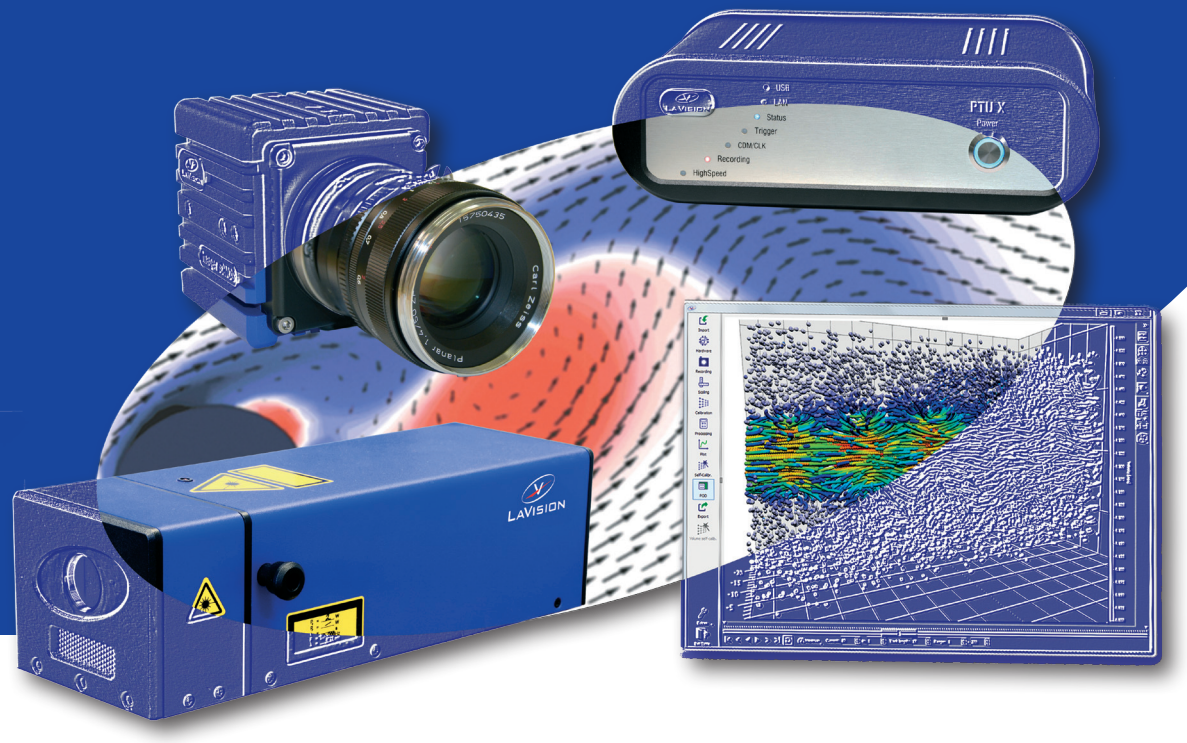
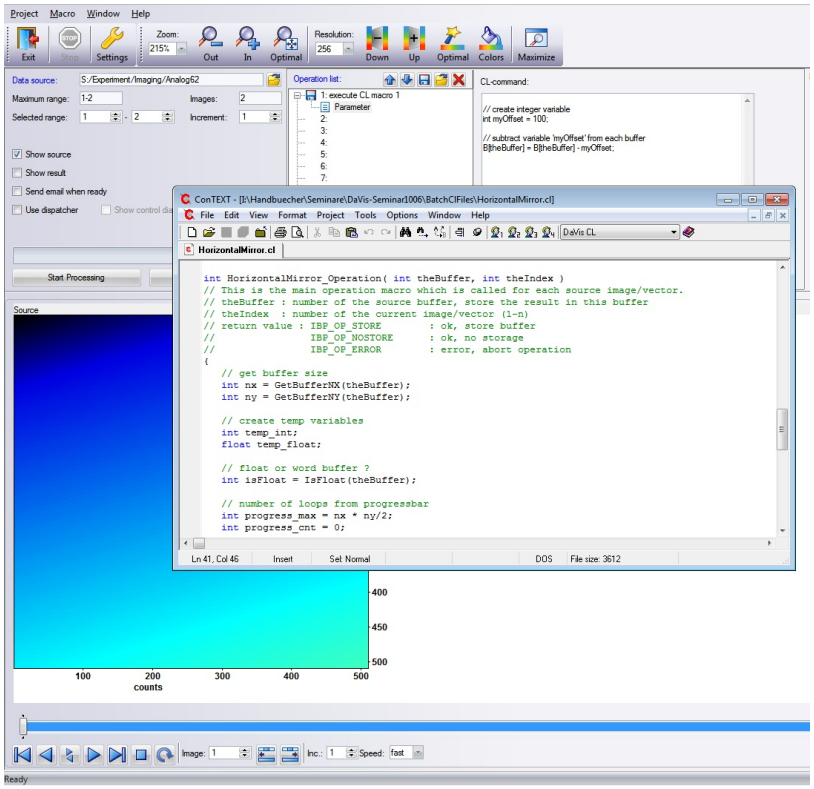


Product Manual

Command Language 10.2

Item Number(s): 1005xxx



LAVISION

FOCUS ON IMAGING

Product Manual for **DaVis 10.2**

LaVision GmbH, Anna-Vandenhoeck-Ring 19, D-37081 Göttingen
Produced by LaVision GmbH, Göttingen
Printed in Germany
Göttingen, December 13, 2022

Document name: 1003002_CommandLanguage_D10.2.pdf

Contents

1 Introduction

1.1 About this manual

The **control structures** and **data structures** of the **DaVis Command Language** (CL) together with a large set of system **subroutines** offer a great flexibility in adapting the software to individual evaluation applications.

Subroutines are system-defined functions for a variety of purposes, such as data I/O, and data processing. In contrast to **macros**, subroutines are only declared but not defined. The executable code is embedded in the main file `DAVIS.EXE` and is therefore not visible for the user.

This manual includes important subroutines and macros of **DaVis** 10 concerning processing and user dialogs. Public descriptions about programming hardware devices and recordings are not available. Please contact **LaVision** about customized recording functions.

Old processing macros from **DaVis** 8 should be executable in general in **DaVis** 10. There is no full compatibility between both versions. In most cases problems are based on changed function names, sometimes on a changed calling syntax. But all those errors result in macro errors with direct error messages. There should be no wrong result of your own macro functions because of the update.

1.2 Conventions

Bold: Links to chapters into this manual, names of menu items and names of button or text items in dialogs are marked in **bold** letters.

Italic: This style is used for links to other *manuals* by **LaVision**.

`Courier`: All CL-code is printed with the `Courier` font: subroutine and macro definitions, variable names, and macro code examples.

2 Macro Menu

2.1 Overview

The **Macro** menu in the **Extras** toolbar menu supports the execution of **DaVis** functions via statements written in the *Command Language* (CL).

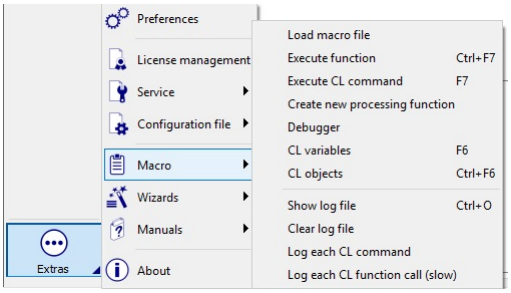


Figure 2.1: The Macro Menu

Newly written macros have to be loaded first with the function **Load Macro File**. The **DaVis** macro files are stored with extension `CL`.

Loaded macros can be executed easily with two different items:

Item **Execute CL command** asks the user to enter any CL statement via a command line (shortcut `F7`). The **Execute CL command** dialog remembers a history of the last 20 *one liners*. So you can easily try some commands and then select an older command.

Item **Execute Function** (shortcut `CTRL+F7`) opens a dialog with a list of all macro functions and system subroutines. The user can search for a macro name, select and executes that function.

Very useful while writing macros is the function **CL Variables** (shortcut `F6`). It displays groups of variables. Their present value is shown and can be changed.

The remaining four functions, the group in the center of the macro menu, operate a **log-file** and history of macro execution. See section on page **??** about changing the log mode during CL execution.

2.1.1 Example of a simple macro creation

This simple example explains how to write a new macro and start its execution.

At first the programmer has to select menu item **Edit Macro File** and select an existing CL file e.g. in the **DaVis** user settings directory: Change to subdirectory User/<name>/CL_Autostart. All CL from this autostart folder are loaded automatically at the startup of **DaVis**. In order to create a new file please click with the right mouse button into the white area of the directory overview, then create a new text file end rename the extension from TXT to CL. Press button **OK** to open the selected file with the default text editor.

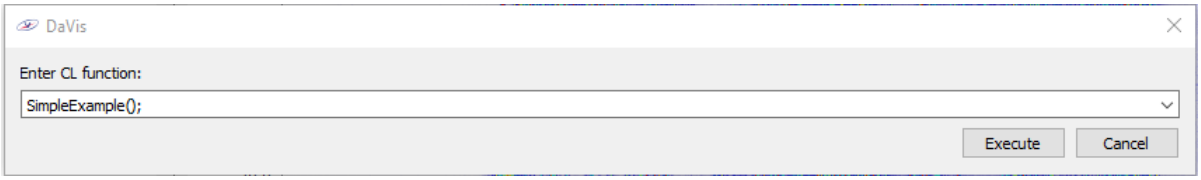
If available, an editor with syntax highlighting should be used, e.g. a C/C++ development software.

Now the programmer has to define a sequence of functions in the Command Language syntax. **For example** create a image buffer of size 256x256 pixel and display it on the screen in a developer view:

```
void SimpleExample()
{
    int hBuffer = 1;
    int size = 256;
    CreateWordBuffer( hBuffer, size, size );
    for (int i=0; i<size; i++)
    {
        Pix[hBuffer,i,i] = i;
    }
    Show(hBuffer);
}
```

After storing the new macro file (select **File** + **Store** in the editor), go back to **DaVis** and select **Load Macro File** in the macro menu. Choose the new macro file and click on the **OK** button. The new macro will be loaded, hopefully without syntax errors.

Then the macro can be executed via the function **Execute Line** or **Execute Function**:



The execution can be interrupted by clicking on the **STOP** button, which is visible in some dialogs in the middle of the tool bar during the execution, or by pressing the ESCAPE key.

2.1.2 The Macro Log File

A useful tool for debugging purposes and for error descriptions is the macro log file. During startup of **DaVis** a log file named `LOG_<data>_<time>.txt` is generated in the **DaVis** sub folder `User/<name>/log` with date and time of the **DaVis** startup, e.g. `LOG_140804_150343.txt`. **DaVis** holds the last ten log files and removes older ones automatically. It can be read with every text editor.

When enabled by a click on menu item **Log each CL command**, **DaVis** writes all directly executed subroutines and system macros into the file. This includes all commands executed from the menu bar, tool bar and from buttons in dialogs.

For some debugging purposes it may be useful to enable another mode, which stores ALL executed commands, including commands which are called by other commands. In mode **Log each CL function call** the macro execution takes a lot of time and creates a large log file. In mode **Log each CL function call** all function calls are printed with their parameters. For line or array parameters the first ten values and the line size are given. So this mode should be enabled only on special purposes by request of a **LaVision** software developer, and only for a short time! Make sure that this logging mode is switched off during normal **DaVis** execution.

When sending an error report to **LaVision**, a log file should be created using the the extended mode. Add a textual error description and, in some cases, a screen shot of the software.

With menu item **Clear Log File** the log file is deleted, and **Show Log File** (shortcut `CTRL+O`) starts the default text editor to display the file.

Example

Example of some **Log-file** entries. This logs are stored with option **Log each CL command**.

The lines include statements written in CL:

- create a image buffer of type word by macro `CreateWordBuffer`
- display buffer 1 on screen with subroutine `Show`

```
// DaVis 10.0 Build (10.41502-64) CL log file date:  10.08.17 time:
13:21:08
CreateWordBuffer(1,256,256);
Show(1);
```

Change log mode

The CL debugger dialog has been removed with **DaVis** 10.1.0.

The log mode of CL execution can be changed by subroutine `Log_Mode`. Call `Log_Mode(logmode)` to change the mode and use `logmode=0` to switch off logging, `logmode=1` to enable soft logging or `logmode=2` enable full logging.

2.2 Autostart folder

If you want DaVis to load your CL files automatically you can use the autostart folder `User/<name>/CL_Autostart`. All CL files in this folder will be loaded automatically at every startup of **DaVis**.

If you want DaVis to load CL files automatically for all users you can use the user-independent autostart folder `User/All users/CL_Autostart`. All CL files in this folder will be loaded automatically at every startup of DaVis.

2.3 Macro Execution

There are three ways to execute a subroutine:

With the dialog **Execute Function** of the **Macro** menu (shortcut CTRL+F7) the user can select one of the system subroutines (or a loaded macro program) and then execute it. We **recommend this way** especially in the beginning, since the user will automatically be guided to enter all necessary parameters.

With the dialog **Execute Line** of the **Macro** menu (shortcut F7) it is possible to enter a subroutine as one statement via a command line. All parameters have to be entered correctly.

It is also possible to include subroutines in a **macro program**. This way sequences of **DaVis** functions that exactly fit to the user's needs are executed with a single command.

2.3.1 Load Macro File

This function loads a macro file of the user's choice. The file can include various macro programs. Each of them can then be executed via the function **Execute Line** or **Execute Function**. A special function can be executed automatically after loading (see subroutine LoadMacroFile on page 71).

A macro file can also include **global static variables**, which are used for storing and retrieving parameters. With subroutine StoreMacroFile(""), all those variables are stored into CLS files with a format like SET files. This is also done automatically during shutdown of **DaVis**.

2.3.2 Execute Function

1. Open the list of all subroutines and loaded macros with the function **Execute Function** in the **Macro** menu or press shortcut CTRL+F7.
2. Enter a **search for** string in the text item on top of the dialog. The list is immediately rebuild after each typed character.

   A list of different substrings can be given when split by the space character. All substrings must be found into the function name in a free order.
3. Select **public only** to restrict the list to all available function. Access to public macros is restricted to the CL file which includes the macro.

