HALCON Version 5.2

MVTec Software GmbH

HALCON/C
User's Manual

20th April 1999

This manual explains the use of the HALCON/C version 5.2 operators

1st Edition.	July 1997
2nd Edition.	November 1997
3rd Edition.	March 1998
4th Edition.	April 1999

Information concerning HALCON:
`http://www.mvtec.com`

Address:	MVTec Software GmbH Orleansstr. 34 D-81667 Munich Germany
E-mail:	`mvtec@mvtec.com`

Contents

Chapter 1

An introductory example

Before going into the details of HALCON, let's have a look at a small example. Given the image of a mandrill in Figure 1.1 (left), the goal is to segment its eyes. This is done by the C program shown in Figure 1.2. The segmentation result is shown in Figure 1.1 (right):

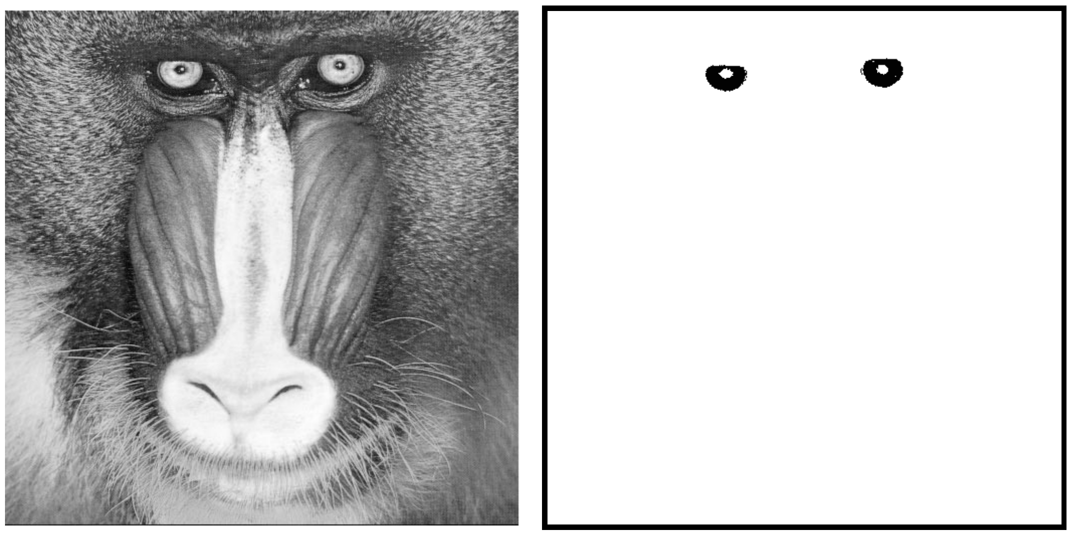

Figure 1.1: To the left, the image of a mandrill is shown (input image). To the right, the result of the segmentation process in the example program can be seen (eyes).

The program is quite self explanatory. We will describe the basic principles nevertheless: First, all image pixels with greyvalues greater than 128 are selected. Then all connected componentes of the region formed by these pixels are calculated. The corresponding HALCON operator calculates a region tuple, and thus splits the image in different regions (objects). From these, the mandrill's eyes are selected by their area and shape.
This little example shows how easy it is to integrate HALCON operators in any C program. Their use is very intuitive: Users don't have to think about the basic data structures and algorithms involved. And since all HALCON operators are hardware independant, users don't even have to care about things like different I/O devices. HALCON has its own memory management and provides a sophisticated runtime environment. Have a look at the program in Figure 1.2 again. Isn't image processing fun this way?
But joking aside. In the following chapters we will describe the details of integrating HALCON operators into C programs. Chapter 2 introduces the four different parameter classes of HALCON operators. We will explain the use of HALCON tuples (Chapter 2.2.2) for

```
#include "HalconC.h"

main()
{
  Hobject mandrill,thresh,conn,area,eyes;  /* required objects           */
  long    WindowHandle;
  open_window(0,0,512,512,0,"visible","",&WindowHandle);  /* open window */
  read_image(&mandrill,"mandrill");     /* read input image ("mandrill") */
  disp_image(mandrill,WindowHandle);    /* display input image           */
  get_mbutton(WindowHandle,_,_,_);      /* wait for mouse click          */
  /* Select image region with pixels in [128,255]                        */
  threshold(mandrill,&thresh,128.0,255.0);
  connection(thresh,&conn);             /* compute connected components  */
  /* select regions with an area of at least 500 pixels                  */
  select_shape(conn,&area,"area","and",500.0,90000.0);
  /* select the eyes in these regions by using the anisometry feature    */
  select_shape(area,&eyes,"anisometry","and",1.0,1.7);
  disp_region(eyes,WindowHandle);       /* display result                */
  get_mbutton(WindowHandle,_,_,_);      /* wait for mouse click          */
  close_window(WindowHandle);           /* close window                  */
  /* delete image objects from the Halcon database                       */
  clear_obj(mandrill); clear_obj(thresh); clear_obj(conn);
  clear_obj(area); clear_obj(eyes);
}
```

