region;
    threshold(img, &region, 0, 122);

    Hobject connected_regions;
    connection(region, &connected_regions);

    Hlong num_regions = 0;
    count_obj(connected_regions, &num_regions);

    printf("Number of Regions: \%" PRIdPTR "\n", num_regions);
}
```

6. Compile the program.
   For details see section 19.4 on page 127 for Windows and section 19.5 on page 128 for Linux.
7. To run the application, type the following command in the same shell:

```
./region_example
```

As a result, you should see the following output `'Number of Regions:  43'`.

Chapter 17

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.

Most HALCON operators can also be called using tuples of parameters instead of single values (so-called tuple mode). Take the operator `threshold` from our example program in the previous chapter, which segments an image and returns the segmented region: If you pass a tuple of images, it will return a tuple of regions, one for each input image. However, in contrast to HDevelop and other programming interfaces, in HALCON/C the tuple mode must be selected explicitly by prefixing the operator with `T_` and by using tuples for all control values (see section 17.2 on page 110 for more details). Whether an operator can be called in tuple mode can be seen in the HALCON reference manual.

HALCON/C provides the data structure `Htuple` for tuples of control parameters (see section 17.2.4 on page 112 for details) and the data structure `Hobject` for image objects (single objects as well as object tuples; see section 17.1).

17.1 Image Objects

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 `Hobject` for image objects (both images and regions). In fact, `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 `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 `obj_to_integer` or `copy_obj`. The number of objects in a tuple can be queried with `count_obj`. To convert the keys (returned from `obj_to_integer`) back to image objects again, the operator `integer_to_obj` has to be used. Note that `integer_to_obj` duplicates the image objects. (This does not necessarily mean that the corresponding gray value arrays are also duplicated. 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 `clear_obj`. Figure 17.1 contains an excerpt from a C program to clarify that approach.

```
...
Hobject   objects;     /* tuple of image objects                  */
Hobject   obj;         /* single image object                     */
Hlong     surrogate;   /* object key, converted to integer        */
Htuple    Tsurrogates; /* tuple of object keys                    */
Htuple    Index,Num;   /* temporary tuple for parameter passing   */
Hlong     i;           /* loop variable                           */
Hlong     num;         /* number of objects                       */

...
count_obj(objects, &num);
/* variant 1: object key -> control parameter                     */
create_tuple_i(&Index,1);
create_tuple_i(&Num,num);
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 17.1: Accessing the i-th image object in a tuple of image objects.

Some HALCON operators like `difference` allow the use of the following specific image objects as input parameters:

`NO_OBJECTS`: An empty tuple of image objects.

`EMPTY_REGION`: An image object with empty region (area = 0).

`FULL_REGION`: An image object with maximal region.

These objects may be returned by HALCON operators, too.

17.2 Control Parameters

HALCON/C supports the following data types as types for control parameters of HALCON operators:

- integers,

- floating point numbers,
- character arrays (strings),
- handles

17.2.1 String Encoding

Regardless of the encoding of the HALCON library (`set_system('filename_encoding', ...)`) the HALCON/C interface expects raw char pointer strings that are passed to HALCON operators and to `Htuple` instances to be UTF-8 encoded. The encoding of the HALCON/C interface (interface encoding) can be changed to local-8-bit encoding via a call of `SetHcInterfaceStringEncodingIsUtf8(false)`. **Note that the encoding must not be changed when using HDevEngine.** The current interface encoding can be requested via `IsHcInterfaceStringEncodingUtf8()`. It is not recommended to switch the interface encoding back and forth. The setting should be adjusted only once at the very beginning of the program (before the first HALCON operator or assignment), because the `Htuple` structure can not store in which encoding the contained strings are present, i.e, for all write and read accesses, the same encoding must be set. Furthermore, the interface encoding is set globally and is therefore not suitable for multithreading programs: Changing the setting in one thread has an effect on other threads. Note also, that UTF-8 encoded strings may take more memory space than strings in local-8-bit-encoding. This must be considered when providing the buffer for strings returned by HALCON operators. As long as the maximum length of the strings returned is not known for sure, it is recommended preferring operator calls in tuple mode over simple mode.

If the string encoding needs to be converted without changing the interface encoding, the following functions for converting `wchar_t*` strings and `char*` strings to or from tuples are provided:

UTF-8 version	local 8-bit version	`wchar_t*` version (Windows only)
`create_tuple_s_from_utf8`	`create_tuple_s_from_local8bit`	`create_tuple_s_from_wcs`
`reuse_tuple_s_from_utf8`	`reuse_tuple_s_from_local8bit`	`reuse_tuple_s_from_wcs`
`init_s_from_utf8`	`init_s_from_local8bit`	`init_s_from_wcs`
`set_s_from_utf8`	`set_s_from_local8bit`	`set_s_from_wcs`
`get_s_to_utf8`	`get_s_to_local8bit`	`get_s_to_wcs`

Table 17.1: Encoding helper functions.

When converted to a different encoding, a string may need a buffer of bigger size. To check on whether you have to allocate more memory for your string you can use the `get_s_to_*`-functions mentioned in the table above (see also figure 17.7). The functions transcode a string to the respective encoding and write it into the buffer provided by the user. If the returned size of the string is greater or equal to the provided buffer, the passed buffer size was too small.

17.2.2 Control Parameter Tuples

Using control parameter tuples in C is not 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 a operator calculates is unknown beforehand.

Basically, there are three ways to determine the default type of control parameters for a given operator:

- Operator description in the HALCON reference manual
- HALCON system operator `get_param_info`
- Description of the HALCON interface in the file `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 `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_`. This means that the following is a call of the HALCON operator `select_shape` (as described in the HALCON reference manual) in simple mode

```
select_shape
```

whereas

```
T_select_shape
```

is a call of the same operator in tuple mode.

17.2.3 Simple Mode

In the so-called *simple mode*, all control parameters of operator calls are variables (or constants) of the data types

- `Hlong` for integers (HALCON type `LONG_PAR`),
- `double` for floating point numbers (`DOUBLE_PAR`), or
- `char*` for character arrays (strings, `STRING_PAR`).

`Hlong` and `double` input control parameters are passed *by value* as usual, the corresponding output control parameters are passed *by reference*, using the &-operator. String parameters are pointers to `char` in both cases. *Note that the memory for output control parameters (in particular strings) has to be provided by the caller!* We recommend allocating memory for at least `1024` characters for string parameters of unknown length. Output parameter values that are of no further interest can be set to `NULL`.

Examples for HALCON operator calls in simple mode can be found in the C programs in figure 17.1 and figure 16.1 on page 107.

Operators with parameters resembling handles must always be called in the tuple mode as described in the following section.

17.2.4 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 are unknown 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_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 arrays of the types

- `Hlong*` for integer arrays (HALCON type `LONG_PAR`),
- `double*` for floating point arrays (`DOUBLE_PAR`),
- `char**` for string arrays (strings, `STRING_PAR`) or
- `Hphandle*` for handle arrays (`HANDLE_PAR`)

Additionally, a `MIXED_PAR` array type is supported that can hold an array with any of the three native value types in arbitrary combination. The usage of these four array types is transparent.

Control parameter tuples must be created, deleted, and manipulated using the appropriate HALCON/C procedures *only* (overview in figures 17.2, 17.3, 17.4 and 17.5).

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. For output parameters that are of no further interest you can pass `NULL`.

The following sections describe the five most important steps when calling a HALCON operator in tuple mode:

- allocate memory (section 17.2.4.1 on page 113)
- construct input parameters (section 17.2.4.2 on page 113)
- call operator (section 17.2.4.3 on page 113)
- process output parameters (section 17.2.4.4 on page 113)
- free memory (section 17.2.4.5 on page 114)

Section 17.2.4.6 on page 114 contains an example.

Finally, section 17.2.4.7 on page 114 describes a generic calling mechanism that can be used in interpreters or graphical user interfaces.

17.2.4.1 Allocate Memory

First, memory must be allocated for all tuples of input control parameters, using `create_tuple` or `create_tuple_type`, respectively (see figures 17.2). Memory for output control parameter tuples is allocated by HALCON/C (a call of `create_tuple` isn't necessary). With `create_tuple_i` etc. you can create a tuple of length 1 and set its value in a single step (see figures 17.3). With `reuse_tuple_i` etc. you can reuse an existing tuple, i.e., destroy and reallocate it and set a single value (see figures 17.4).

17.2.4.2 Create Input Parameters

You set tuple elements using the appropriate procedures `set_*`. `set_s`, which insert a string into a tuple, allocates the needed memory by itself, and then copies the string (see figure 17.5).

17.2.4.3 Call Operator

Then, the HALCON operator is actually called. The operator name is (as already explained) preceded by a `T_` to denote tuple mode.

17.2.4.4 Process Output Parameters

Further processing of the output parameter tuples takes place, using the procedures `length_tuple`, `get_type` (see figure 17.2) and `get_*` (see figure 17.6 and figure 17.7). 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. Also note that output handles might be freed when deleting the last tuple. If a handle should be used further, it should be kept in a tuple.

```
void create_tuple(tuple,length)     or     macro CT(tuple,length)
     Htuple    *tuple;
     Hlong      length;
     /* creates a MIXED_PAR tuple that can hold 'length' entries h  */

void create_tuple_type(tuple,length,type)
     Htuple    *tuple;
     Hlong      length;
     Hlong      type;
     /* creates a tuple of 'type' that can hold 'length' entries.
      * 'type' can hold either LONG_PAR, DOUBLE_PAR, STRING_PAR,
      * HANDLE_PAR, or MIXED_PAR.                                    */

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)          */

Hlong length_tuple(tuple)            or     macro LT(tuple)
     Htuple    tuple;
     /* returns the length of a tuple (number of entries)           */

void copy_tuple(input,output)      or     macro CPT(input,output)
     Htuple    input;
     Htuple    *output;
     /* creates a tuple and copies the entries of the input tuple   */

void resize_tuple(htuple,new_length) or   macro RT(htuple,new_length)
     Htuple    *htuple;
     Hlong      new_length;
     /* creates a tuple with the new size and copies the previous   */
     /* entries                                                     */
```

Figure 17.2: HALCON/C Htuple procedures (1/6).

17.2.4.5 Free Memory

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.

17.2.4.6 Example for the Tuple Mode

An example for the tuple mode can be found in figure 17.8 or the file `example3.c`: The aim is to obtain information about the current HALCON system state. The operator `get_system('?',Values)` (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.

17.2.4.7 Generic Calling Mechanism

There is also 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_by_id(id, ...)
```

calls the HALCON operator `id` in tuple mode, passing input parameters and getting the output parameters (see figure 17.9 on page 121 for the complete signature). The id of an operator can be requested with `get_operator_id`.

```
void create_tuple_i(tuple,value)
     Htuple    *tuple;
     Hlong      val;
     /* creates a tuple with specified integer value               */

void create_tuple_d(tuple,value)
     Htuple    *tuple;
     double    val;
     /* creates a tuple with specified double value                */

void create_tuple_h(tuple,value)
     Htuple    *tuple;
     Hphandle  val;
     /* creates a tuple with specified handle value                */

void create_tuple_s(tuple,value)
     Htuple    *tuple;
     char      *val;
     /* creates a tuple with specified string value                */

void create_tuple_s_from_local8bit(tuple,value)
     Htuple    *tuple;
     char      *val;
     /* creates a tuple with specified string value converted      */
     /* from local-8-bit encoding                                  */

void create_tuple_s_from_utf8(tuple,value)
     Htuple    *tuple;
     char      *val;
     /* creates a tuple with specified string value converted      */
     /* from UTF-8 encoding                                        */

void create_tuple_s_from_wcs(tuple,value)
     Htuple    *tuple;
     wchar_t   *val;
     /* creates a tuple with specified string value converted      */
     /* from wide-character encoding                               */
```

Figure 17.3: HALCON/C Htuple procedures (2/6).

17.3 Vectors

HALCON/C provides the data structure `Hvector` for the use of the vector functionality of the HDevelop language. A HALCON vector is a container that can hold an arbitrary number of elements of the identical data type (i.e., tuples, iconic objects, or vectors) and dimension. The type of a vector, i.e., its dimension and the type of its elements is defined when initializing the vector instance and cannot be changed during its lifetime. A vector with one dimension may be a vector of tuples or a vector of iconic objects. A two-dimensional vector may be a vector of vectors of tuples or a vector of vectors of iconic objects, and so on.

Construction of Vectors

When creating such a vector in HALCON/C you have to differ between vectors of iconic objects and vectors of tuples.

```
void reuse_tuple_i(tuple,val)
     Htuple    *tuple;
     Hlong      val;
     /* reuses a tuple with specified integer value                */

void reuse_tuple_d(tuple,val)
     Htuple    *tuple;
     double    val;
     /* reuses a tuple with specified double value                 */

void reuse_tuple_h(tuple,val)
     Htuple    *tuple;
     Hphandle  val;
     /* reuses a tuple with specified handle value                 */

void reuse_tuple_s(tuple,val)
     Htuple    *tuple;
     char      *val;
     /* reuses a tuple with specified string value                 */

void reuse_tuple_s_from_local8bit(tuple,val)
     Htuple    *tuple;
     char      *val;
     /* reuses a tuple with specified string value converted       */
     /* from local-8-bit encoding                                  */

void reuse_tuple_s_from_utf8(tuple,val)
     Htuple    *tuple;
     char      *val;
     /* reuses a tuple with specified string value converted       */
     /* from UTF-8 encoding                                        */

void reuse_tuple_s_from_wcs(tuple,val)
     Htuple    *tuple;
     wchar_t   *val;
     /* reuses a tuple with specified string value converted       */
     /* from wide-character encoding                               */
```

Figure 17.4: HALCON/C Htuple procedures (3/6).

```
Hvector   vectorObj, vectorTup;

// Create a one-dimensional vector of iconic objects
V_create_object_vector(1,&vectorObj);

// Create a one-dimensional vector of tuples
V_create_tuple_vector(1,&vectorTup);
```

These calls create empty vectors of one dimension, e.g., a vector of tuples. It is also possible to create multi-dimensional vectors, i.e., a vector of vectors of tuples or a vector of vectors of iconic objects and so on, by specifying the number of dimensions in the call.

```
V_create_object_vector(2,&vectorObjMulti);
```

Note that the vector type and its dimension cannot be changed after the creation of the vector.

Accessing and Setting Vector Elements

To access an element in a vector of tuples you may use `V_get_vector_tuple`. The following code line queries a tuple contained in `vectorTup` at position `index` and returns a copy of the result as a `HTuple` in `tuple`.

```
void set_i(tuple,val,index)                          or     macro SI(tuple,val,index)
     Htuple     tuple;
     Hlong      val;
     Hlong      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;
     Hlong      index;
     /* inserts a double with value 'val' into a tuple at           */
     /* position 'index' ('index' in [0,length_tuple(tuple) - 1])   */

void set_h(tuple,val,index)                          or     macro SH(tuple,val,index)
     Htuple     tuple;
     Hphandle   val;
     Hlong      index;
     /* inserts a handle 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;
     Hlong      index;
     /* inserts a copy of string 'val' into a tuple at position     */
     /* 'index' ('index' in [0,length_tuple(tuple) - 1]);           */
     /* memory necessary for the string is allocated by set_s       */

void set_s_from_local8bit(tuple,val,index)           or     macro SS_LOC(tuple,val,index)
     Htuple     tuple;
     char       *val;
     Hlong      index;
     /* inserts a copy of string 'val', converted from              */
     /* local-8-bit encoding, into a tuple at position 'index'      */
     /* ('index' in [0,length_tuple(tuple) - 1]).                   */

void set_s_from_utf8(tuple,val,index)                or     macro SS_U(tuple,val,index)
     Htuple     tuple;
     char       *val;
     Hlong      index;
     /* inserts a copy of string 'val', converted from UTF-8        */
     /* encoding, into a tuple at position 'index' ('index' in      */
     /* [0,length_tuple(tuple) - 1]).                               */

void set_s_from_wcs(tuple,val,index)                 or     macro SS_W(tuple,val,index)
     Htuple     tuple;
     wchar_t    *val;
     Hlong      index;
     /* inserts a copy of string 'val', converted from              */
     /* wide-character encoding, into a tuple at position           */
     /* 'index' ('index' in [0,length_tuple(tuple) - 1]).           */
```

Figure 17.5: HALCON/C Htuple procedures (4/6).

```
V_get_vector_tuple(vectorTup,index,tuple);
```

For accessing elements in vectors of iconic objects `V_get_vector_obj` can be used instead.

```
V_get_vector_obj(image,vectorTup,index);
```

```
int get_type(tuple,index)                              or     macro GT(tuple,index)
     Htuple     tuple;
     Hlong      index;
     /* returns type of value at position 'index' in the tuple.     */
     /* Possible values: LONG_PAR, DOUBLE_PAR or STRING_PAR          */

Hlong get_i(tuple,index)                               or     macro GI(tuple,index)
     Htuple     tuple;
     Hlong      index;
     /* returns the integer at position 'index' in the tuple        */
     /* (a type error results in a run time error)                   */

double get_d(tuple,index)                              or     macro GD(tuple,index)
        Htuple     tuple;
        Hlong      index;
     /* returns the floating point number at position 'index' in    */
     /* the tuple. (a type error results in a run time error)        */

Hphandle get_h(tuple,index)                            or     macro GH(tuple,index)
        Htuple     tuple;
        Hlong      index;
     /* returns the handle at position 'index' in the               */
     /* tuple. (a type error results in a run time error)            */

char *get_s(tuple,index)                               or     macro GS(tuple,index)
      Htuple     tuple;
      Hlong      index;
     /* returns the pointer(!) to the string at position 'index'    */
     /* in the tuple. (a type error results in a run time error)     */
     /* Attention: indices must be in [0,length_tuple(tuple) - 1]    */
```

Figure 17.6: HALCON/C Htuple procedures (5/6).

The element to be accessed is specified by its `index` in form of a tuple. It may either contain a single index of an element in one-dimensional vectors or several indices of the corresponding subvector(s) and its subelement(s) in multi-dimensional vectors.

You may also query a whole subvector of a multi-dimensional vector using `V_get_vector_elem`.

Before accessing the contents of a vector you may set some vector elements first. Again a distinction is made between vectors of tuples, vectors of iconic objects, and multi-dimensional vectors. The following lines show how to set vector elements of one-dimensional vectors.

```
// Set an element of a vector of iconic objects
V_set_vector_obj(image,vectorObj,index);

// Set an element of a vector of tuples
V_set_vector_tuple(vectorTup,index,tuple);
```

The element to be set is again addressed by the specified `index`, which is represented as a `Htuple`. Beside the index, the vector itself and the respective `Hobject` or `Htuple` to be set (e.g., `image` or `tuple`) must be added to the call.

For multi-dimensional vectors the specified index tuple must contain the indices of the subvector(s) and its subelement. The following code lines show the whole process from creating a two-dimensional vector, the image to be set, and the index tuple up to specifying the indices and finally setting the element of the vector.

```
Hlong get_s_to_local8bit(buffer,size,tuple,index)   or     macro GS_LOC(tuple,index)
      char*      buffer;
      Hlong      size;
      Htuple     tuple;
      Hlong      index;
     /* Copies the (local-8-bit encoded) string at position         */
     /* 'index' into 'buffer'. 'size' determines the size of the    */
     /* buffer. The return value determines the size of the         */
     /* string, that would be written. If the returned length of    */
     /* the string is greater or equal to the provided              */
     /* buffer, the buffer was too small.                           */
     /* Attention: indices must be in [0,length_tuple(tuple) - 1]   */

Hlong get_s_to_utf8(buffer,size,tuple,index)        or     macro GS_U(tuple,index)
      char*      buffer;
      Hlong      size;
      Htuple     tuple;
      Hlong      index;
     /* Copies the (UTF-8) string at position                       */
     /* 'index' into 'buffer'. 'size' determines the size of the    */
     /* buffer. The return value determines the size of the         */
     /* string, that would be written. If the returned length of    */
     /* the string is greater or equal to the provided              */
     /* buffer, the buffer was too small.                           */
     /* Attention: indices must be in [0,length_tuple(tuple) - 1]   */

Hlong *get_s_to_wcs(buffer,size,tuple,index)        or     macro GS_W(tuple,index)
      wchar_t*   buffer;
      Hlong      size;
      Htuple     tuple;
      Hlong      index;
     /* Copies the (wide-character) string at position              */
     /* 'index' into 'buffer'. 'size' determines the size of the    */
     /* buffer. The return value determines the size of the         */
     /* string, that would be written. If the returned length of    */
     /* the string is greater or equal to the provided              */
     /* buffer, the buffer was too small.                           */
     /* Attention: indices must be in [0,length_tuple(tuple) - 1]   */
```

Figure 17.7: HALCON/C Htuple procedures (6/6).