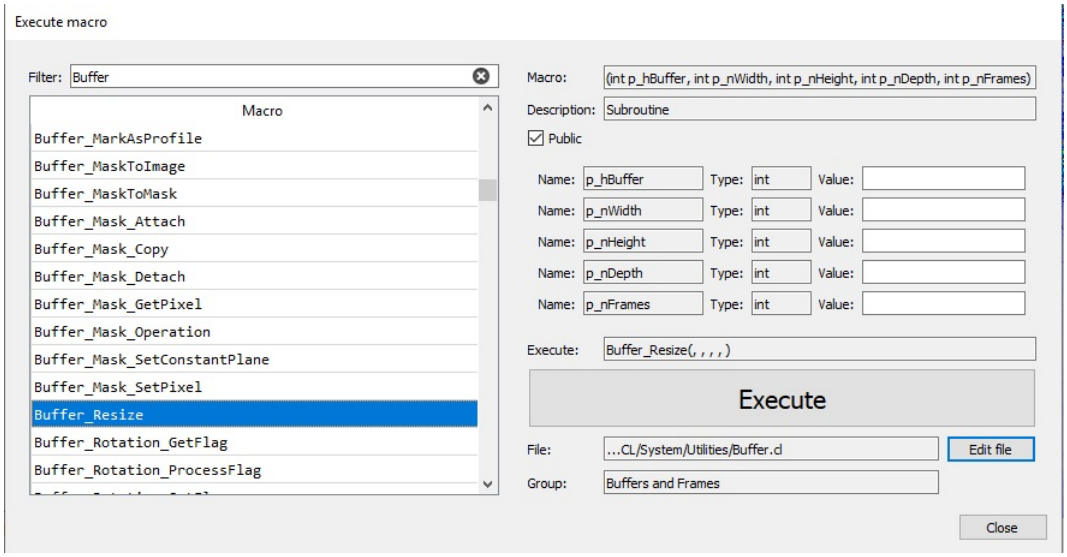


Figure 2.2: The execute function dialog to list all available functions.

4. Optional the function list can be **sorted**. By default all functions are listed in their order of definition.
5. Select a function (subroutine or macro) in the list by clicking on it. Above the list field the function header (return type, name and parameters) and the source is shown (either a macro file or the text "system subroutine"). In the bottom line a short help text appears.
6. Click **Help** to open a manual with further descriptions, if available.
7. Double-click in the list or click **Execute** to start the execution of the selected macro/subroutine.
8. Enter one parameter value after the other in a new dialog box. Click OK.
9. Now the generated CL code is displayed. It is the command to call the selected function and display the result (if any) in the Info window. Click OK to execute. The parameters of the function or the text to be displayed can still be modified.
10. If a function has a return value, it will be displayed in the Info window. In the project browser dialog switch the tabs on bottom left to **Info**.

2.3.3 Execute Line

Enter and execute a single command line.

1. Open a command line box with function Execute Line (press F7).
2. Enter a command (e.g. `B[3]=B[1]+B[2]`).
   Any Command Language statement can be entered. Also several statements in the same line are possible. They have to be separated by a semicolon, for example
   `B[3]=B[1]+B[2]; ExpBuffer(3,4)`
3. Click OK to execute.

Note:

Every valid CL-command is possible: for-loops, if-statements or other "C"-compound statements.

Upper and lower case characters are distinguished, so that, e.g. `varname`, `Varname` and `VarName` are three different names.

2.3.4 Error Dialog

If an error occures while running **DaVis** the dialog box of figure 2.3 appears. The text in the second line describes the error. Button **OK** closes the dialog. The dialog is modal and disables all other functions of **DaVis**.

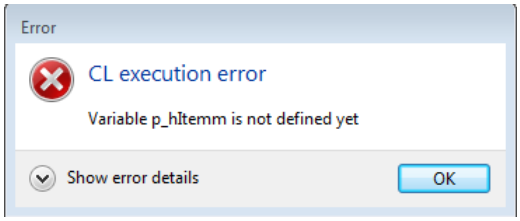


Figure 2.3: Small error dialog with last error message.

The advanced user can get more information, e.g. for developing CL-macros, when pressing button **Show error details**. The dialog will resize and present the layout of figure 2.4.

If the error occurs in a macro file, the name of the file, the recent position (line and column), the name of the macro function and the source text will be shown.

The user can call this dialog box from own macros by the following sub-routine or break the macro execution with this subroutine and an empty `message`:

```
void Error( string p_sMessage )
```

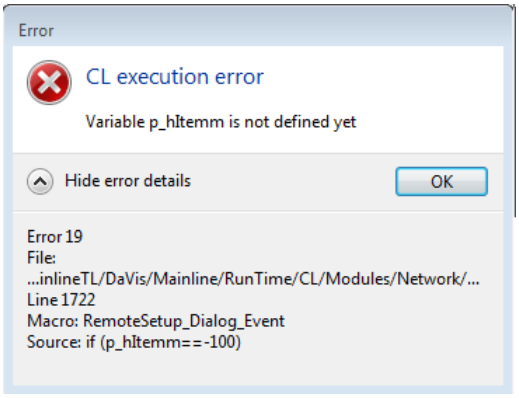


Figure 2.4: Extended error dialog with details about macro file and macro name.

When **DaVis** detects an error, a cleanup function is called, e.g. to switch off the laser, just before displaying the error dialog. The behavior of this function can be changed by the macro programmer, see page 149.

2.4 Variables Dialog

The **Variables** dialog (figure 2.5) displays a list of actually declared variables and their present values. The list can be sorted by group, name or type.

If **view type** is selected every variable is shown with its type, which can be int, float, double or string. Additional type static is shown.

The value of the variable can simply be changed in the editor as long as the variable is not a constant. The new value is directly be assigned to a chosen variable when pressing the Enter key. If the variable is an array, the array index has to be selected to display or edit single elements.

Find String

Enter a (sub)string to view the variables whose name includes the string. A list of different substrings can be given when divided by the space character. All substrings must be found into the variable name in a free order.

If you want to ignore the difference between lower- and uppercase letters, please select **ignore case**.

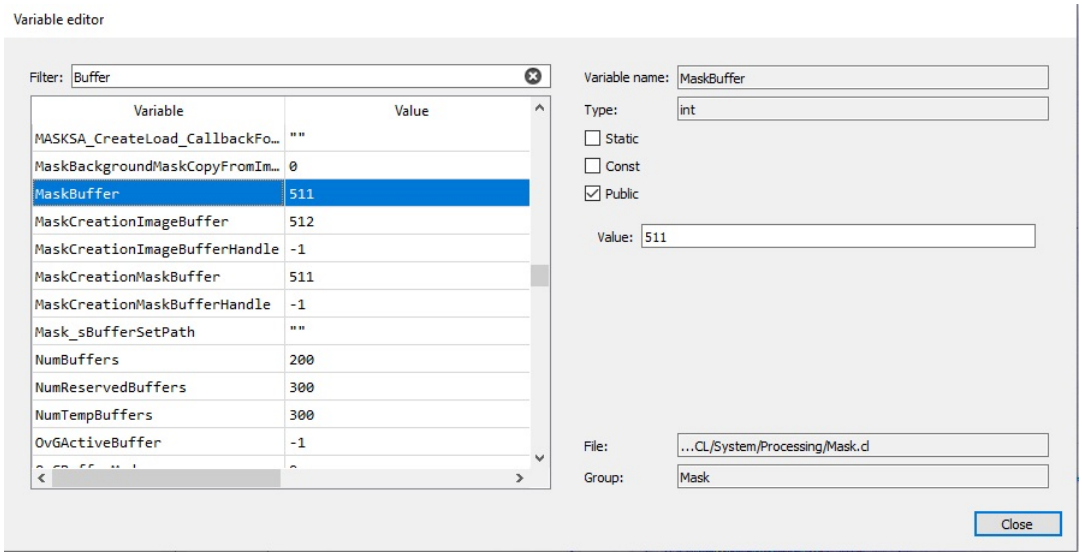


Figure 2.5: The variables dialog to list all available variables and their values.

Group

The variables are internally organized in groups according to their usage. For example, all variables used by processing function **Extract Frame** can be found in group **Batch Operation Extract Frame**.

2.5 Create new Processing Function

This menu item opens the **Processing Function Wizard** dialog. This wizard helps to create the CL code of a new Processing function, stores the automatic generated code into a specified CL file and gives the user all needed information about the structure of own processing. See page 182 for a detailed description.

2.6 Auto Load Changed Macro Files

Before executing a macro **DaVis** optionally checks if a CL-macro file has changed and asks the user, if the changed file should be reloaded. This is used by CL developers and avoids restarts of the software.

Three modes about **check for changed CL files...** are available and can be changed in the **Extras - Preferences** dialog in the **File system** sec-

tion: By default the `User/<name>/CL_Autostart` folder is checked only. The checking can be switched off (**Never**) or processed for **all CL files** loaded so far.

By default a dialog appears if a changed file has been detected and asks the user if the changed files should be loaded. Deselect this mode and load all changed CL files automatically via checkbox **load updated CL files without asking**.

3 CL Language

3.1 Example of a CL File

The following example includes the most important parts of a macro file: global variables and macro functions. When reading this chapter you can always return to this example to see the way, how variables, loops and macros are used.

```
// This is a comment.
// Here the global variables are defined:
float myMaximumValue; // a variable of type float

// The first macro gets a float value as input
// and compares this value to the global variable.
// Return 1 when the new value is larger the the old maximum.
int MyMaximumCompare( float theValue )
{
    if (myMaximumValue < theValue)
    { // new value is a new maximum
        myMaximumValue = theValue;
        return TRUE; // constant for 1
    }
    return FALSE; // constant for 0
}

// Call this macro to execute the upper macro function.
void MyMaximumExample()
{
    int i; // a loop counter
    float value; // stores random values
    for (i=0; i<1000; i++)
    { // try 1000 times:
        value = random(); // get random number
```

```
            if (MyMaximumCompare(value))
            { // print out new maximum value
                InfoText("New maximum:  "+value);
            }
        }
    }
```

3.2 Statements

3.2.1 General Statement Syntax

The syntax for CL statements follows the style of the C programming language and CL has the same arithmetic and logical operators as C.

Each statement must end with a semicolon character (often forgotten by beginners!). More than one statement may be placed on a line:

```
statement1; statement2; statement3;
```

Examples for statements are macro calls like `GetSingleImage()` or arithmetical operations like `result = offset + factor * 7`. Other types of statements are conditional loops, conditional jumps and the `break` and `return` statement.

Text strings are enclosed in quote marks:

```
"this is a textstring"
```

Statements can be **grouped** with the help of brace characters { and } :

```
{
    statement1;
    statement2;
}
```

In contrast to C, CL does not support pre-processor directives (`#define`, `#include` etc.), structures or pointers.

3.2.2 Comments

The beginning of a comment is **denoted with** "//" (two consecutive slash characters). Everything from the // sequence to the end of the line is

treated as a comment. Comments may be on lines by themselves or on the ends of other statements.

A comment can also be specified by beginning the comment with the "/*" character sequence. All following characters are treated as comments up to the matching "*/" sequence.

Example to illustrate both types of comments:

```
// comment line
y = a + b*x; // linear equation
/*
This is a comment.
*/
z = y / 5; /*This is a comment too.*/
```

3.2.3 IF Statement

The form of the IF statement is:

```
if (expr) statement1; [else statement2;]
```

If the expression `expr` is true (not zero), then `statement1` is executed. If the expression is false (zero) and the `else` clause is specified, `statement2` is executed. The `else` clause and the second set of statements are optional.

Example of an `IF` statement controlling a group of statements by enclosing them in braces:

```
if (MaxRect(1,0) > value)
{
    Message ("overflow in buffer 1",0);
    Show(1);
}
```

3.2.4 WHILE Statement

The WHILE statement loops until the controlling expression becomes false (zero).

The form of the WHILE statement is:

```
while (expression)
    statement
```

Proceeding

Each time around the loop the expression is evaluated. If it is true (non zero) the statements are executed and then the process repeats until the expression becomes false. If a BREAK statement is executed within the loop, execution of the loop terminates and control is transferred to the first statement beyond the end of the loop.

Example of a WHILE statement:

```
while (x < 5)
{
    x += xmove;
    y += ymove;
}
```

3.2.5 DO Statement

The DO statement is very similar to the WHILE statement, except the control expression is evaluated at the end of the loop rather than at the beginning. This causes the loop always to be executed at least once. The form of the DO statement is:

```
do
    statement;
while (expression);
```

Proceeding

For each iteration of the loop the statements are executed and then the conditional expression is evaluated. If it is true (non-zero) control transfers to the first statement at the top of the loop.

Example of a DO statement:

```
do
{
    x += xstep;
    y += ystep;
} while (x < limit);
```

3.2.6 SWITCH Statement

The SWITCH statement is used to execute alternative pieces of code depending on the value of an expression. The alternative cases are separated by `case` keywords and ended by the `break` statement.

A default code section is defined by using the keyword `default`.

Example of a SWITCH statement:

```
switch (intvalue) {
    case 1:  InfoText("one"); break;
    case 2:  InfoText("two"); break;
    case 3:  InfoText("three"); break;
    default:  InfoText("illegal value");
}
```

3.2.7 BREAK Statement

The BREAK statement can be used in SWITCH statements. It causes control to go to the statement following the SWITCH statement (see example above). The BREAK statement is not allowed to use in an IF statement like

```
case 1:  if (boolvalue) break; // not allowed!
```

If the BREAK statement is missing, the program executes the next statement, which follows after the next CASE. Sometimes it is useful to leave a BREAK if some cases should execute the same code:

```
case 2:  InfoText("two"); // fall through and execute more:
case 3:  InfoText("three"); break;
case 4:  // no difference between 4 and 5
case 5:  InfoText("four or five"); break;
```

If the value is 2 in the upper example, both strings two and three will be written into the Info window. For the values 4 and 5 the same expression will be executed.

3.2.8 RETURN Statement

The RETURN statement is used in a macro to terminate its execution and return control to the statement following the macro call. If the macro has a return value the RETURN statement is used to return it.

The following **example** is a short macro that returns the square of a float value:

```
float square (float par)
{
    return par*par;
    par = 0;
    // The code behind a return statement is unreachable,
    // hence 'par=0' is not executed.
}
```

3.2.9 FOR Statement

The FOR statement is a **looping control statement** similar to the WHILE statement. In addtion the FOR statement allows to specify initialization expressions that are executed once at the beginning of the loop, and loop-end expressions executed at the end of each loop cycle.

The form of a FOR statement is (`expr`=expression):

```
for (expr1; expr2; expr3) { statements }
```

It is allowed to keep expression 1-3 empty. This example is an endless loop:

```
for ( ; ; ) {...}
```

See below, how expressions 1 to 3 are evaluated when omitted.

