HALCON

a product of MVTec

Technical Updates

HALCON 20.11 *Progress*

MVTec Software GmbH

Building Vision for Business

This technical note describes how to meet technological changes in HALCON, Version 20.11.0.0.

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

About This Technical Note

This technical note describes technological changes in HALCON and how to adapt to them. Each chapter deals with a specific change in a specific version of HALCON. Typically, the chapters are completely unrelated by topic. Each chapter title lists the HALCON version which introduces the described changes.

Contents

Chapter 1

Handles (18.05)

Handles are now a distinct control data type. The new handle data type provides a reference counting system similar to iconic objects. This allows HALCON to keep the ownership of handles and to support an automatic destructor mechanism, i.e., the tool will be cleared automatically when the last handle referencing it is destroyed. Thus, the explicit call of a clear operator is not needed anymore, but can still be used for early freeing of resources.

Previously, there was no way to recognize if a number was just a number or a reference to a tool. Furthermore, it was not possible to check the type of a tool or to check whether a tool was already destroyed. Thus, an explicit clear was needed to avoid resource leaks (especially cumbersome for types where you manage many instances like with `ObjectModel3D`).

1.1 HDevelop Language

The new control data type "handle" is represented as a magic value (e.g. H12BF160) in the variable window. This value cannot not be changed and may differ from execution to execution and version to version. Note that handles are no longer interchangeable with numeric values. Tools are automatically cleared when variables are overwritten or go out of scope (at the end of the procedure).

1.2 HALCON/C++, and HALCON/.NET Language Interface

Handles are represented by a new class `HHandle`. It is the base class of the specialized tool classes (`HShapeModel`, `HBarCode` etc.). `HTuple` supports `HHandle` as an additional element type (both in mixed and pure tuples). `HTuple` elements representing handles are accessed as `HHandle` (*.H) instead of `Hlong` (*.L). This is a common adaptation that will be needed within existing sources when they are to be ported to the new handle mechanism.

`HTuple` is now a disposable class in HALCON/.NET. When a `HTuple` or `HHandle` instance is destroyed, the reference counter of the affected handle(s) is decreased. When the reference counter reaches zero, the physical tool (e.g., a shape model) is destroyed automatically.

1.3 Legacy Code

Code adaptions of existing code for the new data type will usually not be necessary. But if HALCON handles were not held in HALCON data structures like tuples or handle classes, passing the ownership of handles from the user to HALCON might lead to undefined behavior and crashes (as HALCON will free the handle resources automatically when the last HALCON structure holding a handle reference is destroyed).

For these cases HALCON offers a legacy mode to transfer the ownership of handles to the user code. In this mode, handles have to be cleared explicitly. HALCON operators will return handles as numeric values and accept them as input. The legacy handle mode can be switched on and off using `set_system`:

```
set_system('legacy_handle_mode', 'true')
```

Mixing of current and legacy code is not recommended. If absolutely required, handles can be converted to numeric values (see handle_to_integer) and back again (see integer_to_handle).

Cast ambiguities like HShapeModel(tuple) can lead to source code incompatibilities. In C++ these can be avoided by defining HCPP_LEGACY_HANDLE_API before including HalconCpp.h. To avoid cast ambiguities in .NET, please adapt the legacy code, for example by changing tuple to tuple.L.

1.3.1 HALCON/C++

All handle classes provide the methods SetHandle() and GetHandle(), which allow to access the underlying handle for legacy purposes; furthermore, the classes provide an operator that casts an instance of the class into the corresponding handle. These methods are typically used when combining procedural and object-oriented code.

1.4 Behavioral Changes

Serializing HTuple instances that contain handles will now serialize the underlying tool. If the tool is not serializable, an exception is raised.

Cloning a HTuple instance also creates a deep copy via serialization. If this is not desired, use the copy constructor instead. In that case the handle in the tuple copy will contain an additional reference to the same tool.

Chapter 2

UTF-8 Encoding (18.11)

The default encoding which is used for representing strings with non-ASCII characters within the HALCON library is UTF-8 now. This allows to represent arbitrary characters in a unique way independent of the system and the current locale.

2.1 Legacy Encoding Mode

The locale encoding is still available as legacy mode for programs which expect non-ASCII strings to be encoded in the current locale: on western (Windows) systems, e.g., the program still expects `tuple_ord` to return the CP1252 codes (the Euro sign is '128' instead of '8364' in Unicode mode) and workarounds for Japanese encodings should work without changing the code.

The legacy mode can be enabled by setting the environment variable `HALCON_ENCODING` to `LOCALE` or by using `set_system`:

```
set_system('filename_encoding', 'locale')
```

Applications that use the HALCON/C or the HALCON/C++ interface for calling HALCON operators can also set the encoding of the interface independently of the HALCON encoding.

However, the following encoding issues will probably not work as expected in legacy encoding mode:

- Tuple string operators will not work character-wise. They behave as old HALCON versions to keep the legacy mode working as before.
- Changing the filename encoding from 'utf8' to 'locale' or vice versa via `set_system` does not work within JIT-compiled procedures. The strings that are passed to the procedure are encoded according to the filename encoding when entering the procedure. String constants within the procedure are encoded according to the filename encoding during the creation or last edit of the procedure. This is no regression, i.e. the behavior can be observed also in HALCON 12 and 13.
- Error messages with non-ASCII characters were not and will not be displayed correctly on non-latin systems, like Japanese Windows. On Latin systems, only two English messages are affected:
  `H_ERR_CAL_NEGTS` (8445) "`Tilt must be within the range of 0° and 90°`" and
  `H_ERR_CAL_NEGRS` (8446) "`Rot must be within the range of 0° and 360°`".
  German error messages with Umlauts will not be displayed correctly, too.
- On Japanese systems, the procedure `list_image_files` can produce invalid paths if a path contains a Japanese character where the second Shift-JIS byte has the same code as a backslash. This is no regression, i. e. the behavior can be observed also in HALCON 12 and 13.
- The length passed to the operator `read_string` still limits the number of bytes. This affects the number of Asian multi-byte characters that effectively could be entered.

- The global system variables `window_name` and `icon_name` are stored with the encoding of the HALCON library when they are set. When the encoding is changed via `set_system` the names are not transcoded.
- All strings in HDevelop are always encoded in UTF-8, independently of the filename encoding. This implicates that it is not possible to create strings in legacy mode by their local hexadecimal codes (instead of writing them as character, e.g., \xe4 for the Latin-1 character "ä"). The hexadecimal codes are not handled as the expected locale characters, but they are first interpreted as UTF-8 bytes and only converted to local-8-bit encoding when passed to the HALCON library.

Independently of the encoding mode, hostnames must be encoded in pure ASCII. Special characters are not supported. To query the hostname which is used, e.g., for socket connections, use:

```
get_system('hostname', Hostname)
```