```
// Initialize and set the image and indices
Hobject img;
Htuple  indices;
Hvector vectorObjMulti;

V_create_object_vector(2,&vectorObjMulti);
read_image(&img,"Image");
set_i(indices,0,0);
set_i(indices,1,1);

// Set a subvector of a multi-dimensional vector
V_set_vector_obj(img,vectorObjMulti,indices);
```

It is also possible to set a subvector of multi-dimensional vectors using `V_set_vector_elem`.

Note that it is also possible to set non-existing vector elements. If necessary the vector is automatically enlarged and initialized with empty elements.

```
#include "HalconC.h"

main ()
{
  Htuple In, SysFlags, Info;
  Hlong  i, num;

  printf("System information:\n");
  /* prepare query */
  create_tuple(&In, 1);
  /* "?" = list of all informations */
  set_s(In, "?", 0);
  T_get_system(In, &SysFlags);
  destroy_tuple(In);
  num = length_tuple(SysFlags);
  for (i = 0; i < num; i++)
  {
    create_tuple(&In, 1);
    set_s(In, get_s(SysFlags, i), 0);
    printf("%s ", get_s(SysFlags, i));
    T_get_system(In, &Info);
    destroy_tuple(In);
    if (length_tuple(Info) > 0)
    {
      switch (get_type(Info, 0))
      {
      case INT_PAR:
        printf("(Hlong):  %" LONG_FORMAT "d\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;
      case HANDLE_PAR:
        printf("(handle):  %" LONG_FORMAT "d\n", (Hlong)get_h(Info, 0));
        break;
      }
    }
    else
    {
      printf("(--):  no data\n");
    }
    destroy_tuple(Info);
  }
}
```

Figure 17.8: Tuple mode example program: Printing the current HALCON system state.

Destruction of Vectors

If a `Hvector` is not needed for further processing its contents and allocated memory must be freed with `V_destroy_vector`.

```
V_destroy_vector(vectorTup);
```

Additional Information

In addition to the previously mentioned basic information the data structure of `Hvector` provides some more functionality, e.g., inserting or removing vector elements, or concatenation of vectors. Please refer to the corresponding

```
/* generic HALCON operator call style:
 * - the operator is called by an id that is returned by get_operator_id;
 *    attention: this id may differ for different HALCON versions
 * - the tuple arrays are passed directly to the call -> this method is
 *   thread-safe
 *-------------------------------------------------------------------------*/
int    get_operator_id(const char* name);

Herror T_call_halcon_by_id(int id,
                           const Hobject in_objs[],
                           Hobject out_objs[],
                           const Htuple in_ctrls[],
                           Htuple out_ctrls[]);
```

Figure 17.9: Generic calling mechanism for the HALCON/C tuple mode.

header file `Hvector.h` in `%HALCONROOT%\include\halconc` for further information.

Chapter 18

Return Values of HALCON Operators

HALCON operator return values (type `Herror`) can be divided into two categories:

- Messages (`H_MSG_*`) and
- Errors (`H_ERR_*`).

HALCON operators return `H_MSG_TRUE`, if no error occurs. Otherwise, a corresponding error value is returned.

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

```
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 code. Finally, the operator

```
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`.

Chapter 19

Creating Applications With HALCON/C

The HALCON distribution contains examples for creating an application with HALCON/C. The following sections show

- the relevant directories and files (section 19.1 on page 125)
- the list of provided example applications (section 19.2 on page 126)
- the relevant environment variables (section 19.3 on page 126)
- how to create an executable under Windows (section 19.4 on page 127)
- how to create an executable under Linux (section 19.5 on page 128)

19.1 Relevant Directories and Files

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`
  HALCON library (Windows)
- `bin\%HALCONARCH%\halconc.dll`, `lib\%HALCONARCH%\halconc.lib`
  HALCON/C library (Windows)
- `bin\%HALCONARCH%\halconxl.dll`, `halconcxl.dll`,
  `lib\%HALCONARCH%\halconxl.lib`, `halconcxl.lib`
  Corresponding libraries of HALCON XL (Windows)
- `lib/$HALCONARCH/libhalcon.so`
  HALCON library (Linux)
- `lib/$HALCONARCH/libhalconc.so`
  HALCON/C library (Linux)
- `lib/$HALCONARCH/libhalconxl.so,libhalconcxl.so`
  Corresponding libraries of HALCON XL (Linux)
- `include\HProto.h`
  External function declarations
- `%HALCONEXAMPLES%\c\CMakeLists.txt`
  Example CMake file, which can be used to compile the example programs

- %HALCONEXAMPLES%\c\README.md
  Information about building the examples using CMake
- %HALCONEXAMPLES%\c\source\
  Directory containing the source files of the example programs
- images\
  Images used by the example programs
- help\operators_*
  Files necessary for online information
- doc*
  Various manuals (in subdirectories)

19.2 Example Programs

There are several example programs in the HALCON/C distribution (%HALCONEXAMPLES%\c\source\). To experiment with these examples, we recommend creating 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	Shows 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_by_id)
example7.c	Describes the handling of RGB images
example8.c	Demonstrates the creation of an image from user memory
example_extern8.c	Demonstrates the creation of an image from user memory. In contrast to example8.c, the memory is not copied but pointed to.
example_multithreaded1.c	This special case demonstrates the use of HALCON in a multithreaded application. Please note that it does not make sense to run the example on a single-processor or single-core computer.

19.3 Relevant Environment Variables

In the following, we briefly describe the relevant environment variables; see the Installation Guide, section A.4 on page 47, for more information, especially about how to set these variables. 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 on which HALCON is used. For more information, see the Installation Guide, section 1.4 on page 8.

The variable %HALCONEXAMPLES% indicates where the provided examples are installed.

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.

Keep the following in mind in connection with the example programs:

- Default location for images
  The default directory for the HALCON operator `read_image` to look for images is `%HALCONEXAMPLES%\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`. The latter has to be set before starting the program.
- Output terminal under Linux
  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`.

19.4 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 following command near the top of your C file:

```
#include "HalconC.h"
```

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 as well as under Windows, the include file `HalconC.h` includes several Linux-specific headers as well if included under Linux. As they do not exist under Windows, and as the Visual C++ compiler ignores 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.

Make sure that the stacksize is sufficient. Some sophisticated image processing problems require up to 1 MB stacksize, so ensure to configure your compiler accordingly. See your compiler manual for additional information on this topic.

HALCON XL applications:
Please note that you should **use HALCON XL only when you need its features**.

If you want to use HALCON XL, you have to link the libraries `halconxl.lib/.dll` and `halconcxl.lib/.dll` instead of `halcon.lib/.dll` and `halconc.lib/.dll` in your project.

Building HALCON Examples With CMake Under Windows

You can find examples that showcase various HALCON use cases and how they are implemented in C in the directories below `%HALCONEXAMPLES%\c`. You can build them with CMake by using the provided `CMakeLists.txt` file.

To build example programs using CMake, do the following:

1. If required, download CMake (version 3.7.1 or later) from the CMake website and install it.
2. Create a build directory and then run `cmake` to configure the build and create the application:

   ```
   mkdir build
   cd build
   cmake %HALCONEXAMPLES%\c
   cmake --build .
   ```

   To configure, CMake needs to know the location of the HALCON installation, the location of the example files, and which HALCON architecture to use:

   - Specify the location of the HALCON installation via the `HALCON_DIR` CMake option, or via the `%HALCONROOT%` environment variable if the option is not set.
   - Specify the location of the HALCON example files via the `%HALCONEXAMPLES%` environment variable.

- Specify the HALCON architecture with the HALCON_ARCHITECTURE CMake option, or via the %HALCONARCH% environment variable. If neither the option nor the environment variable are set, CMake will try to guess the architecture based on the host build system.

For general information on how to use CMake, see the CMake documentation.

HALCON XL applications: By default, the examples will be built using the normal version of HALCON. If you want to build using HALCON XL, set the option HALCON_XL to ON or 1 in CMake during the configuration step. For this, use the following syntax:

```
cmake -DHALCON_XL=1 %HALCONEXAMPLES%\c
```

3. Optionally, you can use the -G option to specify the generator for a new build tree.
For more information about CMake generators, see the CMake documentation.

19.5 Creating Applications Under Linux

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 following command near the top of your C file:

```
#include "HalconC.h"
```

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 by giving it the -I<pathname> command line flag to denote the include file directory.

To create an application, 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.

HALCON XL applications:
Please note that you should **use HALCON XL only when you need its features**.

If you want to use HALCON XL, you have to link the libraries libhalconcxl.so and libhalconxl.so instead.

To link and run applications under Linux, ensure that the system variable LD_LIBRARY_PATH contains the HALCON library path $HALCONROOT/lib/$HALCONARCH.

Building HALCON Examples With CMake Under Linux

You can find examples that showcase various HALCON use cases and how they are implemented in C in the directories below $HALCONEXAMPLES/c. You can build them with CMake by using the provided CMakeLists.txt file.

To build example programs using CMake, do the following:

1. If required, download CMake (version 3.7.1 or later) from the CMake website and install it.
2. Create a build directory and then run cmake to configure the build and create the application:

```
mkdir build
cd build
cmake $HALCONEXAMPLES/c
cmake --build .
```

To configure, CMake needs to know the location of the HALCON installation, the location of the example files, and which HALCON architecture to use:

- Specify the location of the HALCON installation via the HALCON_DIR CMake option, or via the $HALCONROOT environment variable if the option is not set.
- Specify the location of the HALCON example files via the $HALCONEXAMPLES environment variable.

- Specify the HALCON architecture with the `HALCON_ARCHITECTURE` CMake option, or via the `$HALCONARCH` environment variable. If neither the option nor the environment variable are set, CMake will try to guess the architecture based on the host build system.

For general information on how to use CMake, see the CMake documentation.

HALCON XL applications: By default, the examples will be built using the normal version of HALCON. If you want to build using HALCON XL, set the option `HALCON_XL` to `ON` or `1` in CMake during configuration.

3. Optionally, you can use the `-G` option to specify the generator for a new build tree.
For more information about CMake generators, see the CMake documentation.

Chapter 20

Typical Image Processing Problems

This final chapter shows the possibilities of HALCON and HALCON/C on the basis of several simple image processing problems.

20.1 Thresholding

One of the most common HALCON operators is the following:

```
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.

20.2 Detecting Edges

The following HALCON/C sequence is suitable for edge detection:

```
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`.

20.3 Dynamic Threshold

Among other things, the following code is suitable for edge detection, too:

```
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.

20.4 Simple Texture Transformations

Texture transformations are used to enhance specific image structures. The behavior of the transformation depends on the filters used (HALCON provides many different texture filters).

```
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.

20.5 Eliminating Small Objects

The following morphological operation eliminates small image objects and smoothes the boundaries of the remaining objects:

```
...
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`).

20.6 Selecting Specific Orientations

Yet another application example of morphological operations is the selection of image objects with specific orientations:

```
...
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`).

20.7 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:

```
...
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 21

Introducing HDevEngine

As the name suggests, HDevEngine is the “engine” of HDevelop. This chapter briefly introduces you to its basic concepts. Chapter 22 on page 141 explains how to use it in C++ applications, and chapter 23 on page 153 how to use it in .NET applications (C#, Visual Basic.NET, etc.). Additional information that is independent of the used programming language can be found in chapter 25 on page 187.

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 objects, e.g., C# or Visual Basic.NET. Thus, you can use HDevelop programs 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.

The Easiest Way to Use HDevEngine (in C++ and C#)

The Library Project Export generates an interface, and all the necessary HDevEngine code to transparently use HDevelop procedures in your application, as if they were genuine C++ or C# functions. For more information, see the HDevelopUser’s Guide "Exporting Library Projects", section 10.1 on page 303.

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 in your application.

What is HDevEngine?

HDevEngine is provided as a C++ class library and a .NET assembly. It consists of the following classes:

- `clHDevEngineName` (C++), `clHDevEngineName` (.NET)
  This is the main class of HDevEngine. With it you manage global settings.
- `clHDevProgramName` (C++), `clHDevProgramName` (.NET)
  With this class you load an HDevelop program and get general information about it.
- `clHDevProgramCallName` (C++), `clHDevProgramCallName` (.NET)
  With this class you execute a program and get the values of its variables.
- `clHDevProcedureName` (C++), `clHDevProcedureName` (.NET)
  With this class you load an HDevelop procedure and get general information about it.

- `clHDevProcedureCallName` (C++), `clHDevProcedureCallName` (.NET)
  With this class you pass input parameters to an HDevelop procedure, execute it, and retrieve its output parameters.
- `clHDevOperatorImplName` (C++), `nclIHDevOperatorsName`, `nclHDevOpMultiWindowImplName`, `nclHDevOpFixedWindowImplName` (.NET)
  As noted above, HDevEngine does not implement internal HDevelop operators like `dev_display`. All HDevEngine variants provide a class or interface to create your own implementation for those operators that are useful in your application. HDevEngine/.NET also includes two convenience classes that provide a default implementation of the operators.
- `clHDevEngineExceptionName` (C++), `clHDevEngineExceptionName` (.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.

How to Develop Applications With HDevEngine

With HDevEngine, you can execute complete HDevelop programs or individual local or external HDevelop procedures. Which way is better depends on the stage of development and on your task:

- When developing the image processing part of your application, you will create an HDevelop program. Thus, as a first test of your (programmed) application, it is useful to **execute the HDevelop program** via HDevEngine. This test will already assure that the general configuration of your application (environment variables, procedure path, etc.) is correct.

  The HDevelop program itself should of course use the same procedures that you plan to execute from the programmed application.
- After you finished its development, you integrate the image processing part into your programmed application by **executing** the corresponding **HDevelop procedures**. Typically, you display image processing results by using the methods of the underlying HALCON programming language interface, i.e., HALCON/C++ for HDevEngine/C++, or HALCON/.NET for HDevEngine/.NET (C#, Visual Basic.NET, etc.), but you can also encapsulate recurring display tasks in HDevelop procedures.
- Whether to use local or external procedures depends on the reusability of the procedure. **External procedures** should be used for widely reusable tasks, e.g., opening the connection to the image acquisition device and configuring it, or for standard image processing tasks like bar code or data code reading. Groups of closely related external procedures may be combined into a procedure library to keep them as a unit in a single file.

  In contrast, **local procedures** are suitable for tasks that are not completely reusable, e.g., for training and configuring a shape model to find objects. Then, different applications can use their optimized variant of the procedure instead of creating a single procedure with many parameters and internal switches that suits all applications.

  Using local procedures means that you must load the HDevelop program that contains them. However, as noted above, loading and executing the corresponding HDevelop program is a good test of the general configuration of the application.

Parallel Programming With HDevEngine

HDevEngine is thread-safe and reentrant. Settings like the procedure path and the implementation of the display operators are managed globally for all threads by the main HDevEngine instance. Threads then typically have their own instance of program or procedure calls. They can share instances of the classes for programs and procedures.

Programs or procedures are only loaded upon setting the procedure path. Therefore, to load multiple programs or procedures in parallel safely, special care is required:

- Use the `import` statement to make the external procedures available from within your program. For more information, see also the HDevelop example program `hdevelop/Control/import.hdev`.
- Alternatively, set the procedure path (via `SetProcedurePath`) and create the program or procedure (via `clHDevProgramName`/`clHDevProcedureName`) *before* you start parallelization.

- If the above approaches are not possible, ensure that setting the procedure paths and creating the programs and procedures both happen within a single critical section to prevent this operation from being interrupted by other threads. Otherwise, if another thread calls `SetProcedurePath` after the current thread has called `SetProcedurePath` and before the current thread has finished creating its programs or procedures, the resulting behavior would be undefined.

For more information, compare section 25.1 on page 187.

Note that HDevEngine does not safely support subthreads (started with `par_start`) that continue to run after the main procedure call returns to application code. A procedure that is intended as entry point for an HDevEngine application should join all of its started subthreads before returning.

See also the general information about parallel programming with HALCON in section 2.2 on page 16, in particular the style guide in section 2.2.2 on page 17.

Just-In-Time Compilation of Procedures in HDevEngine

For optimized performance, HDevEngine can also execute compiled procedures using its built-in just-in-time (JIT) compiler. The compilation is turned on or off using an attribute of the `HDevEngine` class. If turned on, the compilation will take place implicitly the first time a procedure is called. It is also possible to explicitly compile the used procedures of a HDevelop program. See section 22.3 on page 151 (C++) and section 23.3 on page 175 (.NET) for more information about using the JIT compiler.

JIT compilation is not supported for procedures that use any of the following features:

- `dev_error_var`
- procedure calls using the `par_start` qualifier
- `par_join`
- `dev_display` with a vector expression as variable
- `for` loop with a vector expression as index variable
- call of any procedure that cannot be JIT compiled due to the above reasons.

If one of these features is found, the corresponding procedure is executed uncompiled as before by HDevEngine.

HDevEngine XL

Like HALCON, the language-dependent versions of HDevEngine are provided in two variants: based on HALCON and based on HALCON XL. The latter use the XL versions of the HALCON library and of HALCON/C++, and HALCON/.NET, respectively.

Chapter 22

HDevEngine in C++ Applications

This chapter explains how to use HDevEngine in C++ applications. Section 22.1 on page 141 quickly summarizes some basic information, e.g., how to compile and link such applications. Section 22.2 on page 142 then explains how to use HDevEngine based on examples.

An overview about the classes of HDevEngine and their methods can be found in section 25.1 on page 187.

22.1 How to Create an Executable Application With HDevEngine/C++

You create executable HDevEngine applications in a way similar to normal HALCON/C++ applications. Chapter 7 on page 49 describes this in detail; here, we summarize the most important points and include the extensions for HDevEngine:

1. In your application, you include the main header file `HalconCpp.h` and HDevEngine's header file `HDevengineCpp.h` and use the corresponding namespaces on Windows and Linux systems:

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

using namespace HalconCpp;
using namespace HDevEngineCpp;
```

2. To compile the application, use the following include paths on Windows systems

```
/I "$(HALCONROOT)\include" /I "$(HALCONROOT)\include\halconcpp"
/I "$(HALCONROOT)\include\hdevengine"
```

   and on Linux systems

```
-I$HALCONROOT/include -I$HALCONROOT/include/halconcpp
-I$HALCONROOT/include/hdevengine
```

3. Link the following libraries on Windows systems

```
/libpath:"$(HALCONROOT)\lib\$(HALCONARCH)" hdevenginecpp.lib halconcpp.lib
```

   and on Linux systems

```
-L$HALCONROOT/lib/$HALCONARCH -lhdevenginecpp -lhalconcpp -lhalcon
```

   HDevEngine XL applications: If you want to use HDevEngine XL, link the following libraries on Windows systems

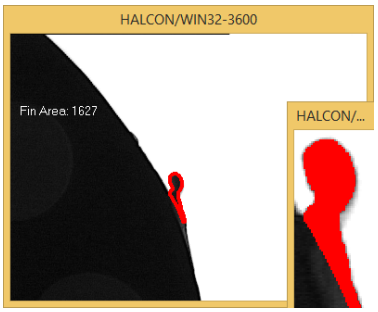


Figure 22.1: Executing an HDevelop program that detects fins on a boundary.