Proceeding

The execution of a FOR statement proceeds as follows:

1. Evaluate expression1: If expression1 is omitted, nothing is executed.

   Typically this expression will include assignment operators ("=") to set initial values for loop variables.

2. Evaluate expression2: If expression2 is omitted, it is always true.

   If its value is false (0) terminate the FOR statement and transfer control to the statement that follows the controlled statement. If expression2 is true, proceed to the next step.

3. Execute the statement(s): If more than one statement are to be controlled, enclose them with brace characters `{ }`.

4. Evaluate expression3: If expression3 is omitted, nothing is executed.

   This expression will typically contain operators such as ++, +=, - or -= to modify the value of a loop variable.

5. Transfer control to step 2, where expression2 is once again evaluated.

Example of a FOR statement:

```
for (time=start; time<end; time+=step)
    { statements; }
```

Remark

Note, that no variable declarations are allowed inside of a loop. In **DaVis** all variables are either global, or they are local to the macro function. Once declared inside of a macro, the variable is destroyed when the macro finished.

The following example will break with an error message:

```
for (time=start; time<end; time+=step)
{
    int value = time*time;
}
```

The correct way to program this is:

```
int value;
for (time=start; time<end; time+=step)
{
    value = time*time;
}
```

3.3 Data Structures

The data structures which are used and manipulated in the **DaVis** Command Language apart from **constants** belong to two groups of variables:

Declared variables which are declared locally or globally: scalar values and one-dimensional vectors of the four basic data types.

Buffer variables which give access to the global data stored in image buffers: pixel, row, column and buffer data variables (see section about **Image Window** in the *DaVis Manual*). These need (and can) not be declared.

3.3.1 Constant Values

There are four types of constant values.

Numeric Decimal

Numeric decimal constants may be written in

their **natural form** (1, 0, -1.5, .0003 etc.) or

in **exponential form**, $\pm n.nnnE \pm ppp$,
where $\pm n.nnn$ is the base value and $\pm ppp$ is the power of ten by which the base is multiplied.

Example: The number 1.5E4 is equivalent to 15000.

Hexadecimal

Hexadecimal constants begin with 0x followed by the hexadecimal number. They are of type integer.

Examples:

`0x100 = 256`

`0xFFFF = 65535`

`0x7FFFFFFF = 2147483647` = maximal int.

Octal

Octal constants begin with a leading 0 followed by a number in the octal system. E.g. the variable i will hold the value 8 in the following example

```
int i = 010;
```

String

String constants must be enclosed in **double quotation marks**.

Example: `"This is a string constant"`

Characters

Characters are enclosed in **single quotation marks '**. They provide a way to convert characters to integer values according to the ASCII code.

Example:

```
'c' - 'a' = 2
```

Again, the special characters described above may be used. For example:

```
'\n' = 10
```

Special Characters

Some **special** characters are defined by preceding a backslash \ . They can be included in strings. See table 3.1 for a complete list. The \ncharacter is often used as separation of list items.

Examples:

```
"The first line \n the second line"
"file C:\\img\\demo.imx"
"a double quote \" "
```

string	ASCII-code	character
\"	34	double quote marks
\r	13	carriage return
\n	10	line feed
\t	9	tabulator
\\	92	backslash
\f	12	form-feed

Table 3.1: List of special characters

3.3.2 Declared Variables

Normal CL variables may be used only after they have been declared. The declaration assigns a data type and optionally an initial value to a variable name. The name should reflect the meaning of the data to be stored.

Global / Local Variables

Global variables are valid everywhere after they have been declared. In contrast, local variables are only valid inside the same macro (the one of their declaration). Both type of variables are declared in the same way. The difference is the place of their definition:

- local variables are declared inside a macro
- global variables are declared in any macro file but outside the macros.

Specials:

Local variables may be initialized with an expression:

```
float var1 = 2.3 + 1000 ;
```

While to global variables only constants may be assigned.

Hint: It is preferable to use local variables. Since the user is always certain about their type and actual value.

Constant Variables

Constant variables are set to a value during the creation of the variable and can not be changed later. Constants can be used in a `case` statement of a switch - case - structure. As a programmer's rule the constant variables

are written in uppercase letters. Most constant values are defined in macro file `Constants.cl`.

```
const int BUFFER_FORMAT_IMAGE = 0;
const int FALSE = 0;
const int TRUE = 1;
```

Static Variables

If global variables are declared as static, their actual value is automatically stored. This storage will take place when leaving **DaVis** or by executing the function **Save Settings** of the **File** menu.

```
static int SaveMe = 3;
```

The length of static strings is restricted to 60.000 characters. Some subroutines concerning so called sets (see chapter on page 175) allow to load or store complete groups of static variables as a group.

S special `CLS` file (CL-Set) is created with the same name as the original `CL` file and in format of a `SET` file. The `CL` files are not touched when storing static variables!

Groups of Variables

In all system files lots of groups are already defined by the following statement:

```
#GROUP NameOfTheGroup
```

The group includes all following variables up to the next `#GROUP` statement. The static variables of a group can be stored and loaded by subroutines `LoadSet`, `StoreSet` and `LoadSetGroup` (see page 73).

Reserved Keywords by CL

Keep in mind that the following keywords **are reserved by CL** and can not be used as the names of variables or parameters:

```
b, c, r, p, f, pix, vox,

(also:  B, C, R, P, F, PIX, Pix, VOX, Vox)

if, else, while, do, for,
```

```
switch, case, break, default,

static, const, return,

int, void, string, float, double, word, line.
```

Basic Data Types

CL has the following three basic data types:

- **Int:** 32 bit signed integral numbers (range -2147483648 .. 2147483647)
- **Float:** floating point numbers with 23 bit mantisse + 8 bit exponent + 1 bit sign (range -1E37...+1E37, smallest value 1.5E-45), 7-8 digits
- **Double:** floating point numbers with 52 bit mantisse + 11 bit exponent + 1 bit sign (range -1.7E308...+1.7E308, smallest value 5.0E-324), 15-16 digits
- **String:** arbitrary length of **text characters**, size is restricted by the system memory only

Declarations

Declarations of scalar CL variables are statements of one of the forms:

```
 int iVar1[=value], iVar2[=value],...;
float fVar1[=value], fVar2[=value],...;
double dVar1[=value], dVar2[=value],...;
string sVar1[=value], sVar2[=value],...;
```

depending on the desired data type.

As seen before, the name of a variable may be followed by an equal sign and then a value to which the variable is initialized. If an initial value is not specified, the variable is initialized to 0 or respectively the empty string.

Examples:

```
float fAverage; // initially 0
int nThreshold = 2500;
string sProgramName = "DaVis 8";
string sDataPath = "C:/DAVIS/CL/";
static string sWillBeStored = "Changes are stored!";
```

1-dimensional Arrays

CL allows to declare one-dimensional arrays of the three basic variable types (int, float and string). The size of the array must be at least 2.

Follow the name of the variable square brackets, which enclose the number of array elements. The following statement declares a **1-dimensional floating point array** (i.e., a vector) with 20 elements:

```
float xvec[20];
```

If two-dimensional arrays are needed, use the **buffer variables** or map the two dimensions (`dim1,dim2`) into one dimension like `my2DArray[ i1 * dim2 + i2 ]`.

There are no limits for the size of arrays besides the system memory. Even arrays of the size 0 and 1 are valid and useful e.g. for executing subroutine `CallDll`, which needs an array as second parameter, but not every called DLL-function needs parameters.

Initial values may also be assigned to arrays by following the declaration with an equal sign and a list of values enclosed in curly braces:

```
int xvec[5] = {2,5,7,1,0};
```

The same syntax is valid later in the code and when calling functions with array parameters:

```
xvec = {2,5,7,1,0};
SomeFunction( {2,5,7,1,0} );
CallDll( "DllName", "FuncName", {} );
```

When used in expressions the subscript values are 0 based. That is, the first element of the array is referenced using a subscript value of 0 and the last element is referenced using a subscript value equal to one less than the number of elements in the array.

Example:

The following statements would declare an array with 100 elements and initialize it:

```
float xsq[100]; int i;
for(i=0; i<100; i++) xsq[i]=i*i;
```

3.3.3 Buffer Variables

Buffer variables are references to the global data which is stored in **image buffers**. It's impossible to address buffer components by this variables. There are the following types of buffer variables:

- **Complete Image Buffer:** `B[n]` accesses the buffer number `n` in the range of 0 to the maximum number of buffers. Buffer 0 is reserved to store profiles and should not be used to store own image data. The expression `B[n]` can be abbreviated to `Bn`, if `n` is a constant value. E.g. `B5` is the same as `B[5]`.
- **Buffer Row:** `R[n,i]` The i'th row of buffer `n` .
- **Buffer Column:** `C[n,j]` The j'th column of buffer `n`.
- **Pixel in a Buffer:** `Pix[n,x,y]` The pixel at column `x`, row `y` in buffer `n`. For RGB buffers use Set/GetColorPixel.
- **Voxel in a Buffer:** `Vox[n,x,y,z,f]` The intensity at column `x`, row `y`, plane `z` and frame `f` in buffer `n`.
- **Profile Line:** `P[i]` The profile `i`. It is the same as `R[0,i]`, since buffer 0 is the profile buffer.
- **Profile Point:** `F[i,j]` The j'th point in profile i :
  It is the same as `PIX[0,j,i]`.

Note:
For convenience, upper case and lower case letters can be used as `B, b, R, r, C, c, PIX, Pix, pix, VOX, Vox, vox, P, p, F, f`.

How to use?

Using buffer variables you can read and write buffer data. **Before** addressing buffer data, the buffer has to be created. This is done, e.g. with subroutine `SetBufferSize` or by copying another already existing buffer.

Some **examples** will demonstrate the use of buffer variables:

- `B2 = B1;` copy the contents of buffer 1 to buffer 2
- `b4=4000;` set all pixels in b4 to the value 4000, b4 must be valid before
- `R[1,7]=R[1,1];` copy row 1 of buffer 1 to row 7

- `B[2]=C[3,0];` set all columns in B2 to values of first column of B3
- `PIX[2,3,4]=99;` set a pixel in buffer 2 to value 99
- `int v=pix[3,2,2];` read a pixel value of buffer 3 into the variable `v`

Line

There is one more data type not mentioned before which is used **only in subroutine or macro declarations**: `line`.

A formal parameter of type line in a subroutine or macro declaration means that the actual parameters on a call to the subroutine can be of one of the following compatible types:

- buffer row (`R[n,i]`) or column (`C[n,i]`)
- profile (`P[n]`)
- (declared) one-dimensional array.

The called macro can access the elements of the line variable by subscribing it like a float array. The length of the passed line is determined with the special CL function `sizeof()`.

Example:

See the following example which calculates the sum of the elements in a line:

```
float SumOfALine( line p_Line )
{
    int i;
    float fSum;
    for (i=0; i<sizeof(p_Line); i++)
        fSum += p_Line[i];
    return fSum;
}
```

3.4 CL Operators

In CL expressions almost "everything" can be combined with "everything". When the data types of two operands in an expression are not the same,

the type of the second operand is transformed to the type of the first and the result will be of the first type. The only exception is the sum of an integer and a float variable, which result will be float. See table 3.2 for examples for the automatic type conversion on operations. See section on **data structures** on page 28 for more information on the **DaVis** data types.

3.4.1 Data Type Conversion

The above explained data type conversion is known in C/C++ as **type cast**. In **DaVis** it is possible to execute an explicit type cast as in the following example:

```
B[3] = (float)B[1] + B[2]
```

Here a temporary FLOAT buffer is created from buffer 1, and then buffer 2 is added. The resulting FLOAT buffer is stored in buffer 3. Without this type cast and if buffer 1 would be a WORD buffer, then the result would be of WORD type and the sum could be out of range: If the sum of pixel intensities is larger than 65535, the result is 65535 because of the restriction to the WORD-range 0...65535.

Another example is the conversion from a FLOAT buffer into a WORD buffer:

```
B[2] = (word)B[1]
```

So especially when using * or / but also for + and –, it might be advantageous to obtain the result in the data type FLOAT in order not to loose any significant digits of the data. For example, when dividing two WORD buffers of about the same intensity, the result of each pixel is usually 0 or 1 if stored as a WORD buffer. Another advantage of FLOAT buffers is, that they may contain negative numbers (useful for subtractions).

A float or int variable can be converted into a string by operation

```
float fVal; int iVal; string str = (string)fVal + (string)iVal
```

3.4.2 Arithmetic operators

The arithmetic operators of table 3.3 may be used in expressions for float values or variables, integer values or variables and most operators even for buffer or line variables.

Operation	Result
varInt + varFloat	float
17 + 30.1	47.1
varFloat + varInt	float
30.1 + 17	47.1
varString + varInt	string
"Test" + 17	"Test17"
varInt + varString	int
17 + "Test"	17
17 + "30 people"	47

Table 3.2: Examples for automatic type conversion

For vector buffers the result of this operations is always a simple vector buffer (see table 5.1 page 93 about the buffer formats). When adding a vector and an image buffer, the result is a valid vector buffer if and only if the size of the image buffer is equal to the size of the vector buffer (either for one component or for all components).

When buffers with scalar components are operated, this components will be copied from the first operand into the result. Scalar components of the second operand will be lost. The operations don't touch scalar components.

Operator	Function
++	add 1 to a variable
–	subtract 1 from a variable
+	addition
-	subtraction or unary minus
*	multiplication
/	division
%	modulo (e.g., 7%3=1) for integer variables and Word buffers only

Table 3.3: Arithmetic operators

The ++ and – operators may be used either **immediately before or after a variable name**. If they are used before the name, the increment or decrement is performed before the value of the variable is used in the expression. If they are used after the name, the value of the variable before being modified is used in the expression and then the increment or decrement takes place.

Example:

This sequence assigns the value 4 to x and 3 to y. At the end of the sequence, both a and b have the value 4.

```
a = 3;
d = 3;
x = ++a;
y = d++;
```

3.4.3 Assignment Operators

The assignment operators of table 3.4 can be used in expressions (`var` = variable, `expr` = expression) for all float and integer variables and for buffers/lines.

Assignment	Function
`var = expr;`	**assign** expression to variable
`var += expr;`	**add** expression to variable
`var -= expr;`	**subtract** expression from variable
`var *= expr;`	**multiply** variable by expression
`var /= expr;`	**divide** variable by expression
`var %= expr;`	calculate the **rest of the division**, modulo operation, working on integer variables and Word buffers only

Table 3.4: Assignment Operators

3.4.4 Comparison Operators

The operators in table 3.5 compare two values and produce a value of 1 if the comparison is **true**, or 0 if the comparison is **false**. Both results are defined as constant values `TRUE` and `FALSE` in **DaVis**.

