Chapter 8

HDevelop Language

This chapter introduces the syntax and the semantics of the HDevelop language. In other words, it illustrates what you can enter into a parameter slot of an operator or procedure call. In the simplest case this is the name of a variable, but it might also be an arbitrary expression like `sqrt(A)`. Besides, control structures (like loops) and the semantics of parameter passing are described.

Note that the HALCON operators themselves are not described in this chapter. For this purpose refer to the HALCON reference manual. All program examples used in this chapter can also be found in the directory `%HALCONEXAMPLES%\hdevelop\Manuals\HDevelop`.

8.1 Basic Types of Parameters

HALCON distinguishes two kinds of data: control data (numbers or strings) and iconic data (images, regions, etc.)

By further distinguishing *input* from *output parameters*, we get four different kinds of parameters. These four kinds always appear in the same order in the HDevelop parameter list. In the reference manual operator signatures are visualized in the following way:

```
operator ( iconic input : iconic output : control input : control output )
```

As you see, iconic input objects are always passed first, followed by the iconic output objects. The iconic data is followed by the control data, and again, the input parameters succeed the output parameters.

Any of the four types of parameters may be empty. For example, the signature of `read_image` reads

```
read_image ( : Image : FileName : )
```

The operator `read_image` has one output parameter for iconic objects `Image` and one input control parameter `FileName`. The parameter types are reflected when entering operators in the operator window. The actual operator call displayed in the HDevelop program window is:

```
read_image (Image, 'Name')
```

The parameters are separated by commas. Input control parameters can either be variables, constants or expressions. An expression is evaluated *before* it is passed to a parameter that receives the result of the evaluation. Iconic parameters must be variables. Control output parameters must be variables, too, as they store the results of an operator evaluation.

8.2 Control Types and Constants

All non-iconic data is represented by so called *control data* (numbers or strings) in HDevelop. The name is derived from their respective functions within HALCON operators where they *control* the behaviour (the effect) of image processing (e.g., thresholds for a segmentation operator). Control parameters in HDevelop may contain arithmetic or logical operations. A control data item can be of one of the following types: `integer`, `real`, `string`, and `boolean`.

`integer` The type `integer` is used under the same syntactical rules as in C. Integer numbers can be input in the standard decimal notation, in hexadecimal by prefixing the number with `0x`, and in octal by prefixing the number with `0` (zero).

For example:

```
4711
-123
0xbeef (48879 in decimal notation)
073421 (30481 in decimal notation)
```

Data items of type `integer` are converted to their machine-internal representations, that is the C type `long` (4 or 8 bytes).

`real` The type `real` is used under the same syntactical rules as in C.

For example:

```
73.815
0.32214
.56
-17.32e-122
32E19
```

Data items of type `real` are converted to their machine-internal representations, that is the C type `double` (8 bytes).

`string` A `string` is a sequence of characters that is enclosed in single quotes ('). Special characters, like the line feed, are represented in the C-like notation, as you can see in table 8.1 (see the reference of the C language for comparison). You can enter arbitrary characters using the format `\x`*nn* where *nn* is a two-digit hexadecimal number, or using the format `\0`*nnn* where *nnn* is a

Meaning	Abbreviation	Notation
line feed	`NL (LF)`	`\n`
horizontal tabulator	`HT`	`\t`
vertical tabulator	`VT`	`\v`
backspace	`BS`	`\b`
carriage return	`CR`	`\r`
form feed	`FF`	`\f`
bell	`BEL`	`\a`
backslash	`\`	`\\`
single quote	`'`	`\'`
arbitrary character (hexadecimal)		`\x`*nn*
arbitrary character (octal)		`\0`*nnn*

Table 8.1: Surrogates for special characters.

three-digit octal number. Less digits may be used if the string is unambiguous. For example, a line feed may be specified as `\xa` unless the string continues with another hexadecimal digit (0-F).

For example: The string `Sobel's edge-filter` has to be specified as `'Sobel\'s edge-filter'`. A Windows directory path can be entered as `'C:\\Programs\\MVTec\\Halcon\\images'`

`boolean` The constants `true` and `false` belong to the type `boolean`. The value `true` is internally represented by the number 1 and the value `false` by 0. This means, that in the expression `Val := true` the effective value of `Val` is set to 1. In general, every integer value other than 0 means `true`. Please note that some HALCON operators take logical values for input (e.g., `set_system`). In this case the HALCON operators expect string constants like `'true'` or `'false'` rather than the boolean values `true` or `false`.

In addition to these general types, there are special constants and the type `tuple`, which are specific to HALCON or HDevelop, respectively. Since HALCON 12.0, HDevelop also supports the variable type `vector` (see section 8.6 on page 375).

constants There are constants for the return value (result state) of an operator. The constants can be used together with the operator `dev_error_var` and `dev_set_check`. These constants represent the normal return value of an operator, so called *messages*. For errors no constants are available (there are more than 400 error numbers internally, see the Extension Package Programmer's Manual).

In table 8.2 all return messages can be found.

Additionally, there are constants for the types of control data. These can be compared to the result of a `type` operation to react to different types of control data.

tuple The control types are only used within the generic HDevelop type *tuple*. A tuple of length 1 is interpreted as an atomic value. A tuple may consist of several numerical data items with *different*

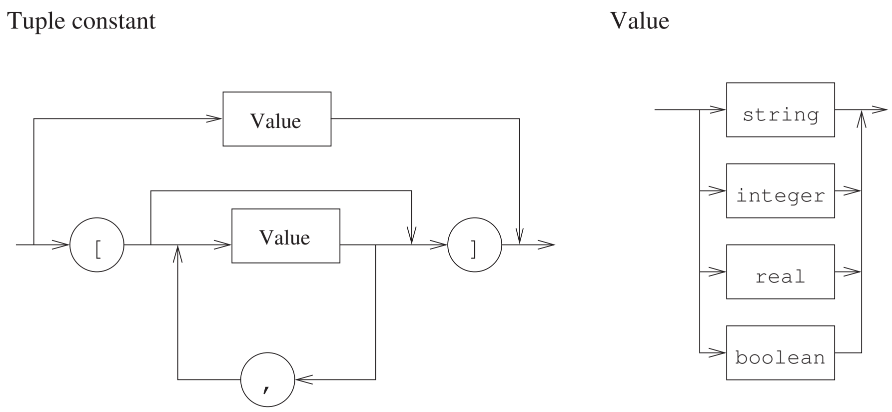

Figure 8.1: The syntax of tuple constants.

types. The standard representation of a tuple is a listing of its elements included into brackets. This is illustrated in figure 8.1.

`[]` specifies the empty tuple. A tuple with just one element is to be considered as a special case, because it can either be specified in the tuple notation or as an atomic value: `[55]` defines the same constant as `55`. Examples for tuples are:

```
[]
4711
0.815
'Text'
[16]
[100.0,100.0,200.0,200.0]
['FileName','Extension']
[4711,0.815,'Hugo']
```

Constant	Meaning	Value
`H_MSG_TRUE`	No error; for tests: (`true`)	2
`H_MSG_FALSE`	For tests: `false`	3
`H_MSG_VOID`	No result could be computed	4
`H_MSG_FAIL`	Operator did not succeed	5

Table 8.2: Return values for operators.

8.3 Variables

The names of variables are built up as usual by composing letters, digits and the underscore '_'. The kind of a variable (iconic or control variable) depends on its position in the parameter list in which the variable identifier is used for the first time (see also section 8.1 on page 349). The kind of the variable is determined during the input of the operator parameters: whenever a new identifier appears, a new variable with the same identifier is created. Control and iconic variables must have different names. The value of a variable (iconic or control) is undefined until the first assignment defines it (the variable has not been instantiated yet). A read access to an undefined variable leads to a runtime error (`Variable <x> not instantiated`).

HDevelop provides a pre-defined variable named _ (single underscore). You can use this variable for output control parameters whose value you are not interested in. Please note that it is not allowed to use this variable for HDevelop-specific operators (chapters `Control` and `Develop` in the HALCON reference manual). **It is not recommended** to use the variable _ in programs that will later be exported to a foreign programming language.

Instantiated variables contain tuples of values. Depending on the kind of the variable, the data items are either iconic objects or control data. The length of the tuple is determined dynamically by the performed operation. A variable can get new values any number of times, but once a value has been assigned the variable will always keep being instantiated, unless you select the menu item Menu `Execute` ▷ `Reset Program Execution`. The content of the variable is deleted before the variable is assigned new values.

The concept of different kinds of variables allows a first ("coarse") typification of variables (control or iconic data), whereas the actual type of the data (e.g., `real`, `integer`, `string`, etc.) is undefined until the variable gets assigned with a concrete value. Therefore, it is possible that the type of a new data item differs from that of the old.

8.3.1 Scope of Variables (local or global)

HDevelop supports local and global variables. All variables are local by default, i.e., they exist only within their procedure. Therefore, local variables with the same name may exist in different procedures without interfering with each other. In contrast, global variables may be accessed in the entire program. They have to be declared explicitly using the operator `global`.

The declaration

```
global tuple File
```

Constand	Meaning	Value
H_TYPE_INT	integer value	1
H_TYPE_REAL	real value	2
H_TYPE_STRING	string value	4
H_TYPE_MIXED	mixed value	8

Table 8.3: Type values for control data.

Language

declares a global *control* variable named File, whereas

```
global object Image
```

declares a global *iconic* variable Image.

The keyword *def* allows to mark one declaration explicitly as the place where the variable is defined, e.g., global def object Image. This is only relevant when exporting the program to a programming language. See the description of the operator global for more information.

Once the global variable is declared, it can be used just like a local variable inside the procedure it has been declared in. If you want to access a global variable in a different procedure, you have to announce this by using the same global ... call (otherwise, a local variable will be created).

```
main procedure:
  * declare global variables
  global tuple File
  global object Image
  ...
  File := 'particle'
  read_image(Image, File)
  process_image()
  * Image has been changed by process_image()
  * File remains unchanged
  ...


process_image procedure:
  * use global variable
  global object Image
  ...
  bin_threshold(Image, Region)
  File := 'fuse'
  read_image(Image, File)
  return()
```

Because procedures have to explicitly announce their use of global variables, existing procedures cannot be broken by introducing global variables in other parts of the program.

By nature, the names of global variables have to be unique in the entire HDevelop program, i.e., all loaded external procedures, the main procedure and all local procedures.

The variable window provides a special tab to list all global variables that are currently declared.

8.4 Operations on Iconic Objects

Iconic objects are exclusively processed by HALCON operators. HALCON operators work on tuples of iconic objects, which are represented by their surrogates in the HALCON data management. The results

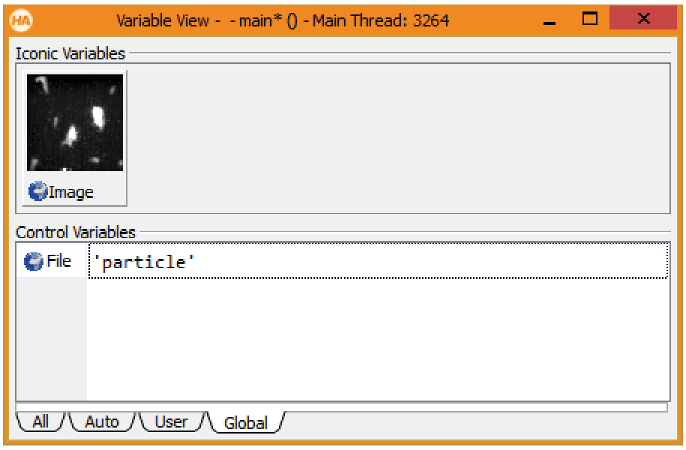

Figure 8.2: Global variables.

of those operators are again tuples of iconic objects or control data elements. For a detailed description of the HALCON operators refer to the HALCON reference manual and the remarks in section 8.5.3 on page 359.