```
/libpath:"$(HALCONROOT)/lib/$(HALCONARCH)" hdevenginecppxl.lib halconcppxl.lib
```

and on Linux systems

```
-L$HALCONROOT/lib/$HALCONARCH -lhdevenginecppxl -lhalconcppxl -lhalconxl
```

22.2 How to Use HDevEngine/C++

This section explains how to use HDevEngine based on example applications, which reside in the subdirectory `%HALCONEXAMPLES%\hdevengine\cpp`. Like the examples for HALCON/C++ described in chapter 7 on page 49, they are provided with CMake files for Linux, and Windows systems.

The example applications show how to

- execute an HDevelop program (section 22.2.1 on page 142),
- execute HDevelop procedures (section 22.2.2 on page 144),
- implement display operators (section 22.2.3 on page 147),
- error handling (section 22.2.4 on page 148).

Section 22.2.5 on page 150 contains additional information for creating multithreaded applications using HDevEngine.

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 `exec_program` (source file `exec_program.cpp`), which checks the boundary of a plastic part for fins. Figure 22.1 on page 142 shows a screenshot of the application.

22.2.1.1 Step 1: Initialization

First, we include the main header files of HALCON/C++ and of HDevEngine and the corresponding namespaces. Note that in this example application the HDevEngine header file is already included via the header file `my_hdevoperatorimpl.h` so we don't have to include the `HDevEngineCpp.h` header file explicitly:

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

using namespace HalconCpp;
using namespace HDevEngineCpp;
```

The `main` procedure just calls a procedure that does all the work of the example. First, we create an instance of the main HDevEngine class `clHDevEngineName`.

```
HDevEngine         my_engine;
```

The path to the HDevelop program and the external procedure path are stored in string variables, with a suitable syntax for the used platform. Note that in Windows applications you can use both / and \ in path strings:

```
std::string halcon_examples =
    (std::string)HSystem::GetSystem("example_dir")[0].S();
std::string program_path(halcon_examples), ext_proc_path(halcon_examples);

program_path += "/hdevengine/hdevelop/fin_detection.hdev";
ext_proc_path += "/hdevengine/procedures";
```

If the HDevelop program calls external procedures, you must **set the external procedure path** with the method `mHDESetProcedurePathName`:

```
my_engine.SetProcedurePath(ext_proc_path.c_str());
```

22.2.1.2 Step 2: Load Program

Now, we create an instance of the class `clHDevProgramName` and **load the HDevelop program** with the method `mHDELoadProgramName`. Note that `mHDELoadProgramName` changes the working directory if a program is loaded successfully.

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 22.2.4 on page 148.

```
  HDevProgram my_program;
try
{
  my_program.LoadProgram(program_path.c_str());
}
catch (HDevEngineException& hdev_exception)
  ...
```

22.2.1.3 Step 3: Execute Program

If the program could be loaded successfully, we **execute the program** with the method and store the returned instance of the class `clHDevProgramCallName` in a variable for later use:

```
HDevProgramCall prog_call = my_program.Execute();
```

22.2.1.4 Step 4: Get Results

That's all you need to do to execute an HDevelop program. You can also access its "results", i.e., its variables with the method `mHDEGetCtrlVarTupleName`. In the example program, the area of the extracted fin is queried and then displayed:

```
HTuple result = prog_call.GetCtrlVarTuple("FinArea");
printf("\nFin Area: %f\n\n", result[0].D());
```

Note that program variables can only be accessed when the program has terminated.

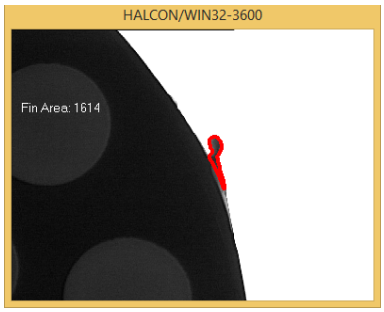


Figure 22.2: Executing an external HDevelop procedure that detects fins on a boundary.

22.2.1.5 General: Display Results

How to display results while the program is running is described in section 22.2.3 on page 147.

22.2.2 Executing HDevelop Procedures

This section describes example applications that execute HDevelop procedures:

- a single external procedure (section 22.2.2.1 on page 144) and
- multiple local and external procedures (section 22.2.2.8 on page 146).

22.2.2.1 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 22.2 on page 144 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 22.2.3 on page 147.

22.2.2.2 Step 1: Initialization

As when executing an HDevelop program, we include the main header files of HALCON/C++ and of HDevEngine and the namespaces. In this example application the HDevEngine header file is included via the header file `my_error_output.h` so we don't need to include it explicitly. The main procedure just calls the procedure `run` that does all the work of the example. We create an instance of the main HDevEngine class `clHDevEngineName` and directly **set the external procedure path** with the method `mHDESetProcedurePathName`. If the external procedure is from a procedure library, the external procedure path may include the name of the library file.

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

using namespace HalconCpp;
using namespace HDevEngineCpp;

void run(void)
{
  std::string halcon_examples =
      (std::string)HSystem::GetSystem("example_dir")[0].S();
  std::string ext_proc_path(halcon_examples);
  ...
  HDevEngine().SetProcedurePath(ext_proc_path.c_str());

  DetectFin();
}
```

22.2.2.3 Step 2: Load Procedure

In the "action" routine, we **load the external procedure** with the constructor of the class `clHDevProcedureName`, specifying the name of the procedure, and store the returned procedure call in an instance of the class `clHDevProcedureCallName`. The call is encapsulated in a `try...catch`-block to handle exceptions occurring in the constructor, 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 22.2.4 on page 148.

```
void DetectFin()
{
  try
  {
    HDevProcedure     proc("detect_fin");
    HDevProcedureCall proc_call(proc);
```

Before executing the procedure, we **open and initialize the graphics window** in which the results are to be displayed and load an example image sequence:

```
    const char* image_sequ_str = "fin";

    HWindow win(00, 100, 384, 288);
    win.SetPart(0, 0, 575, 767);
    win.SetDraw("margin");
    win.SetLineWidth(4);

    HFramegrabber fg("File", 1, 1, 0, 0, 0, 0, "default", -1, "default", -1,
                     "default", image_sequ_str, "default", -1, -1);
```

22.2.2.4 Step 3: Set Input Parameters of Procedure

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 input object** by storing it in the instance of `clHDevProcedureCallName` with the method `mHDESetInputIconicParamObjectName`. Which parameter to set is specified via its index (starting with 1); there is also a method to specify it via its name (see section 25.1.5 on page 198):

```
for (long i = 0; i < 3; i++)
{
  HImage image = fg.GrabImage();

  proc_call.SetInputIconicParamObject(1, image);
```

As as alternative to passing parameters, you can also use **global variables** in HDevEngine (compare the HDevelopUser's Guide, section 8.3.2 on page 252). You set the value of a global variable with the methods `SetGlobalIconicVarObject` or `SetGlobalCtrlVarTuple` and query it with the methods `GetGlobalIconicVarObject` and `GetGlobalCtrlVarTuple`.
However, take care not to overwrite the value of a variable of one program with that of another: Each global variable can have only one value at a time for all running HDevEngine instances.

22.2.2.5 Step 4: Execute Procedure

Now, we **execute the procedure** with the method `mHDEExecuteName`.

```
  proc_call.Execute();
```

22.2.2.6 Step 5: Get Output Parameters of Procedure

If the procedure was executed successfully, we can access its results, i.e., the fin region and its area, with the methods and of the class `clHDevProcedureCallName`; again, you can specify the parameter via its index or name (see section 25.1.5 on page 198).

```
HRegion fin_region = proc_call.GetOutputIconicParamObject(1);
HTuple  fin_area;
proc_call.GetOutputCtrlParamTuple(1, &fin_area);
```

22.2.2.7 Step 6: Display Results of Procedure

Now, we **display the results in the graphics window**. Note how we access the area by selecting the first element of the returned tuple:

```
  char fin_area_str[200];
  sprintf(fin_area_str, "Fin Area: %ld", (long)(fin_area[0].L()));
  win.DispImage(image);
  win.SetColor("red");
  win.DispRegion(fin_region);
  win.SetColor("white");
  win.SetTposition(150, 20);
  win.WriteString(fin_area_str);
```

22.2.2.8 Executing Local and External HDevelop Procedures

The example application `exec_procedures` (source file `exec_procedures.cpp`) executes local and external HDevelop procedures with HDevEngine. It mimics the behavior of the HDevelop program described in section 22.2.1 on page 142. The display of results is partly programmed explicitly and partly delegated to an HDevelop procedure, using the implementation of the internal display operators described in section 22.2.3 on page 147.

Local and external procedures are created and executed in exactly the same way. The only difference is that in order to use a local procedure, you must load the program it is contained in, whereas to load external procedures you must set the procedure path. `clHDevProcedureName` provides different constructors to facilitate this task (see section 25.1.4 on page 195).

22.2.3 Display

In this section, we explain how to provide your own implementation of HDevelop's internal display operators. The files `my_hdevoperatorimpl.h` and `my_hdevoperatorimpl.cpp` contain an example implementation, which is used in the applications `exec_program` (source file `exec_program.cpp`), which was already discussed in section 22.2.1 on page 142, and `exec_procedures` (source file `exec_procedures.cpp`).

In fact, HDevEngine does not provide an implementation of the internal display operators but provides the class `clHDevOperatorImplName`, 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., for `dev_display` and have the same parameters (see section 25.1.6 on page 200 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 file implements the operators `dev_open_window`, `dev_set_window_extents`, `dev_set_part`, `dev_set_window`, `dev_get_window`, `dev_clear_window`, `dev_clear_window`, `dev_close_window`, `dev_display`, `dev_set_draw`, `dev_set_shape`, `dev_set_color`, `dev_set_colored`, `dev_set_lut`, `dev_set_paint`, and `dev_set_line_width`:

```
class MyHDevOperatorImpl : public HDevEngineCpp::HDevOperatorImplCpp
{
public:
  virtual int DevOpenWindow(const HalconCpp::HTuple& row,
                            const HalconCpp::HTuple& col,
                            const HalconCpp::HTuple& width,
                            const HalconCpp::HTuple& height,
                            const HalconCpp::HTuple& background,
                            HalconCpp::HTuple* win_id);
  virtual int DevSetWindowExtents(const HalconCpp::HTuple& row,
                                  const HalconCpp::HTuple& col,
                                  const HalconCpp::HTuple& width,
                                  const HalconCpp::HTuple& height);
  virtual int DevSetPart(const HalconCpp::HTuple& row1,
                         const HalconCpp::HTuple& col1,
                         const HalconCpp::HTuple& row2,
                         const HalconCpp::HTuple& col2);
  virtual int DevSetWindow(const HalconCpp::HTuple& win_id);
  virtual int DevGetWindow(HalconCpp::HTuple* win_id);
  virtual int DevClearWindow();
  virtual int DevCloseWindow();
  virtual int DevDisplay(const HalconCpp::HObject& obj);
  virtual int DevDispText(const HalconCpp::HTuple& string,
                          const HalconCpp::HTuple& coordSystem,
                          const HalconCpp::HTuple& row,
                          const HalconCpp::HTuple& column,
                          const HalconCpp::HTuple& color,
                          const HalconCpp::HTuple& GenParamName,
                          const HalconCpp::HTuple& GenParamValue);
  virtual int DevSetDraw(const HalconCpp::HTuple& draw);
  virtual int DevSetContourStyle(const HalconCpp::HTuple& style);
  virtual int DevSetShape(const HalconCpp::HTuple& shape);
  virtual int DevSetColor(const HalconCpp::HTuple& color);
  virtual int DevSetColored(const HalconCpp::HTuple& colored);
  virtual int DevSetLut(const HalconCpp::HTuple& lut);
  virtual int DevSetPaint(const HalconCpp::HTuple& paint);
  virtual int DevSetLineWidth(const HalconCpp::HTuple& width);
};
```

In addition to these methods, the class contains methods to handle multiple graphics windows. These methods use a second class that manages all open windows. This class is thread-safe and reentrant but not described in detail in this section.

```
  HalconCpp::HTuple GetCurrentWindow()  const;
  size_t            GetCount()          const;
  void              AddWindow(const HalconCpp::HTuple& id);
  HalconCpp::HTuple PopWindow();
  Hlong             SetWindow(const HalconCpp::HTuple& id);

  class WinIdContainer
```

In the executed HDevelop program, two graphics windows are used, one for the main display and one for zooming into the image (see figure 22.1 on page 142).

To use the implementation of `clHDevOperatorImplName`, you include the header file:

```
#include "my_hdevoperatorimpl.h"
```

With the method `mHDESetHDevOperatorImplName`, you pass an instance of your version of `clHDevOperatorImplName` to HDevEngine, which then calls its methods when the corresponding operator is used in the HDevelop program or procedure.

```
    my_engine.SetHDevOperatorImpl(&op_impl);
```

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" or "current". The methods for the internal display operators simply call the corresponding non-internal display operator: For example, a call to `dev_display` in the HDevelop program is "redirected" in to `disp_obj`, with the iconic object to display and the handle of the active window as parameters:

```
int MyHDevOperatorImpl::DevDisplay(const HObject& obj)
{
  HCkDev(DispObj(obj, GetCurrentWindow()));
}
```

Similarly, `dev_set_draw` is redirected in to `set_draw`:

```
int MyHDevOperatorImpl::DevSetDraw(const HTuple& draw)
{
  HCkDev(SetDraw(GetCurrentWindow(), draw));
}
```

As you can see, these operators can be implemented quite easily. The implementation of the operators for handling graphics windows is not described here. We recommend using the example implementation as it is because it provides all the necessary functionality for single- and multithreaded applications.

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 `error_handling` (source file `error_handling.cpp`), which provokes different types of exceptions and "catches" them.

HDevEngine "throws" exceptions in form of the class `clHDevEngineExceptionName`, 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 code (see section 25.1.7 on page 200 for the declaration of the class).

The example code for displaying information about exceptions in a graphics window is contained in the files `my_error_output.cpp` and `my_error_output.h`. You can use it in your application by including the header file:

```
#include "my_error_output.h"
```

The files provide two procedures. The simpler one displays only the error message and waits for a mouse click to continue:

```
void DispMessage(const char* message)
{
  HWindow win(100, 100, ERR_WIN_WIDTH_SIMPLE, ERR_WIN_HEIGHT_SIMPLE, NULL,
              "visible", "");
  win.SetPart(0, 0, ERR_WIN_HEIGHT_SIMPLE - 1, ERR_WIN_WIDTH_SIMPLE - 1);
  win.SetColor("yellow");
  win.SetTposition(10, 10);
  WriteMessageNL(win, message);

  // wait for mouse click to continue
  win.SetColor("red");
  win.SetTposition(ERR_WIN_HEIGHT_SIMPLE / 2 + 10, ERR_WIN_WIDTH_SIMPLE / 2);
  win.WriteString("...click into window to continue");
  win.Click();
}
```

The more complex one prints all available information for the exception (only relevant code shown):

```
void DispErrorMessage(const HDevEngineCpp::HDevEngineException& exception,
                      const char* context_msg /*=NULL*/)
{
  try
  {
    char text[2000];

    HWindow win(100, 100, ERR_WIN_WIDTH_COMPLEX, ERR_WIN_HEIGHT_COMPLEX, NULL,
                "visible", "");

    WriteMessageNL(win, exception.Message());

    sprintf(text, "    Error category: <%d : %s>", exception.Category(),
            exception.CategoryText());
    WriteMessageNL(win, text);

    sprintf(text, "    Error code:   <%d>", exception.HalconErrorCode());
    WriteMessageNL(win, text);

    sprintf(text, "    Procedure:      <%s>", exception.ExecProcedureName());
    WriteMessageNL(win, text);

    sprintf(text, "        Line:         <%d : %s>", exception.ProgLineNum(),
            exception.ProgLineName());
    WriteMessageNL(win, text);
  }
```

This procedure is called when an exception occurs. The example provokes different errors and displays the corresponding information; some of them are described in the following. Figure 22.3 on page 150 displays an exception that occurred because the application tried to load a non-existing HDevelop program (category).

```
  try
  {
    program.LoadProgram(wrong_program_path.c_str());
  }
  catch (HDevEngineException& hdev_exception)
  {
    DispErrorMessage(hdev_exception,
                     "Error #1: Try to load a program that does not exist");
  }
```

The same exception category occurs when a program is loaded whose external procedures are not found (see figure 22.4 on page 150).

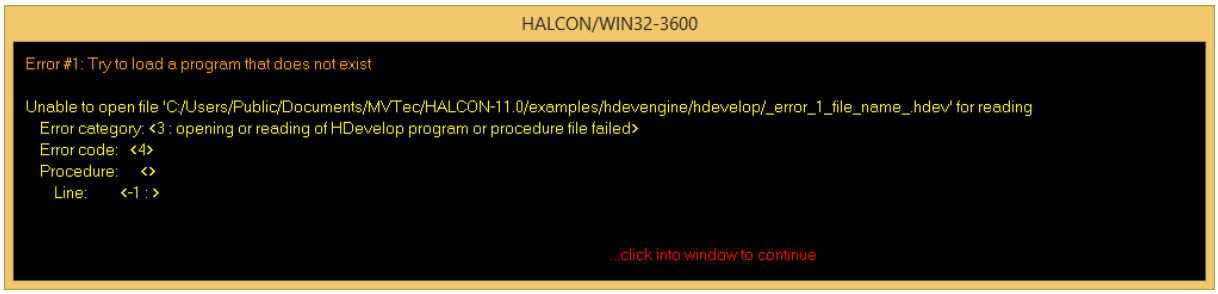


Figure 22.3: Content of the exception if an HDevelop program could not be found.

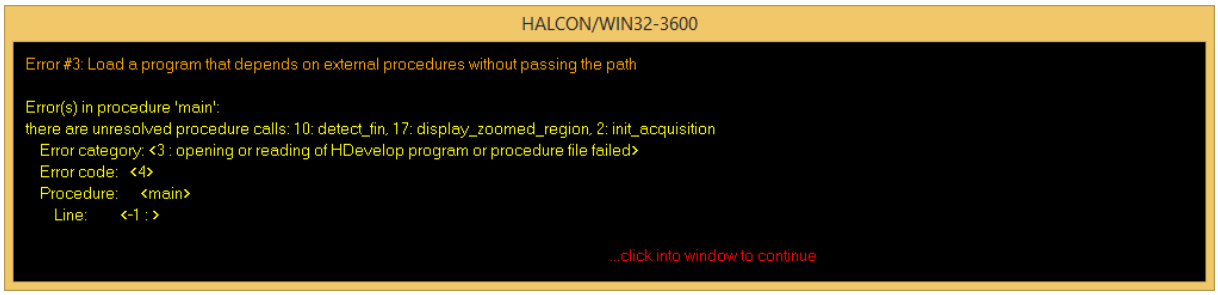


Figure 22.4: Content of the exception if external procedures of an HDevelop program could not be loaded.

The exception displayed in figure 22.5 on page 150 occurs because an input iconic parameter is not initialized (category). It contains very detailed information about where the error occurred and why.

The exception displayed in figure 22.6 on page 151 is provoked by calling an operator with an invalid parameter (category).

With the method `UserData` (see section 25.1.7 on page 200), you can also access user exception data that is thrown within an HDevelop program or procedure by the operator `throw` similarly to the operator `dev_get_exception_data`.

In case of an exception (which is not caught within the procedure) the procedure call is cleaned up. This means all subthreads are destroyed and all values of input and output parameters are cleared. Therefore, we recommend that you always set all input parameters before executing a call even if some of them did not change.

Note that you can configure the behavior of HDevEngine when loading programs or procedures that contain invalid lines or unresolved procedure calls with the method `SetEngineAttribute` (see section 25.1.1 on page 188).

22.2.5 Creating Multithreaded Applications

In the example `mfc\exec_procedures_mt_mfc` , three threads execute HDevelop procedures for image acquisition, data code reading, and visualization in parallel (see figure 22.7 on page 151). Please have a look at the example

Figure 22.5: Content of the exception if an input parameter was not initialized.

Figure 22.6: Content of the exception if an error occurred in a HALCON operator call.

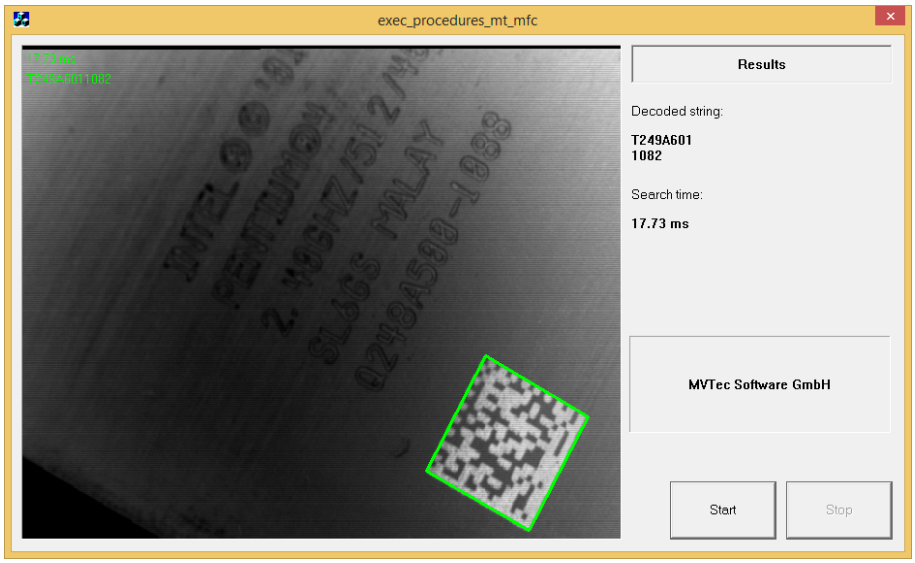


Figure 22.7: Example program with three threads performing image acquisition, data code reading, and visualization in parallel.

source files (in the directory `mfc\exec_procedures_mt_mfc\source\`) to see how the threads synchronize their input and output data.

The example `exec_programs_mt` (source file `exec_programs_mt.cpp`) shows how one or several different HDevelop programs can be executed in different threads in parallel. Note that it is kept very general and does not realize a specific application.

The HDevelop program(s) must be passed as command line arguments. Optionally, you can pass for every program the number of threads and/or how often the program should be performed consecutively within each thread. The command line parameters are explained when calling the executable without parameters.

22.2.6 Executing an HDevelop Program with Vector Variables

The example application `use_vector_variables` shows how to load and execute an HDevelop program that contains vector variables in HDevengine/C++. In the example two vectors are used for processing: one containing the input images and one containing scaling factors. When executing the program the gray values of the input images are scaled according to the scaling factors. Please have a look at the example source file `use_vector_variables.cpp` for more details on how to work with vector variables in HDevengine/C++.

22.3 Using the Just-in-time Compiler With HDevEngine/C++

The just-in-time compilation of procedures needs to be enabled in your instance of the `clHDevEngineName` class:

```
bool exec_compiled = true;
...
HDevEngine my_engine;
// enable or disable execution of compiled procedures
my_engine.SetEngineAttribute("execute_procedures_jit_compiled",
                             exec_compiled ? "true" : "false");
```

Procedures (and procedures referenced by it) are compiled at the moment a corresponding instance of `clHDevProcedureCallName` or `clHDevProgramCallName` is created.

You can also explicitly pre-compile all used procedures of a HDevelop program or procedure using the method `CompileUsedProcedures` of `clHDevProgramName` or `clHDevProcedureName`, respectively.

In the following example, all used procedures of a procedure call are just-in-time compiled:

```
HDevProgram   my_program(program_path.c_str());
HDevProcedure proc_fib(my_program, "fib");
...
proc_fib.CompileUsedProcedures();
```

Chapter 23

HDevEngine in .NET Applications

This chapter explains how to use HDevEngine in C# and Visual Basic.NET applications. Section 23.1 on page 153 quickly summarizes some basic information about creating HDevEngine applications with .NET Core and .NET Framework. Section 23.2 on page 153 then explains how to use HDevEngine/.NET based on examples.

23.1 Basics

A short reference of the C++ classes for the HDevEngine can be found in section 25.1 on page 187. The .NET classes are very similar; their exact definition can be seen in the online help of Visual Studio (see section 11.1.1 on page 68).

23.1.1 Adding HDevEngine/.NET to a .NET Core Application

No HDevEngine-specific actions are required. See section 10.3.1 on page 62.

23.1.2 Adding HDevEngine/.NET to a .NET Framework Application

To use HDevEngine in Visual Studio .NET, you must add a reference to the HDevEngine/.NET assembly `hdevenginedotnet.dll` via the Solution Explorer. See section 10.3.3 on page 64.

HDevEngine XL applications: If you want to use HDevEngine/.NET XL, you must add the XL versions of the HALCON/.NET and HDevEngine/.NET assembly instead.

23.2 Examples

23.2.1 Creating an HDevEngine/.NET Application

This section explains how to create a simple HDevEngine/.NET application with .NET Core. For a more comprehensive description, read section 23.2.2 on page 154.

1. Install the .NET Core SDK for your system.
2. Run the following commands in a command line:

```
dotnet new console -n hdevengine-example
cd hdevengine-example
dotnet add package MVTec.HalconDotNet -v 25110
```

3. Change the content of `Program.cs` to:

```
using System;
using System.Diagnostics;
using System.IO;

using HalconDotNet;

namespace hdevengine_example
{
  class Program
  {
    static void Main(string[] args)
    {
      string ExampleDir = HSystem.GetSystem("example_dir");
      string ProcedurePath = "/hdevengine/procedures";

      HDevEngine Engine = new HDevEngine();
      Engine.SetProcedurePath(Path.GetFullPath(ExampleDir + ProcedurePath));

      HImage Image = new HImage("fin2");

      HDevProcedure Procedure = new HDevProcedure("detect_fin");
      HDevProcedureCall ProcCall = new HDevProcedureCall(Procedure);

      ProcCall.SetInputIconicParamObject("Image", Image);
      ProcCall.Execute();

      HTuple FinArea = ProcCall.GetOutputCtrlParamTuple("FinArea");
      Console.WriteLine(String.Format("Fin Area: {0}", FinArea.I));
    }
  }
}
```

4. To run the application type the following command in a command line:

```
dotnet run
```

As as result, you should see the following output `'Fin Area:  1634'`.

23.2.2 Using HDevEngine/.NET

This section explains how to use HDevEngine/.NET with the help of example applications written in C#, which reside in the subdirectory `%HALCONEXAMPLES%\hdevengine\c#`. Most examples use Windows Forms, which is only available for .NET Core on Windows. The HDevEngine/.NET specific information also applies to .NET Core in general.

For some examples, Visual Basic.NET versions are available, which reside in the subdirectory `%HALCONEXAMPLES%\hdevengine\vb.net`. They are identical except for the standard differences between the two languages.

The example applications show how to

- execute an HDevelop program (section 23.2.3 on page 155),
- executeHDevelop procedures (section 23.2.4 on page 156),
- implement display operators (section 23.2.5 on page 161),
- error handling (section 23.2.6 on page 162),
- multithreading (section 23.2.7 on page 164).

Figure 23.1: Executing an HDevelop program that detects fins on a boundary.

23.2.3 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 on page 155 shows a screenshot of the application; it contains two buttons to load and execute the HDevelop program.

23.2.3.1 Step 1: Initialization

First, we create a global instance of the main HDevEngine class `clHDevEngineName`.

```
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 `mHDESetProcedurePathName`:

```
String ProgramPathString;

private void ExecProgramForm_Load(object sender, System.EventArgs e)
{
  string ExampleDir = HSystem.GetSystem("example_dir");
  string ProcedurePath = "/hdevengine/procedures";
  MyEngine.SetProcedurePath(Path.GetFullPath(ExampleDir + ProcedurePath));

  ProgramPathString = Path.GetFullPath(
    ExampleDir + "/hdevengine/hdevelop/fin_detection.hdev"
  );
}
```

Note that the latter is only necessary if the HDevelop program calls external procedures.

23.2.3.2 Step 2: Load Program

When you click the button to load the HDevelop program, an instance of the class `clHDevProgramName` is created, with the path of the program as parameter. Furthermore, an instance of `clHDevProgramCallName` is created for later use. Note that the working directory will be changed if a program is loaded.

Exceptions occurring in the constructors, e.g., because the file name was not specified correctly, are handled with the standard C# error handling mechanism:

```
private void LoadBtn_Click(object sender, System.EventArgs e)
{
  try
  {
    HDevProgram Program = new HDevProgram(ProgramPathString);
    ProgramCall = new HDevProgramCall(Program);
  }
  catch (HDevEngineException Ex)
  {
    MessageBox.Show(Ex.Message, "HDevEngine Exception");
    return;
  }

  ...
}
```

More information on error handling can be found in section 23.2.6 on page 162.

23.2.3.3 Step 3: Execute Program

When you click the button to execute the program, the method `mHDEExecuteName` is called:

```
private void RunProgram()
{
  try
  {
    try
    {
      ProgramCall.Execute();
    }
  ...
  }
}
```

23.2.3.4 Step 4: Get Results

That's all you need to do to execute a HDevelop program. You can also access its "results", i.e., its variables with the method `mHDEGetCtrlVarTupleName`. In the example program, the area of the extracted fin is queried and then displayed:

```
double FinArea;
FinArea = ProgramCall.GetCtrlVarTuple("FinArea");
Window.SetTposition(150, 20);
Window.WriteString("Fin Area: ");
```

Note that program variables can only be accessed when the program has terminated.

23.2.3.5 General: Display Results

How to display results while the program is running is described in section 23.2.5 on page 161.

23.2.4 Executing HDevelop Procedures

This section describes example applications that execute HDevelop procedures:

- A single external procedure (section 23.2.4.1 on page 157)
- Multiple local and external procedures (section 23.2.4.8 on page 159).

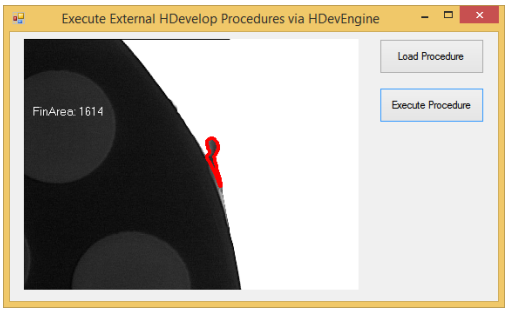


Figure 23.2: Executing an external HDevelop procedure that detects fins on a boundary.

23.2.4.1 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 on page 157 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.

23.2.4.2 Step 1: Initialization

As when executing an HDevelop program, we create a global instance of the main HDevEngine class `clHDevEngineName` and set the external procedure path with the method `mHDESetProcedurePathName` upon loading the form (code for constructing the path omitted). If the external procedure is from a procedure library, the external procedure path may include the name of the library file.

```
private HDevEngine MyEngine = new HDevEngine();

private void ExecExtProcForm_Load(object sender, System.EventArgs e)
{
  string ProcedurePath = "/hdevengine/procedures";
  ...
  MyEngine.SetProcedurePath(Path.GetFullPath(ExampleDir + ProcedurePath));
}
```

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 `HSmartWindowControl` (also see section 11.7 on page 80 and section 11.5 on page 78). 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 `HSmartWindowControl` and initialize the window:

```
private HWindow Window;

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

  Window.SetDraw("margin");
  Window.SetLineWidth(4);
}
```

23.2.4.3 Step 2: Load Procedure

When you click the button `Load`, the HDevelop procedure is loaded with the constructor of the class `clHDevProcedureName`, specifying the name of the procedure, and a corresponding procedure call is created

as an instance of the class clHDevProcedureCallName. Exceptions occurring in the constructors, 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 23.2.6 on page 162.

```
private void LoadBtn_Click(object sender, System.EventArgs e)
{
  try
  {
    HDevProcedure Procedure = new HDevProcedure("detect_fin");
    ProcCall = new HDevProcedureCall(Procedure);
  }
  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();
  Framegrabber.OpenFramegrabber("File", 1, 1, 0, 0, 0, 0, "default",
     -1, "default", -1, "default", "fin.seq", "default", -1, -1);
```

23.2.4.4 Step 3: Set Input Parameters of Procedure

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 clHDevProcedureCallName with the method mHDESetInputIconicParamObjectName. Which parameter to set is specified via its name (as an alternative, you can specify it via its index):

```
HImage Image = new HImage();
HRegion FinRegion;
HTuple FinArea;

for (int i = 0; i <= 2; i++)
{
  Image.GrabImage(Framegrabber);
  Image.DispObj(Window);

  ProcCall.SetInputIconicParamObject("Image", Image);
```

As as alternative to passing parameters, you can also use **global variables** in HDevEngine (compare the HDevelopUser's Guide, section 8.3.2 on page 252). You set the value of a global variable with the methods SetGlobalIconicVarObject or SetGlobalCtrlVarTuple and query it with the methods GetGlobalIconicVarObject and GetGlobalCtrlVarTuple.
However, take care not to overwrite the value of a variable of one program with that of another: Each global variable can have only one value at a time for all running HDevEngine instances.

23.2.4.5 Step 4: Execute Procedure

Now, we execute the procedure with the method mHDEExecuteName.

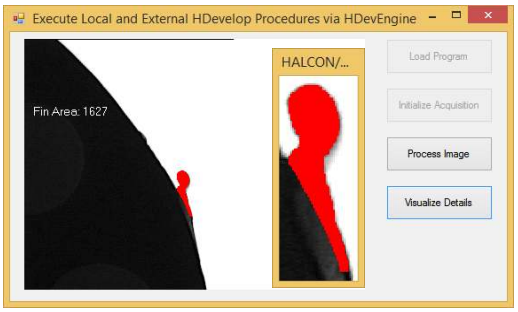


Figure 23.3: Screenshot of the application.

```
ProcCall.Execute();
```

23.2.4.6 Step 5: Get Output Parameters of Procedure

If the procedure was executed successfully, we can access its results, i.e., the fin region and its area, with the methods `nmHDEGetOutputIconicParamRegion` and of the class `clHDevProcedureCallName`; 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 .

```
FinRegion = ProcCall.GetOutputIconicParamRegion("FinRegion");
FinArea = ProcCall.GetOutputCtrlParamTuple("FinArea");
```

23.2.4.7 Step 6: Display Results of Procedure

Finally, we display the results in the graphics window:

```
this.Invoke((MethodInvoker)delegate
{
  Image.DispObj(Window);
  Window.SetColor("red");
  Window.DispObj(FinRegion);
  Window.SetColor("white");
  Window.SetTposition(150, 20);
  Window.WriteString("FinArea: " + FinArea.D);
});
```

23.2.4.8 Executing Local and External HDevelop Procedures

The example application `ExecProcedures` executes local and external HDevelop procedures with HDevEngine. It mimics the behavior of the HDevelop program described in section 23.2.3 on page 155. The display of results is partly programmed explicitly and partly delegated to an HDevelop procedure, using the implementation of the internal display operators described in section 23.2.5 on page 161. Figure 23.3 on page 159 shows a screenshot of the application.

In the following, we briefly describe parts of the code.

Local and external procedures are created and executed in exactly the same way. The only difference is that in order to use a local procedure, you must load the program it is contained in, whereas to load external procedures you must set the procedure path. In the example, the image processing procedure is local, the other external. Note that the code for constructing the program and procedure path is omitted.

```
private HDevProcedureCall InitAcqProcCall;
private HDevProcedureCall ProcessImageProcCall;
private HDevProcedureCall VisualizeDetailsProcCall;

private void ExecProceduresForm_Load(object sender, System.EventArgs e)
{
  string ProcedurePath = "/hdevengine/procedures";
  ...
  MyEngine.SetProcedurePath(Path.GetFullPath(ExampleDir + ProcedurePath));
}
```

```
private void LoadBtn_Click(object sender, System.EventArgs e)
{
  try
  {
    HDevProgram Program = new HDevProgram(ProgramPathString);

    HDevProcedure InitAcqProc = new HDevProcedure(Program, "init_acquisition");
    HDevProcedure ProcessImageProc = new HDevProcedure(Program, "detect_fin");
    HDevProcedure VisualizeDetailsProc =
          new HDevProcedure(Program, "display_zoomed_region");

    InitAcqProcCall = new HDevProcedureCall(InitAcqProc);
    ProcessImageProcCall = new HDevProcedureCall(ProcessImageProc);
    VisualizeDetailsProcCall = new HDevProcedureCall(VisualizeDetailsProc);
  ...
}
```

One of the procedures opens the image acquisition device. It returns the corresponding handle, which we store in an instance of the class `HFramegrabber`.

```
private HFramegrabber Framegrabber;

private void InitAcqBtn_Click(object sender, System.EventArgs e)
{
  InitAcqProcCall.Execute();
  Framegrabber =
      new HFramegrabber(InitAcqProcCall.GetOutputCtrlParamTuple("AcqHandle").H);
  ...
}
```

In the example application, the device is closed when the application terminates and calls the finalizer of the class `HFramegrabber`, which in turn calls the operator `CloseFramegrabber`. If you use an HDevelop procedure for closing the connection to the device, you would invalidate the handle so that the finalizer raises an exception.

As in the previous example, the results of image processing (button `Process Image`) are displayed "manually" by calling HALCON/.NET operators. In contrast, when you click the button `Visualize Details`, an HDevelop procedure is executed that zooms onto the extracted fin. For this, we pass an implementation of HDevelop's internal display operators (see section 23.2.5 on page 161 for more information about the implementation classes) and remove it again after the procedure has been executed.

```
private void VisualizeDetailsBtn_Click(object sender, System.EventArgs e)
{
  MyEngine.SetHDevOperators(MyHDevOperatorImpl);

  VisualizeDetailsProcCall.SetInputIconicParamObject("Image", Image);
  VisualizeDetailsProcCall.SetInputIconicParamObject("Region", FinRegion);
  VisualizeDetailsProcCall.SetInputCtrlParamTuple("ZoomScale", 2);
  VisualizeDetailsProcCall.SetInputCtrlParamTuple("Margin", 5);
  VisualizeDetailsProcCall.Execute();

  MyEngine.SetHDevOperators(null);
}
```

The instance of the implementation class is initialized with the HALCON window of the form.

```
private HDevOpMultiWindowImpl MyHDevOperatorImpl;

private void WindowControl_Load(object sender, EventArgs e)
{
  Window = WindowControl.HalconWindow;
  ...
  MyHDevOperatorImpl = new HDevOpMultiWindowImpl(Window);
}
```

If the class `nclHDevOpMultiWindowImplName` is initialized without specifying the window, a new HALCON window will open automatically to emulate the behavior of HDevelop. Consequently, using the operator `dev_open_window` in your HDevelop program or procedure will open another window. The newly opened window is set active automatically.

23.2.5 Display

In contrast to the C++ version of HDevEngine, HDevEngine/.NET already provides convenience implementations of HDevelop's internal display operators in form of two classes:

- `nclHDevOpFixedWindowImplName` directs all display operators to a single graphics window (passed in the constructor), even if the HDevelop program or procedure uses multiple windows.
- `nclHDevOpMultiWindowImplName` 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.

In the example code, some of the actual program and procedure execution is delegated to a background thread. This is good practice because long-running execution would otherwise lead to an unresponsive GUI. Further, HDevelop code using interactive drawing objects would not work with `HSmartWindowControl` if the GUI thread is blocked.

While HALCON display operators are thread-safe, access to Windows Forms elements is not (for example, to enable buttons or set label texts). Therefore, result visualization is delegated back to the GUI thread using `Invoke()` calls in these examples.