Figure 1.2: Introductory example program

supplying operators with tuples of control parameters in great detail: Using tuples, the two `select_shape(...)` calls in our example program could be combined into only one call. Chapter 3 is dedicated to the return values of HALCON operators. Chapter 4 gives an overview over all the include-files and C-libraries necessary for compiling C programs and shows a sample `makefile`. Finally, Chapter 5 contains example solutions for some common problems in image processing (like edge detection).

Chapter 2

HALCON parameter classes

HALCON distinguishes four different classes of operator parameters:

- Input image objects
- Output image objects
- Input control parameters
- Output control parameters

Input parameters are passed *by value*, output parameters are passed *by reference* (using the &-operator). An exception to this rule are output control parameters of type `char*`. Here, the caller has to provide the memory and only a pointer to that memory is passed to the operator.
As a rule of thumb, all HALCON operators can also be called using tuples of parameters instead of single values. Take the `connection(...)` operator from our example program in the previous chapter: It calculates a tuple of output image objects (the connected components). Of course there are several HALCON operators that cannot be called with tuples for some or all parameters. Whether this is the case for specific operators is described in detail in the HALCON reference manual. Unfortunately, C doesn't provide a generic list constructor (e.g., like the one in PROLOG). Therefore, the use of tuples of control parameters is a little elaborate. Using tuples of image objects on the other hand is in no way different from using single values.
HALCON/C provides the data structure `Htuple` for tuples of control parameters (see Chapter 2.2.2 for details) and the data structure `Hobject` for image objects (single objects as well as object tuples — see Chapter 2.1).

2.1 Image objects

Image processing isn't possible without actual images. By using image objects, HALCON provides a 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 need not be connected and may include "holes." They may even be larger than the image format. Internally, regions are stored by run length encoding.

Images consist of at least one image array and a corresponding region. The image region denotes the array pixels that are "defined" (i.e., HALCON operators working on greyvalues 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 greyvalue, but by a vector of up to n greyvalues (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 greyvalue transformations. Segmentation operators are the transition from images (greyvalues) 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 data base 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. It may be noted that `integer_to_obj(...)` duplicates the image objects (Don't worry, this doesn't mean necessarily, that the corresponding greyvalue arrays are duplicated too. As long as there is only read-access, a duplication of the references is sufficient). Therefore, all extracted objects have to be deleted explicitly from the HALCON data base, using `clear_obj(...)`. Figure 2.1 contains an excerpt from a C program to clarify that approach.
Some HALCON operators like `neighbor(...)` or `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.

2.2 Control parameters

HALCON/C supports the following data types as types for control parameters of HALCON operators:

- integers,
- floating point numbers,
- character arrays (strings)

As already mentioned in the introduction to this chapter, using control parameter tuples in C isn't as elegant as using image object tuples. To circumvent the missing generic lists in C, it was necessary to introduce two different working modes into HALCON/C: The simple mode and the tuple mode. If a tuple is necessary for at least one control parameter, the tuple mode

```
...
Hobject   objects;     /* tuple of image objects                    */
Hobject   obj;         /* single image object                       */
long      surrogate;   /* object key, converted to integer          */
Htuple    Tsurrogates; /* tuple of object keys                      */
Htuple    Index,Num;   /* temporary tuple for parameter passing     */
long      i;           /* loop variable                             */
long      num;         /* number of objects                         */

...
count_obj(&num);
/* variant 1: object key -> control parameter                       */
create_tuple(&Index,1); set_i(Index,1,0);
create_tuple(&Num,1);   set_i(Num,num,0);
T_obj_to_integer(objects,Index,Num,&Tsurrogates);
for (i=0; i<num; i++)
{
  surrogate = get_i(Tsurrogates,i);
  /* process single object                                          */
}
/* variant 2: copying objects individually                          */
for (i=1; i<=num; i++)
{
  copy_obj(objects,&obj,i,1);
  /* process single object                                          */
}
...
```