8.5 Expressions for Input Control Parameters

In HDevelop, the use of expressions like arithmetic operations or string operations is limited to control input parameters; all other kinds of parameters must be assigned by variables.

8.5.1 General Features of Tuple Operations

This section intends to give you a short overview over the features of tuples and their operations. A more detailed description of each operator mentioned here is given in the following sections.

Please note that in all following tables variables and constants have been substituted by letters which indicate allowed data types. These letters provide information about possible limitations of the areas of definition. The letters and their meaning are listed in table 8.4. Operations on these symbols can only be applied to parameters of the indicated type or to expressions that return a result of the indicated type.

The symbol names `i`, `a`, `l`, and `s` can denote atomic tuples (tuples of length 1) as well as tuples with arbitrary length.

Operations are normally described assuming atomic tuples. If the tuple contains more than one element, most operators work as follows:

- If one of the tuples is of length one, all elements of the other tuples are combined with that single value for the chosen operation.

Symbol	Types
i	`integer`
a	arithmetic, that is: `integer` or `real`
b	`boolean`
s	`string`
v	all types (atomic)
t	all types (tuple)

Table 8.4: Symbols for the operation description.

Input	Result
`5 * 5`	`25`
`[5] * [5]`	`25`
`[1,2,3] * 2`	`[2,4,6]`
`[1,2,3] * 2.1 + 10`	`[12.1,14.2,16.3]`
`[1,2,3] * [1,2,3]`	`[1,4,9]`
`[1,2,3] * [1,2]`	`runtime error`
`'Text1' + 'Text2'`	`'Text1Text2'`
`17 + '3'`	`'173'`
`'Text ' + 3.1 * 2`	`'Text 6.2'`
`3.1 * (2 + 'Text')`	`runtime error`
`3.1 + 2 + ' Text'`	`'5.1 Text'`
`3.1 + (2 + 'Text')`	`'3.12 Text'`

Table 8.5: Examples for arithmetic operations with tuples and strings.

- If both tuples have a length greater than one, both tuples must have the same length (otherwise a runtime error occurs). In this case, the selected operation is applied to all elements with the same index. The length of the resulting tuples is identical to the length of the input tuples.
- If one of the tuples is of length 0 (`[]`), a runtime error occurs.

In table 8.5 you can find some examples for arithmetic operations with tuples. Pay special attention to the order in which the string concatenations are performed. The basic arithmetic operations in HDevelop are `+`, `-`, `*`, `/`. Please note that `+` is a dimorphic operation: If both operands are numeric, it adds numbers. If at least one of the operands is a `string`, it concatenates both operands as strings.

8.5.2 Assignment

In HDevelop, an assignment is treated like an operator. To use an assignment you have to select the operator `assign (Input, Result)`. This operator has the following semantics: It evaluates `Input` (right side of assignment) and stores it in `Result` (left side of assignment). However, in the program text the assignment is represented by the usual syntax of the assignment operator: `Result := Input`. The following example outlines the difference between an assignment in C syntax and its transformed version in HDevelop:

The assignment in C syntax

```
u = sin(x) + cos(y);
```

is defined in HDevelop using the assignment operator as

```
assign (sin(x) + cos(y), u)
```

which is displayed in the program window as:

```
u := sin(x) + cos(y)
```

If the result of the expression does not need to be stored into a variable, the expression can directly be used as input value for any operator. Therefore, an assignment is necessary only if the value has to be used several times or if the variable has to be initialized (e.g., for a loop).

The assignment operator `assign_at (Index, Value, Result)` is used to modify tuple elements. The call:

```
assign_at (Radius-1, Area, Areas)
```

is not presented in the program text as an operator call, but in the more intuitive form as:

```
Areas[Radius-1] := Area.
```

As an example:

```
Areas := [1,2,3]
Areas[1] := 9
```

sets `Areas` to `[1,9,3]`.

To construct a tuple with `assign_at`, normally an empty tuple is used as initial value and the elements are inserted in a loop:

```
Tuple := []
for i := 0 to 5 by 1
  Tuple[i] := sqrt(real(i))
endfor
```

As you can see from the examples, the indices of a tuple start at 0.

An insertion into a tuple can generally be performed in one of the following ways:

1. In case of appending the value at the 'back' or at the 'front', the tuple concatenation operation , (comma) can be used. Here the operator assign is used with the following parameters:

   ```
   assign ([Tuple,NewVal],Tuple)
   ```

   which is displayed as

   ```
   Tuple := [Tuple,NewVal]
   ```

2. If the index position is somewhere in between, the operator tuple_insert has to be used.

   To insert the tuple [11,12,13] into the tuple [1,2,3] at position 1, use

   ```
   tuple_insert ([1,2,3], 1, [11,12,13], Result)
   ```

   resulting in [1,11,12,13,2,3].

In the following example regions are dilated with a circle mask and afterwards the areas are stored into the tuple Areas. In this case the operator assign_at is used.

```
read_image (Mreut, 'mreut')
threshold (Mreut, Region, 190, 255)
Areas := []
for Radius := 1 to 50 by 1
  dilation_circle (Region, RegionDilation, Radius)
  area_center (RegionDilation, Area, Row, Column)
  Areas[Radius-1] := Area
endfor
```

Please note that first the variable Areas has to be initialized in order to avoid a runtime error. In the example Areas is initialized with the empty tuple ([]). Instead of assign_at the operator assign with tuple concatenation

```
Areas := [Areas,Area]
```

could be used, because the element is appended at the back of the tuple. More examples can be found in the program assign.hdev.

Operation	Meaning	HALCON operator
`t := [t1,t2]`	concatenate tuples	tuple_concat
`i := |t|`	get number of elements of tuple `t`	tuple_length
`v := t[i]`	select element `i` of tuple `t`; 0 <= `i` < `|t|`	tuple_select
`t := t[i1:i2]`	select from element `i1` to element `i2` of tuple `t`	tuple_select_range
`t := subset(t,i)`	select elements specified in `i` from `t`	tuple_select
`t := select_mask(t1,t2)`	select all elements from `t1` where the corresponding mask value in `t2` is greater than 0	tuple_select_mask
`t := remove(t,i)`	remove elements specified in `i` from `t`	tuple_remove
`i := find(t1,t2)`	get indices of all occurrences of `t2` within `t1` (or `-1` if no match)	tuple_find
`t := uniq(t)`	discard all but one of successive identical elements from `t`	tuple_uniq
`t := [i1:i2:i3]`	generate a sequence of values from `i1` to `i3` with an increment value of `i2`	tuple_gen_sequence
`t := [i1:i2]`	generate a sequence of values from `i1` to `i2` with an increment value of one	tuple_gen_sequence

Table 8.6: Basic operations on tuples (control data) and the corresponding HALCON operators.

8.5.3 Basic Tuple Operations

A basic tuple operation may be selecting one or more values, combining tuples (concatenation) or getting the number of elements (see table 8.6 for operations on tuples containing control data).

The concatenation accepts one or more variables or constants as input. They are all listed between the brackets, separated by commas. The result again is a tuple. Please note the following: `[[t]] = [t] = t`.

`|t|` returns the number of elements of a tuple. The indices of elements range from zero to the number of elements minus one (i.e., `|t|-1`). Therefore, the selection index has to be within this range. [1]

```
Tuple := [V1,V2,V3,V4]
```

[1] Please note that the index of objects (e.g., select_obj) ranges from 1 to the number of elements.

control	iconic
`[]`	`gen_empty_obj()`
`[t1,t2]`	`concat_obj(p1, p2, q)`
`|t|`	`count_obj(p, num)`
`t[i]`	`select_obj(p, q, i+1)`
`t[i1:i2]`	`copy_obj(p, q, i1+1, i2-i1+1)`

Table 8.7: Equivalent tuple operations for control and iconic data.

```
for i := 0 to |Tuple|-1 by 1
  fwrite_string (FileHandle,Tuple[i]+'\n')
endfor
```

In the following examples the variable `Var` contains `[2,4,8,16,16,32]`:

`[1,Var,[64,128]]`	`[1,2,4,8,16,16,32,64,128]`
`|Var|`	`6`
`Var[4]`	`16`
`Var[2:4]`	`[8,16,16]`
`subset(Var,[0,2,4])`	`[2,8,16]`
`select_mask(Var,[1,0,0,1,1,1])`	`[2,16,16,32]`
`remove(Var,[2,3])`	`[2,4,16,32]`
`find(Var,[8,16])`	`2`
`uniq(Var)`	`[2,4,8,16,32]`

Further examples can be found in the program `tuple.hdev`. The HALCON operators that correspond to the basic tuple operations are listed in table 8.6 on page 359.

Note that these direct operations cannot be used for iconic tuples, i.e., iconic objects cannot be selected from a tuple using `[]` and their number cannot be directly determined using `||`. For this purpose, however, HALCON operators are offered that carry out the equivalent tasks. In table 8.7 you can see tuple operations that work on control data (and which are applied via `assign` or `assign_at`) and their counterparts that work on iconic data (and which are independent operators). In the table the symbol `t` represents a control tuple, and the symbols `p` and `q` represent iconic tuples.

8.5.4 Tuple Creation

The simplest way to create a tuple, as mentioned in section 8.2 on page 350, is the use of constants together with the operator `assign` (or in case of iconic data one of its equivalents shown in table 8.7):

```
assign ([],empty_tuple)
assign (4711,one_integer)
assign ([4711,0.815],two_numbers)
```

This code is displayed as

```
empty_tuple := []
one_integer := 4711
two_numbers := [4711,0.815]
```

This is useful for constant tuples with a fixed (small) length. More general tuples can be created by successive application of the concatenation or the operator assign_at together with variables, expressions or constants. If we want to generate a tuple of length 100, where each element has the value 4711, it might be done like this:

```
tuple := []
for i := 1 to 100 by 1
  tuple := [tuple,4711]
endfor
```

Because this is not very convenient a special function called gen_tuple_const is available to construct a tuple of a given length, where each element has the same value. Using this function, the program from above is reduced to:

```
tuple := gen_tuple_const(100,4711)
```

A fast way to create a sequence of values with a common increment is to use tuple_gen_sequence. For example, to create a tuple containing the values 1..1000, use

```
tuple_gen_sequence(1,1000,1,Sequence)
```

An alternative syntax to the above is to write:

```
Sequence := [1:1:1000]
```

If the increment value is one (as in the above example), it is also possible to write:

```
Sequence := [1:1000]
```

If we want to construct a tuple with the same length as a given tuple there are two ways to get an easy solution. The first one is based on gen_tuple_const:

```
tuple_new := gen_tuple_const(|tuple_old|,4711)
```

Language

Operation	Meaning	HALCON operator
`b := is_int(t)`	test for integer values	tuple_is_int
`b := is_mixed(t)`	test for mixed values	tuple_is_mixed
`b := is_number(t)`	test for numerical values	tuple_is_number
`b := is_real(t)`	test for real values	tuple_is_real
`b := is_string(t)`	test for string values	tuple_is_string
`i := type(t)`	get type value	tuple_type

Table 8.8: Type operations.

The second one is a bit tricky and uses arithmetic functions:

```
tuple_new := (tuple_old * 0) + 4711
```

Here we get first a tuple of the same length with every element set to zero. Then, we add the constant to each element.

In the case of tuples with different values we have to use the loop version to assign the values to each position:

```
tuple := []
for i := 1 to 100 by 1
  tuple := [tuple,i*i]
endfor
```

In this example we construct a tuple with the square values from 1^2 to 100^2.

8.5.5 Type Operations

The type operations allow to test or query the value type of control data. See table 8.3 on page 353 for the corresponding type constants.

There are also corresponding operations that test each element of the input tuple.

8.5.6 Basic Arithmetic Operations

See table 8.10 for an overview of the available basic arithmetic operations.

All operations are left-associative, except the right-associative unary minus operator. The evaluation usually is done from left to right. However, parentheses can change the order of evaluation and some operators have a higher precedence than others (see section 8.5.16).