Finally, using `dev_*` operators for visualization via `nclHDevOpFixedWindowImplName` or `nclHDevOpMultiWindowImplName` is limited for multithreaded applications (executing multiple procedures in parallel). This is because there is only one global "active" window at any given time as controlled by `dev_set_window`, hence threads cannot control their output windows independently. This behavior is consistent with the behavior in HDevelop (which also has only one active window) when using multiple threads via `par_start`.

If parallel visualization is desired, we recommend writing explicit visualization code using HALCON operators that output directly to the desired window. For more information, see section 23.2.7 on page 164.

The example program `ExecProgram` uses `nclHDevOpMultiWindowImplName`. To use this class (or `nclHDevOpFixedWindowImplName`), you pass an instance of it to `clHDevEngineName` with the method `nmHDESetHDevOperatorsName`:

```
private void WindowControl_Load(object sender, 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 22.2.3 on page 147) by creating a class implementing the interface `nclIHDevOperatorsName` and overloading its methods `mHDEDevOpenWindowName`, `mHDEDevDisplayName`, etc.

23.2.6 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 23.4 on page 162 shows a screenshot of the application.

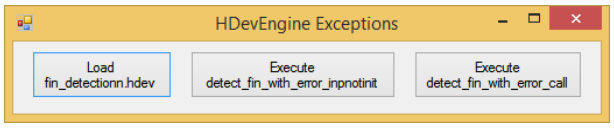


Figure 23.4: Provoking exceptions in HDevEngine.

HDevEngine throws exceptions as instances of the class `clHDevEngineExceptionName`, 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 code (also see section 25.1.7 on page 200).

In the example application, the following procedure displays all the information contained in `clHDevEngineExceptionName` in a message box:

```
private void DisplayException(HDevEngineException Ex)
{
  string FullMessage = "Message: <" + Ex.Message + ">" +
    ",  Error in program / procedure: <" + Ex.ProcedureName + ">" +
    ",  program line: <" + Ex.LineText + ">" +
    ",  line number: <" + Ex.LineNumber + ">" +
    ",  HALCON Error Number: <" + Ex.HalconError + ">";

  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.

```
try
{
  HDevProgram Program = new HDevProgram(ProgramPathString);
  new HDevProgramCall(Program);
}

catch (HDevEngineException Ex)
{
  DisplayException(Ex);
  return;
}
```

Figure 23.5 on page 163 displays an exception that occurred because the application tried to load a non-existing HDevelop program (category). 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):

```
procedure detect_fin_with_error_inpnotinit (Image: FinRegion: : FinArea)
  binary_threshold (NotExistingImage, Dark, 'max_separability', 'dark', UsedThreshold)
  ...
```

Figure 23.6 on page 163 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).

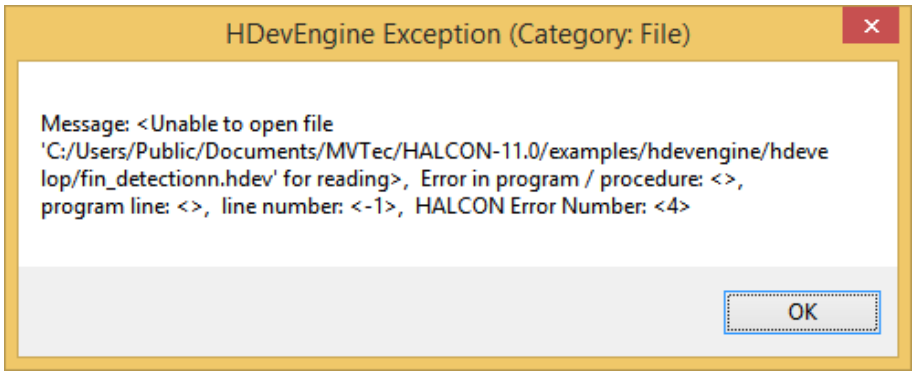


Figure 23.5: Content of the exception if an HDevelop program could not be loaded.

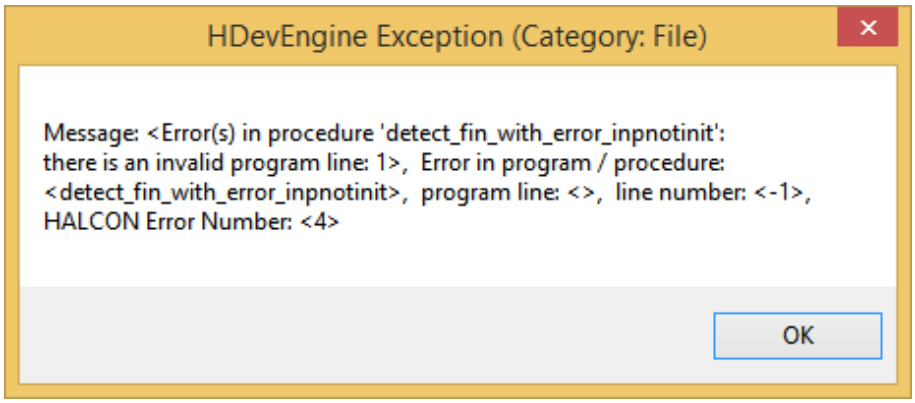


Figure 23.6: Content of the exception if an input parameter was not initialized.

```
procedure detect_fin_with_error_call (Image: FinRegion: : FinArea)
  binary_threshold (Image, Dark, 'max_separability', 'dark', UsedThreshold)
  difference (Image, Dark, Background)
  dev_set_color ('blue')
  dev_display (Background)
  closing_circle (Background, ClosedBackground, -1)
  ...
```

Figure 23.7 on page 163 shows the content of the exception.

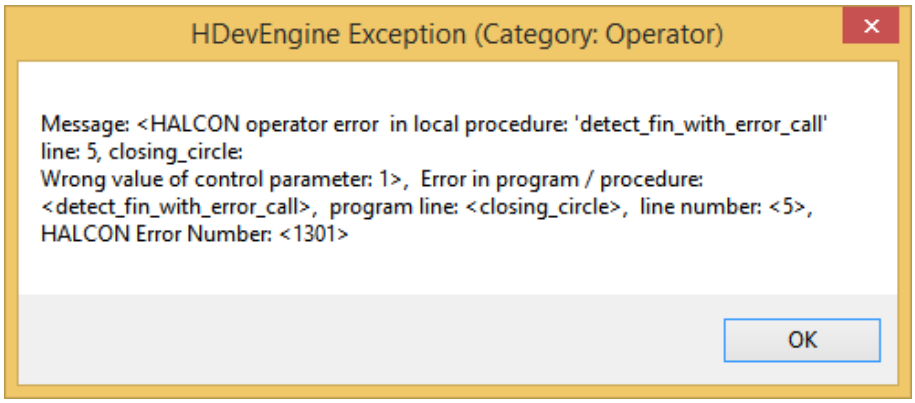


Figure 23.7: Content of the exception if an error occurred in a HALCON operator call.

With the method `UserData` (see section 25.1.7 on page 200), you can also access user exception data that is thrown within an HDevelop program or procedure by the operator `throw` similarly to the operator

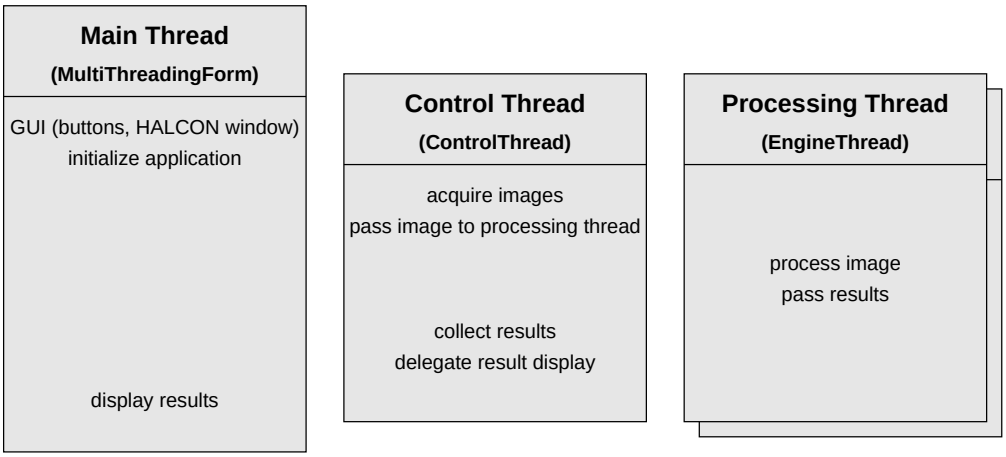


Figure 23.8: Tasks of the threads.

`dev_get_exception_data`.

In case of an exception (which is not caught within the procedure) the procedure call is cleaned up. This means all subthreads are destroyed and all values of input and output parameters are cleared. Therefore, we recommend that you always set all input parameters before executing a call even if some of them did not change.

Note that you can configure the behavior of HDevEngine when loading programs or procedures that contain invalid lines or unresolved procedure calls with the method `SetEngineAttribute` (see section 25.1.1 on page 188).

23.2.7 Creating Multithreaded Applications

HALCON provides two C# example applications that use multithreading with HDevEngine/.NET:

- In `MultiThreading`, the application is sped up by **executing the same HDevelop procedure in parallel** using two threads.
  This example is described in detail in section 23.2.7.1 on page 164.
- In contrast, `MultiThreadingTwoWindows` **executes different procedures in parallel**.
  This example is very similar to the previous one. Therefore, in section 23.2.7.2 on page 170 only the differences are described.

In the following, we briefly list the most important rules to observe when creating multithreaded HDevEngine applications. Also have a look at the general information about parallel programming using HALCON in section 2.2 on page 16, in particular the style guide in section 2.2.2 on page 17.

- When multiple threads execute HDevelop programs in parallel, **each thread must create its own instance of the corresponding** `clHDevProgramCallName`.
- **External procedure path and the implementation of HDevelop's display operators are always set globally** for all instances of HDevEngine. We recommend setting them via a separate HDevEngine instance to keep the code more readable.

23.2.7.1 Executing a Procedure in Parallel by Multiple Threads

The example application `MultiThreading` presented in this section exploits multi-core or multi-processor systems by executing the same HDevelop procedure (task) in parallel by two threads. The procedure finds bottle caps using shape-based matching.

Figure 23.8 on page 164 shows an overview of the structure of the application. It consists of four threads: The main thread (i.e., the form) is in charge of the graphical user interface (GUI), which is depicted in figure 23.9 on page 165. It consists of a HALCON window for the display of results and buttons to initialize, start, and stop the application.

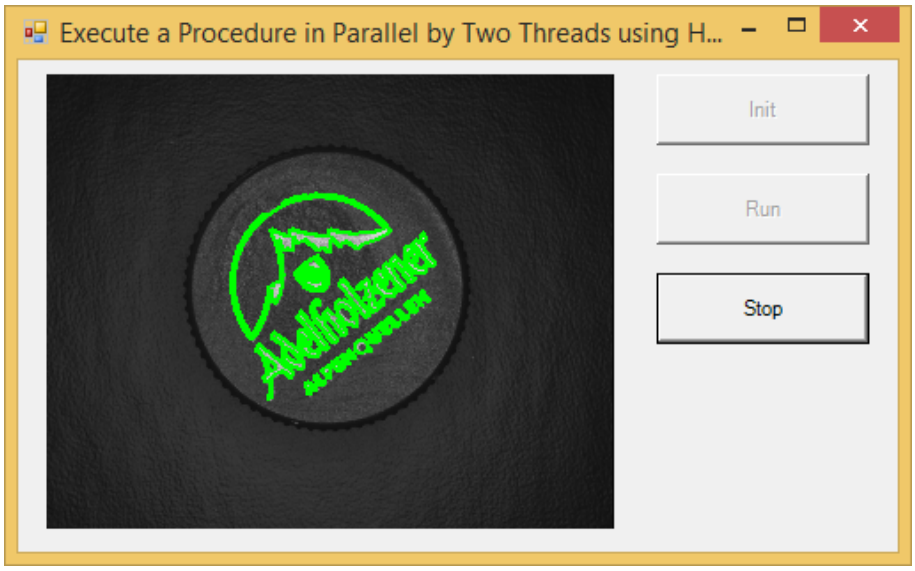


Figure 23.9: Screenshot of the application.

The main thread also initializes the application by training the shape model via an HDevelop procedure and by creating and initializing the other three threads: two processing threads and the so-called control thread, which controls the two processing threads.

The control thread acquires the images and passes them to the processing threads, which then process the images and pass back the results. The control thread collects the results, but does not display them itself, because all activities in the HALCON window must be performed by the thread that created it, i.e., the main thread.

Now, we take a closer look at the corresponding code. Please note that we do not show all details; in particular, error handling, and termination including memory management are left out.

Initialization

The application is initialized in the event handler of the `Init` button (file: `MultiThreadingForm.cs`).

```
private void InitButton_Click(object sender, System.EventArgs e)
```

Step 1: Switch off automatic operator parallelization

```
HOperatorSet.SetSystem("parallelize_operators", "false");
```

First, the automatic operator parallelization is switched off, otherwise the two mechanisms (multithreading and operator parallelization) would use more than the available number of cores / processors and thus slow down the application instead of speeding it up (see the style guide in section 2.2.2 on page 17). If you have a system with more than two cores or processors, you can consider to allocate some of them to the automatic operator parallelization as described in section 2.5.1 on page 19.

Step 2: Set external procedure path

Then, we create an instance of `clHDevEngineName` and set the path for searching the HDevelop procedures (code for constructing the path omitted). If the external procedure is from a procedure library, the external procedure path may include the name of the library file.

```
HDevEngine MyEngine = new HDevEngine();
string ProcedurePath = "/hdevengine/procedures";
  ...
MyEngine.SetProcedurePath(Path.GetFullPath(ExampleDir + ProcedurePath));
```

Step 3: Train the shape model

To initialize the image processing part, we execute an HDevelop procedure that trains the shape model of the caps.

```
HDevProcedureCall ProcTrain;

HDevProcedure Procedure = new HDevProcedure("train_shape_model");
ProcTrain = new HDevProcedureCall(Procedure);
ProcTrain.Execute();
```

Step 4: Store the model data

The procedure returns the handle of the shape model and the model contours. We store both in variables of the form so that the processing threads can access them.

```
public HTuple ModelID;
public HXLD ModelContours;

ModelID = ProcTrain.GetOutputCtrlParamTuple("ModelID");
ModelContours = ProcTrain.GetOutputIconicParamXld("ModelContours");
```

Step 5: Create and initialize the processing engines

The actual image processing is encapsulated in the class `EngineThread` (file: `EngineThread.cs`). The main members of this class are a thread and instances of `clHDevEngineName` and `clHDevProcedureCallName`. Besides, an `EngineThread` contains variables for accessing the shape model data trained in the main thread and an event that signals that the "engine" is ready for the next image.

```
public class EngineThread
{
  Thread WorkerObject = null;
  HDevProcedureCall ProcCall;
  HTuple ModelID;
  HXLD ModelContours;
  public AutoResetEvent EngineIsReady;

  public EngineThread(MultiThreadingForm mainForm)
  {
    ModelID = mainForm.ModelID;
    ModelContours = mainForm.ModelContours;
    EngineIsReady = new AutoResetEvent(true);
  }
```

The main thread creates and initializes two instances of this class and also stores their events (file: `MultiThreadingForm.cs`).

```
EngineThread WorkerEngine1;  // Processing thread.
EngineThread WorkerEngine2;  // Processing thread.
AutoResetEvent Engine1Ready;
AutoResetEvent Engine2Ready;

WorkerEngine1 = new EngineThread(this);
WorkerEngine1.Init();
Engine1Ready = WorkerEngine1.EngineIsReady;

WorkerEngine2 = new EngineThread(this);
WorkerEngine2.Init();
Engine2Ready = WorkerEngine2.EngineIsReady;
```

An `EngineThread` initializes itself by creating the procedure call for detecting the caps in the images. Because the input parameters of the procedure that concern the shape model are the same for each call, they can be set once in advance (file: `EngineThread.cs`).

```
public void Init()
{
  HDevProcedure Procedure = new HDevProcedure("detect_shape");
  ProcCall = new HDevProcedureCall(Procedure);
  ProcCall.SetInputCtrlParamTuple("ModelID", ModelID);
  ProcCall.SetInputIconicParamObject("ModelContours", ModelContours);
}
```

Step 6: Initialize image acquisition

Finally, we initialize the image acquisition. The handle is stored in a variable of the form, so that the control thread can access it (file: `MultiThreadingForm.cs`).

```
private HFramegrabber AcqHandle;

string ImagePath = Path.GetFullPath(ExampleDir + "/images/cap_illumination");
AcqHandle = new HFramegrabber("File", 1, 1, 0, 0, 0, 0, "default", -1,
  "default", -1, "default", ImagePath, "default", -1, -1);
```

Image Processing

When you click the `Run` button, the application starts to process images in a loop.

Step 1: Starting the processing threads and the control thread

First, the main thread starts the processing engines (file: `MultiThreadingForm.cs`).

```
private void RunButton_Click(object sender, System.EventArgs e)
{
  WorkerEngine1.Run();
  WorkerEngine2.Run();
```

The corresponding method creates and starts their thread and sets the “ready” signal (file: `EngineThread.cs`).

```
public void Run()
{
  EngineIsReady.Set();
  WorkerObject = new Thread(new ThreadStart(Process));
  WorkerObject.Start();
}
```

Then, the main thread starts the control thread (file: `MultiThreadingForm.cs`):

```
ControlThread = new Thread(new ThreadStart(Run));
ControlThread.Start();
```

Step 2: Triggering the processing threads from the control thread

The control thread’s action is contained in the method `Run` (file: `MultiThreadingForm.cs`). As long as the `Stop` is not pressed (please take a look at the project’s code for more information), it waits until one of the processing engine is ready.

```
EngineThread WorkerEngine;   // Variable to switch between processing threads.

public void Run()
{
  HImage Image;

  while (!StopEventHandle.WaitOne(0, true))
  {
    if (Engine1Ready.WaitOne(0, true))
      WorkerEngine = WorkerEngine1;
    else if (Engine2Ready.WaitOne(0, true))
      WorkerEngine = WorkerEngine2;
    else
      continue;

    Image = AcqHandle.GrabImageAsync(-1);
    WorkerEngine.SetImage(Image);
```

Then, it acquires the next image and passes it to the engine, which stores it in a member variable (file: `EngineThread.cs`).

```
private HImage InputImage = null;

public void SetImage(HImage Img)
{
  InputImage = Img;
}
```

Step 3: Processing the image

In their action method (`Process`), the processing threads wait for the image to be set (file: `EngineThread.cs`). The actual image processing is performed by the HDevelop procedure, passing the image as input parameter.

```
public void Process()
{
  while (!DelegatedStopEvent.WaitOne(0, true))
  {
    if (InputImage == null)
      continue;

    ProcCall.SetInputIconicParamObject("Image", InputImage);
    ProcCall.Execute();
```

Step 4: Passing the results to the control thread

To pass the results, a class is defined that stores the relevant data: the processed image and the position, orientation, and the contours of the found cap.

```
public class ResultContainer
{
  public HImage InputImage;
  public HXLD FoundContours;
  public double Row;
  public double Column;
  public double Angle;
}
```

After executing the procedure, the processing thread accesses its results and stores them in a new instance of the result class (“result container”), together with the processed image.

```
ResultContainer Result;
HTuple ResultTuple;

Result = new ResultContainer();
Result.InputImage = InputImage;
Result.FoundContours = ProcCall.GetOutputIconicParamXld("ResultObject");
ResultTuple = ProcCall.GetOutputCtrlParamTuple("ResultData");
Result.Row = ResultTuple[0];
Result.Column = ResultTuple[1];
Result.Angle = ResultTuple[2];
```

The processing thread then passes the result container to the control thread by appending it to a list.

```
ResultMutex.WaitOne();
ResultList.Add(Result);
ResultMutex.ReleaseMutex();
```

This list is a member variable of the main thread (file: `MultiThreadingForm.cs`). It is protected by a mutex so that the threads can access it safely.

```
public ArrayList ResultList;
public Mutex ResultDataMutex;

public MultiThreadingForm()
{
  ResultDataMutex = new Mutex();
  ResultList = new ArrayList();
}
```

The processing threads store references to the list and to the mutex in own member variables (file: `EngineThread.cs`).

```
ArrayList ResultList;
Mutex ResultMutex;

public EngineThread(MultiThreadingForm mainForm)
{
  ResultList = mainForm.ResultList;
  ResultMutex = mainForm.ResultDataMutex;
}
```

Step 5: "Ready again"

Finally, the processing thread signals that it is ready for the next image by setting the corresponding event and by setting the input image to `null`.

```
InputImage = null;
this.EngineIsReady.Set();
```

Result Display

Step 1: Checking whether new results are available

Let's return to the action method (`Run`) of the control thread (file: `MultiThreadingForm.cs`). After triggering a processing thread by passing the image to process, it checks whether the result list contains new items.

```
int Count = -1;

ResultDataMutex.WaitOne();
Count = ResultList.Count;
ResultDataMutex.ReleaseMutex();
```

Step 2: Delegating the display

The control thread does not perform the display of results itself but delegates it to the main thread (running the form) with the method `Invoke`.

```
for (; Count > 0; Count--)
  Invoke(DelegatedDisplay);
```

The necessary members are defined by the form.

```
delegate void FuncDelegate();
FuncDelegate DelegatedDisplay;

public MultiThreadingForm()
{
  DelegatedDisplay = new FuncDelegate(DisplayResults);
}
```

Note that all HALCON visualization operators are automatically delegated to the correct thread as described in chapter section 2.3 on page 18.