Figure 2.1: Accessing the i-th image object in a tuple of image objects

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 operators calculates isn't fixed beforehand.
Mentioning the control parameter types — How is the default type of control parameters determined for a given operator? Basically there are three ways:

1. The operator description in the HALCON reference manual,
2. the HALCON system operator `get_param_info(...)` and
3. the 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 preceeding `T_`. That means,

```
select_shape(...)
```

is a call of the HALCON operator `select_shape` (as described in the HALCON reference manual) in simple mode, whereas

```
T_select_shape(...)
```

is a call of the same operator in tuple mode.

2.2.1 Simple mode

In simple mode, all operators described in the HALCON reference manual can be used in a very intuitive way in own C programs. All control parameters are variables (or constants) of the data types

- `long` for integers (HALCON type `INT_PAR`),
- `double` for floating point numbers (`DOUBLE_PAR`) or
- `char*` for character arrays (strings, `STRING_PAR`).

`long` 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 *Please note, that the memory for output control parameters (esp. strings) has to be provided by the caller!*. Output parameter values that are of no further interest can be denoted by the anonymous variables

- "`_`" or "`_i`" for `long`-parameters,
- "`_d`" for `double`-parameters and
- "`_s`" for `char*`-parameters

As an example for the use of anonymous variables see the small introductory program in Figure 1.2: We coded `get_mbutton(WindowHandle,_,_,_)`, because the actual button pressed is of no further interest.
Examples for HALCON operator calls in simple mode can be found in the C programs in Figures 1.2 and 2.1.

2.2.2 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. Furthermore, HALCON allows tuples with mixed types as control parameter values (e.g., integers mixed with floating point numbers).
Therefore, in addition to the very intuitive simple mode there is another mode in HALCON/C: The tuple mode. Using this mode is a little more elaborate. If at least one of the control parameters of a HALCON operator is passed as a tuple, the tuple mode has to be used for all control parameters (Mixing of both modes isn't possible). Furthermore, the tuple mode also has to be used if the *number* or *type* of the calculated values aren't known beforehand.
Syntactically, tuple mode is distinguished from simple mode by a `T_` preceeding 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 values of the types

- `long` for integers (HALCON type `INT_PAR`),
- `double` for floating point numbers (`DOUBLE_PAR`) or
- `char*` for character arrays (strings, `STRING_PAR`)

in arbitrary combination. Control parameter tuples must be created, deleted, and manipulated, using the appropriate HALCON/C operators *only* (Overview in Figures 2.2 and 2.3).
The rules for parameter passing are valid in tuple mode, too: Input control parameters (type `Htuple`) are passed *by value* as usual, output control parameters are passed *by reference*, using the &-operator. Output parameters, that are of no further interest can be denoted by the anonymous variable "`_t`" instead of a "dummy" tuple.
Let's summarize the five most important steps, when calling a HALCON operator in tuple mode:

1. **step:** First, memory must be allocated for all tuples of input control parameters, using `create_tuple`. Memory for output control parameter tuples is allocated by HALCON/C (a call of `create_tuple` isn't necessary).

2. **step:** Now, the input control parameter tuples are constructed, using the appropriate `set_*` operators. `set_s`, which inserts a string into a tuple allocates the needed memory by itself and then copies the string.

3. **step:** Then, the HALCON operator is actually called. The operator name is (as already explained) preceeded by a `T_` to denote tuple mode.

4. **step:** Further processing of the output parameter tuples takes place, using the operators `length_tuple`, `get_type` and `get_*`. When processing strings (using `get_s`), please note that the allocated memory is freed automatically upon deleting the tuple with `destroy_tuple`. If the string has to be processed even after the deletion of the tuple, the whole string must be copied first. The maximal string length (incl. termination character "\0") in HALCON is `MAX_STRING` (1024 in HALCON version 5.x).

5. **step:** Finally the memory allocated by all the tuples (input and output) has to be freed again. This is done with `destroy_tuple`. If you still need the values of the tuple variables, remember to copy them first. Now, the whole series can start again — using different or the same tuple variables.