Operation	Meaning	HALCON operator
`t := is_int_elem(t)`	elementwise test for integer values	tuple_is_int_elem
`t := is_real_elem(t)`	elementwise test for real values	tuple_is_real_elem
`t := is_string_elem(t)`	elementwise test for string values	tuple_is_string_elem
`t := type_elem(t)`	get type value elementwise	tuple_type_elem

Table 8.9: Elementwise type operations.

Operation	Meaning	HALCON operator
`a1 / a2`	division	tuple_div
`a1 * a2`	multiplication	tuple_mult
`a1 % a2`	modulus	tuple_mod
`a1 + a2`	addition	tuple_add
`a1 - a2`	subtraction	tuple_sub
`-a`	negation	tuple_neg

Table 8.10: Basic arithmetic operations.

The arithmetic operations in HDevelop match the usual definitions. Expressions can have any number of parentheses.

The division operator (`a1 / a2`) can be applied to `integer` as well as to `real`. The result is of type `real`, if at least one of the operands is of type `real`. If both operands are of type `integer`, the division is an integer division. The remaining arithmetic operators (multiplication, addition, subtraction, and negation) can be applied to either `integer` or `real` numbers. If at least one operand is of type `real`, the result will be a `real` number as well.

Examples:

Expression	Result
`4/3`	1
`4/3.0`	1.3333333
`(4/3) * 2.0`	2.0

Simple examples can be found in the program `arithmetic.hdev`.

8.5.7 Bit Operations

This section describes the operators for bit processing of numbers. The operands have to be integers.

The result of `lsh(i1,i2)` is a bitwise left shift of `i1` that is applied `i2` times. If there is no overflow this is equivalent to a multiplication by 2^{i2}. The result of `rsh(i1,i2)` is a bitwise right shift of `i1` that

Operation	Meaning	HALCON operator
`lsh(i1,i2)`	left shift	`tuple_lsh`
`rsh(i1,i2)`	right shift	`tuple_rsh`
`i1 band i2`	bitwise and	`tuple_band`
`i1 bxor i2`	bitwise xor	`tuple_bxor`
`i1 bor i2`	bitwise or	`tuple_bor`
`bnot i`	bitwise complement	`tuple_bnot`

Table 8.11: Bit operations.

is applied `i2` times. For non-negative `i1` this is equivalent to a division by 2^{i2}. For negative `i1` the result depends on the used hardware. For `lsh` and `rsh` the result is undefined if the second operand has a negative value or the value is larger than 32. More examples can be found in the program `bit.hdev`.

8.5.8 String Operations

There are several string operations available to modify, select, and combine strings. Furthermore, some operations allow to convert numbers (`real` and `integer`) to strings.

`v$s`	convert `v` using specification `s`
`v1 + v2`	concatenate `v1` and `v2`
`strchr(s1,s2)`	search character `s2` in `s1`
`strstr(s1,s2)`	search substring `s2` in `s1`
`strrchr(s1,s2)`	search character `s2` in `s1` (reverse)
`strrstr(s1,s2)`	search substring `s2` in `s1` (reverse)
`strlen(s)`	length of string
`s{i}`	select character at position `i`; 0 <= `i` <= `strlen(s)-1`
`s{i1:i2}`	select substring from position `i1` to position `i2`
`split(s1,s2)`	split `s1` in substrings at `s2`
`regexp_match(s1,s2)`	extract substrings of `s1` matching the regular expression `s2`
`regexp_replace(s1,s2,s3)`	replace substrings of `s1` matching the regular expression `s2` with `s3`
`regexp_select(s1,s2)`	select tuple elements from `s1` matching the regular expression `s2`
`regexp_test(s1,s2)`	return how many tuple elements in `s1` match the regular expression `s2`

Table 8.12: String operations.

$ (string conversion)

See also: `tuple_string`.

$ converts numbers to strings or modifies strings. The operation has two operands: The first one (left of the $) is the number that has to be converted. The second one (right of the $) specifies the conversion. It is comparable to the format string of the `printf()` function in the C programming language. This format string consists of the following four parts

<flags><width>.<precision><conversion>

or as a regular expression:

```
[-+ #]?([0-9]+)?(\.[0-9]*)?[doxXfeEgGsb]?
```

(which roughly translates to zero or more of the characters in the first bracket pair followed by zero or more digits, optionally followed by a dot which may be followed by digits followed by a conversion character from the last bracket pair).

Some conversion examples might show it best:

Input	Output
`23 $ '10.2f'`	`'     23.00'`
`23 $ '-10.2f'`	`'23.00     '`
`4 $ '.7f'`	`'4.0000000'`
`1234.56789 $ '+10.3f'`	`' +1234.568'`
`255 $ 'x'`	`'ff'`
`255 $ 'X'`	`'FF'`
`0xff $ '.5d'`	`'00255'`
`'total' $ '10s'`	`'     total'`
`'total' $ '-10s'`	`'total     '`
`'total' $ '10.3'`	`'       tot'`

flags Zero or more flags, in any order, which modify the meaning of the conversion specification. Flags may consist of the following characters:

- `-` The result of the conversion is left justified within the field.
- `+` The result of a signed conversion always begins with a sign, `+` or `-`.
- **Space** If the first character of a signed conversion is not a sign, a space character is prefixed to the result.
- `#` The value is to be converted to an "alternate form". For `d` and `s` (see below) conversions, this flag has no effect. For `o` conversion (see below), it increases the precision to force the first digit of the result to be a zero. For `x` or `X` conversion (see below), a non-zero result has `0x`

or `0X` prefixed to it. For `e`, `E`, `f`, `g`, and `G` conversions, the result always contains a radix character, even if no digits follow the radix character. For `g` and `G` conversions, trailing zeros are not removed from the result, contrary to usual behavior.

width An optional string of decimal digits to specify a minimum field width. For an output field, if the converted value has fewer characters than the field width, it is padded on the left (or right, if the left-adjustment flag - has been given) to the field width.

precision The precision specifies the minimum number of digits to appear for integer conversions (the field is padded with leading zeros), the number of digits to appear after the radix character for the `e` and `f` conversions, the maximum number of significant digits for the `g` conversion, or the maximum number of characters to be printed from a string conversion. The precision takes the form of a period `.` followed by a decimal digit string. A null digit string is treated as a zero.

conversion A conversion character indicates the type of conversion to be applied:

`d`, `o`, `x`, `X` The integer argument is printed in signed decimal (`d`), unsigned octal (`o`), or unsigned hexadecimal notation (`x` and `X`). The `x` conversion uses the numbers and lower-case letters `0123456789abcdef`, and the `X` conversion uses the numbers and upper-case letters `0123456789ABCDEF`. The precision component of the argument specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits than the specified minimum, it is expanded with leading zeroes. The default precision is 1. The result of converting a zero value with a precision of 0 is no characters.

`f` The floating-point number argument is printed in decimal notation in the style `[-]`*ddd*`.`*ddd*, where the number of digits after the radix character, `.`, is equal to the precision specification. If the precision is omitted from the argument, six digits are output; if the precision is explicitly 0, no radix appears.

`e`,`E` The floating-point-number argument is printed in the style `[-]`*d*`.`*ddd*`e+`*dd*, where there is one digit before the radix character, and the number of digits after it is equal to the precision. When the precision is missing, six digits are produced; if the precision is 0, no radix character appears. The `E` conversion character produces a number with `E` introducing the exponent instead of `e`. The exponent always contains at least two digits. However, if the value to be printed requires an exponent greater than two digits, additional exponent digits are printed as necessary.

`g`, `G` The floating-point-number argument is printed in style `f` or `e` (or in style `E` in the case of a `G` conversion character), with the precision specifying the number of significant digits. The style used depends on the value converted; style `e` is used only if the exponent resulting from the conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the result. A radix character appears only if it is followed by a digit.

`s` The argument is taken to be a string, and characters from the string are printed until the end of the string or the number of characters indicated by the precision specification of the argument is reached. If the precision is omitted from the argument, it is interpreted as infinite and all characters up to the end of the string are printed.

In no case does a nonexistent or insufficient field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result.

Examples for the string conversion can be found in the program `string.hdev`.

+ (string concatenation)

The string concatenation (+) can be applied in combination with strings or all numerical types; if necessary, the operands are first transformed into strings (according to their standard representation). At least one of the operands has to be already a string so that the operator can act as a string concatenator. In the following example a file name (e.g., `'Name5.tiff'`) is generated. For this purpose two string constants (`'Name'` and `'.tiff'`) and an integer value (the loop-index `i`) are concatenated:

```
for i := 1 to 5 by 1
  read_image (Image, 'Name'+i+'.tiff')
endfor
```


`str(r)chr`

See also: `tuple_strchr`, `tuple_strrchr`.

`str(r)chr(s1,s2)` returns the index of the first (last) occurrence of one of the character in `s2` in string `s1`, or `-1` if none of the characters occur in the string. `s1` may be a single string or a tuple of strings.

`str(r)str`

See also: `tuple_strstr`, `tuple_strrstr`.

`str(r)str(s1,s2)` returns the index of the first (last) occurrence of string `s2` in string `s1`, or `-1` if `s2` does not occur in the string. `s1` may be a single string or a tuple of strings.

`strlen`

See also: `tuple_strlen`.

`strlen(s)` returns the number of characters in `s`.

`{}`

See also: `tuple_str_bit_select`.

`s{i}` selects a single character (specified by index position) from `s`. The index ranges from zero to the length of the string minus 1. The result of the operator is a string of length one.

`s{i1:i2}` returns all characters from the first specified index position (`i1`) up to the second specified position (`i2`) in `s` as a string. The index ranges from zero to the length of the string minus 1.

`split`

See also: `tuple_split`.

`split(s1,s2)` divides the string `s1` into single substrings. The string is split at those positions where it contains a character from `s2`. As an example the result of

```
split('/usr/image:/usr/proj/image',':')
```

consists of the two strings

```
['/usr/image','/usr/proj/image']
```

Regular Expressions

HDevelop provides string functions that use Perl compatible regular expressions. Detailed information about them can be found in the Reference Manual at the descriptions of the corresponding operators, which have the same name but start with `tuple_`. In particular, at the description of `tuple_regexp_match` you find further information about the used syntax, a list of possible options, and a link to suitable literature about regular expressions.

`regexp_match`

See also: `tuple_regexp_match`.

`regexp_match(s1,s2)` searches for elements of the tuple `s1` that match the regular expression `s2`. It returns a tuple with the same size as the input tuple (exceptions exist when working with capturing groups, see the description of `tuple_regexp_match` in the Reference Manual for details). The resulting tuple contains the matching results for each tuple element of the input tuple. For a successful match the matching substring is returned. Otherwise, an empty string is returned.

`regexp_replace`

See also: `tuple_regexp_replace`.

`regexp_replace(s1,s2,s3)` replaces substrings in `s1` that match the regular expression `s2` with the string given in `s3`. By default, only the *first* matching substring of each element in `s1` is replaced. To replace all occurrences, the option `'replace_all'` has to be set in `s2` (see `tuple_regexp_replace`).

For example:

```
assign(regexp_replace(List, '\\.jpg$', '.png'), List)
```

substitutes file names that look like JPEG images with PNG images.

`regexp_select`

See also: `tuple_regexp_select`.

`regexp_select(s1,s2)` returns only the elements of the tuple `s1` that match the regular expression `s2`. In contrast to `regexp_match`, the original tuple elements instead of the matching substrings are returned. Tuple elements that do not match the regular expression are discarded.

For example:

```
assign(regexp_select(List, '\\.jpg$'), Selection)
```

sets `Selection` to all the strings from `List` that look like file names of JPEG images. Please note that the backslash character has to be escaped to be preserved.

Operation	Meaning	HALCON operator	Alternative notation
`t1 < t2`	less than	`tuple_less`	
`t1 > t2`	greater than	`tuple_greater`	
`t1 <= t2`	less or equal	`tuple_less_equal`	
`t1 >= t2`	greater of equal	`tuple_greater_equal`	
`t1 == t2`	equal	`tuple_equal`	`t1 = t2`
`t1 != t2`	not equal	`tuple_not_equal`	`t1 # t2`

Table 8.13: Comparison operations.