Step 3: Displaying the results

The actual display is performed by the method `DisplayResults`. Each time it is called, it removes an item from the result list and displays the processed image with the contours of the found cap. Then, it frees the corresponding HALCON-internal memory.

```
public void DisplayResults()
{
  ResultDataMutex.WaitOne();
  Result = (ResultContainer)ResultList[0];
  ResultList.Remove(Result);
  ResultDataMutex.ReleaseMutex();

  Window.ClearWindow();
  Window.DispImage(Result.InputImage);
  Window.DispObj(Result.FoundContours);

  Result.InputImage.Dispose();
  Result.FoundContours.Dispose();
}
```

23.2.7.2 Executing Multiple Procedures in Parallel by Multiple Threads

In contrast to the previous section, the example application `MultiThreadingTwoWindows` presented here executes different HDevelop procedures (tasks) in parallel by two threads. One task is to find bottle caps using shape-based matching, the other to read ECC 200 data codes.

Figure 23.10 on page 171 shows an overview of the structure of the application. Like the application described in the previous section, it consists of four threads: The main thread (i.e., the form) is in charge of the graphical user interface (GUI), which is depicted in figure 23.9 on page 165. It consists of a HALCON window for the display of results and buttons to initialize, start, and stop the application.

The main thread also initializes the application by creating and initializing the other three threads: two processing threads and the so-called control thread, which controls the two processing threads. In contrast to the previous application, here the processing threads initialize the image processing tasks by training the shape model and the data code model, respectively, via HDevelop procedures.

The control thread acquires the images and passes them to the processing threads, which then process the image and pass back the results. The control thread collects the results, but does not display them itself, because all activities in the HALCON window must be performed by the thread that created it, i.e., the main thread. In contrast to the previous application the results of the two tasks are displayed in two separate windows.

Below, we take a closer look at the corresponding code, restricting ourselves, however, to the parts that are different to the previous application.

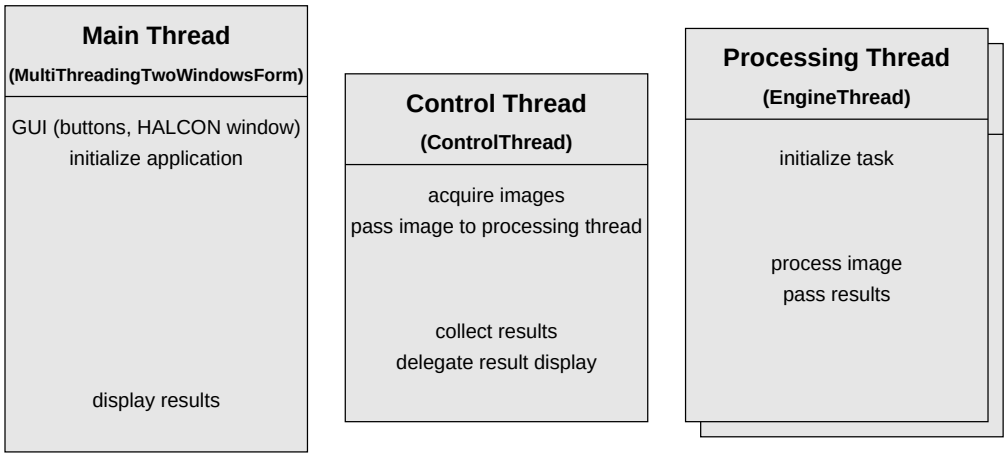


Figure 23.10: Tasks of the threads.

Figure 23.11: Screenshot of the application.

Initialization

As in the previous example, the application is initialized in the event handler of the `Init` button (file: `MultiThreadingTwoWindowsForm.cs`).

Step 1: Create and initialize the processing engines

The processing engines are created and initialized similarly to the previous example, with some exceptions: First, the shape and the data code model are now trained by the processing threads instead of the control thread (see the step below). Secondly, the processing engines now also have a variable that indicates "their" HALCON window (file: `EngineThread.cs`).

```
public class EngineThread
{
  ...
  public int WindowIndex = -1;
  ...
```

The control thread sets this variable after creating the engines (file: `MultiThreadingTwoWindowsForm.cs`).

```
private void InitButton_Click(object sender, System.EventArgs e)
{
  ...
  WorkerEngine1.WindowIndex = 1;
  ...
  WorkerEngine2.WindowIndex = 2;
```

Step 2: Train the shape and data code model

The training of the shape and data code model is now performed by the initialization method of the processing threads, which now has a parameter that specifies the task of the processing thread (file: `MultiThreadingTwoWindowsForm.cs`).

```
WorkerEngine1.Init("shape");
  ...
WorkerEngine2.Init("datacode");
```

The HDevelop procedures for training the models and for performing the image processing have similar names for the two tasks, so that their names can be generated automatically (file: `EngineThread.cs`). The task name itself is stored in a variable of the class `EngineThread`.

```
public class EngineThread
{
  HDevProcedureCall ProcCall;
  string Task;
  HTuple ModelID;
  HXLD ModelContours;
  ...

public void Init(string Task)
{
  string TrainMethod = "train_" + Task + "_model";
  string ProcessingMethod = "detect_" + Task;
  HDevProcedureCall ProcTrain;

  this.Task = Task;
```

Then, the model of the shape or datacode, respectively, is trained by executing the corresponding HDevelop procedure and the returned model data is stored in variables of the class.

```
HDevProcedure Procedure = new HDevProcedure(TrainMethod);
ProcTrain = new HDevProcedureCall(Procedure);
ProcTrain.Execute();
ModelID = ProcTrain.GetOutputCtrlParamTuple("ModelID");
if (Task.Equals("shape"))
{
  ModelContours = ProcTrain.GetOutputIconicParamXld("ModelContours");
}
```

Step 3: Store the model data

Finally, those input parameters of the image processing procedure that are the same for each call are set (file: `EngineThread.cs`).

```
HDevProcedure Procedure = new HDevProcedure(ProcessingMethod);
ProcCall = new HDevProcedureCall(Procedure);
ProcCall.SetInputCtrlParamTuple("ModelID", ModelID);
if (Task.Equals("shape"))
{
  ProcCall.SetInputIconicParamObject("ModelContours", ModelContours);
}
```

Step 4: Initialize image acquisition

The two image processing tasks are performed in different images, therefore, two image acquisition devices are opened by the main thread (file: `MultiThreadingTwoWindowsForm.cs`, code not shown).

Image Processing

Step 1: Triggering the processing threads

The control thread's action is contained in the method `Run` (file: `MultiThreadingTwoWindowsForm.cs`). As long as the `Stop` is not pressed, it checks whether the processing engines are ready and, if this is the case, acquires and passes images..

```
public void Run()
{
  HImage Image;

  while (!StopEventHandle.WaitOne(0, true))
  {
    if (Engine1Ready.WaitOne(0, true))
    {
      Image = AcqHandle1.GrabImageAsync(-1);
      WorkerEngine1.SetImage(Image);
    }
    if (Engine2Ready.WaitOne(0, true))
    {
      Image = AcqHandle2.GrabImageAsync(-1);
      WorkerEngine2.SetImage(Image);
    }
```

Step 2: Passing the results to the control thread

The class storing the result data differs significantly from the one in the previous example: It now also contains a variable that indicates the window in which to display the results and a flag that shows whether the processing was successful. Because the processing results differ between the two tasks, they are encapsulated in a tuple (file: `EngineThread.cs`).

```
public class ResultContainer
{
  public int WindowIndex; // 1 -> shape, 2 -> datacode.
  public HImage InputImage;
  public HXLD FoundContours;
  public HTuple ResultData;
  public bool DetectionSuccessful;
}
```

After executing the procedure, the processing thread accesses its results and stores them in a new instance of the result container, together with the processed image and the window index.

```
public void Process()
{
  ResultContainer Result;

  Result = new ResultContainer();
  ...
  Result.InputImage = InputImage;
  DetectionSuccessful = ProcCall.GetOutputCtrlParamTuple("DetectionSuccessful").S;
  if (DetectionSuccessful.Equals("true"))
  {
    Result.DetectionSuccessful = true;
    Result.FoundContours = ProcCall.GetOutputIconicParamXld("ResultObject");
    Result.ResultData = ProcCall.GetOutputCtrlParamTuple("ResultData");
  }
  else
  {
    Result.DetectionSuccessful = false;
  }
  Result.WindowIndex = WindowIndex;
```

Result Display

As in the previous example, the display of results is performed by the main thread in the method `ResultDisplay` (file: `MultiThreadingTwoWindowsForm.cs`). The main difference is that the display now is switched between the two HALCON windows, based on the variable in the result container.

```
public void DisplayResults()
{
  HWindow Window;

  if (Result.WindowIndex == 1)
  {
    Window = Window1;
  }
  else
  {
    Window = Window2;
  }
```

Furthermore, the display method now checks the success of the image processing to avoid accessing non-existing result elements. For both tasks, the resulting contours, i.e., the found shape or data code region, respectively, are displayed. For the data code task, also the read code is displayed.

```
Window.ClearWindow();
Window.DispImage(Result.InputImage);
if (Result.DetectionSuccessful)
{
  Window.DispObj(Result.FoundContours);
  // Additional display for data code result: code.
  if (Result.WindowIndex == 2)
  {
    Row = (int)Result.ResultData[0].D;
    Col = (int)Result.ResultData[1].D;
    Window.SetTposition(Row, Col);
    Window.WriteString((string)Result.ResultData[2].S);
  }
}
else
{
  Window.SetColor("red");
  Window.SetTposition(20, 20);
  Window.WriteString("Detection failed!");
  Window.SetColor("green");
}
```

23.2.8 Executing an HDevelop Program with Vector Variables

The example application `UseVectorVariables` shows how to load and execute an HDevelop example that contains vector variables in HDevengine/C#. In the example two vectors are used for processing: one containing the input images and one containing scaling factors. When executing the program the gray values of the input images are scaled according to the scaling factors. Please have a look at the source file `UseVectorVariablesForm.cs` for more details on how to work with vector variables in HDevengine/.NET.

23.3 Using the Just-in-time Compiler With HDevEngine/.NET

The just-in-time compilation of procedures needs to be enabled in your instance of the `clHDevEngineName` class:

```
...
Engine = new HDevEngine();

// Enable or disable execution of compiled procedures.
Engine.SetEngineAttribute("execute_procedures_jit_compiled", "true");
```

Procedures (and procedures referenced by it) are compiled at the moment a corresponding instance of `clHDevProcedureCallName` or `clHDevProgramCallName` is created.

You can also explicitly pre-compile all used procedures of a HDevelop program or procedure using the method `CompileUsedProcedures` of `clHDevProgramName` or `clHDevProcedureName`, respectively.

In the following example, all used procedures of a procedure call are just-in-time compiled:

```
Program = new HDevProgram(ProgramPathString);
// Get local procedure.
Proc = new HDevProcedure(Program, name);
...
Proc.CompileUsedProcedures();
```

Chapter 24

HDevEngine In Python Applications

This chapter explains how to use HDevEngine in Python applications. Section 24.1 on page 177 quickly summarizes some basic information about creating HDevEngine applications with Python. Section 24.3 on page 178 provides a more comprehensive overview.

24.1 Introduction

The HDevEngine/Python interface is similar to the HDevEngine/.NET and HDevEngine/C++ interface. Most concepts and functions have a one-to-one mapping.

A short reference of the C++ classes for the HDevEngine can be found in section 25.1 on page 187. The Python classes are similar.

This chapter only describes HDevEngine/Python specifics. If a topic e.g., error handling is not mentioned, the relevant part in Part IV on page 91 applies.

24.1.1 A First Example

This section explains how to create a simple HDevEngine/Python application. For a more comprehensive description, read section 24.3 on page 178.

1. Install HALCON 25.11.
2. Install Python 3.8 or newer on your system.
3. Set up your Python environment of choice, e.g., using `python -m venv path_to_new_virtual_environment`
4. Run the following commands in a shell:

```
mkdir hdevengine_example
cd hdevengine_example
pip install mvtec-halcon==25110
```

5. Create a file named `hdevengine_example.py` and change the content to:

```
import os

import halcon as ha

if __name__ == '__main__':
    example_dir = ha.get_system_s('example_dir')
    procedure_path = os.path.join(example_dir, 'hdevengine', 'procedures')

    hdev_engine = ha.HDevEngine()
    hdev_engine.set_procedure_path(procedure_path)

    img = ha.read_image('fin2')

    procedure = ha.HDevProcedure.load_external('detect_fin')
    proc_call = ha.HDevProcedureCall(procedure)

    proc_call.set_input_iconic_param_by_name('Image', img)
    proc_call.execute()

    fin_area = proc_call.get_output_control_param_by_name('FinArea')[0]
    print(f'Fin Area: {fin_area}')
```

6. To run the application, type the following command in the same shell:

```
python hdevengine_example.py
```

As as result, you should see the following output `'Fin Area:  1634'`.

24.2 Creating Applications With HDevEngine/Python

24.2.1 Adding HDevEngine/Python to a Python Application

No HDevEngine-specific actions are required. For more information, see section 14.2 on page 93.

24.3 HDevEngine/Python Interface

24.3.1 Global Functionality

While you can have multiple Python instances of HDevEngine/Python, they all share the same implementation as a mutable singleton.

This mostly affects configuration. Specific program and procedure call instances are more or less independent from each other.

Here HALCON is queried for the example directory path, based on which the path containing the `'detect_fin'` procedure is constructed. Then an `HDevEngine` instance is initialized and the procedure search path set to it.

```
example_dir = ha.get_system_s('example_dir')
procedure_path = os.path.join(example_dir, 'hdevengine', 'procedures')

hdev_engine = ha.HDevEngine()
hdev_engine.set_procedure_path(procedure_path)
```

24.3.1.1 Attributes

```
def set_attribute(self, name: str, value: HTupleType) -> None:
def get_attribute(self, name: str) -> HTupleType:
```

These functions let you set specific attributes. For example, by default, the attribute 'ignore_invalid_results' is set to true. This means that by default, an empty region object or an empty tuple is returned if a variable or parameter is queried that was not previously set by the program or procedure. Think of it like this: Should accessing an uninitialized variable return a default or should it raise an exception?

```
hdev_engine = ha.HDevEngine()
assert hdev_engine.get_attribute('ignore_invalid_results') == 1

hdev_engine.set_attribute('ignore_invalid_results', 0)
assert hdev_engine.get_attribute('ignore_invalid_results') == 0
```

For HDevEngine attributes, 0 maps to false, and 1 to true.

See section 25.1 on page 187 for a list of available attributes.

24.3.1.2 Debug Server

```
HDevEngine

def start_debug_server(self) -> None:
def stop_debug_server(self) -> None:
```

These functions let you start and stop the server required for debugging the engine execution. You can control the port via the attribute 'debug_port'.

24.3.1.3 Procedure Search Paths

```
HDevEngine

def set_procedure_path(self, path: str) -> None:
def add_procedure_path(self, path: str) -> None:
```

These functions let you set and extend the list of paths searched when trying to load a procedure.

24.3.1.4 Global Metadata

```
HDevEngine

def get_procedure_names(self) -> List[str]:
def get_loaded_procedure_names(self) -> List[str]:
def get_global_control_var_names(self) -> List[str]:
def get_global_iconic_var_dimension(self, name: str) -> int:
def get_global_control_var_dimension(self, name: str) -> int:
```

These functions let you query global metadata, such as which procedures are loaded, the vector dimensions of global variables, and more.

24.3.1.5 Reading and Writing Global Variables

```
HDevEngine

def set_global_iconic_var(self, name: str, value: HObject) -> None:
def set_global_iconic_vector_var(self, name: str, value: IconicVectorType) -> None:
def set_global_control_var(self, name: str, value: HTupleType) -> None:
def set_global_tuple_vector_var(self, name: str, value: TupleVectorType) -> None:
def get_global_iconic_var(self  , name: str) -> HObject:
def get_global_iconic_vector_var(self, name: str) -> TupleVectorType:
def get_global_control_var(self, name: str) -> HTupleType:
def get_global_tuple_vector_var(self, name: str) -> TupleVectorType:
```

These functions let you read and write global variables, both iconic and control.

24.3.2 Calling HDevelop Procedures

The general concept behind calling HDevelop procedures from Python is as follows:

1. Initialize an `HDevProcedure` instance by loading a procedure from the file system.
2. Create a specific `HDevProcedureCall` instance from the `HDevProcedure` instance.
3. Set iconic and control input parameters.
4. Call `execute`.
5. Read output parameters.

```
procedure = ha.HDevProcedure.load_external('detect_fin')
proc_call = ha.HDevProcedureCall(procedure)

proc_call.set_input_iconic_param_by_name('Image', img)
proc_call.execute()

fin_area = proc_call.get_output_control_param_by_name('FinArea')[0]
print(f'Fin Area: {fin_area}')
```

24.3.2.1 Loading Procedures

```
HDevProcedure

@staticmethod
def load_external(name: str) -> 'HDevProcedure':

@staticmethod
def load_local(
    program: Union[HDevProgram, str],
    name: str
) -> 'HDevProcedure':
```

Procedures are loaded via these static methods. Use only these functions to initialize `HDevProcedure` instances.

```
hdev_engine.set_procedure_path(proc_dir)

external_proc = ha.HDevProcedure.load_external('detect_fin')

program = ha.HDevProgram(os.path.join(proc_dir, 'program.hdev'))
local_proc_program = ha.HDevProcedure.load_local(program, 'count_nuts')
local_proc_name = ha.HDevProcedure.load_local('program.hdev', 'count_nuts')
```

While Python allows calling static methods on instances, e.g.: `ha.HDevProcedure().load_external` this is not recommended as it needlessly wastes resources.

24.3.2.2 Unloading Procedures

```
HDevEngine

def unload_procedure(self, name: str) -> None:
def unload_all_procedures(self) -> None:
```

These functions let you unload procedures again, once they are loaded. For example, this can be useful to free up unused memory.