3.4.5 Logical Operators

The logical operators from table 3.6 may be used. Variables and constant values are either `TRUE` if not zero or `FALSE` if equal to zero.

Operator	Function
==	equal
!=	not equal
<=	less than or equal
>=	greater than or equal
<	less than
>	greater than

Table 3.5: Comparison Operators

Operator	Function
!	logical NOT (negates TRUE or FALSE)
&&	AND
||	OR

Table 3.6: Logical Operators

3.4.6 Bit Operators

Table 3.7 gives a list of all available bit operators. Bit values are often used for binary flags, e.g. the state of TTL-I/O-lines. This flags should be stored in integer variables. Constant values are often defined with hexadecimal numbers or, but not possible in **DaVis**, as binary numbers. The values are defined as $\sum_{i=0}^{n} b_i \times 2^i$ with $n \leq 30$ and $b_i \in \{0, 1\}$. Now an operation $\odot$ works on all bits like:

$$
\begin{aligned}
A &= B \odot C && (3.1)\\
&= (\sum_{i=0}^{n} b_i \times 2^i) \odot (\sum_{i=0}^{n} c_i \times 2^i) && (3.2)\\
&= \sum_{i=0}^{n} (b_i \odot c_i) \times 2^i && (3.3)
\end{aligned}
$$

Examples

- Logical variables:

```
int logic1 = TRUE; // = 1
int logic2 = FALSE; // = 0
```

Operator	Function
~	logical NOT (negates TRUE or FALSE)
&	AND: a_i is 1 if b_i and c_i are 1
|	OR: a_i is 1 if b_i or c_i is 1
^	XOR: a_i is 1 if b_i is different to c_i
<<	Bitshift left: $b_i = b_{(i+c)}$
>>	Bitshift right: $b_i = b_{(i-c)}$

Table 3.7: Bit Operators

- Bitflag variables:

```
int flags1 = 19; // = 0x13 = (bin)00010011
int flags2 = 6; // = 0x06 = (bin)00000110
```

- Logical vs. bit operations:

```
logic1 && logic2 = 1 = TRUE
flags1 && flags2 = 1 = TRUE
flags1 &  flags2 = 2 = (bin)00000010
flags1 |  flags2 = 23 = (bin)00010111
flags1 || flags2 = 1 = TRUE
```

3.4.7 String Operators

Strings may be added or compared to each other or assigned to a numerical value:

```
"aaa" + "bbb" = "aaabbb"

"hallo" == "HALLO"; // = FALSE

"LaVision" > "PIV"; // = TRUE

float f = "1.23 counts"; // assigns 1.23 to f
```

Special case: It is possible to assign a string containing many numbers to a 1-dimensional integer or float **array**:

```
int data[3] = {3 4 5};

//data[0] = 3

//data[1] = 4

//data[2] = 5
```

or:

```
int data[3];

data = "3 4 5";
```

Conversely, a numerical value can be converted to a string:

```
string s; s = "value" + 2.3;

// assigns "value2.3" to s
```

3.4.8 Other Operators

Operations can be performed with two more operands.

The **conditional operator** has the form:

```
(operand1 ?  operand2 :  operand3)
```

The value of operand1 is evaluated. If it is TRUE (not zero) then the value of operand2 is the result of the expression. If the value of operand1 is FALSE (zero) then operand3 is the result of the expression.

There is another **special operator [...]** which encloses subscripts on arrays and buffer variables.

3.4.9 Operator Precedence

The operator precedence, in decreasing order, is as follows:

1. subscript (square brackets)
2. unary minus
3. logical NOT
4. ++ and –
5. multiplication, division and modulo
6. addition and subtraction
7. relational (comparison)
8. logical AND
9. logical OR

10. conditional
11. assignment

Parentheses `(...)` may be used to group terms.

3.5 Macro Programs

Macro files (file extension `CL`) contain an arbitrary number of macros and global variables. The global variables must not be defined before the first usage in a macro, and a macro must not be defined before the first call from within another macro.

When calling macros from other macro files, the other macro file must only be loaded at run time before the call. There is no need to "include" a macro file into another macro file.

The macro files are created or changed using a text editor such as Windows NOTEPAD, or any other text editor or word processor. The best choice is the usage of a programming editor with syntax highlighting.

3.5.1 Macro Definition

Every CL program code must be inside a macro definition. The code is then executed by calling the macro and optionally passing parameters to it.

Macro definitions have the following form (the used symbols are explained below):

```
return_type macroname ([type1[&] par1, type2[&] par2, ...])
{ statements }
```

The above symbols have the following meanings (terms in square brackets are optional):

- **return type**: Data type of the **value** that is returned by the macro. If the macro doesn't return a value, the return_type must be specified as `void`.
- **macro name**: The naming rules for macros are the same as for declared variables.

- **type**: Data type of a formal parameter. If the type specification is followed by "`&`", the parameter is passed by reference, otherwise it is passed by value. The difference is explained below.
- **par**: A formal parameter. This is a variable which has the specified type and the value passed when the macro is called.

3.5.2 Macro Call

A call to a macro has the form

```
macroname ([par1, par2, ...]);
```

where `par1`, `par2`, ... are the actual parameters which are passed to the macro.

Macros without parameters are defined and called with empty brackets, e.g. `macroname();`

Passing a Parameter

There are two mechanisms of passing a parameter to a macro, depending on the type of the formal parameter:

- **Pass-by-value:** the value of the actual parameter is passed to the macro, e.g.: `macro1(7);`
  The called macro can use this value but assignments to the parameter inside the macro can't affect variables outside it.
- **Pass-by-reference** (indicated by "`&`")**:** a reference to a variable is passed to a macro, e.g.: `macro2(var);`
  The called macro can read the value of the variable and assign new values to it. The new value can then be used in the code calling the macro.

Example:

How to define and call two CL macros.

1. Declaration of two macros:

```
int macro1(int par) // pass-by-value
```

```
{ par*=10; return par;

}

void macro2(int& par) // pass-by-reference

{ par*=10;

}

int var=5, result; // variables used below
```

2. Calling the macros:

```
result = macro1(7); // result := 70

result = macro1(var); // result := 50;
        // var is unchanged

macro2(7); // impossible: a variable must be passed rather
            // than a constant

macro2(var); // var := 50
```

Recursive Macro Calls

Recursive macro calls are allowed but will break with a macro failure at a certain recursion depth. Of cause every recursive code can be rewritten into a non-recursive macro.

Calling a recursive macro means, that a macro calls itself, either on direct way (macro A calls macro A) or via a stack of other macros (macro A calls macro B and macro B calls macro A).

A simple example for recursive macros is the calculation of factorials:

```
int GetFactorial ( int n )
{
    if (n>1) return n * GetFactorial(n-1);
    return 1;
}
```

3.5.3 Private and Public Variables and Macros

By default all global variables and all macros can be used from every macro file in **DaVis**.

For a clean programming interface it could be better to hide functions and variables to other macro files, so they can not call functions like a dialog event handler or change the values of some global variables.

Therefore **DaVis** 7 uses the keywords `#PUBLIC` and `#PRIVATE`. Starting with the beginning of a CL file, all symbols are declared as public automatically. All symbols following a `#PRIVATE` line can be accessed from this CL file only.

Is is useful to write the first part of a CL file like a header file in C/C++: Define the public global variables and the public macros, which need no body of code in the public part of the CL file. The body can follow later in the file in a private area.

Friends of Macro Files

For complete packages or very large macro systems (e.g. the hardware device manager or the Data and Display Attributes dialog) the macros are spread over more than one CL file. But all this CL files must be able to access private variables and functions of at least the main macro file of this function group.

Therefore a number of CL files can be combined with keyword `#FRIEND`:

```
#FRIEND NameOfForeignMacroFile
```

Now the foreign macro file is allowed to access private variables and macros of the CL file, which defines the friend. The name must be given without path and without extension.

Example for Public/Private

In the following example the function `MyPublicFunction` can be executed from everywhere in **DaVis**, while a call to `MyPrivateFunction` creates an error message.

```
float MyPublicFunction( float val1, float val2 );
```

```
#PRIVATE

float MyPublicFunction( float val1, float val2 )
{ return MyPrivateFunction(val1,val2);
}

float MyPrivateFunction( float val1, float val2 )
{ return sqrt( val1*val1 + val2*val2 );
}
```

3.6 Loading CL files during startup

During the start of **DaVis** all CL files from the program subfolder `CL` are loaded from internal initialization of the macro language. Then macro function `StartUp` is called and loads more CL files e.g. depending on the available packages and licence. At the end of this startup procedure all `CL` files from the autostart folder `User/<name>/CL_Autostart` are loaded, followed by file `User.cl` which is loaded last. At this point the **DaVis** main window is visible.

The `User.cl` has been created automatically during startup since **DaVis** 5 but has been removed in **DaVis** 10.2.1. If existing in an old configuration file it will be loaded also in newer versions.

4 Subroutines

4.1 About Subroutines

Subroutines are system-defined functions for a variety of purposes (such as data I/O, and data processing). In contrast to macros, subroutines are only declared but not defined. The executable code is embedded in the main file `DAVIS.EXE` or in dynamic links libraries (`DLL`s) and is therefore not visible for the user. Such subroutines are declared with the keyword `intern`, e.g.

```
intern int MacroThread_Interface( int theMode,
    int theHandle, string theMacroCommand );
```

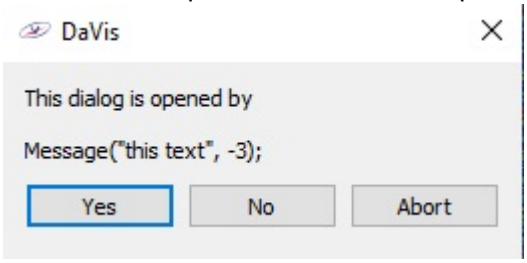
The subroutines are declared in different function groups and `CL` files. In the `Execute Function` dialog the list of displayed macros and subroutines can be restricted to groups or files. The subroutines for buffer and dialog handling are described in chapters starting on page 85 and on page 151.

4.2 Keyboard & Screen I/O

Message

```
int Message( string message, int mode );
```

Display a short text in a message box (default mode is 0, more modes are listed in table 4.1). The box is opened in a centered position.



In most cases this used to give the user a simple information and wait until the user presses the **OK** button. Other operation modes allow the creation

of a dialog with two buttons (**OK** and **No** or **OK** and **Abort**) or with all three buttons.

Mode	Display Message
0	with OK-button.
-1	with yes/no-button. Return 0 if yes is pressed, 1 if no is pressed.
-2	with OK/abort-button. Return 0 if OK and 1 if abort is pressed.
-3	with yes/no/abort-button. Return 0/1/2 for yes/no/abort.
> 0	display message for this time in milliseconds.

Table 4.1: Modes for subroutine `Message`.

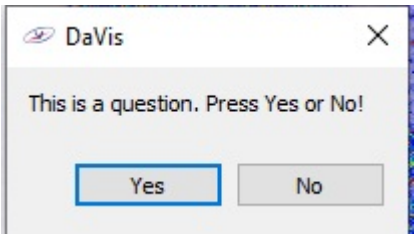
Question

```
int Question( string question );
```

Display a question in a centered message box and get the answer (yes/no) from the user. The user's input is returned as Yes = 1 or No = 0.

The user has the possibility to stop the execution of the macro by clicking the STOP button or by pressing the `ESCAPE` key. The execution of a macro is **not** stopped, when **No** has been entered

The STOP button is not displayed when the `question` string starts with characters `"~\n"`. Then as message text all characters after the line break are used.



InfoText / ScrollInfoText / ClearInfoText

```
void InfoText( string theText );
```

Append a line of text to the info window. More than one line can be appended in the same call with '`\n`') as line break, e.g.

```
InfoText("Line 1\nThe second line.")
```

With a special string the programmer can define the line number to be displayed: `InfoText("\J<line>")`, where `line` is the line number (started

with value 0 for the first line) or value -1 for the last line and end of text. This function can easily be used as macro

```
void ScrollInfoText( int theLineN );
```

Another special function is implemented to clear the complete text of the Info window and set the tabulator width, which is stored in variable `InfoTextTabSize`. `InfoText("\C")`. Again there is a macro to call this function:

```
void ClearInfoText();
```

GetInfoText

```
string GetInfoText( int fromLine, int toLine );
```

Return the given lines from the info window. Negative values are used to define the last lines of the text.

Examples: `GetInfoText(0,-1)` returns the complete info text, `GetInfoText(-10,-1)` returns the last ten lines.

Macro `SaveInfoText(string filename)` stores the complete info window text in a file.

Error

```
void Error (string message);
```

Open **DaVis** error dialog box (see page 17) and display `message`. If `message` is the empty string, the macro is aborted, but the error dialog is not displayed.

EnterString / Float / Int

```
int EnterString( string prompt, string& input );
```

```
int EnterFloat( string prompt, float& input );
```

```
int EnterInt( string prompt, int& input );
```

Display the string prompt in a message box and let the user enter a character string (or a real (floating point) value or a 32-bit signed integer value).

As always, the user has the possibility to stop the execution of the macro by clicking the STOP button or by pressing the key ESCAPE. The return value will be 0 if canceled or 1 if the OK button has been pressed.

CheckKey

```
int CheckKey();
```

Get the ASCII code of the keyboard memory (0 if no key has been pressed) and return immediately. This function is, for example, used to realize the keyboard control while continuously grabbing images.

Table 4.2 shows some special return codes. Standard character keys can be checked like:

```
if (CheckKey()=='A') InfoText("pressed key A");
```

Code	Key
97,...,122	a,...,z
65,...,90	A,...,Z
48,...,57	0,...,9
301	cursor down
302	cursor up
303	cursor right
304	cursor left
305	page up
306	page down
307	home / pos1
308	end
312	insert
127	delete
9	tabulator
13	return/enter
368,369,...	F1, F2, ...

Table 4.2: Values returned by `CheckKey` for some important keys.

Wait

```
int Wait(int ms);
```

Wait the specified number of milliseconds. The execution is not interruptible. If `ms` is a positive value, the waiting time is very exact in a range of microseconds, but the waiting is active and the CPU usage is high. On negative values `ms` the waiting is passive with low CPU usage, but the error is in a range of milliseconds.