`regexp_test`

See also: `tuple_regexp_test`.

`regexp_test(s1,s2)` returns the number of elements of the tuple `s1` that match the regular expression `s2`. Additionally, a short-hand notation of the operator is available, which is convenient in conditional expressions:

```
s1 =~ s2
```

8.5.9 Comparison Operations

In HDevelop, the comparison operations are defined not only on atomic values, but also on tuples with an arbitrary number of elements. They always return values of type `boolean`. Table 8.13 shows all comparison operations.

`t1 == t2` and `t1 != t2` are defined on all types. Two tuples are equal (`true`), if they have the same length and all the data items on each index position are equal. If the operands have different types (`integer` and `real`), the integer values are first transformed into `real` numbers. Values of type `string` cannot be mixed up with numbers, i.e., `string` values are considered to be not equal to values of other types.

The four comparison operations compute the lexicographic order of tuples. On equal index positions the types must be identical, however, values of type `integer`, `real`, and `boolean` are adapted automatically. The lexicographic order applies to strings, and the boolean `false` is considered to be smaller than the boolean `true` (`false < true`). In the program `compare.hdev` you can find examples for the comparison operations.

8.5.10 Elementwise Comparison Operations

These comparison operations compare the input tuples `t1` and `t2` elementwise.

If both tuples have the same length, the corresponding elements of both tuples are compared. Otherwise, either `t1` or `t2` must have length 1. In this case, the comparison is performed for each element of the

1st Operand	Operation	2nd Operand	Result
`1`	`==`	`1.0`	`true`
`[]`	`==`	`[]`	`true`
`''`	`==`	`[]`	`false`
`[1,'2']`	`==`	`[1,2]`	`false`
`[1,2,3]`	`==`	`[1,2]`	`false`
`[4711,'Hugo']`	`==`	`[4711,'Hugo']`	`true`
`'Hugo'`	`==`	`'hugo'`	`false`
`2`	`>`	`1`	`true`
`2`	`>`	`1.0`	`true`
`[5,4,1]`	`>`	`[5,4]`	`true`
`[2,1]`	`>`	`[2,0]`	`true`
`true`	`>`	`false`	`true`
`'Hugo'`	`<`	`'hugo'`	`true`

Table 8.14: Examples for the comparison of tuples.

Operation	Meaning	HALCON operator	Alternative notation
`t1 [<] t2`	less than	`tuple_less_elem`	
`t1 [>] t2`	greater than	`tuple_greater_elem`	
`t1 [<=] t2`	less or equal	`tuple_less_equal_elem`	
`t1 [>=] t2`	greater of equal	`tuple_greater_equal_elem`	
`t1 [==] t2`	equal	`tuple_equal_elem`	`t1 [=] t2`
`t1 [!=] t2`	not equal	`tuple_not_equal_elem`	`t1 [#] t2`

Table 8.15: Elementwise comparison operations.

longer tuple with the single element of the other tuple. As a precondition for comparing the tuples elementwise two corresponding elements must either both be (integer or floating point) numbers or both be strings.

1st Operand	Operation	2nd Operand	Result
`[1,2,3]`	`[<]`	`[3,2,1]`	`[1,0,0]`
`['a','b','c']`	`[==]`	`'b'`	`[0,1,0]`
`['a','b','c']`	`[<]`	`['b']`	`[1,0,0]`

Table 8.16: Examples for the elementwise comparison of tuples.

Operation	Meaning	HALCON operator
`l1 and l2`	logical 'and'	`tuple_and`
`l1 xor l2`	logical 'xor'	`tuple_xor`
`l1 or l2`	logical 'or'	`tuple_or`
`not l`	negation	`tuple_not`

Table 8.17: Boolean operations.

8.5.11 Boolean Operations

The boolean operations `and`, `xor`, `or`, and `not` are defined only for tuples of length 1. `l1 and l2` is set to `true` (1) if both operands are `true` (1), whereas `l1 xor l2` returns `true` (1) if exactly one of both operands is true. `l1 or l2` returns `true` (1) if at least one of the operands is `true` (1). `not l` returns `true` (1) if the input is `false` (0), and `false` (0), if the input is `true` (1).

8.5.12 Trigonometric Functions

All these functions work on tuples of numbers as arguments. The input can either be of type `integer` or `real`. However, the resulting type will be of type `real`. The functions are applied to all tuple values, and the resulting tuple has the same length as the input tuple. For `atan2` the two input tuples have to be of equal length. table 8.18 shows the provided trigonometric functions. For the trigonometric functions the angle is specified in radians.

Operation	Meaning	HALCON Operator
`sin(a)`	sine of `a`	`tuple_sin`
`cos(a)`	cosine of `a`	`tuple_cos`
`tan(a)`	tangent of `a`	`tuple_tan`
`asin(a)`	arc sine of `a` in the interval $[-\pi/2, \pi/2], a \in [-1, 1]$	`tuple_asin`
`acos(a)`	arc cosine `a` in the interval $[-\pi/2, \pi/2], a \in [-1, 1]$	`tuple_acos`
`atan(a)`	arc tangent `a` in the interval $[-\pi/2, \pi/2], a \in [-\infty, +\infty]$	`tuple_atan`
`atan2(a1,a2)`	arc tangent `a1/a2` in the interval $[-\pi, \pi]$	`tuple_atan2`
`sinh(a)`	hyperbolic sine of `a`	`tuple_sinh`
`cosh(a)`	hyperbolic cosine of `a`	`tuple_cosh`
`tanh(a)`	hyperbolic tangent of `a`	`tuple_tanh`

Table 8.18: Trigonometric functions.

Operation	Meaning	HALCON operator
`exp(a)`	exponential function $e^{\mathtt{a}}$	`tuple_exp`
`log(a)`	natural logarithm $\ln(\mathtt{a})$, a > 0	`tuple_log`
`log10(a)`	decadic logarithm, $\log_{10}(a)$, a > 0	`tuple_log10`
`pow(a1,a2)`	$a1^{a2}$	`tuple_pow`
`ldexp(a1,a2)`	$a1 \cdot 2^{a2}$	`tuple_ldexp`

Table 8.19: Exponential functions.

8.5.13 Exponential Functions

All these functions work on tuples of numbers as arguments. The input can either be of type `integer` or `real`. However, the resulting type will be of type `real`. The functions are applied to all tuple values and the resulting tuple has the same length as the input tuple. For `pow` and `ldexp` the two input tuples have to be of equal length.

See table 8.19 for the provided exponential functions.

8.5.14 Numerical Functions

The numerical functions shown in table 8.20 work on different data types.

The functions `min` and `max` select the minimum and the maximum values of the tuple values. All of these values either have to be of type `string`, or `integer/real`. It is not allowed to mix strings with numerical values. The resulting value will be of type `real`, if at least one of the elements is of type `real`. If all elements are of type `integer` the resulting value will also be of type `integer`. The same applies to the function `sum` that determines the sum of all values. If the input arguments are strings, string concatenation will be used instead of addition.

The functions `mean`, `deviation`, `sqrt`, `deg`, `rad`, `fabs`, `ceil`, `floor` and `fmod` work with `integer` and `real`; the result is always of type `real`. The function `mean` calculates the mean value and `deviation` the standard deviation of numbers. `sqrt` calculates the square root of a number.

`cumul` returns the different cumulative sums of the corresponding elements of the input tuple, and `median` calculates the median of a tuple. For both functions, the resulting value will be of type `real`, if at least one of the elements is of type `real`. If all elements are of type `integer` the resulting value will also be of type `integer`. `select_rank` returns the element at rank `i` and works for tuples containing `int` or `real` values. The index `i` is of type `int`.

`deg` and `rad` convert numbers from radians to degrees and from degrees to radians, respectively.

`real` converts an `integer` to a `real`. For `real` as input it returns the input. `int` converts a `real` to an `integer` and truncates it. `round` converts a `real` to an `integer` and rounds the value. For `integer` it returns the input. The function `abs` always returns the absolute value that is of the same type as the input value.

The following example (file name: `euclid_distance.hdev`) shows the use of some numerical functions:

Operation	Meaning	HALCON operator
min(t)	minimum value of the tuple	tuple_min
min2(t1,t2)	elementwise minimum of two tuples	tuple_min2
max(t)	maximum value of the tuple	tuple_max
max2(t1,t2)	elementwise maximum of two tuples	tuple_max2
sum(t)	sum of all tuple elements or string concatenation	tuple_sum
mean(a)	mean value	tuple_mean
deviation(a)	standard deviation	tuple_deviation
cumul(a)	cumulative sums of a tuple	tuple_cumul
median(a)	median of a tuple	tuple_median
select_rank(a,i)	element at rank i of a tuple	tuple_select_rank
sqrt(a)	square root $\sqrt{a}$	tuple_sqrt
deg(a)	convert radians to degrees	tuple_deg
rad(a)	convert degrees to radians	tuple_rad
real(a)	convert integer to real	tuple_real
int(a)	truncate real to integer	tuple_int
round(a)	convert real to integer	tuple_round
abs(a)	absolute value of a (integer or real)	tuple_abs
fabs(a)	absolute value of a (always real)	tuple_fabs
ceil(a)	smallest integer value not smaller than a	tuple_ceil
floor(a)	largest integer value not greater than a	tuple_floor
fmod(a1,a2)	fractional part of a1/a2, with the same sign as a1	tuple_fmod
sgn(a)	elementwise sign of a tuple	tuple_sgn

Table 8.20: Numerical functions.

```
V1 := [18.8,132.4,33,19.3]
V2 := [233.23,32.786,234.4224,63.33]
Diff := V1 - V2
Distance := sqrt(sum(Diff * Diff))
Dotvalue := sum(V1 * V2)
```

First, the Euclidian distance of the two vectors V1 and V2 is computed, by using the formula:

$$d = \sqrt{\sum_i (V1_i - V2_i)^2}$$

The difference and the multiplication (square) are successively applied to each element of both vectors.

Operation	Meaning	HALCON operator
sort(t)	sorting in increasing order	tuple_sort
sort_index(t)	return index instead of values	tuple_sort_index
inverse(t)	reverse the order of the values	tuple_inverse
is_number(v)	test if value is a number	tuple_is_number
number(v)	convert string to a number	tuple_number
environment(s)	value of an environment variable	tuple_environment
ord(a)	ASCII number of a character	tuple_ord
chr(a)	convert an ASCII number to a character	tuple_chr
ords(s)	ASCII number of a tuple of strings	tuple_ords
chrt(i)	convert a tuple of integers into a string	tuple_chrt
rand(a)	create random numbers	tuple_rand

Table 8.21: Miscellaneous functions.

Afterwards `sum` computes the sum of the squares. Then the square root of the sum is calculated. After that the dot product of `V1` and `V2` is determined by the formula:

$$\langle V1, V2 \rangle = \sum_i (V1_i * V2_i)$$

8.5.15 Miscellaneous Functions

`sort` sorts the tuple values in ascending order, that means, that the first value of the resulting tuple is the smallest one. But again: strings must not be mixed up with numbers. `sort_index` sorts the tuple values in ascending order, but in contrast to `sort` it returns the index positions (0..) of the sorted values.

The function `inverse` reverses the order of the tuple values. Both `sort` and `inverse` are identical, if the input is empty, if the tuple is of length 1, or if the tuple contains only one value in all positions, e.g., `[1,1,...,1]`.

`is_number` returns `true` for variables of the type `integer` or `real` and for variables of the type `string` that represent a number.

The function `number` converts a `string` representing a number to an `integer` or a `real` depending on the type of the number. Note that strings starting with `0x` are interpreted as hexadecimal numbers, and strings starting with `0` (zero) as octal numbers; for example, the string `'20'` is converted to the integer `20`, `'020'` to `16`, and `'0x20'` to `32`. If called with a `string` that does not represent a number or with a variable of the type `integer` or `real`, `number` returns a copy of the input.