24.3.2.3 Procedure Metadata

Access procedure metadata via these read-only member variables:

```
HDevProcedure

- name : str
- short_description : str
- loaded : bool
- input_iconic_param_names : List[str]
- output_iconic_param_names : List[str]
- input_control_param_names : List[str]
- output_control_param_names : List[str]
- input_iconic_param_dimensions : List[int]
- output_iconic_param_dimensions : List[int]
- input_control_param_dimensions : List[int]
- output_control_param_dimensions : List[int]
```

```
HDevProcedure

def get_used_procedure_names(self) -> List[str]:
def get_info(self, slot: str) -> HTupleType:
def get_param_info(self, name: str, slot: str) -> HTupleType:
def get_input_iconic_param_info(self, idx: int, slot: str) -> HTupleType:
def get_output_iconic_param_info(self, idx: int, slot: str) -> HTupleType:
def get_input_control_param_info(self, idx: int, slot: str) -> HTupleType:
def get_output_control_param_info(self, idx: int, slot: str) -> HTupleType:
def query_slots(self) -> List[str]:
def query_param_slots(self) -> List[str]:
```

Use these functions to query procedure metadata.

24.3.2.4 JIT Compiling

```
HDevProcedure

def compile_used_procedures(self) -> bool:
```

Compile all procedures that are used by the program and that can be compiled with a just-in-time compiler. Procedures that could not be compiled are called by the HDevEngine interpreter in the usual way. To check which procedure could not be compiled and what the reason is for that, start HDevelop and check the compilation states there.

This functions returns whether all used procedures were JIT compiled.

24.3.2.5 Procedure Call Initialization

```
proc_call = ha.HDevProcedureCall(procedure)
```

The only way to initialize an `HDevProcedureCall` instance is using a loaded `HDevProcedure` instance.

24.3.2.6 Setting Input Parameters

```
HDevProcedureCall

def set_input_control_param_by_index(self, idx: int, value: HTupleType) -> None:
def set_input_tuple_vector_by_index(self, idx: int, value: TupleVectorType) -> None:
def set_input_control_param_by_name(self, name: str, value: HTupleType) -> None:
def set_input_tuple_vector_by_name(self, name: str, value: TupleVectorType) -> None:
def set_input_iconic_param_by_index(self, idx: int, value: HObject) -> None:
def set_input_iconic_vector_by_index(self, idx: int, value: IconicVectorType) -> None:
def set_input_iconic_param_by_name(self, name: str, value: HObject) -> None:
def set_input_iconic_vector_by_name(self, name: str, value: IconicVectorType) -> None:
```

These functions let you set input parameters. We recommend always setting all input parameters again every time you want to execute the procedure call. This helps make your code more robust in the face of error conditions.

Indices here start at 1 instead of 0.

24.3.2.7 Execution

```
HDevProcedureCall

def execute(self) -> None:
```

Use this function when you have set all input parameters to execute the procedure.

24.3.2.8 Reading Output Parameters

```
HDevProcedureCall

def get_output_control_param_by_index(self, idx: int) -> HTupleType:
def get_output_tuple_vector_by_index(self, idx: int) -> TupleVectorType:
def get_output_control_param_by_name(self, name: str) -> HTupleType:
def get_output_tuple_vector_by_name(self, name: str) -> TupleVectorType:
def get_output_iconic_param_by_index(self, idx: int) -> HObject:
def get_output_iconic_vector_by_index(self, idx: int) -> IconicVectorType:
def get_output_iconic_param_by_name(self, name: str) -> HObject:
def get_output_iconic_vector_by_name(self, name: str) -> IconicVectorType:
```

Use these functions to read output parameters after the call to `execute` has finished successfully.

Indices here start at 1 instead of 0.

24.3.2.9 Waiting for the Debugger

```
HDevProcedureCall

def wait_for_debug_connection(self) -> None:
```

Use this function in conjunction with `start_debug_server` to debug procedure execution.

24.3.2.10 Reseting

```
HDevProcedureCall

def reset(self) -> None:
```

This is mainly for situations when you want to abort execution from another thread or possibly free native resources even while some instances are still alive.

24.3.3 Calling HDevelop Programs

The general concept behind calling HDevelop programs from Python is as follows:

1. Initialize an `HDevProgram` instance by loading a program from the file system.
2. Create a specific `HDevProgramCall` instance from the `HDevProgram` instance.
3. Call `execute`.
4. Read variables.

```
example_dir = ha.get_system_s('example_dir')
program_path = os.path.join(
    example_dir,
    'hdevelop',
    'Transformations',
    'Poses'
)

program = ha.HDevProgram(os.path.join(program_path, 'pose_compose.hdev'))
program_call = ha.HDevProgramCall(program)
program_call.execute()

pose = program_call.get_control_var_by_name('PoseComposeAlternative')
rounded_pose = [round(x, 8) for x in pose]

assert rounded_pose == [0.3, -0.0498838, 0.33986422, 77.0, 90.0, 0.0, 0]
```

24.3.3.1 Loading Programs

```
program = ha.HDevProgram(os.path.join(program_path, 'pose_compose.hdev'))
```

Programs are loaded via the initializer of `HDevProgram`, which expects a full path to an HDevelop program in the format of the operating system, including file name.

24.3.3.2 Program Metadata

Access program metadata via these read-only member variables:

```
HDevProgram

- name : str
- loaded : bool
- inconic_var_names : List[str]
- control_var_names : List[str]
- inconic_var_dimensions : List[int]
- control_var_dimensions : List[int]
```

```
HDevProgram

def get_used_procedure_names(self) -> List[str]:
def get_local_procedure_names(self) -> List[str]:
```

Use these functions to query program metadata.

24.3.3.3 JIT Compiling

```
HDevProgram

def compile_used_procedures(self) -> bool:
```

Compile all procedures that are used by the program and that can be compiled with a just-in-time compiler. Procedures that could not be compiled are called by the HDevEngine interpreter in the usual way. To check which procedure could not be compiled and what the reason is for that, start HDevelop and check the compilation states there.

Returns whether all used procedures were JIT compiled.

24.3.3.4 Program Call Initialization

```
program_call = ha.HDevProgramCall(program)
```

The only way to initialize an `HDevProgramCall` instance is using a valid `HDevProgram` instance.

24.3.3.5 Execution

```
HDevProgramCall

def execute(self) -> None:
```

Use this function to execute the program.

24.3.3.6 Reading Variables

```
HDevProgramCall

def get_control_var_by_index(self, idx: int) -> HTupleType:
def get_tuple_vector_var_by_index(self, idx: int) -> TupleVectorType:
def get_control_var_by_name(self, name: str) -> HTupleType:
def get_tuple_vector_var_by_name(self, name: str) -> TupleVectorType:
def get_iconic_var_by_index(self, idx: int) -> HObject:
def get_iconic_vector_var_by_index(self, idx: int) -> IconicVectorType:
def get_iconic_var_by_name(self, name: str) -> HObject:
def get_iconic_vector_var_by_name(self, name: str) -> IconicVectorType:
```

Use these functions to read variables after the call to `execute` has finished successfully.

Indices here start at 1 instead of 0.

24.3.3.7 Waiting for the Debugger

```
HDevProgramCall

def wait_for_debug_connection(self) -> None:
```

Use this function in conjunction with `start_debug_server` to debug program execution.

24.3.3.8 Reseting

```
HDevProgramCall

def reset(self) -> None:
```

This is mainly for situations when you want to abort execution from another thread or possibly free native resources even while some instances are still alive.

24.3.4 Dev Operators

Inside HDevelop, `dev_*` operators can be used for convenience. When embedding an HDevelop program or procedure inside your application, the potential `dev_*` operator calls have no straightforward mapping. It might be desirable to for example map `dev_*` operator calls to visualization within your application. Thus, HDevEngine/Python provides a base class, which you can inherit from and overwrite with logic appropriate for your application.

Take for example this simple HDevelop program, which reads an image and then displays this PCB image in a dev window.

```
read_image(Image, 'pcb')
dev_display(Image)
```

By default, this program will not open a window when using HDevEngine/Python. However, you can specify your own logic of what should happen when `dev_display` is called.

For example, this Python code registers a very basic implementation of `dev_display`.

```
class DevImpl(ha.HDevOperatorBase):
    @staticmethod
    def dev_display(object):
        print(object)

hdev_engine = ha.HDevEngine()
hdev_engine.set_hdev_operator_impl(DevImpl())
```

To get an overview which `dev_*` operators are available and which signatures each of them expects, take a look at the implementation of `HDevOperatorBase`.

Note that only one implementation can be registered at a time. This applies to all current and future `HDevEngine` instances, until changed.

By default no implementation is registered. Once you have registered an implementation you can return to the default behavior by calling:

```
hdev_engine.unset_hdev_operator_impl()
```

Note that calling `register_dev_operators` might have surprising lifetime effects on your Python variables. See the documentation of said function for more details.

Every function that was not overwritten by your implementation of `HDevOperatorBase` will raise an exception if called.

It is safe to raise exceptions from within Python code implementing `dev_*` operators. However, only a generic `HDevEngineError` will be raised and the original exception traceback will be logged to stderr. This is due to technical limitations.

24.3.5 HALCON Vectors

HDevEngine/Python maps HALCON vectors to nested Python lists. There are two separate, incompatible types of vectors: `tuple_vector` and `iconic_vector`. The following examples demonstrate the Python representation that is used to read and write HALCON vectors in HDevEngine/Python:

```
empty_tuple_vector = ha.HDevEmptyVector(dimension=1)
1d_tuple_vector = [[23, 'a'], ['ec', 2.5, 77]]
2d_tuple_vector = [[[8], ['b', 'c']], [], [[]]]

empty_iconic_vector = ha.HDevEmptyVector(dimension=1)
1d_iconic_vector = [img1, img2, img3]
2d_iconic_vector = [[img1, img2], [img1, img3, img4], []]
```

For `tuple_vector`, the inner most list is treated as HALCON tuple.

All elements of a vector must have the same dimension:

```
[ [234], [2, 5] ] # ok
[ 234, [2, 5] ] # not ok
```

24.3.6 Multithreading

Because HDevEngine is a mutable singleton, some functionality affects your entire application:

1. Global functionality: See section 24.3.1 on page 178.
2. Loading programs and procedures.
3. Registering `dev_*` operator implementations.

A multi-threaded application must take care when working with this sort of shared mutable state. For example, you cannot safely load multiple procedures in parallel using the global procedure path if they require different path settings, without synchronizing appropriately inside your application. Not applying the required care will lead to undefined behavior, including crashes and worse.

To avoid the aforementioned issues, we recommend configuring HDevEngine, as well as loading procedures and programs, at the beginning of your program before starting additional application threads.

In contrast to `HDevEngine`, `HDevProcedure`, and `HDevProgram`, the call instances `HDevProcedureCall` and `HDevProgramCall` are independent from each other.

Chapter 25

General Information

This chapter contains an overview about the main classes of HDevEngine and their methods (section 25.1 on page 187) and miscellaneous application tips (section 25.3 on page 203). Remote debugging of HDevEngine applications from HDevelop is described in section 25.2 on page 201.

25.1 Overview of the Classes

Note in the following, we print only the declaration of the classes for HDevEngine/C++. In the other variants of HDevEngine, the methods and properties have the same names.

25.1.1 clHDevEngineName

.NET: clHDevEngineName

```
 ****************************************************************************
 ** class HDevEngine
 **==========================================================================
 ** Class for managing global engine settings:
 **  + external procedure path
 **  + implementation of dev_ operators (HDevOperatorImpl)
 **  + Attention: all changes made to one HDevEngine instance are global
 **    for all .dev programs or .dvp procedure that are executed in one
 **    application
 ****************************************************************************
 ****************************************************************************/
class LIntExport HDevEngine
{
public:
  HDevEngine();

  // Via engine attributes the behavior of the engine can be configured
  // currently the following flags are supported:
  //  "ignore_unresolved_lines"  [default: false, 0]
  //     - if set to true (or "true"), program lines that refer to an
  //       unresolved procedure are ignored, i.e., the program or procedure is
  //       executed without the corrupted program line;
  //       this may lead to an unexpected behavior or an error during the
  //       program execution
  //     - as the default an exception is thrown while creating the program or
  //       procedure instance
  //  "ignore_invalid_lines"  [default: false, 0]
  //     - if set to true (or "true"), invalid program lines  are ignored,
  //       i.e., the program or procedure is executed without the corrupted
  //       program line;
  //       this may lead to an unexpected behavior or an error during the
  //       program execution
  //     - as the default an exception is thrown while creating the program or
  //       procedure instance
  //  "ignore_invalid_results"  [default: true, 1]
  //     - if set to false (or "false") throw an exception if the accessed
  //       procedure output parameter or program variable is invalid
  //     - the following methods are concerned:
  //         HenProgramCall::GetIconicVarObject()
  //         HenProgramCall::GetCtrlVarTuple()
  //         HenProcedureCall::GetOutputIconicParamObject()
  //         HenProcedureCall::GetOutputCtrlParamTuple()
  //     - as the default an empty region object or an empty tuple is returned
  //       if the object was not set within the program or procedure
```

(continued on next page)

(continued declaration of clHDevEngineName)

```
  //  "docu_language"  [default: "" -> en_US]
  //     - could be set to "en_US","de_DE",  other languages
  //  "docu_encoding"  [default: "" -> "utf8"]
  //     - if set to "native" all natural language strings are converted
  //       to native encoding
  //  "execute_procedures_jit_compiled"  [default: false, 0]
  //     - if set to true (or "true"), procedures are tried to being compiled
  //       with a just-in-time compiler for faster execution
  //  "debug_port"  [default: 57786]
  //     - specifies the port number of the socket where the debug server
  //       waits for incoming connections
  //  "debug_password"  [default: ""]
  //     - specifying a password provides a basic layer of protection
  //       against misuse. For security reasons, it is highly recommended
  //       to always supply a password. If a password is set, it must be
  //       entered in HDevelop to allow the connection
  //  "debug_wait_for_connection"  [default: false]
  //     - if set to true, the engine switches into "stopped state"
  //       after starting the debug server (see below). This has the effect
  //       that any application thread that enters procedure execution
  //       via HDevEngine will stop on the first line of script code.
  //       This way, you can start debugging from the beginning of your code
  //       upon connecting from HDevelop
  void              SetEngineAttribute(const char* name, const HalconCpp::HTuple& value);
  HalconCpp::HTuple GetEngineAttribute(const char* name);

  // Set path(s) for external procedures
  //  - several paths can be passed together separating them by ';' or ':'
  //    on Windows or UNIX-like systems resp.
  //  - NULL removes all procedure paths and unloads all external procedures
  //    (Attention: procedures that are used by programs (HDevProgram) or
  //    procedures (HDevProcedures) remain unchanged until the program or
  //    procedure is reloaded explicitly.  The appropriate calls must be
  //    recreated or reassigned by the reloaded program or procedure.)
  //  - additional calls of SetProcedurePath will remove paths set before
  //    and unload all external procedures
  void SetProcedurePath(const char* path);
  void AddProcedurePath(const char* path);
#ifdef _WIN32
  void SetProcedurePath(const wchar_t* path);
  void AddProcedurePath(const wchar_t* path);
#endif
  // Get names of all available external procedures
  HalconCpp::HTuple GetProcedureNames() const;
  // Get names of all loaded external procedures
  HalconCpp::HTuple GetLoadedProcedureNames() const;
  // Unload a specific procedure <proc_name>
  void UnloadProcedure(const char* proc_name);
  // Unload all external procedures
  void UnloadAllProcedures();

  // Starts the debug server that allows to attach HDevelop as
  // as debugger to step through engine code. With default settings
  // server waits on port 57786 and engine runs normally until HDevelop
  // is connected and F9 is pressed to stop execution.
  void StartDebugServer();
```

(continued on next page)

(continued declaration of clHDevEngineName)

```
  // global variable access
  HalconCpp::HTuple GetGlobalIconicVarNames() const;
  HalconCpp::HTuple GetGlobalCtrlVarNames() const;
  // get dimension of a global variable
  int GetGlobalIconicVarDimension(const char* var_name) const;
  int GetGlobalCtrlVarDimension(const char* var_name) const;
  // get value of a global variable
  HalconCpp::HObject       GetGlobalIconicVarObject(const char* var_name);
  HalconCpp::HTuple        GetGlobalCtrlVarTuple(const char* var_name);
  HalconCpp::HObjectVector GetGlobalIconicVarVector(const char* var_name);
  HalconCpp::HTupleVector  GetGlobalCtrlVarVector(const char* var_name);
  // these method is provided for efficiency:
  // the results are copied directly into the tuple variable provided by
  // the user without additional copying
  void GetGlobalCtrlVarTuple(const char* var_name, HalconCpp::HTuple* tuple);
  // set global variable
  void SetGlobalIconicVarObject(const char* var_name, const HalconCpp::HObject& obj);
  void SetGlobalCtrlVarTuple(const char* var_name, const HalconCpp::HTuple& tuple);
  void SetGlobalIconicVarVector(const char* var_name, const HalconCpp::HObjectVector& vector);
  void SetGlobalCtrlVarVector(const char* var_name, const HalconCpp::HTupleVector& vector);


  // Set implementation for HDevelop internal operators
  void SetHDevOperatorImpl(HDevOperatorImplCpp* hdev_op_impl);
};
```

25.1.2 clHDevProgramName

.NET: clHDevProgramName

```
 ****************************************************************************
 ** class HDevProgram
 **============================================================================
 ** Class for managing HDevelop programs
 ****************************************************************************
class LIntExport HDevProgram
{
public:
  // Create a program from a .dev program file
  HDevProgram(const char* file_name = NULL);

#ifdef _WIN32
  HDevProgram(const wchar_t* file_name);
#endif

  // Copy constructor
  HDevProgram(const HDevProgram& hdev_prog);
  HDevProgram(const Data& data);

  // Assignment operation
  HDevProgram& operator=(const HDevProgram& hdev_prog);

  // Destructor
  virtual ~HDevProgram();

  // Load a program if not yet done during construction
  void LoadProgram(const char* file_name);

#ifdef _WIN32
  void LoadProgram(const wchar_t* file_name);
#endif

  // check whether the program was successfully loaded
  bool IsLoaded() const;

  // Get the program name
  const char* GetName() const;

  // Get the names of all local and the used external procedures
  HalconCpp::HTuple GetUsedProcedureNames() const;
  HalconCpp::HTuple GetLocalProcedureNames() const;

  // Compile all procedures that are used by the program and that can be
  // compiled with a just-in-time compiler.
  // The method returns true when all used procedures could be compiled by the
  // just-in-time compiler.
  // Procedures that could not be compiled are called normally by the
  // HDevEngine interpreter.
  // To check which procedure could not be compiled and what the reason is for
  // that start HDevelop and check there the compilation states.
  bool CompileUsedProcedures();
```

(continued on next page)

(continued declaration of clHDevProgramName)

```
  // create a program call for execution
  HDevProgramCall CreateCall() const;

  // This is a method provided for convenience:
  // execute the program and return the program call for
  // accessing the variables of the program's main procedure
  HDevProgramCall Execute() const;

  // get some information about the variables of the program's main procedure:
  //  - get the variable names as a tuple
  HalconCpp::HTuple GetIconicVarNames() const;
  HalconCpp::HTuple GetCtrlVarNames() const;

  //  - get the number of iconic and control variables
  size_t GetIconicVarCount() const;
  size_t GetCtrlVarCount() const;

  //  - get the names of the variables
  //   (indices of the variables run from 1 to count)
  const char* GetIconicVarName(size_t var_idx) const;
  const char* GetCtrlVarName(size_t var_idx) const;

  //  - get the dimensions of the variables
  //   (indices of the variables run from 1 to count)
  int GetIconicVarDimension(size_t var_idx) const;
  int GetCtrlVarDimension(size_t var_idx) const;
};
```