Before we end this chapter with a short example program, we will explain an alternative **generic calling mechanism** for HALCON operators in tuple mode. This mechanism is intended for the use in interpreters or graphical user interfaces:

```
            T_call_halcon(ProcName)
```

calls the HALCON operator `ProcName` in tuple mode. To do so, the operator parameters have to be set first, using

```
void create_tuple(tuple,length)     or     macro CT(tuple,length)
     Htuple    *tuple;
     long      length;
     /* creates a tuple that can hold 'length' entries             */

void destroy_tuple(tuple)           or     macro DT(tuple)
     Htuple    tuple;
     /* deletes a tuple (if the tuple contains string entries,     */
     /* the memory allocated by the strings is freed, too)         */

long length_tuple(tuple)            or     macro LT(tuple)
    Htuple    tuple;
    /* returns the length of a tuple (number of entires)           */

void set_i(tuple,val,index)         or     macro SI(tuple,val,index)
     Htuple    tuple;
     long      val;
     long      index;
     /* inserts an integer with value 'val' into a tuple at        */
     /* position 'index' ('index' in [0,length_tuple(tuple) - 1])  */

void set_d(tuple,val,index)         or     macro SD(tuple,val,index)
     Htuple    tuple;
     double    val;
     long      index;
     /* inserts a double with value 'val' into a tuple at          */
     /* position 'index' ('index' in [0,length_tuple(tuple) - 1])  */

void set_s(tuple,val,index)         or     macro SS(tuple,val,index)
     Htuple    tuple;
     char      *val;
     long      index;
     /* inserts a copy of string 'val' into a tuple at             */
     /* position 'index' ('index' in [0,length_tuple(tuple) - 1]). */
     /* The memory necessary for the string is allocated by set_s. */
```

Figure 2.2: HALCON/C Htuple operators (part one)

`set_in_opar`, `set_out_opar`, `set_in_tpar` und `set_out_tpar`

Accessing these parameters is still possible with the ordinary tuple operators. Figure 2.4 summarizes the operators of the generic HALCON/C calling interface.
But now to the mentioned example program (see Figure 2.5 or the file `example3.c`: The aim is to get informations about the current HALCON system state. The HALCON operator `get_system('?',Values)` (here in PROLOG notation) 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.

```
int get_type(tuple,index)           or     macro GT(tuple,index)
    Htuple    tuple;
    long      index;
    /* returns the type of the value at position 'index' in the     */
    /* tuple. Possible values: INT_PAR, DOUBLE_PAR or STRING_PAR    */

long get_i(tuple,index)             or     macro GI(tuple,index)
    Htuple    tuple;
    long      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;
       long      index;
    /* returns the floating point number 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;
     long      index;
    /* returns the pointer(!) to the string at position 'index' in   */
    /* the tuple. (a type error results in a run time error)         */

    /* Attention: all indices must be in [0,length_tuple(tuple) - 1] */
```

Figure 2.3: HALCON/C Htuple operators (part two)

```
void set_in_opar(obj,par)           or     macro IO(obj,par)
     Hobject   obj;
     int       par;
     /* defines 'obj' as input image object parameter no. 'par'     */
     /* (inside the input image object parameter parameter class)   */

void set_out_opar(obj,par)          or     macro OO(obj,par)
     Hobject   *obj;
     int       par;
     /* defines 'obj' as output image object parameter no. 'par'    */
     /* (inside the output image object parameter parameter class)  */

void set_in_tpar(tuple,par)         or     macro IT(tuple,par)
     Htuple    tuple;
     int       par;
     /* defines 'tuple' as input control parameter no. 'par'        */
     /* (inside the input control parameter parameter class)        */

void set_out_tpar(tuple,par)        or     macro OT(tuple,par)
     Htuple    *tuple;
     int       par;
     /* defines 'tuple' as output control parameter no. 'par'       */
     /* (inside the output control parameter parameter class)       */

Herror T_call_halcon(ProcName)      or     macro TC(ProcName)
     char      *ProcName;
     /* calls the Halcon operator 'ProcName' using tuple mode;      */
     /* input and output parameters of 'ProcName' must be declared  */
     /* using set_in_*par and set_out_*par first                    */
```