`environment` returns the value of an environment variable. Input is the name of the environment variable as a string.

`ord` gives the ASCII number of a character as an `integer`. `chr` converts an ASCII number to a character.

`band`
`bxor bor`
`and`
`xor or`
`!= == # =`
`<= >= < >`
`+ -`
`/ * %`
`- (unary minus) not`
`$`

Table 8.22: Operation precedence (increasing from top to bottom).

`ords` converts a tuple of strings into a tuple of (ASCII) integers. `chrt` converts a tuple of integers into a string.

8.5.16 Operation Precedence

See table 8.22 for the precedence of the operations for control data. Some operations (like functions, `| |`, `t[]`, etc.) are left out, because they mark their arguments clearly.

8.6 Vectors

A vector is a container that can hold an arbitrary number of elements, all of which must have the exact same variable type (i.e., tuple, iconic object, or vector). The variable type "vector" is specific to HDevelop. It is available in HDevelop 12.0 or higher. Please note that programs utilizing vector variables cannot be executed in older versions of HDevelop.

A vector of tuples or objects is called one-dimensional, a vector of vectors of tuples or objects is two-dimensional, and so on. Once a vector is defined, its type cannot change within the program, i.e., its dimension has to remain constant, and vectors of tuples must not be assigned iconic objects or vice versa.

This is the definition of a vector in EBNF (Extended Backus-Naur Form) grammar:

```
vector     = "{" list "}" ;
list       = tuplelist | objectlist | vectorlist ;
tuplelist  = tuple, {",", tuple} ;
objectlist = object, {",", object} ;
vectorlist = vector, {",", vector} ;
tuple      = "[" control "]" ;
control    = string | integer | real | boolean ;
```

Language

Construction of Vectors

A vector is defined by providing a comma-separated list of its elements in curly brackets.

```
vectorT := {[1], [2], [3]}      // one-dimensional vector
```

This is equivalent to

```
vectorT := {1, 2, 3}            // tuples of length 1 do not require square brackets
```

Of course, variable names or arbitrary expressions can be used instead of constants.

```
t1 := 1
vectorT := {t1, t1 * 2, 3}
```

The following example defines a vector of iconic objects.

```
read_image (Image, 'clip')
threshold (Image, Region, 0, 63)
connection (Region, ConnectedRegions)
vectorO := {Image, Region, ConnectedRegions}
```

The following example defines a two-dimensional vector variable.

```
vectorV := {vectorT, {[4,5,6,7], [8,9]}}
```

It is also possible to define vector variables using the `.at()` and `.insert()` expressions (see below).

Accessing and Setting Vector Elements

A single vector element is accessed using the `.at()` expression, the argument of which ranges from 0 to the number of vector elements minus 1. Several `.at()` expressions can be combined to access the subelements of multi-dimensional vectors. It is a runtime error to access non-existing vector elements.

```
tuple  := vectorT.at(0)         // tuple  := 1
region := vectorO.at(1)         // region := Region
vector := vectorV.at(0)         // vector := {[1,2,3]}
tuple  := vectorV.at(1).at(1)   // tuple  := [8, 9]
```

The `.at()` expression is also used to set vector elements. Writing to a non-existing vector element is allowed. If necessary, the vector is automatically filled with empty elements.

```
vectorT.at(2) := 33            // vectorT := {[1], [2], [33]}
vectorE.at(4) := 'text'        // vectorE := {[], [], [], [], 'text'}
```

The `.insert()` expression specifies an index position and a value. It shifts the values from the given index to the end by one positon, and sets the value at the index to the new value.

The `.remove()` expression performs the opposite operation. It removes the value at the specified position and moves all following values to the left.

```
vectorT.insert(1, 99)          // vectorT := {[1], [99], [2], [33]}
vectorT.remove(2)              // vectorT := {[1], [99], [33]}
```

Like with the `.at()` expression, the vector is automatically filled with empty elements if necessary.

```
vectorNew.insert(2,3)         // vectorNew := {[], [], [3]}
```

The `.concat()` expression concatenates two vectors of the same type and dimension.

```
vectorC := vectorT.concat(vectorNew)   // vectorC := {[1], [99], [33], [], [], [3]}
```

Getting the Number of Vector Elements

The number of vector elements is queried using the `length()` expression.

```
i := vectorT.length()              // i := 3
j := vectorV.length()              // j := 2
k := vectorV.at(0).length()        // k := 3
l := vectorV.at(1).at(0).length()  // l := 4
```

Clearing a Vector Variable

The `.clear()` expressions removes all elements from the corresponding vector variable. Note however that the cleared vector still keeps its variable type.

```
vectorT.clear()
vectorT := vectorO              // illegal: vectorT is a tuple vector
vectorT := vectorV              // illegal: vectorT is one-dimensional
```

Modifying Expressions

The expressions `.clear()`, `.insert()`, and `.remove()` are special in that they modify the input vector. In addition, they return the modified input vector and can be used in assignments.

Testing Vector Variables for (In)equality

The expressions == and `!=` are used to test two vector variables for equality or inequality, respectively.

Converting Vectors to Tuples and Vice-versa

A vector variable can be flattened to a tuple using the convenience operator convert_vector_to_tuple. It concatenates all tuple values that are stored in the input vector and stores them in the output tuple.

```
convert_vector_to_tuple (vectorV, T)    // T := [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The convenience operator convert_tuple_to_vector_1d stores the elements of the input tuple as single elements of the one-dimensional output vector.

```
convert_tuple_to_vector_1d (T, V)   // V := {[1],[2],[3],[4],[5],[6],[7],[8],[9]}
```

8.7 Reserved Words

The identifiers listed in table 8.23 are reserved words and their usage is strictly limited to their predefined meaning. They cannot be used as variable names.

8.8 Control Flow Operators

The operators introduced in this section execute a block of operators conditionally or repeatedly. Usually, these operators come in pairs: One operator marks the start of the block while the other marks the end. The code lines inbetween are referred to as the body of a control flow structure.

When you enter a control flow operator to start a block, HDevelop also adds the corresponding closing operator by default to keep the program code balanced. In addition, the IC is placed between the control flow operators. This is fine for entering new code blocks. If you want to add control flow operators to existing code, you can also add the operators individually. Keep in mind, however, that a single control flow operator is treated as invalid code until its counterpart is entered as well.

In the following, <condition> is an expression that evaluates to an integer or boolean value. A condition is false if the expression evaluates to 0 (zero). Otherwise, it is true. HDevelop provides the following operators to control the program flow:

if ... endif This control flow structure executes a block of code conditionally. The operator if takes a condition as its input parameter. If the condition is true, the body is executed. Otherwise the execution is continued at the operator call that follows the operator endif.

To enter both if and endif at once, select the operator if in the operator window and make sure the check box next to the operator is ticked.

```
if (<condition>)
  ...
endif
```

abs	acos	and	asin
assign	assign_at	atan	atan2
band	bnot	bor	break
bxor	case	catch	ceil
chr	chrt	comment	continue
cos	cosh	cumul	default
deg	deviation	else	elseif
endfor	endif	endswitch	endtry
endwhile	environment	exit	exp
export_def	fabs	false	find
floor	fmod	for	gen_tuple_const
global	H_MSG_FAIL	H_MSG_FALSE	H_MSG_TRUE
H_MSG_VOID	H_TYPE_ANY	H_TYPE_INT	H_TYPE_MIXED
H_TYPE_REAL	H_TYPE_STRING	if	ifelse
insert	int	inverse	is_int
is_int_elem	is_mixed	is_number	is_real
is_real_elem	is_string	is_string_elem	ldexp
log	log10	lsh	max
max2	mean	median	min
min2	not	or	ord
ords	pow	rad	rand
real	regexp_match	regexp_replace	regexp_select
regexp_test	remove	repeat	replace
return	round	rsh	select_mask
select_rank	sgn	sin	sinh
sort	sort_index	split	sqrt
stop	strchr	strlen	strrchr
strrstr	strstr	subset	sum
switch	tan	tanh	throw
true	try	type	type_elem
uniq	until	while	xor

Table 8.23: Reserved words.

`ifelse` (`if` ... `else` ... `endif`) Another simple control flow structure is the condition with alternative. If the condition is true, the block between `if` and `else` is executed. If the condition is false, the part between `else` and `endif` is executed.

Language

To enter all three operators at once, select the operator `ifelse` in the operator window and make sure the check box next to the operator is ticked.

```
if (<condition>)
  ...
else
  ...
endif
```

`elseif` This operator is similar to the `else`-part of the previous control flow structure. However, it allows to test for an additional condition. The block between `elseif` and `endif` is executed if <condition1> is false and <condition2> is true. `elseif` may be followed by an arbitrary number of additional `elseif` instructions. The last `elseif` may be followed by a single `else` instruction.

```
if (<condition1>)
  ...
elseif (<condition2>)
  ...
endif
```

This is syntactically equivalent and thus a shortcut for the following code block:

```
if (<condition1>)
  ...
else
  if (<condition2>)
    ...
  endif
endif
```

`while` **...** `endwhile` This is a looping control flow structure. As long as the condition is `true`, the body of the loop is executed. In order to enter the loop, the condition has to be true in the first place. The loop can be restarted and terminated immediately with the operator `continue` and `break`, respectively (see below).

To enter both `while` and `endwhile` at once, select the operator `while` in the operator window and make sure the check box next to the operator is ticked.

```
while (<condition>)
 ...
endwhile
```

`repeat` **...** `until` This loop is similar to the `while` loop with the exception that the condition is tested at the end of the loop. Thus, the body of a `repeat` ... `until` loop is executed at least once. Also in contrast to the `while` loop, the loop is repeated if the condition is false, i.e., *until* it is finally true.

To enter both `repeat` and `until` at once, select the operator `until` in the operator window and make sure the check box next to the operator is ticked.

```
repeat
  ...
until (<condition>)
```

for ... endfor The for loop is controlled by a start and an end value and an increment value, step, that determines the number of loop steps. These values may also be expressions, which are evaluated immediately before the loop is entered. The expressions may be of type `integer` or of type `real`. If all input values are of type `integer`, the loop variable will also be of type `integer`. In all other cases the loop variable will be of type `real`.

Please note that the for loop is displayed differently in the program window than entered in the operator window. What you enter in the operator window as `for(`*start*`,`*end*`,`*step*`,`*index*`)` is displayed in the program window as:

```
for <index> := <start> to <end> by <step>
  ...
endfor
```

To enter both for and endfor at once, select the operator for in the operator window and make sure the check box next to the operator is ticked.

The start value is assigned to the index variable. The loop is executed as long as the following conditions are true: 1) The step value is positive, and the loop index is smaller than or equal to the end value. 2) The step value is negative, and the loop index is greater than or equal to the end value. After a loop cycle, the loop index is incremented by the step value and the conditions are evaluated again.

Thus, after executing the following lines,

```
for i := 1 to 5 by 1
  j := i
endfor
```

`i` is set to 6 and `j` is set to 5, while in

```
for i := 5 to 1 by -1
  j := i
endfor
```

`i` is set to 0, and `j` is set to 1.

The loop can be restarted and terminated immediately with the operator continue and break, respectively. (see below).

Please note, that in older versions of HDevelop (prior to HALCON 11), the expressions for start and termination value were evaluated only once when *entering the loop*. A modification of a variable that appeared within these expressions had no influence on the termination of the loop. The same applied to the modifications of the loop index. It also had no influence on the termination. The loop value was assigned to the correct value each time the for operator was executed. See the reference manual of the for operator for more information.

Language

If the for loop is left too early (e.g., if you press Stop and set the PC) and the loop is entered again, the expressions will be evaluated, as if the loop were entered for the first time.

In the following example the sine from 0 up to 6π is computed and printed into the graphical window (file name: sine.hdev):

```
old_x := 0
old_y := 0
dev_set_color ('red')
dev_set_part(0, 0, 511, 511)
for x := 1 to 511 by 1
    y := sin(x / 511.0 * 2 * 3.1416 * 3) * 255
    disp_line (WindowID, -old_y+256, old_x, -y+256, x)
    old_x := x
    old_y := y
endfor
```

In this example the assumption is made that the window is of size 512×512. The drawing is always done from the most recently evaluated point to the current point.

continue The operator continue forces the next loop cycle of a for, while, or repeat loop. The loop condition is tested, and the loop is executed depending on the result of the test.

In the following example, a selection of RGB color images is processed. Images with channel numbers other than three are skipped through the use of the operator continue. An alternative is to invert the condition and put the processing instructions between if and endif. But the form with continue tends to be much more readable when very complex processing with lots of lines of code is involved.