4.3 File Handling

4.3.1 File and directory handling

Delete / Rename

```
void DeleteFile( string filename, int mode );
```

Delete a disk file. The `filename` may not include a path. If the file does not exist, if the file is opened by some other software or if the user privileges don't allow the operation, then the macro stops with an error message (`mode=0`). In `mode=1` the macro does not stop.

```
void Delete( string filename );
```

Delete a file from disk. The `filename` may not include a path. Retired subroutine, same as `DeleteFile(filename,0)`.

```
void Rename( string oldname, string newname);
```

Rename a file. Both names may include a path, and both pathes must be on the same drive.

CopyFile

```
int CopyFile( string source, string dest);
```

Copy a file and return 0 if no error occurred. Otherwise the return value is the error code: source file not found (1), can't open destination file (2), error during copy process (3, e.g. harddisc full).

MoveFile

```
int MoveFile(string source, string dest);
```

Move file or directory and return 0 if no error occured.

XCopyFile

```
int XCopyFile(string source, string destpath, int mode);
```

Copy file(s) and return 0 if no error occurred (wildcards are allowed). Set bit 0 of the `mode` flag to copy subdirectories.

For example, copy complete directory structure: `XCopyFile("d:\\data\\*.*", "e:\\backup", 0x01)`

FileExists

```
int FileExists(string filename);
```

Check if a disk file with a given name exists. Return TRUE (1) if the file exists, otherwise FALSE (0). Specify file including complete path.

If `filename` is a drive name, the function returns the drive type. The return value is 2 for a floppy, 3 for a harddisk, 4 for a netdrive, 5 for a cdrom or 0 if device not exists.

Example: `type = FileExists(“a:” )`
The type should be 2, because `a:` is a floppy drive on most systems.

GetDirectory

```
string GetDirectory (string thePath, int mode);
```

Get the list of files (mode=0) or subdirectories (mode=1) in a given directory.

The specified path may contain wildcards, e.g. `C:\nIMAGES\nFLAME*.IMX`

GetFileDate / Size

```
string GetFileDate (string filename);
```

```
int GetFileSize (string filename);
```

Get date and time of the last change or size (in byte) of a file. If the file does not exist, the return values are the empty string or -1.

IsFileWritable

```
int IsFileWritable (string filename);
```

Return TRUE if the existing file has a write access and can be overwritten or deleted.

GetDriveInformation

```
string GetDriveInformation(string theDrive, int theMode);
```

Get information for `theDrive`, e.g. `"d:"`, about the drive type (`theMode=0`, return 2 (floppy), 3 (harddisk), 4 (netdrive), 5 (cdrom) or 0 if device not exists). Return the drive name (`theMode=1`), the free drive space in megabytes (2) ot the total drive space in megabytes (3).

GetDirName / ChDir / MkDir / RmDir / RmDirTree

```
string GetDirName();
```

Get name of current directory path.

```
void ChDir(string toDirectory);
```

Change the current directory path.

It is the default path to which any other function (e.g. GetDirectory, Rename, ...) refers, if no other path is explicitly specified.

```
void MkDir(string newDirectory);
```

Create new subdirectory in the current directory path.

```
void RmDir(string delDirectory);
```

Delete empty subdirectory in the current directory path.

```
void RmDirTree(string delDirectory);
```

Delete all files in subdirectory and delete subdirectory in the current directory path.

4.3.2 Macros for file name access

MakeNiceFileName

```
string MakeNiceFileName( string filename )
```

This function creates a valid filename. All forbidden characters, e.g. the '*' and '?', are replaced by an apostrophe.

Extract path, filename and extension

```
string FileGetPath( string pathAndFileName )
```

Return the path name only and throw away the filename.

```
string FileGetName( string pathAndFileName )
```

Return the filename only and throw away the path.

```
string FileGetExtension( string pathAndFileName )
```

Return the extension only.

```
string FileStripExt( string pathAndFileName )
```

Return the path and filename without extension.

```
string FileStripPath( string pathAndFileName, string path )
```

If the path of `pathAndFileName` is equal to `path`, then strip the path and return the filename only.

4.3.3 Dialogs for file selection

FileSelectBoxOpen/Save

```
string FileSelectBoxOpen(string boxtitle, string filetypes, int allowmultiple);
```

Open a fileselectbox with title **boxtitle** to open a file(s) of a type of list **filetypes**, which is a special string like

```
"Text (TXT, DOC)\n*.txt;*.doc\nBitmap (BMP)\n*.bmp"
```

The string is a list of pairs of a type description and its type extension. All elements are divided by a carriage return ('\n').

If `allowmultiple=0` only one file can be selected. The function returns the selected filenames or an empty string. If more than one file is selected, the returned string has the form `"path\nname1\nname2\n...\nnameN"`.

Only existing files can be selected.

`string FileSelectBoxSave(string boxtitle, string filetypes, string filename);`

Open a fileselectbox with title boxtitle to save a file with a type of list filetypes. The function returns the selected/entered filename or an empty string. Is is possible to enter the name of an unexisting (=new) file.

Calls to retired subroutines `FileSelectionBox` and `DirSelectionBox` should be replaced by `FileSelectBoxSave`.

Note:

All selection box dialogs use string variable `DefaultPathFileSelectBox` as default file path (the path which opens first in the fileselectbox). If the user changes the path in the dialog box, the new path is stored in `DefaultPathFileSelectBox`.

Standard file selection boxes in menu **File – Load** or **Save** are using their own path, which will not be changed by the above defined file selection boxes.

If a special path should be used in the macro functions, a string variable has to be declared to store the special path. After defining this variable the above defined subroutines can be called. Otherwise every call to one of the above functions could change the default path and destroy a combination of load ... action... store or load ... action ... load from same path.

```
string yourownpath;
void yourfunctionsave()
{
    string filename;
    DefaultPathFileSelectBox = yourownpath;
    filename = FileSelectBoxSave("type","title","");
    yourownpath = DefaultPathFileSelectBox;
    // some code to save filename
}
```

4.3.4 Working with Text and Binary Files

The functions to access text and binary files are more less equal to the well known C-functions: At first a file is opened by the `Open` command, which returns a so called **file handle** for later accessing this file by subroutines `ReadLine`, `ReadInt`, `ReadIntBin`, `ReadFloat`, `ReadFloatBin`, `ReadFile`, `ReadBytes` and `Write`. At the end the file must be closed with the `Close` command.

Open

```
int Open( string file, string mode );
```

Open a file for reading or writing. Return the file handle or 0 on errors. This could happen if a file for reading access does not exist or if a file with writing access could not be created.

Parameter `mode` should be one of the characters `r` for reading the file or `w` for writing. With `mode = "a"` the file is opened for writing access, but all writing operations are appended to the existing file.

With the optional `mode = "O"` flag a file will not be closed by a call to `Close(-1)`. In this case a file can stay open at the end of a macro run. By default all open files are closed at the end of a macro.

When the file should be read by `ReadIntBin`, `ReadFloatBin` or `ReadBytes`, the character `b` must be appended to the `mode` to tell the system about the so called binary file: use either `mode="rb"` or `mode="wb"`.

Close

```
void Close( int theHandle );
```

Close a file, which has been opened by the `Open` command. Use parameter -1 to close all files, which have been opened by `Open` and without the `mode = "O"` flag. This function is called automatically whenever stopping a macro by hand or on errors.

ReadLine

```
void ReadLine( int theHandle, string& theLine );
```

Read a line from a text file into a string variable. Up to 10 000 characters can be read at once. Longer rows will be splitted.

ReadInt / ReadFloat

```
void ReadInt( int theHandle, int& theNumber );
```

Read an integral number from a text file into an integer variable.

```
void ReadFloat( int theHandle, float& theNumber );
```

Read a floating point number from a text file into a float variable.

ReadIntBin / ReadFloatBin

```
void ReadIntBin( int theHandle, int& theNumber, int theByteOrder );
```

Read a 32-bit integral number from a binary file into an integer variable. The order of the bytes which are combined into the integer value depends on the system which has been used to create the file. Set `theByteOrder` to 0 for an Intel-based PC system or to 1 for an Unix workstation system.

```
void ReadFloatBin( int theHandle, float& theNumber, int theByteOrder
);
```

Read a floating point number from a binary file into a float variable.

ReadFile

```
void ReadFile( int theHandle, string& theText );
```

Read a whole file into a string variable.

ReadBytes / ReadBytesToBuffer

```
int ReadBytes( int p_hFile, string& p_rsData, int p_nLength );
```

Read part of a binary file (`p_nLength` bytes) into a string. Return number of read bytes. This functions reads everything up to the end of the file with `theLength=0`.

The reading operation starts at the recent position of the file, so one can call some `ReadIntBin` and `ReadFloatBin` commands e.g. to read a file header before using the `ReadBytes` subroutine.

```
int ReadBytesToBuffer( int p_hFile, int p_hBuffer, int p_nLength );
```

Same as `ReadBytes`, but the data is read into a buffer of type byte. The buffer with handle `p_hBuffer` is created internally. If the number of bytes/characters (`p_nLength`) to be read is negative, then read up to the end of the file.

EndOfFile

```
int EndOfFile( int theHandle );
```

Determine if the end-of-file is reached when reading (Yes=1, No=0).

Write

```
void Write( int theHandle, string theText );
```

Write a character string to a text file.

4.4 Strings

The characters in a string are indexed from 0 to (stringlength-1).

4.4.1 General Functions for Strings

ToUpper / ToLower

```
string ToUpper (string str);
```

Convert all characters in a string to upper case.

```
string ToLower (string str);
```

Convert all characters in a string to lower case.

StrLen

```
int StrLen (string str);
```

Get the length of a string.

CountChar

```
int CountChar( string str, int ch );
```

Count the occurences of a character in a string.

Example: `CountChar("1221221", '1') = 3`

Strip

```
string Strip (string str, int mode, int char);
```

Strip leading (mode=1) or trailing (mode=2) characters (typically blanks) or both (mode=3) off a string.

ExStr

```
string ExStr(string str, int from, int to);
```

Extract a substring from a string.

Example:

```
string s = "LaVision";
string s1 = ExStr(s,2,4)
```

The string variable `s1` includes the string `Vis`.

PosInStr

```
int PosInStr(string str, string search, int fromPos);
```

Search a string in another string; returns the position (0...strlen-1) or -1 if not found. If `fromPos<0` the string is searched from right to left.

ReplaceChar

```
string ReplaceChar (string str, int ch, int newchar);
```

Replaces all occurrences of a character by another and returns the new string.

ReplaceStrings

```
string ReplaceStrings( string& theString, string theSearch, string theReplace );
```

Replace all occurrences of `theSearch` in `theString` by `theReplace`.

ReplaceStringsEx

```
string ReplaceStringsEx( string& theString, string theSearch, string theReplace, int theIgnoreCase );
```

Replace all occurrences of `theSearch` in `theString` by `theReplace`. If `theIgnoreCase` is set to `TRUE`, the search is done case insensitive.

Hex / Bin / Decimal / DecimalBin / IsHex / IsBin

```
string Hex(int decimal, int length);
```

```
string Bin(int decimal, int length);
```

Convert a (non-negative) decimal number to a hexadecimal or binary string. If the string is shorter then length, some '0'-characters are added in front of the string.

Examples:

```
string sh = Hex(12345,4); //-> "3039"
```

```
string sb = Bin(19,6); //-> "010011"
```

```
int Decimal(string hex);
```

```
int DecimalBin(string bin);
```

Convert a hexadecimal or binary string into a decimal (integer) number.

```
int IsHex(string hex);
```

```
int IsBin(string bin);
```

Return 1, if the string has a hexadecimal or binary format, or 0 for a wrong format.

FormatNumber

```
string FormatNumber (string format, float number);
```

Convert a number to a formatted string for output. The format string is like in the C function 'printf', but works on floating point variables only.

Examples:

```
FormatNumber("e", exp(1))
```
→"2.718282e+00" exponential form

```
FormatNumber("10g", exp(1))
```
→" 2.71828" 10 characters, leading blanks

```
FormatNumber("10.5g", exp(1))
```
→" 2.7183" 10 characters, 5 significant digits

```
FormatNumber("10.2f", exp(1))
```
→" 2.72" 10 characters, 2 digits after '.'

```
FormatNumber("010.2f",exp(1))
```
→"0000002.72" like above, with leading zeroes

Get / SetChar

```
int GetChar(string str, int pos);
```

Get the character at a specified position in a string.

```
void SetChar(string& str, int pos, int newchar);
```

Set (replace) the character at a specified position within a string.

Example:

```
string str = "LaVision"
int t = GetChar(str,2);
```

Now the integer variable `t` includes the ASCII code "86" for the letter "V".

Time

```
string Time(int format);
```

Get the recent time in a desired format (see table 4.3). The format's character has to be enclosed in single quote marks.

Example: `string day = Time('A');`

Format	Resulting date and time
0	(default) date, year and time
a	abbreviated name of week-day
A	complete name of week-day
b	abbreviated name of month
B	complete name of month
c	date, time and year
C	century (19 or 20)
d	two digits for the month's day (01 - 31)
D	date in the format MM/DD/YY
G	date in the format DD.MM.YYYY
g	date in the format DD.MM.YY
H	two digits for the hour (00 - 23)
I	two digits for the hour (01 - 12)
m	two digits for the month (01 - 12)
M	two digits for the minute (00 - 59)
p	AM or PM
S	two digits for the second (00 - 59)
T	milliseconds with three digits
w	week-day, starting with Sunday: 0 (0 - 6)
x	date (e.g. "Thu Sep 03, 1998")
X	time (HH:MM:SS)
Y	four digits for the year (e.g. 1998)
y	two digits for the year (00 - 99)
z	time with milliseconds (HH:MM:SS.mmm)
Z	number of milliseconds (60*1000*minute + 1000*second + ms)

Table 4.3: Time formats of subroutine `Time`.

4.5 Strings, Tokens, Lists and Tables

Often in **DaVis** a string is used to store information in the style of lists and tables. This lists and tables include a number of so called **tokens** (in lists) or **items** (in tables). The tokens are separated by a **delimiter**. For lists two different delimiters are used, one for the columns and one for the rows.

All of the following functions get as parameters a string (the list or table), the delimiter and in most cases the index of the requested token. The index is always counted starting with 1.

4.5.1 Working with Tokens

GetTokenN and GetElementN

```
string GetTokenN(string str, string delim, int n);
```

```
string GetElementN(string str, string delim, int n);
```

Get the n-th token from `str` (n = 1,2,...). The substrings are divided by the delimiter `delim`. If the n-th token does not exist, an empty string is returned.