Figure 2.4: Generic calling mechanism for the HALCON/C tuple mode

```
#include "HalconC.h"

main ()
{
  Htuple    In,SysFlags,Info;        /* tuple variables            */
  long      i,num;

  printf("system informations:\n");
  create_tuple(&In,1);               /* prepare first query        */
  set_s(In,"?",0);                   /* only value of 'In': "?"    */
  T_get_system(In,&SysFlags);        /* first query                */
  destroy_tuple(In);                 /* free parameter             */
  num = length_tuple(SysFlags);      /* number of system flags     */
  for (i=0; i<num; i++)
  {      /* determine the value of the i-th system flag:           */
     create_tuple(&In,1);             /* prepare query              */
     set_s(In,get_s(SysFlags,i),0);   /* insert i-th system flag    */
     printf("%s ",get_s(SysFlags,i)); /* print name                 */
     T_get_system(In,&Info);          /* get corresponding info     */
     destroy_tuple(In);               /* free parameter             */
     switch(get_type(Info,0))
     {        /* print the value according to the flag's type:     */
       case INT_PAR:    printf("(int): %ld\n",get_i(info,0));
                        break;
       case DOUBLE_PAR: printf("(double): %f\n",get_d(info,0));
                        break;
       case STRING_PAR: printf("(string): %s\n",get_s(info,0));
                        break;
     }
     destroy_tuple(Info);             /* free parameter             */
   } /* for(i=... */
}
```

Figure 2.5: Tuple mode example program: Printing the current HALCON system state

Chapter 3

HALCON operator return values

HALCON operator return values (type `Herror`) can be divided into two categories:

- Messages (`H_MSG_*`) and
- Errors (`H_ERR_*`).

According to its procedural concept, HALCON distinguishes four kinds of messages:

- `H_MSG_TRUE`: The operator finished without error and returns the boolean value true.
- `H_MSG_FALSE`: The operator finished without error and returns the boolean value false.
- `H_MSG_VOID`: The operator finished without error, but doesn't return a value.
- `H_MSG_FAIL`: the operator finished without error (!) and returns "operation failed". This could mean that the operator doesn't consider itself relevant for the operation or that a specific event didn't happen.

Nearly all HALCON operators return `H_MSG_TRUE`, if no error occurs.
Errors in HALCON operators usually result in an exception, i.e., a program abort with the appropriate error message in HALCON/C (default exception handling). However, users can disable this mechanism (with a few exceptions, like errors in `Htuple` operators), using

```
set_check("~give_error");
```

to provide their own error handling routines. In that case, the operator `error_text(...)` is very useful: This operator returns the plain text message for any given error number. 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 4

Generation of HALCON/C Applications

The HALCON/C system consists of the files summarized in Figure 4.1.
Please make sure to reserve a stack size of 6 MByte when linking an application with the HALCON libraries.
Your own C programs that use HALCON operators, have to include the file `HalconC.h`. Do this by adding the include-command:

```
#include "HalconC.h"
```

near the top of your C file. Using this syntax, the compiler looks for `HalconC.h` in the current directory only. Alternatively you can tell the compiler where to find the file, giving it the

```
-I<pathname>
```

command line flag to denote the include-file directory. `HalconC.h` contains all user-relevant definitions of the HALCON system and the declarations necessary for the C interface.
The library `libhalconc.a/.so/.sl` contains the various components of the HALCON/C interface. `libhalcon.a/.so/.sl` is the actual HALCON-library. The corresponding NT libraries are `halconc.lib/.dll` and `halcon.lib/.dll`.
Finally, there are several example programs in the HALCON/C distribution:

`example1.c` reads an image and demonstrates several graphics operators.

`example2.c` introduces several image processing operators.

`example3.c` An example for the usage of 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 tuple mode (`T_call_halcon(...)`).

`example7.c` describes the handling of RGB images.

`include/HalconC.h`: include-file; contains all user-relevant definitions of the HALCON system and the declarations necessary for the C interface.

`lib/architecture/libhalconc.a/.so/.sl`: The HALCON/C library (UNIX).

`lib/architecture/libhalcon.a/.so/.sl`: The HALCON library (UNIX).

`halconc.lib/.dll`: The HALCON/C library (Windows NT).

`halcon.lib/.dll`: The HALCON library (Windows NT).

`include/HProto.h`: external function declarations.

`examples/C/example1.c, ..., examples/C/example9.c`: example programs.

`examples/C/makefile.*`: Sample makefiles to compile the example programs for several arcitectures.

`images/`: Several images used by the example programs.

`help/english.*`: Files necessary for online informations.

`doc/*`: Various manuals (in subdirectories).

`doc/ps/manual/*`: postscript manuals.

`doc/pdf/manual/*`: postscript manuals.

`doc/html/*`: HTML manuals

`doc/html/`: HALCONoperator reference (HTML).