```
i := |Images|
while (i)
  Image := Images[i]
  count_channels (Image, Channels)
  if (Channels != 3)
    continue
  endif
  * extensive processing of color image follows
endwhile
```

break The opeator break enables you to exit for, while, and repeat loops. The program is then continued at the next line after the end of the loop.

A typical use of the operator break is to terminate a for loop as soon as a certain condition becomes true, e.g., as in the following example:

```
Number := |Regions|
AllRegionsValid := 1
* check whether all regions have an area <= 30
for i := 1 to Number by 1
```

```
    ObjectSelected := Regions[i]
    area_center (ObjectSelected, Area, Row, Column)
    if (Area > 30)
      AllRegionsValid := 0
      break ()
    endif
  endfor
```

In the following example, the operator `break` is used to terminate an (infinite) `while` loop as soon as one clicks into the graphics window:

```
  while (1)
    grab_image (Image, FGHandle)
    dev_error_var (Error, 1)
    dev_set_check ('~give_error')
    get_mposition (WindowHandle, R, C, Button)
    dev_error_var (Error, 0)
    dev_set_check ('give_error')
    if ((Error = H_MSG_TRUE) and (Button != 0))
      break ()
    endif
  endwhile
```

`switch` ... `case` ... `endswitch` The `switch` block allows to control the program flow via a multiway branch. The branch targets are specfied with `case` statements followed by an integer constant. Depending on an integer control value the program execution jumps to the matching case statements and continues to the next `break` statement or the closing `endswitch` statement. An optional `default` statement can be defined as the last jump label within a `switch` block. The program execution jumps to the `default` label if no preceding `case` statement matches the control expression.

```
  ...
  switch (Grade)
    case 1:
      Result := 'excellent'
      break
    case 2:
      Result := 'good'
      break
    case 3:
      Result := 'acceptable'
      break
    case 4:
    case 5:
      Result := 'unacceptable'
      break
    default:
      Result := 'undefined'
```

```
endswitch
...
```

`stop` The operator `stop` stops the program after the operator is executed. The program can be continued by pressing the `Step Over` or `Run` button.

`exit` The `exit` operator *terminates* the HDevelop session.

`return` The operator `return` returns from the current procedure call to the calling procedure. If `return` is called in the main procedure, the PC jumps to the end of the program, i.e., the program is finished.

`try` ... `catch` ... `endtry` This control flow structure enables dynamic exception handling in HDevelop. The program block between the operators `try` and `catch` is watched for exceptions, i.e., runtime errors. If an exception occurs, diagnostic data about what caused the exception is stored in an exception tuple. The exception tuple is passed to the `catch` operator, and program execution continues from there. The program block between the operators `catch` and `endtry` is intended to analyze the exception data and react to it accordingly. If no exception occurs, this program block is never executed.

See section "Error Handling" on page 384, and the reference manual, e.g., the operator `try` for detailed information.

`throw` The operator `throw` allows to generate user-defined exceptions.

8.9 Error Handling

This section describes how errors are handled in HDevelop programs. When an error occurs, the default behavior of HDevelop is to stop the program execution and display an error message box. While this is certainly beneficial at the time the program is developed, it is usually not desired when the program is actually deployed. A finished program should react to errors itself. This is of particular importance if the program interacts with the user.

There are basically two approaches to error handling in HDevelop:

- tracking the return value (error code) of operator calls
- using exception handling

A major difference between these approaches is the realm of application: The first method handles errors inside the procedure in which they occur. The latter method allows errors to work their way up in the call stack until they are finally dealt with.

8.9.1 Tracking the Return Value of Operator Calls

The operator `dev_set_check` specifies if error message boxes are displayed at all.

To turn message boxes off, use

```
dev_set_check('~give_error')
```

HDevelop will then ignore any errors in the program. Consequently, the programmer has to take care of the error handling. Every operator call provides a return value (or error code) which signals success or failure of its execution. This error code can be accessed through a designated error variable:

```
dev_error_var(ErrorCode, 1)
```

This operator call instantiates the variable ErrorCode. It stores the error code of the last executed operator. Using this error code, the program can depend its further flow on the success of an operation.

```
...
if (ErrorCode != H_MSG_TRUE)
  * react to error
endif
* continue with program
...
```

The error message related to a given error code can be obtained with the operator get_error_text. This is useful when reporting errors back to the user of the program.

If the error is to be handled in a calling procedure, an appropriate output control variable has to be added to the interface of each participating procedure, or the error variable has to be defined as a global variable (see section 8.3.1).

```
global tuple ErrorCode
dev_error_var(ErrorCode, 1)
...
```

8.9.2 Exception Handling

HDevelop supports dynamic exception handling, which is comparable to the exception handling in C++ and C#.

A block of program lines is watched for run-time errors. If an error occurs, an exception is raised and an associated exception handler is called. An exception handler is just another block of program lines, which is invisible to the program flow unless an error occurs. The exception handler may directly act on the error or it may pass the associated information (i.e., the exception) on to a parent exception handler. This is also known as *rethrowing an exception*.

In contrast to the tracking method described in the previous section, the exception handling requires HDevelop to be set up to stop on errors. This is the default behavior. It can also be turned on explicitly:

```
dev_set_check('give_error')
```

Furthermore, HDevelop can be configured to let the user choose whether or not an exception is thrown, or to throw exceptions automatically. This behavior is set in the preferences tab `General Options -> Experienced User`.

An HDevelop exception is a tuple containing data related to a specific error. It always contains the error code as the first item. The operator `dev_get_exception_data` provides access to the elements of an exception tuple.

HDevelop exception handling is applied in the following way:

```
...
try
  * start block of watched program lines
  ...
catch(Exception)
  * get error code
  ErrorCode := Exception[0]
  * react to error
endtry
* program continues normally
...
```

8.10 Parallel Execution

The HDevelop language supports the parallel execution of procedure and operator calls as subthreads of the main thread. Once started, subthreads are identified by a thread ID which is an integer process number depending on the operating system. The execution of subthreads is independent of the thread they have been started from. Therefore, the exact point in time when a specific threat finishes cannot be predicted. If you want to access data returned from a group of threads, it is required to explicitly wait for the corresponding threads to finish.

HDevelop supports up to 20 threads by default but this number can be configured in the preferences if required (see section "`General Options -> General Options`" on page 94). The main reason for limiting the number of simultaneous threads at all, is to prevent the user from inadvertendly generating a huge number of threads due to a programming error. In that case, the system load as well as the memory consumption may grow so high that HDevelop can become unresponsive.

8.10.1 Starting a Subthread

To start a new thread, prefix the corresponding operator or procedure call with the `par_start` qualifier:

```
par_start <ThreadID> : gather_data()
...
```

This call starts the hypothetical procedure `gather_data()` as a new subthread in the background and continues to execute the subsequent program lines. The thread ID is returned in the variable `ThreadID`

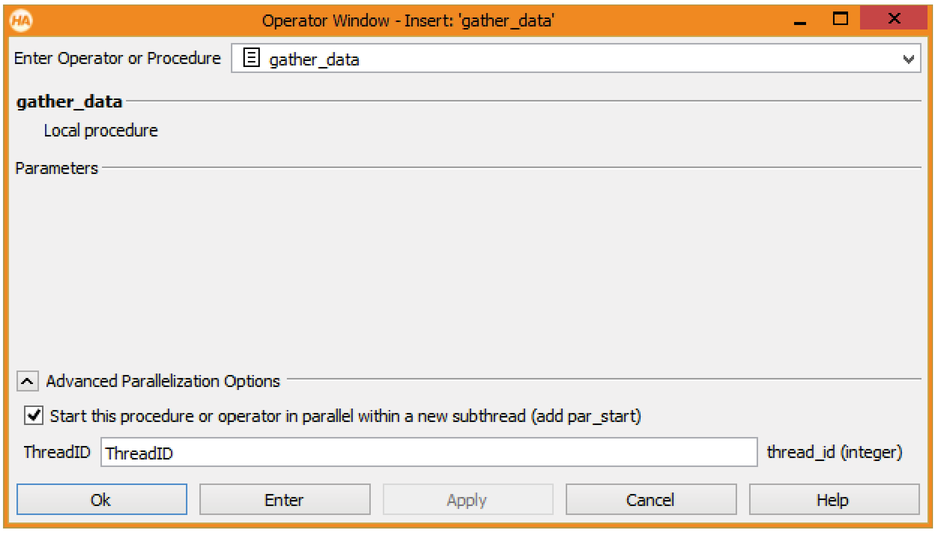

Figure 8.3: Operator window with parallelization options.

which must be specified in angle brackets. Note that `par_start` is not an actual operator but merely a qualifier that modifies the calling behavior. Therefore, it is not possible to select `par_start` in the operator window.

You can also start procedure or operator calls as a subthread from the operator window (see figure 8.3). To do this, open the section `Advanced Parallelization Options` at the bottom of the operator window, tick the check box and enter the name of the variable that will hold the thread ID. If you double-click on a program containing the `par_start` qualifier, the parallelization options will also be displayed in the operator window. For certain program lines (e.g., comments, declarations, loops, or assignments) `par_start` is not supported and the corresponding options will also not be available in the operator window. For a general description of the operator window see section “Operator Window” on page 170.

It is supported to start multiple threads in a loop. In that case the thread IDs need to be collected so that all threads can be referenced later:

```
ThreadIDs := []
for Index := 1 to 5 by 1
  par_start <ThreadID> : gather_data()
  ThreadIDs := [ThreadIDs, ThreadID]
endfor
```

It is often more convenient to collect the thread IDs in a vector variable (page 375):

```
for Index := 1 to 5 by 1
  par_start <ThreadIDs.at(Index - 1)> : gather_data()
endfor
```

Special care must be taken when the subthread returns data in an output variable. In particular, output variables must not be accessed in other threads while the subthread is still running. Otherwise the data is not guaranteed to be valid.

Likewise, its must be ensured that multiple threads do not interfere with their results. Suppose the procedure gather_data is started as multiple threads like above but returns data in an output control variable:

```
for Index := 1 to 5 by 1
  par_start <ThreadIDs.at(Index - 1)> : gather_data(Result)  // BEWARE!!!
endfor
```

In the above example, all the threads would return their result in the same variable which is certainly not what was intended. The final value of Result would be the (unpredictable) return value of the thread that finishes last, and all other results would be lost.

An easy solution to this problem is to collect the returned data in a vector variable as shown previously with the thread IDs:

```
for Index := 1 to 5 by 1
  par_start <ThreadIDs.at(Index - 1)> : gather_data(Result.at(Index - 1))
endfor
```

Here, each invocation of gather_data returns its result in a unique slot of the vector variable Result.

8.10.2 Waiting for Subthreads to Finish

Use the operator par_join to wait for the completion of a single thread or a group of threads.

As an example why this is necessary suppose we want to call a procedure that performs some magic calculation in the background and returns a count number as a result. In the subsequent program lines we want to use that number for further calculations.

```
par_start <ThreadID> : count_objects(num)
...
for i := 1 to num by 1     // BEWARE: num might be uninitialized
  ...
endfor
```

Simply relying on the subthread to be fast enough is most likely going to fail. Therefore, an explicit call to par_join is required beforehand.