25.1.3 clHDevProgramCallName

.NET: clHDevProgramCallName

```
 ****************************************************************************
 ** class HDevProgramCall
 **============================================================================
 ** Class for managing the execution of an HDevelop program
 ****************************************************************************
class LIntExport HDevProgramCall
{
public:
  // Create an empty HDevelop program call instance
  HDevProgramCall();
  // Create an HDevelop program call from a program
  HDevProgramCall(const HDevProgram& prog);
  // Copy constructor
  HDevProgramCall(const HDevProgramCall& hdev_prog_call);
  HDevProgramCall(const Data& data);
  // Assignment operation
  HDevProgramCall& operator=(const HDevProgramCall& hdev_prog_call);
  // Destructor
  virtual ~HDevProgramCall();

  // Get the program
  HDevProgram GetProgram() const;

  // Execute program
  void Execute();

  // Stop execution on first line of program. This is intended for debugging
  // purposes when you wish to step through a specific program call. It only
  // has an effect when a debug server is running and it will only stop once.
  void SetWaitForDebugConnection(bool wait_once);

  // Clear program and reset callstack
  //  - this method stops the execution of the program after the current
  //    program line
  void Reset();

  // Get the objects / values of the variables by name or by index
  //   (indices of the variables run from 1 to count)
  HalconCpp::HObject GetIconicVarObject(size_t var_idx);
  HalconCpp::HObject GetIconicVarObject(const char* var_name);

  HalconCpp::HObjectVector GetIconicVarVector(size_t var_idx);
  HalconCpp::HObjectVector GetIconicVarVector(const char* var_name);

  HalconCpp::HTuple GetCtrlVarTuple(size_t var_idx);
  HalconCpp::HTuple GetCtrlVarTuple(const char* var_name);

  HalconCpp::HTupleVector GetCtrlVarVector(size_t var_idx);
  HalconCpp::HTupleVector GetCtrlVarVector(const char* var_name);
```

(continued on next page)

(continued declaration of clHDevProgramCallName)

```
  // these methods are provided for efficiency:
  // the results are copied directly into the tuple variable provided by
  // the user without additional copying
  void GetCtrlVarTuple(size_t var_idx, HalconCpp::HTuple* tuple);
  void GetCtrlVarTuple(const char* var_name, HalconCpp::HTuple* tuple);
};
```

25.1.4 clHDevProcedureName

.NET: clHDevProcedureName

```
 ***************************************************************************
 ** class HDevProcedure
 **=========================================================================
 ** Class for managing HDevelop procedures
 ***************************************************************************
class LIntExport HDevProcedure
{
public:
  // Create HDevelop procedure from external or local procedure
  HDevProcedure(const char* proc_name = NULL);
  HDevProcedure(const char* prog_name, const char* proc_name);
  HDevProcedure(const HDevProgram& prog, const char* proc_name);

#ifdef _WIN32
  HDevProcedure(const wchar_t* prog_name, const char* proc_name);
#endif

  // Copy constructor
  HDevProcedure(const HDevProcedure& hdev_proc);
  HDevProcedure(const Data& data);
  // Assignment operation
  HDevProcedure& operator=(const HDevProcedure& proc);
  // Destructor
  ~HDevProcedure();

  // Load a procedure if not yet done during construction
  void LoadProcedure(const char* proc_name);
  void LoadProcedure(const char* prog_name, const char* proc_name);
  void LoadProcedure(const HDevProgram& prog, const char* proc_name);

#ifdef _WIN32
  void LoadProcedure(const wchar_t* prog_name, const char* proc_name);
#endif

  // Check whether the procedure was successfully loaded
  bool IsLoaded() const;

  // Get the name of the procedure
  const char* GetName() const;

  // Get the short description of the procedure. The encoding of the
  // description will be in local 8 bit or utf-8, depending on the
  // HALCON/C++ interface encoding. Note there is no wchar_t overload for
  // Windows applications compiled with UNICODE support; however, you can
  // access the description using GetInfo("short").S().TextW() instead.
  const char* GetShortDescription() const;

  // Get all refered procedures
  HalconCpp::HTuple GetUsedProcedureNames() const;
```

(continued on next page)

(continued declaration of clHDevProcedureName)

```
// Compile all procedures that are used by the procedure and that can be
// compiled with a just-in-time compiler.
// The method returns true when all used procedures could be compiled by the
// just-in-time compiler.
// Procedures that could not be compiled are called normally by the
// HDevEngine interpreter.
// To check which procedure could not be compiled and what the reason is for
// that start HDevelop and check there the compilation states.
bool CompileUsedProcedures();

// Create a program call for execution
HDevProcedureCall CreateCall() const;

// Get name of input/output object/control parameters
HalconCpp::HTuple GetInputIconicParamNames() const;
HalconCpp::HTuple GetOutputIconicParamNames() const;
HalconCpp::HTuple GetInputCtrlParamNames() const;
HalconCpp::HTuple GetOutputCtrlParamNames() const;

// Get number of input/output object/control parameters
int GetInputIconicParamCount() const;
int GetOutputIconicParamCount() const;
int GetInputCtrlParamCount() const;
int GetOutputCtrlParamCount() const;

// Get name of input/output object/control parameters
//   (indices of the parameters run from 1 to count)
const char* GetInputIconicParamName(int par_idx) const;
const char* GetOutputIconicParamName(int par_idx) const;
const char* GetInputCtrlParamName(int par_idx) const;
const char* GetOutputCtrlParamName(int par_idx) const;

// Get dimension of input/output object/control parameters
//   (indices of the parameters run from 1 to count)
int GetInputIconicParamDimension(int par_idx) const;
int GetOutputIconicParamDimension(int par_idx) const;
int GetInputCtrlParamDimension(int par_idx) const;
int GetOutputCtrlParamDimension(int par_idx) const;

// Get info of procedure documentation
HalconCpp::HTuple GetInfo(const char* slot) const;
// Get info of parameter documentation by name
HalconCpp::HTuple GetParamInfo(const char* par_name, const char* slot) const;
// Get info of parameter documentation by index
```

(continued on next page)

(continued declaration of clHDevProcedureName)

```
  // Get info of parameter documentation by index
  //   (indices of the parameters run from 1 to count)
  HalconCpp::HTuple GetInputIconicParamInfo(int par_idx, const char* slot) const;
  HalconCpp::HTuple GetOutputIconicParamInfo(int par_idx, const char* slot) const;
  HalconCpp::HTuple GetInputCtrlParamInfo(int par_idx, const char* slot) const;
  HalconCpp::HTuple GetOutputCtrlParamInfo(int par_idx, const char* slot) const;

  // Query possible slots for procedure/parameter info
  HalconCpp::HTuple QueryInfo() const;
  HalconCpp::HTuple QueryParamInfo() const;
};
```

25.1.5 clHDevProcedureCallName

.NET: clHDevProcedureCallName

```
 ****************************************************************************
 ** class HDevProcedureCall
 **==========================================================================
 ** Class for executing an HDevelop procedure and managing the parameter
 ** values
 ****************************************************************************
class LIntExport HDevProcedureCall
{
public:
  // Create an empty HDevelop procedure call instance
  HDevProcedureCall();
  // Create HDevelop procedure call instance
  HDevProcedureCall(const HDevProcedure& hdev_proc);
  // Copy constructor
  HDevProcedureCall(const HDevProcedureCall& hdev_proc_call);
  HDevProcedureCall(const Data& data);
  // Assignment operation
  HDevProcedureCall& operator=(const HDevProcedureCall& hdev_proc_call);
  // Destructor
  ~HDevProcedureCall();

  // Get the procedure
  HDevProcedure GetProcedure() const;

  // Execute program
  void Execute();

  // Stop execution on first line of procedure. This is intended for debugging
  // purposes when you wish to step through a specific procedure call. It only
  // has an effect when a debug server is running and it will only stop once.
  void SetWaitForDebugConnection(bool wait_once);

  // Clear procedure and reset callstack
  //  - this method stops the execution of the procedure after the current
  //    program line
  void Reset();
```

(continued on next page)

(continued declaration of clHDevProcedureCallName)

```
  // Set input object/control parameter
  void SetInputIconicParamObject(int par_idx, const HalconCpp::HObject& obj);
  void SetInputIconicParamObject(const char* par_name, const HalconCpp::HObject& obj);
  void SetInputIconicParamVector(int par_idx, const HalconCpp::HObjectVector& vector);
  void SetInputIconicParamVector(const char* par_name, const HalconCpp::HObjectVector& vector);

  void SetInputCtrlParamTuple(int par_idx, const HalconCpp::HTuple& tuple);
  void SetInputCtrlParamTuple(const char* par_name, const HalconCpp::HTuple& tuple);
  void SetInputCtrlParamVector(int par_idx, const HalconCpp::HTupleVector& vector);
  void SetInputCtrlParamVector(const char* par_name, const HalconCpp::HTupleVector& vector);

  // Get the objects / values of the parameters by name or by index
  //   (indices of the variables run from 1 to count)
  HalconCpp::HObject       GetOutputIconicParamObject(int par_idx) const;
  HalconCpp::HObject       GetOutputIconicParamObject(const char* par_name) const;
  HalconCpp::HObjectVector GetOutputIconicParamVector(int par_idx) const;
  HalconCpp::HObjectVector GetOutputIconicParamVector(const char* par_name) const;
  HalconCpp::HTuple        GetOutputCtrlParamTuple(int par_idx) const;
  HalconCpp::HTuple        GetOutputCtrlParamTuple(const char* par_name) const;
  HalconCpp::HTupleVector  GetOutputCtrlParamVector(int par_idx) const;
  HalconCpp::HTupleVector  GetOutputCtrlParamVector(const char* par_name) const;

  // These methods are provided for efficiency:
  // the results are copied directly into the tuple variable provided by
  // the user without additional copying
  void GetOutputCtrlParamTuple(int par_idx, HalconCpp::HTuple* tuple) const;
  void GetOutputCtrlParamTuple(const char* par_name, HalconCpp::HTuple* tuple) const;
};
```

Note that HDevEngine/.NET provides additional methods that return iconic output parameters of a procedure call in the corresponding class (, nmHDEGetOutputIconicParamRegion,).

25.1.6 clHDevOperatorImplName

.NET: nclIHDevOperatorsName

```
 ****************************************************************************
 ** class HDevOperatorImplCpp
 **==========================================================================
 ** Class for the implemention of HDevelop internal operators
 ****************************************************************************
class LIntExport HDevOperatorImplCpp
{
public:
  HDevOperatorImplCpp();
  // Copy constructor
  HDevOperatorImplCpp(const HDevOperatorImplCpp& hdev_op_impl);
  HDevOperatorImplCpp(const Data& data);
  // Assignment operation
  HDevOperatorImplCpp& operator=(const HDevOperatorImplCpp& hdev_op_impl);
  // Destructor
  virtual ~HDevOperatorImplCpp();

  virtual int DevClearWindow();
  virtual int DevCloseWindow();
  virtual int DevSetWindow(const HalconCpp::HTuple& win_id);
  virtual int DevGetWindow(HalconCpp::HTuple* win_id);
  virtual int DevDisplay(const HalconCpp::HObject& obj);
  virtual int DevDispText(const HalconCpp::HTuple& string, const HalconCpp::HTuple& coordSystem,
                          const HalconCpp::HTuple& row, const HalconCpp::HTuple& column,
                          const HalconCpp::HTuple& color, const HalconCpp::HTuple& genParamName,
                          const HalconCpp::HTuple& genParamValue);
  virtual int DevSetWindowExtents(const HalconCpp::HTuple& row, const HalconCpp::HTuple& col,
                                  const HalconCpp::HTuple& width, const HalconCpp::HTuple& height);
  virtual int DevSetDraw(const HalconCpp::HTuple& draw);
  virtual int DevSetContourStyle(const HalconCpp::HTuple& style);
  virtual int DevSetShape(const HalconCpp::HTuple& shape);
  virtual int DevSetColored(const HalconCpp::HTuple& colored);
  virtual int DevSetColor(const HalconCpp::HTuple& color);
  virtual int DevSetLut(const HalconCpp::HTuple& lut);
  virtual int DevSetPaint(const HalconCpp::HTuple& paint);
  virtual int DevSetPart(const HalconCpp::HTuple& row1, const HalconCpp::HTuple& col1, const HalconCpp::HTu
                         const HalconCpp::HTuple& col2);
  virtual int DevSetLineWidth(const HalconCpp::HTuple& width);
  virtual int DevOpenWindow(const HalconCpp::HTuple& row, const HalconCpp::HTuple& col, const HalconCpp::HT
                            const HalconCpp::HTuple& height, const HalconCpp::HTuple& background,
                            HalconCpp::HTuple* win_id);
};
```

25.1.7 clHDevEngineExceptionName

.NET: clHDevEngineExceptionName

```
 ****************************************************************************
 ** class HDevEngineException
 **==========================================================================
 ** Class for HDevelop engine exceptions
 ****************************************************************************
class LIntExport HDevEngineException
{
public:
  // Exception categories
  enum ExceptionCategory
  {
    Exception,           // Generic
    ExceptionInpNotInit, // Error input parameters not initialized
    ExceptionCall,       // Error HALCON or HDevelop operator call
    ExceptionFile        // Error opening or reading HDevelop file
  };

  // Create HDevelop engine exception
  HDevEngineException(const char* message, ExceptionCategory category = Exception, const char* exec_proc_name = "
                      int prog_line_num = -1, const char* prog_line_name = "", Herror h_err_nr = H_MSG_VOID,
                      const HalconCpp::HTuple& user_data = HalconCpp::HTuple());
  HDevEngineException(const HDevEngineException& exc);
  HDevEngineException(const Data& data);
  HDevEngineException& operator=(const HDevEngineException& exc);
  virtual ~HDevEngineException();

  // Error text
  const char* Message() const;
  // Category of exception
  ExceptionCategory Category() const;
  const char*       CategoryText() const;
  // Name of executed procedure
  const char* ExecProcedureName() const;
  // Number of executed procedure or operator program line
  int ProgLineNum() const;
  // Name of executed procedure or operator program line
  const char* ProgLineName() const;
  // HALCON error code
  Herror HalconErrorCode() const;

  HalconCpp::HTuple UserData() const;
  void              UserData(HalconCpp::HTuple& user_Data) const;
};
```


25.2 Debugging HDevEngine From HDevelop

HDevEngine applications can be debugged remotely using HDevelop. The application must explicitly enable remote debugging itself. This will start a debug server, which accepts debug connections from HDevelop. How to attach to an application from HDevelop is described in the HDevelopUser's Guide, chapter 9 on page 299. The following sections describe remote debugging from the perspective of HDevEngine. These three example programs show how to deploy the debug server in an application:

- `hdevengine/c#/UseDebugServer`
- `hdevengine/cpp/source/use_debug_server`
- `hdevengine/python/remote_debug`

25.2.1 Configuring the Debug Server

The debug server is configured using calls to `HDevEngine::SetEngineAttribute`. It is recommended to set up the debug server *before* starting it. For example, to set up a password:

```
private HDevEngine MyEngine = new HDevEngine();
MyEngine.SetEngineAttribute("debug_password", "mysecretpassword");
```

The following attributes are supported:

Attribute	Default value	Description
debug_port	57786	Specifies the port number of the socket where the debug server waits for incoming connections.
debug_password	""	Specifying a password provides a basic layer of protection against misuse. For security reasons, it is highly recommended to always supply a password. If a password is set, it must be entered in HDevelop to allow the connection.
debug_wait_for_connection	false	If true, the engine switches into "stopped state" after starting the debug server (see below). This has the effect that any application thread that enters procedure execution via HDevEngine will stop on the first line of script code. This way, you can start debugging from the beginning of your code upon connecting from HDevelop.

25.2.2 Controlling the Debug Server

HDevEngine provides two member functions to start and stop the debug server.

```
MyEngine.StartDebugServer();
...
MyEngine.StopDebugServer();
```

Usually, HDevEngine executes HDevelop code continuously without any interruption. However, once the debug server has started, the execution of HDevelop code can be interrupted for debugging purposes.

There are multiple ways for HDevEngine to enter a stopped state. The first two are available through the API. They work regardless of whether HDevelop is actually attached. Please note that the only way to continue program execution is to actually attach HDevelop to the debug server and control the execution from there.

- `HDevEngine::SetEngineAttribute` (before starting the debug server):

  ```
  MyEngine.SetEngineAttribute("debug_wait_for_connection","true");
  // execution will stop on first line of HDevelop code that is executed
  ```

- `HDevProcedureCall::SetWaitForDebugConnection` (for a specific procedure call):

  ```
  HDevProcedure mProcPreprocess;
  HDevProcedureCall mCallPreprocess;
  ...
  mProcPreprocess = new HDevProcedure("preprocess_nuts");
  mCallPreprocess = mProcPreprocess.CreateCall();
  mCallPreprocess.SetWaitForDebugConnection(true);
  mCallPreprocess.Execute(); // execution will stop on first line of this call
  ```

Other ways to stop the program execution are triggered only if HDevelop is attached to the debug server:

- A `Stop` command (F9) is sent from HDevelop.
- An activated break point on a program line or variable is reached by HDevEngine.
- An unhandled error occurs. Normally, HDevEngine would throw a `HDevException` in this case, but in debug mode a notification is sent out to HDevelop so that the error condition can be examined there. The delayed exception will be thrown when the program execution continues.

Regardless of the reason the stopped state is always global, i.e., other application threads will also stop when executing HDevelop code.

25.2.3 Security Implications

Like any remotely accessible server, careful attention must be paid to prevent unauthorized access to the debug server. Although the firewall must be configured to allow connections to the configured port, it is your responsibility to limit the access to authorized clients only.

As a minimal security measure, always set up a password for the debug server. Please note that the debug server itself provides no measures against brute-force or denial-of-service attacks.

Note the following if the debug server is running and no password is set:

- The execution of the external application can be stopped by a third party.
- Unprotected procedures can be viewed and their code copied.

Apart from the transmission of passwords, the communication between HDevelop and the debug server is not encrypted. Protected procedures are transmitted in their encrypted binary form. If you worry about other sensitive data like images, you should use a VPN (e.g., an SSH tunnel) when connecting from the outside to the local network.

25.2.4 Limitations

- Debugging of HDevelop programs (i.e., started with `HDevProgramCall`) is not supported.
- Debugging applications that use `par_start` is not supported.
- JIT-compiled procedures cannot be debugged.

Further limitations of remote debugging from the HDevelop side are listed in the HDevelopUser's Guide, section 9.9 on page 302.

25.3 Tips and Tricks

25.3.1 Troubleshooting

Executed program or procedure raises exception for display operators like `set_tposition` when not using implementation of display operators

If you are not using an implementation of HDevelop's internal display operators (`dev_*`), calls to these operators in the executed HDevelop program or procedure are simply ignored. However, if the program or procedure contains other, "external" display operators like `set_tposition` or `write_string`, which need a window handle as input parameter, the program / procedure will raise an exception at this operator if the window handle has not been instantiated.

Typically, this problem will not arise when executing programs, because you will in most cases use an implementation of the display operators. When executing procedures, we recommend that you leave out the external display operators that use window handles. If this is not possible, you could place them in a separate procedure and use the implementation of display operators for just this procedure. Alternatively, initialize the window handle with a value like `-1` and test it before executing the external display operators.

External procedures cannot call local procedures

Note that local procedures can call external procedures but not the other way round.

25.3.2 Loading and Unloading Procedures

To ensure that programs developed with HDevelop can be executed with HDevEngine without any further action, all standard procedures (see the HDevelopUser's Guide, section 5.4 on page 45) are automatically loaded when HDevEngine is started.

In most applications there is no need to delete loaded HDevelop procedures explicitly using `mHDEUnloadProcedureName`. Possible reasons might be to free memory or to reload a procedure from a procedure or library file that was changed on disk.

When creating a procedure with the class `clHDevProcedureName`, the procedure is loaded and copied together with all the procedures it uses, and will not be affected by further calls to unload a procedure.
For library procedures, the entire library will be unloaded. For non-library external procedures, `mHDEUnloadProcedureName` deletes only the specified procedure. In both cases, dependent procedures will not be unloaded automatically.
To delete the automatically loaded procedures, you can query their names using `mHDEGetLoadedProcedureNamesName`. Afterwards, you can delete them if you are sure that they are not used by another loaded procedure, or you can use `mHDEUnloadAllProceduresName` to unload all external procedures.
Note that after calling `mHDEUnloadProcedureName` or `mHDEUnloadAllProceduresName`, `mHDEGetProcedureNamesName` still returns the names of the unloaded procedures.

Index