While `GetTokenN` counts two or more delimiter as one, if they have no other characters between the delimiter, the subroutine `GetElementN` works with such empty tokens.

Example:

```
str = "Token 1\nToken 2\nToken 3\nToken 4";
GetTokenN(str,"\n",2) ♢ "Token 2"
```

FindToken

```
int FindToken( string theString, string theToken, string delim );
```

The *reverse* function to `GetTokenN` returns the token number (1..n) of `theToken` in string `theString` with `delim` as delimiter. If the token could not be found, value -1 is returned.

Example:

```
str = "Token 1\nToken 2\nToken 3\nToken 4";
FindToken (str,"Token 2","\n") ♢ 2
```

DeleteTokenN

```
string DeleteTokenN( string theString, string delim, int theN );
```

Delete token number (1..n) in `theString` with `delim` as delimiter

Example:

```
str = "Token 1\nToken 2\nToken 3\nToken 4";
DeleteTokenN(str,"\n",3) ♢ "Token 1\nToken 2\nToken 4"
```

ReplaceTokenN

```
string ReplaceTokenN( string theString, string delim, int theN,
string theNewToken );
```

Replace token number (1..n) in `theString` with `delim` as delimiter by `theNewToken`.

Example:

```
str = "Token 1\nToken 2\nToken 3";
ReplaceTokenN (str,"\n",2,"Test") ♢ "Token 1\nTest\nToken 3"
```

SortStringByMode

```
string SortStringByMode(string theInputString, int theDelimiter,
line& oldIndexList, int theMode);
```

Sort the string list, return the sorted list. The items in the list are seperated by a delimiter character.

For `theMode = 0` sort string by index list. For `theMode = 1` sort string and return the index table of the old list. For `theMode = 2` sort string, ignore case and return the index table of the old list. Add value 256 to `theMode` to sort in inverse direction.

Example:

```
string sort = GetDirectory("c:/davis/*.*",0);
int oldlist[1000];
InfoText(sort);
InfoText("");
InfoText(SortStringByMode(sort,'\n',oldlist,0));
```

4.5.2 Working with Tables

Some subroutines and macros can be used to program string tables. The general functions need both delimiters for columns and rows as parameters, the functions for **standard tables** (`StdTable`) don't need them. Standard tables use the tabulator `\t` as delimiter for columns and the line break `\n` as delimiter for rows.

The position of columns and rows are counted starting with 1.

CreateTable

```
string CreateTable( string delimColumn, string delimRow, int sizeX,
int sizeY );
```

Create a table with the given delimiters and the defined size. Return the table as string.

GetTableItem

```
string GetTableItem( string table, string delimColumn, string delimRow,
int x, int y );
```

Get item at position (x,y) of a string table with delimiters for columns and rows.

SetTableItem

```
string SetTableItem( string table, string delimColumn, string delimRow,
int x, int y, string itemText );
```

Set item at position (x,y) of a string table with delimiters for columns and rows. If table is empty, then a complete table is created with size (x,y). Return the new table.

Same Functions for Standard Tables

```
string CreateStdTable( int sizeX, int sizeY );
```

```
string GetStdTableItem( string table, int x, int y );
```

```
string SetStdTableItem( string table, int x, int y, string itemText );
```

```
void GetStdTableSize( string table, int& sizeX, int& sizeY );
```

Returns the size (rows and columns) of the given table.

4.6 Scalar Arithmetic

`float sin(float arg);` the sine function.

`float cos(float arg);` the cosine function.

`float tan(float arg);` the tangent function.

`float atan(float arg);` the arcus tangent function.

All trigonometric functions take the argument as a value in **radiant** (RAD).

```
float atan2(float y, float x);
```

The arcus tangent function calculates atan(x/y), but takes quadrants into account (no division by 0).

Example:

```
float arg = 1, result;
result = sin(arg); // the result is 0.841
```

`float abs(float arg);` absolute value of arg

`float exp (float arg);` exponential function

`float log(float arg);` natural logarithm function

`float sqrt (float arg);` the square root.

```
float power(float arg, float exponent);
```

Raise a number arg to a power exponent.

```
float random();
```

Random number between 0.0 and 0.99999.

```
void randomize();
```

Initialize the random number generator.

```
int min(int i1, int i2);
```

`int max(int i1, int i2);`

Get the minimum (maximum) of two integer numbers.

`float fmin(float i1, float i2);`

`float fmax(float i1, float i2);`

Get the minimum (maximum) of two float numbers.

`int floor(float x);`

Returns the highest integer number, that is smaller or equal to x: Returns $n * modulowithn * modulo <= x < (n + 1) * modulo$ with n as integer.

`int ceil(float x);`

Returns the smallest integer number, that is higher or equal to x

`float rounddigit( float value, int digit );`

Round `value` to a number of leading `digits`. For example `rounddigit( 3.14159, 2 ) = 3.1` and `rounddigit( 3.14159, 4 ) = 3.142`

4.6.1 Mathematical Functions

The following macros are defined in file `CL/System/Utilities/Math.CL`.

`float acos( float x );`

`float acosd( float x );` arcus cosinus in degrees

`float asin( float x );`

`float asind( float x );` arcus sinus in degrees

`int sgn( float x );` signum function: -1 for negative x, +1 for positive x

`float fmod1( float x );` Returns x mod 1.0

`int range(int theMin, int theValue, int theMax);` Returns theValue between theMin and theMax.

`int frange(float theMin, float theValue, float theMax);` Returns theValue between theMin and theMax.

4.7 Serial COM Port

All serial port subroutines need the number of the COM port as first parameter. In normal cases the PC has two ports (`port=1/2`), but **DaVis** supports up to 19 serial ports.

The data transfer can be executed in a special mode to receive strings with a size larger than some kilobytes or to receive a complete word buffer (could be used for raw data instead of character strings). The special modes are defined by subroutine `SetComParameter`.

SetBaudRate

```
void SetBaudRate(int port, int baud);
```

Set the baud rate (bits per second) of the serial port. Most common values are 1200, 2400, 9600 and 19200. The maximum rate depends on your PC, but it can be 128 000 or even more.

SetComParameter

```
void SetComParameter( int port, int bytesize, int parity, int stopbits);
```

Set special parameters of a serial port. Default settings are 8,0,0 for every port. If single parameters should be changed, use value -1 for the unchanged parameters.

`bytesize` : 5 to 8

`parity` **:** 0=no, 1=odd, 2=even, 3=mark, 4=space

`stopbits`**:** 0=1 bit, 1=1.5 bits, 2=2 bits

The use of 5 data bits with 2 stop bits is an invalid combination, as is 6, 7, or 8 data bits with 1.5 stop bits.

Use `bytesize<0` to change some special settings:

`bytesize = -1`: use overlapped mode if `parity!=0`

`bytesize = -2`: set buffer size for `SendCom` to value of `parity`

`bytesize = -3`: close serial port

`bytesize = -4`: not used yet

`bytesize = -5`: echo mode: undefined (0), no echo send by device (1), echo send by device (2)

`bytesize = -6`: msec pause between single characters during send or receive

The following macro functions can be used as shortcuts for the special parameter settings:

```
void SetComDefault( int port );

void SetComOverlappedMode( int port, int enable );

void SetComBufferSize( int port, int buffersize );

void CloseComPort (int port);

void SetComFastMode (int port);
```

SendCom

```
string SendCom( int thePort, string str2send, int theTimeout, int termchar);
```

Send a character string to a device connected via a serial port. Returns the received string, which length is smaller than 4096 characters by default.

`str2send`**:** string to be sent (may be "", in which case it is only checked, if something is received)

`theTimeout`**:** wait at most this number of milliseconds for response from device; don't wait and return immediately if `theTimeout<0`

`termchar`**:** if this char is received, return immediately: usually this is 13 = end-of-line (CR)

Example:

```
SendCom(1,"", 0, 0 ); // clear input buffer (COM1:)
receiveStr = SendCom(1, "hi", 1000, 13)
```

Wait for response at most 1 sec or until a carriage return comes.

Fast sending and receiving:

The fast version of sending and receiving strings is disabled by default for downwards compatibility with earlier versions of **DaVis**. If the echo mode

is disabled by `SetComParameter(port,-5,1,0)` and the character pause is switched of by `SetComParameter(port,-6,0,0)`, it is possible to send large strings with a high baudrate.

ReceiveCom

```
string ReceiveCom (int thePort, int theStringLength, int theTimeout)
```

Receive a character string of length `theStringLength` from a serial port. Returns if the complete string has been received or if timeout.

Note: The maximum time waited by this subroutine is calculated from the given timeout (in milliseconds), from the string length and the baud rate of this serial port.

ReceiveComBuffer

```
int ReceiveComBuffer (int thePort, int theWordValues, int theTimeout,
int theBuffer)
```

Receive a word array of size `theWordValues` from a serial port and store the array in `theBuffer`., which is set to a size one line with `theWordValues` pixel. The number of received words is returned.

Send / ReceiveByte

```
void SendByte (int port, int byte);
```

Send a character to a device connected via a serial port.

```
int ReceiveByte (int port, int timeout);
```

Receive a character from a device connected via a serial port. Wait for response at most timeout milliseconds. Returns a negative value if no character has been received until timeout.

4.8 Miscellaneous Functions

4.8.1 Handling Macros and Macro Files

Executing Strings as CL Commands

```
void CL_command( string theCmdLine );
```

```
int CL_command_Err( string theCmdLine );
```

Interpret string as command language code and execute. While the first subroutine displays the error dialog on runtime errors, the second function returns 0 on successfully execution and another value on errors.

After a call to `int CL_command_Err` the error stack information from macro file `cl/Error.cl` can be used. An easy way is a call to macro `DisplayErrorStack()`, which lists the complete error stack in the Info window. Remember to clean the error stack after usage with function `ClearErrorStack()`. Otherwise the next error will be stored on top of the old error in the stack.

Note: Both executions are running context free in the same way as a macro, which is started from the **Execute Line** dialog. Those commands can not access local variables from the calling macro or private variables from the (local) macro file. To use those variables, include the values into the command string, or better call virtual functions as explained below and give the variable as parameters.

Virtual Functions: Virtual functions are called from string variables, which can be used the same way as in direct macro call:

```
string VirtualFunction = "MyFatherFunction";
VirtualFunction(theFirstParameter,theSecondParameter);
```

LoadMacroFile

```
void LoadMacroFile(string theFileName);
```

```
int LoadMacroFileErr(string theFileName);
```

Install a CL macro file with user written macros. While the first functions displays the error dialog on loading errors, the second function returns the error code and goes on with macro processing. After the macro file has been installed successful, the macros of that file can be called. Some returned error codes are listed in table 4.4.

Error code	Message
0	macro file loaded without errors
1	syntax error
3	unable to open file, e.g. file not found

Table 4.4: Error codes returned by subroutine LoadMacroFileErr.

It is only necessary to install a macro file **once per session**, even if it is edited, changed and saved. Depending on a setting in the **Global Options** the changed macro file is either loaded automatically or the user will get a message and has to choose if he wants to load the file.

If a macro file has already been loaded, another call to LoadMacroFile will not reload the file until its modification time has changed.

Autostart Macro: If there is a function with a special name in the loaded macro file, this macro will be called immediately. The name must be the same as the value of theFileName but in uppercase letters. All characters of theFileName which are not in "0"..."9" or "A"..."Z" will be changed to the underscore "_". The function gets no parameters and returns no value:

```
void THEFILENAME()
```

Example: When loading a macro file with name Camera 01-a.cl the autostart macro looks like this:

```
void CAMERA_01_A ()
{
    // do something
}
```

Note: The following macros should be used to load system macros in the **DaVis** directory or subdirectory CL. The first macro appends the string

".CL" to the filename and shows a message in the status line, the second macro only loads the macro file, if it has not been loaded successful before.

```
void LoadCLFile(string theFileName);
```

```
void CheckLoadCLFile(string theFileName);
```

UnloadMacroFile

```
void UnloadMacroFile(string theFileName);
```

Remove macro file from memory. All variables and macros, which are included in this file, will be removed!

StoreMacroFile

```
int StoreMacroFile(string theFileName);
```

Save the static variables of a loaded CL macro file. If `theFileName` is the empty string, all macro files will be saved.

Load / StoreSet

In **DaVis** all variables and macros are grouped by the following keyword:

```
#GROUP NameOfTheGroup
```

The name of the group can be used to store all static variables of this group into a `SET` file.

```
void StoreSet( string filename, string grouplist );
```

Store static variables to a `SET` file. The `grouplist` is a list of all group names which should be included. The single names must be divided by a '\n' character. The default value can be found as variable `CustomerGroups`:

```
CustomerGroups = "Software Modules\n
      Hardware (ports, addresses, etc.)"
```

```
string LoadSet( string filename );
```

Load parameter set (without buffer loading). This macro returns a list of all loaded groups and internally calls `LoadSetGroup`.

```
string LoadSetGroups( string theFilename, string theGroupList );
```

Load only the given group(s) of the set. The group list must be divided by '\n'. If theGroupList="", all groups will be loaded from the file. A list of all loaded groups is returned.

```
void LoadSetInfo( string filename );
```

Load only set info (without parameters or buffers). This macro calls LoadSetGroup.

ReadSetVariable

```
line ReadSetVariable( string filename, string theVariableToRead );
```

Read a variable from the setfile and return the value. The global variable theVariableToRead is not touched by this function!

If theVariableToRead is the empty string, a string list of all groups in the set file is returned.

Examples:

```
InfoText(ReadSetVariable("sets\\test.set","SetTime"));
```

```
float camicpar[1000];
camicpar = ReadSetVariable("calibration.set", "CamIcPar");
InfoText(""+camicpar[0]+" "+camicpar[1]+" "+camicpar[2]);
```

TestSetVariable

```
int TestSetVariable( string filename, string theVariableToTest );
```

Test if the variable exists in the setfile and return 0, if the variable is not stored in the set, or return the type: unknown (1), int (2), float (3), string (4), int array (12), float array (13 or string array (14). The type depends on the existing variable in **DaVis** and not on the stored value in the set file. That's the reason why the type can be unknown: The macro file which includes the variable is not loaded.