```
par_start <ThreadID> : count_objects(num)
...
par_join(ThreadID)
for i := 1 to num by 1
  ...
endfor
```

Note that in HDevelop it is not stricly required to use par_join because the main thread will always outlive the subthreads. However, ommitting it might lead to trouble if the program is going to be executed in HDevEngine or exported to a programming language. Similarly, access to global variables might need some additional synchronization if the program is going to be exported.

Given the example from the previous section, waiting for the finishing of all the threads that were started in a loop is achieved using the following lines.

```
convert_vector_to_tuple(ThreadIDs, Threads)
par_join(Threads)
```

Please note that the thread IDs had been collected in a vector variable. Thus, the conversion to a tuple is necessary for par_join to work properly.

The par_join operator blocks the further execution of the procedure it has been called from until all specified threads have finished. In the subsequent program lines, results from the corresponding threads can then be accessed reliably.

8.10.3 Execution of Threads in HDevelop

In general, threads in HDevelop are only executed in parallel when the program runs continuously after pressing F5. In all other execution modes only the selected thread is started and all other threads remain stopped unless an explicit user interaction advances their execution. Active break points, stop instructions, runtime errors or uncaught exceptions also cause all threads to stop so their current state can be evaluated. This convention enables a clearly defined debugging process because it eliminates uncontrollable side-effects from other threads. Any editing action in the program window will also cause a concurrently running program to stop.

Threads cannot be “killed” externally. They can be stopped between operator calls or by aborting interruptible operators. If any thread executes a long-running operator that cannot be interrupted at the time HDevelop tries to stop the program execution, a corresponding message will be displayed in the status line, and the corresponding thread will eventually stop after the operator has finished.

Selected Thread

Exactly one of the threads is the so-called *selected* thread; by default this corresponds to the main thread of the program. The position of the PC, the status of the call stack, and the state of the variables in the variable window are linked to the selected thread. The selected thread can change automatically to a thread that stops, e.g., by a break point, stop instruction, an uncaught exception, or a draw operator.

How to select a specific thread is described in section “Inspecting Threads” on page 390. All run modes other then continuous execution apply only to the selected thread. Program lines unrelated to the selected thread will be grayed out in the program window.

Threads and Just-in-time Compiled Procedures

Procedures can be executed as compiled bytecode instead of being interpreted by the HDevelop interpreter. This is described in section “Just-In-Time Compilation” on page 50. There is one notable difference when debugging threaded HDevelop programs with compiled procedures. If the program is running continuously and is then being stopped (either by user action or a break point/`stop` instruction), the current state of the compiled procedures (variables, PC) cannot be inspected. You can still step into the procedure calls but this will cause the corresponding thread to be re-executed which may cause unexpected side-effects. Note that this is not an issue when single-stepping into the thread calls in the first place because in that case procedures are always executed by the HDevelop interpreter.

Thread Lifetime

A thread exists as long as it is still being referenced by a variable even if its execution has finished. This is necessary to reference the thread in a `par_join` instruction, or to set back the PC of the corresponding thread for debugging. However, the lifetime of a thread ends if the PC is manually set back to a program line before its invocation. Apart from that the lifetime of all subthreads ends when the program is reset using F2. During its lifetime a thread is listed in the `Thread View / Call Stack` window from where it can be selected and managed.

Error Handling

Each thread can specify its own error handling, e.g., using `dev_set_check`('~give_error'). New subthreads inherit the error handling mode from their parent thread. Exception handling using `try` .. `catch` works only within a thread, i.e., in the main thread it is not possible to catch an exception that is thrown in a subthread.

8.10.4 Inspecting Threads

The current execution status of the program and its threads is displayed in the combined thread view/call stack window. Select `Execute` ▷ `Thread View / Call Stack` or click in the tool bar (see also `Thread View / Call Stack` (page 104)). The upper half of the window lists all existing threads while the lower half displays the call stack of the selected thread. To illustrate the interaction with this window consider the following (silly) example.