Figure 4.1: HALCON/C file structure

`example8.c` demonstrates the creation of an image from user memory.

`example9.c` describes some additional handling of RGB images.

`makefile` sample makefile for `example1.c`.

In the following some important environment variables shall be introduced. In principle, they are the same both under UNIX and Windows NT but in the latter case will be set automatically during the setup. So we will only consider their handling in an UNIX environment.
While a HALCON program is running, it accesses several files (e.g., `help/*`). To tell HALCON where to look for these files, the environment variable `HALCONROOT` has to be set. `HALCONROOT` points to the HALCON home directory. If HALCON is installed in the directory `/usr/halcon`, `HALCONROOT` has to be set accordingly, using the command `setenv HALCONROOT /usr/halcon` (`csh` syntax). `HALCONROOT` is also used in the sample makefile.
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`. See the HALCON C-Interface Programmer's Manual for more details.

Two things are important in connection with the example programs: The default directory for the HALCON operator `read_image(...)` to look for images is `/usr/halcon/images`. If the images reside in different directories, the appropriate path must be set in `read_image(...)` or the default image directory must be changed, using `set_system("image_dir","/...")`. This is also possible with the environment variable `HALCONIMAGES`. It has to be set before starting the program. An appropriate definition could look like: `setenv HALCONIMAGES /usr/halcon/images`. It is also possible to put the images in several directories. In that case, all the directories (separated by colons under Unix and semicolons under Windows NT) have to be set in `HALCONIMAGES`.

```
PROG_NAME     = example1
DESTINATION   = $(PROG_NAME)
#HALCONROOT   =
#ARCHITECTURE =

BIN           = .
SOURCE        = .
H_INCLUDE     = $(HALCONROOT)/include
H_LIB         = $(HALCONROOT)/lib/$(ARCHITECTURE)

LIBRARIES     = -lhalcon -lhalconc -lm -lX11
CC            = cc

$(BIN)/$(DESTINATION) : $(PROG_NAME).o
      $(CC) -o $(BIN)/$(DESTINATION) -L$(H_LIB) \
      $(PROG_NAME).o \
      $(LIBRARIES)

$(PROG_NAME).o: $(PROG_NAME).c
      $(CC) -c -I$(H_INCLUDE) $(SOURCE)/$(PROG_NAME).c
```

Figure 4.2: Sample makefile for `example1.c` (compiles program `example1`)

The second remark concerns the output terminal. 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`, like `setenv DISPLAY hostname:0.0` (`csh` syntax).
To end this chapter, Figure 4.2 shows a sample makefile for the program `example1.c` (see file `makefile`, too). If necessary, the paths in the makefile have to be replaced by the corresponding paths in your actual system before compiling the example program.
Things are a bit different with VisualC++. Mainly there is no such thing like a makefile. Instead you have to create a new project which has to contain the `halcon.lib/.dll` and the `halconcpp.lib/.dll` files additionally to the desired source file. For the examples the project should be of the WIN 32 ConsoleApplication type. Please note that the VisualC++ compiler implicitly calls “Update all dependencies” if a new file is added to a project. Since HALCON runs under Unix as well as under Windows NT, the include file `HalconC.h` includes several Unix-specific headers as well if included under Unix. Since they don’t exist under NT, and the VisualC++ compiler is dumb enough to ignore the operating-system-specific cases in the include files, you will get a number of warning messages about missing header files. These can safely be ignored.

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

Chapter 5

Common image processing problems

This final chapter shows the possibilities of HALCON and HALCON/C on the basis of several simple image processing problems.

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 greyvalues between 0 and 120 (channel 1) are selected.
- The remaining image regions are split into connection components.
- By suppressing regions that are too small, noise is eliminated.

Edge detection: 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 desired 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(...)`.

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 grey-values lighter than their surrounding (11 x 11) pixels.

Simple texture transformations: Texture transformations are used to enhance specific image structures. The behaviour of the transformation depends on the filters used (HALCON provides 16 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, that are combined later, using `add_image(...)`, `mult_image(...)` or a similar operator.

Elimination of 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 an segmentation operator that calculates a tuple of image objects (`Seg`).

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 an segmentation operator that calculates a tuple of image objects (`Seg`).

Smoothing of 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 smooting of region boundaries, circular masks are suited best.
- The mask size determines the degree of the smoothing.
- `segmentation(...)` is used to denote an segmentation operator that calculates a tuple of image objects (`Seg`).

List of Figures