StoreSetVariableGroup

```
int StoreSetVariableGroup(string filename, string theClVar,
    string theClVarGroup, string theNewValue);
```

Store a new CL variable in a dataset with `theClVar` as valid CL variable, `theClVarGroup` as group of `theClVar` and `theNewValue` as new value of `theClVar`, e.g.

```
StoreSetVariable("d:/mydata.set", "MyClVariable1",
    "MyClGroup", "3");
```

Store (and overwrite) new groups in a dataset with `theClVar=""`, `theClVarGroup` as new group and `theNewValue=""`, e.g

```
StoreSetVariable("d:/mydata.set", "", "MyClGroup", "");
```

Assign a new value to an existing CL variable in a dataset with `theClVar` as valid CL variable, `theClVarGroup=""` and `theNewValue` as new value of `theClVar`, e.g.

```
StoreSetVariable("d:/mydata.set", "MyClVariable1", "", "3");
```

IsValidSymbol

```
int IsValidSymbol(string theSymbolName);
```

Returns the type of a symbol: 0=not existing, 1=macro function, 2=subroutine, -1=variable.

If `theSymbolName = " [groupname]"`, the return value is TRUE if the group includes static variables.

GetVariable(Its)Type

```
int GetVariableType(string theVariableName, int& theDimension);
```

```
int GetVariableItsType(line& theVariable, int& theDimension);
```

Both subroutines return the type (0=not existing, 1=int, 2=float, 3=string) and dimension (size of an array) of a variable. Function `GetVariableType` gets the name of the requested variable as parameter and therefore works for global variables only. Function `GetVariableItsType` gets the variable itself as parameter and can be used for local variables. This is especially

useful for line parameters, when the macro wants to get information about the line type. The size of a line parameter can be requested with function `sizeof(theVariable)`.

Examples with `dim` as existing integer variable:

- `GetVariableType("CamType",dim)` returns value 1 for the integer variable and `dim=6` for the size of the array.
- `GetVariableItsType(CamType,dim)` returns the same as above.
- `{ int i; GetVariableType("i",dim); }` returns 0, because `i` is a local variable.
- `{ int i; GetVariableItsType(i,dim); }` returns 1, because this function can request information about local variables.
- `void TestVar(line& parm)`
  `{ int dim; int type = GetVariableItsType(parm,dim); }`
  When called as `float arr[11]; TestVar(arr)`, the variables will be set to `type=2` and `dim=11`.

4.8.2 Program Version and Packages

When programming for different versions of **DaVis** the CL-code sometimes needs to execute different functions depending on the software version. Therefore some variable are set to version values during the startup:

`ProgramVersion` is a string like `"10.0.2"`. The release number is coded into integer variables `ProgramVersionBranchId` and `ProgramVersionChangesetId`. Both are rarely used in customized macros when special new features must be checked during runtime and the macro is used with several software versions.

The Linux version of **DaVis** includes letter `'L'` in variable `ProgramVersion` and can be checked by macro `IsLinuxVersion()`.

String variable `ProgramVersionDate` gives the release date.

The dongle defines the available packages. During the startup of **DaVis** the dongle is checked and the variables `SupportX`, e.g. `SupportPiv2D`, are set concerning to the dongle and the licence code of variable `DongleCode`. The `SupportX` variables are readonly in **DaVis** and can not be changed!

4.8.3 Calling Windows Programs

GetEnvironmentVariable

```
string GetEnvironmentVariable( string varname );
```

Get the value of an environment variable or of internal system information. All available environment variables are listed in Windows® when opening a command shell and then typing the command `set`. Some examples can be found in table 4.5.

NUMBER_OF_-PROCESSORS	Number of processers in the PC, supported since WindowsNT
HOMEPATH	Home folder of the Windows user.

Table 4.5: Examples for subroutine `GetEnvironmentVariable`.

ExecuteProgram

```
ExecuteProgram( string theFileName, int WaitForEnd );
```

Let the operating system (Windows) execute a file with extension EXE or PIF or BAT. A new process is started, that runs independently from **DaVis**. The waiting modes are listed in table 4.6 and defined as constants `EXECPROG_x` in **DaVis**.

WaitForEnd	
0	DaVis and the new process in the operating system keep on running parallel.
1 = EXECPROG_WAIT	DaVis waits until the independent process in the operating system is completed. Afterwards, new DaVis actions can be made.
2 = EXECPROG_-NO_SHOW	Minimize the window of the executed program.

Table 4.6: Waiting modes of subroutine `ExecuteProgram`.

If `theFileName` has the extension ".htm" or ".html" the preferred browser is called which shows `theFileName`. Note: `theFileName` has to be a relative path to the **DaVis** program path, e.g. `"help\\user.htm"` will load the

file "C:\DaVis\help\user.htm". The second parameter has no effect for browser-calls (**WaitForEnd**=0).

The default editor is opened for a file name with extension ".cl".

It is possible to execute a command line instruction by a call like the following, which creates a list of all PCs in a network domain:

```
ExecuteProgram("cmd.exe /c net view > "
    +ProgramPath+"NetComputer.txt",1);
```

This command opens a PDF file in the standard viewer:

```
ExecuteProgram ("cmd.exe /c start "+ ProgramPath+
    "Macros\Doku\nMultiCalc.pdf",0);
```

4.9 Fast User Functions via DLLs

A user macro with pixel access is running very slow. A loop about all pixels of a buffer should be programmed in a programming language like C or C++, compiled for fast execution as a Dynamic Link Library (DLL). **DaVis** is able to call functions of DLLs via subroutines `CallDll` and `CallDllEx`.

Three files can be found in in subdirectory `DemoDll` of the **DaVis** directory: `CL_DLL_Interface.h` and `CL_DLL_Interface.cpp` handle the connection to the **DaVis** software, so the programmer can execute a number of important **DaVis** functions from the own C-code. Both files must be included into own DLL projects, but should not be changed!

An small C-code `CL_DLL_Example.cpp` shows examples of DLL functions, which can be executed by **DaVis** with the corresponding calls in macro file `DemoDLL.CL`.

A project is included in the `DemoDLL` directory for the the Microsoft® Visual C++ compiler. The compilation should work immediately without problems.

4.9.1 Upgrading DaVis 8.x DLLs

Older customer DLLs from DaVis 8 or earlier versions should be compiled again using the new interface files. Some interface functions have been removed, others have been added during lifetime of DaVis 8.

Another reason for recompiling old DLLs is the change from 32 bit DaVis 7 or 8 to 64 bit DaVis 10. Old 32 bit DLLs are not running with 64 bit DaVis 10.

4.9.2 CallDll

```
int CallDll( string dll_name, string function_name,
line& parameter );
```

Call an external user-defined function in a Dynamic Link Library (DLL).

Writing own DLL-functions is useful for

- implementing fast image processing with 'C'-level speed
- writing native drivers, execute native windows functions
- do special action not supported by CL-functions

The DLL has to be in a certain format. See files in **DaVis** subdirectory `DemoDLL` as examples of how to write your own library.

4.9.3 CallDllEx

```
int CallDllEx( string dll_name, string function_name,
line& theIntPars, line& theFloatPars, line& theStringPars );
```

Call an external user-defined function in a Dynamic Link Library (DLL). Three arrays of types int, float and string can be given to the DLL function as parameters. The array is allowed to store results in this arrays, but the array size must be large enough.

A special function in the C-sources of the examples can be used to pass a string result to **DaVis**:

```
DllEx_SetStringParm( char** parsString, int setParI,
                     const char* setParString );
```

Give the string array parameter as first argument, followed by the item of the array to be replaced. The last value points to the new string. See example below.

4.9.4 UnloadDll

```
void UnloadDll(string theDllName);
```

This function can be used during the debugging stage of the DLL, in order to compile a modified DLL and reload it into **DaVis**. Otherwise one would have to close **DaVis** and restart it to be able to access the modified library.

If the function name is empty (""), subroutine `CallDll` unloads the library from **DaVis**.

4.9.5 How DaVis works with Dynamic Link Libraries

When the library is loaded the first time, a function

```
InitDll( void(*)() )
```

is automatically called. This function has as a parameter a function pointer, with which it is possible to access many low-level data structures and functions inside **DaVis**. This is necessary for manipulating e.g. buffer data. For more details see file `CL_DLL_Interface.cpp`.

At the end `InitDll()` calls a user function named

```
void MyInitDLLExtensions()
```

This function is included in the example file `CL_DLL_Example.cpp` and can be used to execute own initialization code during the first loading of the DLL.

Next the DLL-function `<function_name>` itself is executed.

It must be of a format

```
extern "C" int _export <function_name>
( int* parameter );
// (Borland C++ notation)
```

or for Visual C++

```
extern "C" int _export <function_name>
( float* parameter );
```

The parameters in the integer or float array can be used to transfer data to the DLL-function or to return values.

4.9.6 Example for DLL-function parameters

Function `AddTwoNumbers()` in `CL_DLL_Example.cpp` can be called by the following CL-code (see file `DEMODLL.CL`). This example gets two float numbers as parameters, adds both numbers and returns the sum in the third item of the float array. Additionally a text is written into the InfoWindow and a Message box appears on screen.

```
float pars[3] = { 1.2, 2.4, 0 };
CallDll("demodll.dll", "AddTwoNumbers", pars );
InfoText("Dll-function returns as a third
    parameter:  " + pars[2] );
```

If the DLL-function has an integer array for the parameters instead, you must set up an CL-integer array correspondingly.

```
extern "C" int EXPORT AddTwoNumbers( float* thePars )
{
    thePars[2] = thePars[0] + thePars[1];
    char str[80];
    sprintf( str,
        "InfoText(\n"parameter:  %g + %g = %g\n")",
        thePars[0], thePars[1], thePars[2] );
    ExecuteCommand( str ); // display in info window
    sprintf( str, "parameter:  %g + %g = %g",
        thePars[0], thePars[1], thePars[2] );
    return Message( str, 0 ); // display as message
}
```

4.9.7 Passing string parameters to a DLL-function with CallDll

In the old `CallDll` function it is only possible to pass float or int arrays as parameters. String parameters have to be stored in global CL variables, before they can be read out by the DLL-function.

A CL definition `string MyStringVariable;` is needed in a CL macro file. Then the DLL function can read out the value of this string variable by a call like `const char* myString GetString("MyStringVariable",0);`.

4.9.8 Using CallDllEx with string value as result

```
extern "C" int EXPORT ExampleInterface72(
            int* parsInt, float* parsFloat, char** parsString )
{
    parsInt[1] = parsInt[0]+1;
    parsFloat[1] = parsFloat[0]+3.147f;
    // for string results we have to free the old string
    // and create memory for the new string
    char text[256];
    sprintf(text,"%s and more",parsString[0]);
    DllEx_SetStringParm( parsString, 1, text );
    return 0;
}
```

4.9.9 Example for a DLL-function with buffer access

The most important (and most critical) function for the access to buffer raw data during a DLL execution is the function `GetBufferRowPtr`. This function executes some code in the `DaVis.exe` and returns a void-pointer to a line of Word- or float-values, which depends on the buffer type. Use function `IsFloat()` to determine the type.

There are two critical sections in this function: At first you have to program the correct type cast. In the example below the buffer type is known as Word, so the pointer returned by `GetBufferRowPtr` is casted to a Word pointer. A cast to a float pointer would cause wrong values during a read access to the array, and it could cause a crash of DaVis during a write access because of the different item sizes of 2 byte for one Word item and 4 byte for one float item. At second a writing or reading access with negative indexes or with indexes above the line length could crash DaVis.

Note: When using multi frame buffers in a DLL the function `GetVolBufRowPtr` must be used instead of `GetBufferRowPtr`.

```
extern "C" int EXPORT SetBufferToValue( int* pars )
{
    int theBufferNumber = pars[0],
    theNx = pars[1],
    theNy = pars[2];
```

```
    Word theValue = (Word)pars[3];
    // create buffer of type Word with theNX columns
    // and theNY rows
    if ( SetBufferSize( theBufferNumber, theNx, theNy, 0 ) )
        return -1; // error, maybe out of memory
    int row, col;
    for ( row = 0; row < theNy; row++ )
    {
        Word* rowPtr =
        (Word*)GetBufferRowPtr( theBufferNumber, row );
        for ( col = 0; col < theNx; col++ )
            rowPtr[col] = theValue;
    }
    char str[80];
    sprintf( str,
        "BufferName(%d,\n"value = %d\n")",
            theBufferNumber, theValue );
    ExecuteCommand( str ); // set buffer name
    Show( theBufferNumber );
    return 0;
}
```

4.9.10 Breaking a long time calculation

When programming a time intensive function in a DLL, it is recommended to break on the user's request to keep **DaVis** responsively.

Therefore the DLL function `int ProcessEvents()` should be called during the execution to ask **DaVis**, if the user wants to stop the execution. This function returns TRUE if the DLL has to return. Additionally the function gives **DaVis** the possibility to process Windows events, e.g. repainting of images.

5 Image and Vector Buffers

5.1 About Buffers

DaVis stores image and vector data in so called **Buffers**. In the simplest cases a buffer includes a two dimensional array of **pixel** with raw intensities (counts). Most subroutines of **DaVis** are referring a pixel with two coordinates x and y, both with a range from 0 to the width or height minus 1. In CL an easy access is possible via function `Pix[theBufferNumber,x,y]`. Additionally a buffer stores **attributes** like scaling information, which are used during the image display, and acquisition and processing information.

5.1.1 About Multi-Frame Buffers

Because of the multi camera and multi exposure support in **DaVis** and because of 3D systems the buffers must be able to include more than a 2D plane of pixels.

So the buffers have four dimension internally: width, height, depth (the z-direction) and frames.

In a 3D buffer every point is named as **voxel** instead of **pixel** for 2D buffers.

For example a double exposure camera creates a buffer with a free width and height, a depth of 1 and two frames: one frame per exposure. A system with two double exposure cameras creates a buffer with four frames, the first two frames of the first camera and frames three and four of the second camera. The frames again are numbered from 0 to the frame size minus 1.

5.1.2 About Typed Scalars

The fifth dimension of a buffer is the **Typed Scalar** dimension: Every pixel/voxel/vector can store more than one intensity. E.g. a vector field

gets additional information about peak ratio and height information. Those values are stored as **Typed Scalars** for each position in the buffer.

There are a number of access functions to work with **Typed Scalars**, all following the naming style `Buffer_TS_name`.

5.1.3 About Buffer Format

The type of a buffer and its number of components are defined by the **buffer format**: Examples for formats are the simple image of float or word intensities, or a vector field of 2D vectors or of 3D vectors, or a RGB colored buffer. Vector components are always stored in floating point values.

5.1.4 About Buffer Mask

All buffer types can add a mask as additional information for each pixel/voxel/vector. The image display and the statistics subroutines are using valid values only. When moving the mouse cursor above a masked pixel, the real intensity value is displayed in the tool tip at the mouse cursor.