```
for Index:= 1 to 5 by 1
  par_start <ThreadIDs.at(Index - 1)> : wait_seconds(Index)
endfor
wait_seconds(2)
stop()
```

After pressing F5, the program will start five subthreads, and eventually reach the `stop` instruction, leaving some subthreads still running while others will already have finished. The corresponding thread view is displayed in figure 8.4. Note that the unfinished threads are in a stopped state because of another thread (in this case caused by the `stop` instruction in the main thread).

The thread view lists the properties of all threads in a table. The status icon in the first column of each thread visualizes the current execution state. The currently selected thread (1) is marked by the yellow arrow in the status icon and is also highlighted in bold text. The other five threads are the subthreads started from the main thread. To select another thread, double-click it in the thread view. This will also

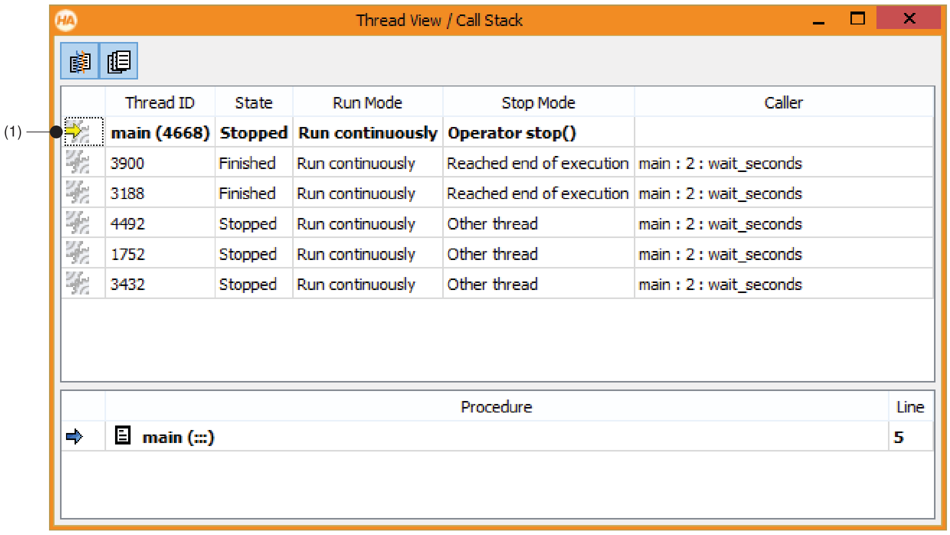

Figure 8.4: Inspecting threads.

update the PC, the call stack, and the variables according to the selected thread. The active procedure of the selected thread will be displayed in the program window.

The meaning of the columns of the thread view is as follows:

Column	**Description**
Thread ID	The unique number assigned to the thread when it is started.
State	The current execution state of the thread.
Run Mode	The mode that the thread was started in the last time.
Stop Mode	The reason for stopping the thread.
Caller	The position (procedure and line number) from where the thread was started.
References (hidden by default)	The number of references to this thread.

Single-stepping over a program line containing a `par_start` will initialize the corresponding thread without actually starting it. To debug a specific thread, press F7 when the PC is on the corresponding `par_start` line. This will automatically make the new subthread the selected thread. If the PC is already past the invocation line, first select the thread in the thread view window. This will automatically display the correct procedure in the program window, with the PC on the first line if the thread was started by a procedure call. For threads started by an operator call like in the example above, the PC will be located on the corresponding call, and the rest of the program will be grayed out (see figure 8.5). The notification line (1) in the program window shows the thread ID of the selected subthread, and allows fast access to the thread view window (2). Clicking (3) switches back to the main thread.

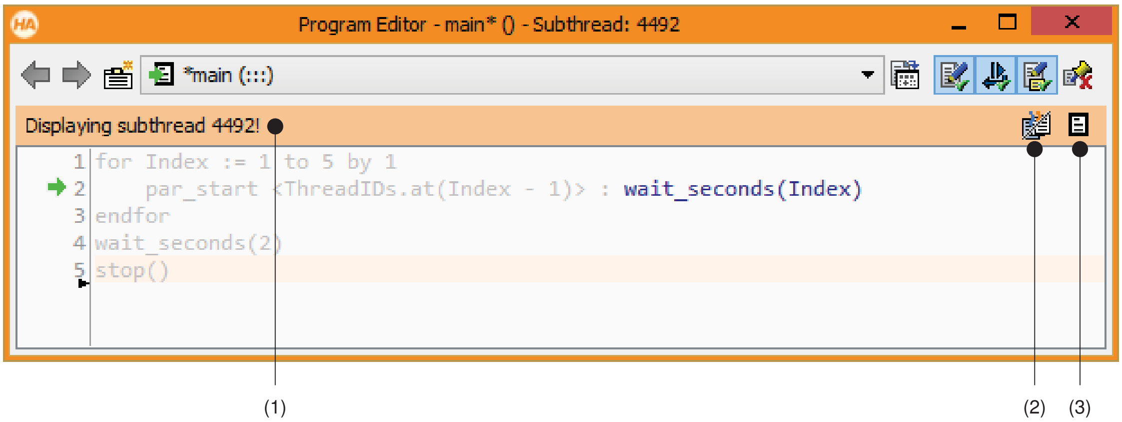

Figure 8.5: Selected subthread in the program window.

8.10.5 Suspending and Resuming Threads

Threads can explicitly be suspended and resumed in the thread view window. Suspending a thread only works between operator calls, i.e., if the thread is currently running, the current operator will still be executed before the thread is frozen.

To suspend a thread, right-click on the thread entry and select `Suspend Thread`. Suspended threats are "frozen" in their current state and will defer subsequent run commands until the threads are resumed again, i.e., run commands will change the run status of the suspended threads but the actual execution will be prevented.

To resume a suspended thread again, right-click on the thread entry and select `Resume Thread`.

8.11 Summary of HDevelop Tuple Operations

Functionality	HDevelop Operation	HALCON operator
concatenation	`[t1,t2]`	`tuple_concat`
number of elements	`|t|`	`tuple_length`
select tuple element	`t[i]`	`tuple_select`
select tuple slice	`t[i1:i2]`	`tuple_select_range`
select elements	`subset(t,i)`	`tuple_select`
remove tuple elements	`remove(t,i)`	`tuple_remove`
lookup tuple values	`find(t1,t2)`	`tuple_find`
unify tuple elements	`uniq(t)`	`tuple_uniq`

tuple creation	`gen_tuple_const(i1,i2)`	`tuple_gen_const`
tuple creation	`[i1:i2:i3]`	`tuple_gen_sequence`
tuple creation	`[i1:i2]`	`tuple_gen_sequence`
division	`a1 / a2`	`tuple_div`
multiplication	`a1 * a2`	`tuple_mult`
modulo	`a1 % a2`	`tuple_mod`
addition	`a1 + a2`	`tuple_add`
subtraction	`a1 - a2`	`tuple_sub`
negation	`-a`	`tuple_neg`
left shift	`lsh(i1,i2)`	`tuple_lsh`
right shift	`rsh(i1,i2)`	`tuple_rsh`
bitwise and	`i1 band i2`	`tuple_band`
bitwise xor	`i1 bxor i2`	`tuple_bxor`
bitwise or	`i1 bor i2`	`tuple_bor`
bitwise complement	`bnot i`	`tuple_bnot`
string conversion	`v$s`	`tuple_string`
string concatenation	`v1 + v2`	`tuple_concat`
search character	`strchr(s1,s2)`	`tuple_strchr`
search character (reverse)	`strrchr(s1,s2)`	`tuple_strrchr`
search string	`strstr(s1,s2)`	`tuple_strstr`
search string (reverse)	`strrstr(s1,s2)`	`tuple_strrstr`
length of string	`strlen(s)`	`tuple_strlen`
select character	`s{i}`	`tuple_str_bit_select`
select substring	`s{i1:i2}`	`tuple_str_bit_select`
split string	`split(s1,s2)`	`tuple_split`
regular expression match	`regexp_match(s1,s2)`	`tuple_regexp_match`
regular expression replace	`regexp_replace(s1,s2,s3)`	`tuple_regexp_replace`
regular expression select	`regexp_select(s1,s2)`	`tuple_regexp_select`
regular expression test	`regexp_test(s1,s2)`	`tuple_regexp_test`
less than	`t1 < t2`	`tuple_less`
greater than	`t1 > t2`	`tuple_greater`

less or equal	`t1 <= t2`	`tuple_less_equal`
greater or equal	`t1 >= t2`	`tuple_greater_equal`
equal	`t1 == t2`	`tuple_equal`
not equal	`t1 != t2`	`tuple_not_equal`
less than (elementwise)	`t1 [<] t2`	`tuple_less_elem`
greater than (elementwise)	`t1 [>] t2`	`tuple_greater_elem`
less or equal (elementwise)	`t1 [<=] t2`	`tuple_less_equal_elem`
greater or equal (elementwise)	`t1 [>=] t2`	`tuple_greater_equal_elem`
equal (elementwise)	`t1 [==] t2`	`tuple_equal_elem`
not equal (elementwise)	`t1 [!=] t2`	`tuple_not_equal_elem`
logical and	`l1 and l2`	`tuple_and`
logical xor	`l1 xor l2`	`tuple_xor`
logical or	`l1 or l2`	`tuple_or`
negation	`not l`	`tuple_not`
sine	`sin(a)`	`tuple_sin`
cosine	`cos(a)`	`tuple_cos`
tangent	`tan(a)`	`tuple_tan`
arc sine	`asin(a)`	`tuple_asin`
arc cosine	`acos(a)`	`tuple_acos`
arc tangent	`atan(a)`	`tuple_atan`
arc tangent2	`atan2(a1,a2)`	`tuple_atan2`
hyperbolic sine	`sinh(a)`	`tuple_sinh`
hyperbolic cosine	`cosh(a)`	`tuple_cosh`
hyperbolic tangent	`tanh(a)`	`tuple_tanh`
exponential function	`exp(a)`	`tuple_exp`
natural logarithm	`log(a)`	`tuple_log`
decadic logarithm	`log10(a)`	`tuple_log10`
power function	`pow(a1,a2)`	`tuple_pow`
ldexp function	`ldexp(a1,a2)`	`tuple_ldexp`
minimum	`min(t)`	`tuple_min`
elementwise minimum	`min2(t1,t2)`	`tuple_min2`

maximum	`max(t)`	`tuple_max`
elementwise maximum	`max2(t1,t2)`	`tuple_max2`
sum function	`sum(t)`	`tuple_sum`
mean value	`mean(a)`	`tuple_mean`
standard deviation	`deviation(a)`	`tuple_deviation`
cumulative sum	`cumul(a)`	`tuple_cumul`
median	`median(a)`	`tuple_median`
element rank	`select_rank(a,i)`	`tuple_select_rank`
square root	`sqrt(a)`	`tuple_sqrt`
radians to degrees	`deg(a)`	`tuple_deg`
degrees to radians	`rad(a)`	`tuple_rad`
integer to real	`real(a)`	`tuple_real`
real to integer	`int(a)`	`tuple_int`
real to integer	`round(a)`	`tuple_round`
absolute value	`abs(a)`	`tuple_abs`
floating absolute value	`fabs(a)`	`tuple_fabs`
ceiling function	`ceil(a)`	`tuple_ceil`
floor function	`floor(a)`	`tuple_floor`
fractional part	`fmod(a1,a2)`	`tuple_fmod`
elementwise sign	`sgn(a)`	`tuple_sgn`
sort elements	`sort(t)`	`tuple_sort`
sort elements (returns index)	`sort_index(t)`	`tuple_sort_index`
reverse element order	`inverse(t)`	`tuple_inverse`
test for numeric value	`is_number(v)`	`tuple_is_number`
string to number	`number(v)`	`tuple_number`
environment variable	`environment(s)`	`tuple_environment`
character to ASCII number	`ord(a)`	`tuple_ord`
ASCII number to character	`chr(a)`	`tuple_chr`
tuple of strings to ASCII numbers	`ords(s)`	`tuple_ords`
tuple of integers to string	`chrt(i)`	`tuple_chrt`
random number	`rand(a)`	`tuple_rand`

test for integer values	`is_int(t)`	tuple_is_int
test for mixed values	`is_mixed(t)`	tuple_is_mixed
test for numerical values	`is_number(t)`	tuple_is_number
test for real values	`is_real(t)`	tuple_is_real
test for string values	`is_string(t)`	tuple_is_string
get type value	`type(t)`	tuple_type
test for integer values (elementwise)	`is_int_elem(t)`	tuple_is_int_elem
test for real values (elementwise)	`is_real_elem(t)`	tuple_is_real_elem
test for string values (elementwise)	`is_string_elem(t)`	tuple_is_string_elem
get type value (elementwise)	`type_elem(t)`	tuple_type_elem

8.12 HDevelop Error Codes

21000	HALCON operator error
21001	User defined exception ('throw')
21002	User defined error during execution
21003	User defined operator does not implement execution interface
21010	HALCON license error
21011	HALCON startup error
21012	HALCON operator error
21020	Format error: file is not a valid HDevelop program or procedure
21021	File is no HDevelop program or has the wrong version
21022	Protected procedure could not be decompressed
21023	Protected procedure could not be compressed and encrypted for saving
21030	The program was modified inconsistently outside HDevelop.
21031	The program was modified outside HDevelop: inconsistent procedure lines.
21032	The program was modified outside HDevelop: unmatched control statements
21033	Renaming of procedure failed
21034	Locked procedures are not supported for the selected action.
21035	Procedures with advanced language elements are not supported for the selected action.
21036	Procedures with vector variables are not supported for the selected action.
21040	Unable to open file
21041	Unable to read from file
21042	Unable to write to file
21043	Unable to rename file
21044	Unable to open file: invalid file name
21050	For this operator the parallel execution with par_start is not supported
21051	Thread creation failed
21052	Thread creation failed: exceeded maximum number of sub threads
21055	Old program version: Not supported for hdevelop_demo
21056	Wrong program check sum: HDevelop Demo cannot open the program or procedure if it was changed outside HDevelop. This is not allowed for HDevelop Demo

21057	Program was saved without a check sum: This is not supported by HDevelop Demo
21058	Inserting procedures is not supported in hdevelop_demo
21060	Iconic variable is not instantiated
21061	Control variable is not instantiated (no value)
21062	Wrong number of control values
21063	Wrong value type of control parameter
21064	Wrong value of control parameter
21065	Control parameter does not contain a variable
21066	Control parameter must be a constant value
21067	Wrong number of control values in condition variable
21068	Wrong type: Condition variable must be an integer or boolean
21070	Variable names must not be empty
21071	Variable names must not start with a number
21072	Invalid variable name
21080	For loop variable must be a number
21081	Step parameter of for loop must be a number
21082	End parameter of for loop must be a number
21083	Variable names must not be a reserved expression
21084	Case label value has already appeared in switch block
21085	Default label has already appeared in switch block
21090	A global variable with the specified name but a different type is already defined
21091	Access to an unknown global variable
21092	Access to an invalid global variable
21100	Access to an erroneous expression
21101	Wrong index in expression list
21102	Empty expression
21103	Empty expression argument
21104	Syntax error in expression
21105	Too few function arguments in expression
21106	Too many function arguments in expression
21107	The expression has no return value

21108	The expression has the wrong type
21110	The expression has the wrong type: iconic expression expected
21112	The expression has the wrong type: control expression expected
21114	The expression has the wrong vector dimension
21116	Vector expression expected
21118	Single value expression expected instead of a vector
21120	Expression expected
21121	lvalue expression expected
21122	Variable expected
21123	Unary expression expected
21124	Expression list expected
21125	Function arguments in parentheses expected
21126	One function argument in parentheses expected
21127	Two function arguments in parentheses expected
21128	Three function arguments in parentheses expected
21129	Four function arguments in parentheses expected
21130	Five function arguments in parentheses expected
21131	Right parenthesis ')' expected
21132	Right curly brace '}' expected
21133	Right square bracket ']' expected
21134	Unmatched right parenthesis ')' found
21135	Unmatched right curly brace '}' found
21136	Unmatched right square bracket ']' found
21137	Second bar '|' expected
21138	Function name expected before parentheses
21139	Unterminated string detected
21140	Invalid character in an expression identifier detected
21141	Parameter expression expected
21142	Parameter expression is not executable
21143	Vector method after . expected
21144	Vector method 'at' after . expected

21145	Modifying vector methods are not allowed within parameters
21200	Syntax error in operator expression
21201	Identifier (operator or variable name) expected
21202	Syntax error in parameter list
21204	Parenthesis expected
21205	No parenthesis expected
21206	List of parameters in parenthesis expected
21207	Wrong number of parameters
21208	Unexpected characters at end of line
21209	Assign operator ':=' expected
21210	Expression after assign operator ':=' expected
21211	Expression in brackets '[ ]' for the assign_at index expected
21212	In for statement, after keyword 'by' expression for parameter 'Step' expected
21213	In for statement, after keyword 'to' expression for parameter 'End' expected
21214	In for statement, after assign operation (':=') expression for parameter 'Start' expected
21215	In for statement, after 'for .. := .. to ..' keyword 'by' expected
21216	In for statement, after 'for .. := ..' keyword 'to' expected
21217	In for statement, assign operation ':=' for initializing the index variable expected
21218	After 'for' keyword, assignment of 'Index' parameter expected
21219	In for statement, error after 'by' keyword in expression of parameter 'Step'
21220	In for statement, error after 'to' keyword in expression of parameter 'End' or the following 'by' keyword
21221	In for statement, error after assignment operation (':=') in expression of parameter 'Start' or the following 'to' keyword
21222	In for statement, invalid variable name in parameter 'Index' or error in the following assignment operation (':=')
21223	for statement not complete
21224	In for statement, space after 'for' expected
21225	In for statement, space after 'to' expected
21226	In for statement, space after 'by' expected
21227	Wrong type: The switch statement requires an integer value as parameter
21228	Wrong type: The case statement requires a constant integer value as parameter

21229	At the end of the case and the default statement a colon is expected
21230	Unknown operator or procedure
21231	Qualifier ’par_start’ before ’:’ or ’<ThreadID>’ expected
21232	’<ThreadID>’ variable in angle brackets after ’par_start’ expected
21233	ThreadID variable after ’par_start<’ expected
21234	Closing angle bracket (’>’) after ’par_start<ThreadID’ expected
21235	Colon (’:’) after ’par_start<ThreadID>’ expected
21236	Operator or procedure call after ’par_start :’ expected
21900	Internal value has the wrong type
21901	Internal value is not a vector
21902	Index into internal value is out of range
21903	Internal value is not instantiated
22000	Internal operation in expression failed
22001	Internal operation in a constant expression failed
22010	Parameters are tuples with different size
22011	Division by zero
22012	String exceeds maximum length
22100	Parameter is an empty tuple
22101	Parameter has more than one single value
22102	Parameter is not a single value
22103	Parameter has the wrong number of elements
22104	Parameter contains undefined value(s)
22105	Parameter contains wrong value(s)
22106	Parameter contains value(s) with the wrong type
22200	First parameter is an empty tuple
22201	First parameter has more than one single value
22202	First parameter is not a single value
22203	First parameter has the wrong number of elements
22204	First parameter contains undefined value(s)
22205	First parameter contains wrong value(s)
22206	First parameter contains value(s) with the wrong type

22300	Second parameter is an empty tuple
22301	Second parameter has more than one single value
22302	Second parameter is not a single value
22303	Second parameter has the wrong number of elements
22304	Second parameter contains undefined value(s)
22305	Second parameter contains wrong value(s)
22306	Second parameter contains value(s) with the wrong type
23100	The generic parameter value is unknown
23101	The generic parameter name is unknown
30000	User defined exception