All unmasked pixel e.g. will be painted in color 0 by the image view. When moving the mouse cursor above a masked pixel, the real intensity value is displayed in the status information. This real value can be accessed by the `Pix[bufnum,x,y]` operator and by all buffer operations, but there is no guarantee that buffer operations leave the mask untouched.

There are a number of access functions to work with the **Mask**, most following the naming style `Buffer_Mask_name`.

5.1.5 About Buffer Attributes

DaVis supports special attributes for every buffer, which can be saved together with the raw buffer data in the file formats IM7, IMG, IMX, VC7 and VEC and in the stream format. Each buffer can store additional named attributes (strings and arrays of the types float, int or word). These are used for different purpose like the `Overlay` definition or (analog) Device Data information. A complete table of predefined supported buffer attributes is not published, but descriptions about the existing attributes are available in the attribute dialog of the view information.

All attributes are copied, too, when copying a buffer into another buffer. E.g. the command `B7=B2` copies the raw data, the scaling information, attributes, mask, and scalar fields from buffer 2 into buffer 7.

Usually the frame operations mentioned above copy **frame attributes** from source to destination. A frame attribute is defined by a name with preceding frame index like `CameraName0` for the first frame and `CameraName1` for the second frame. If a frame operation takes the first frame as source and second the third frame as destination, then e.g. the value of `CameraName0` in the source buffer can be found in attribute `CameraName2` in the destination.

Attributes without preceding index are called **global attributes**. A short list of exceptions includes attribute names, which index is NOT the frame index but a counter for the attributes. Those attributes are global attributes, and the most important exception is the device data class `DevData`.

5.2 Image Buffers

The simplest functions to create a buffer are `CreateImageBuffer`, `CreateWordBuffer`, `CreateFloatBuffer` and `CreateVectorBuffer`.

If a buffer exists, the function `ResizeBuffer` can be called to change the size of the buffer. With subroutine `Show` the buffer is displayed on screen.

A number of functions are implemented to ask a buffer for its size, e.g `GetBufferNX` for the width, or buffer format, e.g. `IsFloat` and `IsVector`. Other functions give easy access to frames or planes of a buffer, e.g. copy a single frame of a multi frame buffer into a new (single frame) buffer.

CreateImageBuffer

```
void CreateImageBuffer( int buf, int nx, int ny, int nf, int isFloat );
```

Create an image buffer in buffer number `buf` with the defined size of width (`nx`), height (`ny`) and number of frames (`nf`). The pixel values are either stored as word values (`isFloat=FALSE`) or as float values (`isFloat=TRUE`). This macro is a shortcut for subroutine `SetVolumeBufferSize`.

CreateWordBuffer

```
void CreateWordBuffer( int buf, int nx, int ny );
```

Create an image buffer of WORD data type in buffer number `buf` with the defined size.

CreateFloatBuffer

```
void CreateFloatBuffer( int buf, int nx, int ny );
```

Create an image buffer of FLOAT data type in buffer number `buf` with the defined size.

ResizeBuffer

```
void ResizeBuffer( int theBuffer, int theNX, int theNY, int theNZ,
int theNF )
```

Subroutine `SetVolumeBufferSize` is changing the size of a buffer without maintaining values of every pixel. This macro on the other hand holds every pixel at its 4D-position. This macro is working for vector buffers, too.

SetBufferSize

```
void SetBufferSize (int p_hBuffer, int width, int height, int isfloat);
```

This function has been a subroutine up to **DaVis** 6 but is retired now and implemented as macro. It should not be used any longer. Better call `CreateImageBuffer` and likewise functions.

Warning: If `SetBufferSize` is executed with `width = 0` or with `height = 0` the buffer will be destroyed (together with comments, scales, etc.).

Note: Setting the buffer size with this macro always changes the image format to type `BUFFER_FORMAT_IMAGE` !

GetBufferSize

```
void GetBufferSize (int p_hBuffer, int& width, int& height, int& isfloat);
```

Get the size (width and height) and the type of a buffer. For other than single frame single plane buffers, the `height` is a product of frame-height, frame-depth and frame-size.

For WORD buffers the parameter **isfloat=0** and for FLOAT buffers **isfloat=1**.

GetVolumeBufferSize

```
void GetVolumeBufferSize( int buf, int& nx, int& ny, int& nz, int& nf )
```

Return the size of the four volume buffer dimensions.

GetBufferNX/Y/Z/F/Frames

```
int GetBufferNX( int theBuffer );
int GetBufferNY( int theBuffer );
int GetBufferNZ( int theBuffer );
int GetBufferNF( int theBuffer );
```

This subroutines are declared in macro file `CL/System/Utilities/Buffer.cl`. They return the width (X), height (Y) or depth (Z) or frame number (F) of a buffer.

IsEmpty

```
int IsEmpty (int p_hBuffer);
```

Return `TRUE` if the buffer is empty i.e. its size is zero.

IsFloat

```
int IsFloat (int p_hBuffer);
```

Return 1 if the buffer is a FLOAT buffer, 2 for a DOUBLE buffer and 0 for a WORD or RGB buffer.

Show

```
void Show (int p_hBuffer);
```

Show the contents of a buffer in an image window of an automatically defined view type: simple image view or simple vector view. If the buffer has a small height, it is shown as a profile. This functions does not open a new window if a view of any type is open for the buffer.

5.3 Vector Buffers

CreateVectorBuffer

```
void CreateVectorBuffer( int buf, int nx, int ny, int nf,
    int theVectorType );
```

Create a vector buffer in buffer number `buf` with the defined size. The vector type (2D, 3D, or extended formats) is given by value `theVectorType`. Table 5.1 about the buffer formats lists all available vector types. This macro is a shortcut for subroutine `SetVolumeBufferSize`.

SetVectorBufferSize

```
void SetVectorBufferSize( int buffer, int nx, int ny, int format,
    int grid_spacing );
```

Create vector buffer with specified format (see table 5.1) and vector grid-spacing and with the given number of vectors in x- and y-direction.

```
void SetVolumeVectorBufferSize( int buffer, int nx, int ny, int nz,
int nframes, int format, int grid_spacing )
```

Same as above but with additional depth (=plane) and frame numbers.

GetVectorBufferSize

```
void GetVectorBufferSize( int buffer, int& nx, int& ny, int& format,
    int& grid_spacing );
```

Retrieve number of vectors, format and grid-spacing.

IsVector / Is3DVector / IsSimpleVector / IsPeakVector

```
int IsVector (int p_hBuffer);
```

Return 1 if the buffer is a vector buffer.

```
int Is3DVector (int p_hBuffer);
```

Return 1 if the buffer is a vector buffer with three components.

```
int IsSimpleVector (int p_hBuffer);
```

Return 1 if the buffer is a vector buffer without extended information.

```
int IsPeakVector (int p_hBuffer);
```

Return 1 if the buffer is a vector buffer with peak ratio information.

5.4 Advanced Buffer Access

SetVolumeBufferSize

```
void SetVolumeBufferSize (int p_hBuffer, int width, int height,
int depth, int frames, int bufferformat, int isfloat );
```

Create a volume buffer with full planes (not sparse). See table 5.1 for a list of all defined `bufferformats`. This subroutine should not be used to resize an existing buffer, because the data may loose its frame and component order. Call `ResizeBuffer` instead to hold every pixel at its correct position.

CreateVolumeBuffer

```
void CreateVolumeBuffer( int p_hBuffer, int width, int height,
    int depth, int frames, int bufferformat, int isfloat,
    int nComponents_RETIRED );
```

Create a volume buffer with full planes (not sparse). The last parameter about additional components has been retired in version 8.1. If such components are used, they must be replaced by typed scalars.

GetVolumeBufferInfo

```
void GetVolumeBufferInfo( int p_hBuffer, int& width, int& height,
    int& depth, int& frames, int& issparse_RETIRED,
    int& bufferformat, int& nComponents_RETIRED,
    int& nVectors, int& nScalars_RETIRED );
```

Get the information of a volume buffer, return `depth=1` and `frames=1` for normal buffers. Use `IsFloat()` to request information about the data type. The `subtype` is the same as from subroutine `GetImageFormat()`. Value `nVectors` gives the number of vector components.

GetImageFormat

```
int GetImageFormat (int p_hBuffer);
```

Get the format of the buffer. See table 5.1 for a detailed format list.

A simple **type conversion** can be executed with the following commands:

```
B[out] = (word) B[in]; // result is a word buffer
B[out] = (float) B[in]; // result is a float buffer
```

BUFFER_FORMAT_x	**Buffer Type**
IMAGE	Standard image format f(x,y)
WORD	Standard word image format, for creation only
FLOAT	Standard float image format, for creation only
DOUBLE	Standard double image format, for creation only.
FLOAT_VALID	Retired float image format.
FLOAT_VALID_BOOL	Retired float image format.
VECTOR_2D_ EXTENDED	Extended (PIV) vector format Vx(x,y) and Vy(x,y) = two-dimensional 2D-vector field, additional information for four stored vectores at each position and for the selection of the used vector choice
VECTOR_2D	Simple vector format Vx(x,y) and Vy(x,y) = two-dimensional 2D-vector field.

continued on next page. . .

BUFFER_FORMAT_x	Buffer Type
VECTOR_3D	Same as BUFFER_FORMAT_VECTOR_2D with extra Vz component.
MEMPACKWORD	Byte buffer created by some 8-bit-cameras for faster acquisition rates. The buffer will be enlarged to 16 bit at the first writing access, so it is impossible to create and fill this buffer type by CL.
RGB32	32 bit RGB buffer to be used with `SetColorPixel`, `GetColorPixel`, `RGBFilter` and `RGBCombine`, see section below.

Table 5.1: Buffer formats to be used by `SetVolumeBufferSize` and `GetImageFormat`. The names of the constant values must be read like `BUFFER_FORMAT_IMAGE`.

5.5 Mask Information

Check Existing Mask

```
int Buffer_HasMask(int p_hBuffer);
```

Return `TRUE` if the buffer includes a mask. If there is no mask, the function will return (`FALSE`) and function `IsPixelMasked` will return `TRUE` always.

Pixel Access of the Mask

```
void Buffer_Mask_SetPixel(int p_hBuffer, int theX, int theY, int theZ,
int theF,
    int bMask);

int Buffer_Mask_GetPixel(int p_hBuffer, int theX, int theY, int theZ,
int theF);
```

The mask flag can be set or requested for a single pixel by this two subroutines. For faster access please use the following function:

```
void Buffer_MaskToImage( int p_hBufferWithMask, int p_hImageBuffer
);

void Buffer_ImageToMask( int p_hImage, int p_hDest );
```

Copy mask information between buffers. Use the mask in an image buffer for easy manipulation.

5.6 RGB Color Buffers

To create RGB color buffers please call subroutine `CreateVolumeBuffer` with parameter `bufferformat = BUFFER_FORMAT_RGB`.

A RGB buffer can easily be converted into a grayscale buffer with the following command, which uses the global variable `DefaultBMPvisual` to define the conversion mode (average all components=0, use graying function=1):

```
B[p_hBuffer] = (word)B[p_hBuffer]
```

IsRGB

```
int IsRGB(int p_hBuffer);
```

Return 1 is the buffer is a 32 bit RGB buffer, else 0 for a grayscale buffer.

Set / Buffer_RGB_GetPixel

```
Buffer_RGB_GetPixel (p_hBuffer, int mode, int x, int y,
    float& par1, float& par2, float& par3);
```

```
void SetColorPixel (int p_hBuffer, int mode, int x, int y,
    float par1, float par2, float par3);
```

Get or set color information of a pixel in a color buffer. See subroutine `ConvertRGBImage` on page 117 for likewise function.

RGBFilter

```
void RGBFilter( int bin, int bout, float factorRed, float factorGreen,
    float factorBlue, int normalizeToOne );
```

Create a float buffer with
*pixel = red*factorRed + green*factorGreen + blue*factorBlue*.

This subroutine can be used to filter colors from the RGB buffer, e.g. `RGBFilter(1,2,0,1,1,1)` creates a buffer with the yellow color component.

RGBCombine

```
void RGBCombine (int binRed, int binGreen, int binBlue, int bout);
```

Create a RGB buffer from three “color component buffers” .

RGBTupelMul

```
void RGBTupelMul (int bin, int bout, float factorRed, float factorGreen,
float factorBlue);
```

Multiply every component by the given factor. The destination range of every pixel component is 0..255.

RGBTupelRange

```
void RGBTupelRange (int bin, int bout, int isMax);
```

Get minimum (`isMax=0`) or maximum (`isMax=1`) intensity of the components of every pixel:

```
Pix[dest,x,y] = min{R(x,y),G(x,y),B(x,y)}
```

5.7 Buffer Frame Handling

Programming macros for the usage of buffer frames need additional commands compared to simple working with the complete buffer: The programmer needs information about the frame size of a buffer, about the frames to work with or to work with all frames. A lot of functions and even subroutines are not able to work with single frames, they are implemented to work with complete buffers only. Therefore the programmer has to extract single frames into temporary buffers, then call the restricted functions, and at the end copy the results into multi frame buffers.

A number of functions to extract single frames of a buffer or to combine some buffers or frames into a multi frame buffer are implemented in macro file `CL/System/Utilities/Buffer.cl`.

The subroutines, which are able to work with single frames, need more than one parameter to define the input source or the output destination: they need frame information.

5.7.1 General Parameters for Frame Operations

For each input buffer and for each output buffer all standard functions need one parameter only: the buffer number.

The frame operation functions need at least the frame index, starting with index 0 and going up to $N - 1$ for a buffer with N frames. The frame parameter can be set to value -1, if the programmer wants a function to work with all frames.

Some functions are even designed to work with single planes. They need a third parameter to define the plane number. Again, value -1 can be used to let the function work for all planes.

For both frames and planes the output parameter can be set to a non-negative value only. Value -1 in the meaning of "all frames/planes" is used as value 0, so the result of the first input frame/plane is stored in the first frame/plane of the destination buffer.

If the destination buffer is empty, it will be created with the needed size and type. The destination will always be created when the frame parameter on the input size is -1, because in this case something happens to all frames, and all results must be stored in the destination.

If the destination buffer is not empty when calling a frame or plane operation, the type of the destination will not be changed and the buffer will not be resized! If the operations creates a larger output or different type, the result is fitted into the destination buffer: cutting off the outer areas of the larger result or e.g. converting float values into a word destination buffer.

When working with a single plane or frame, the result can be stored in the source buffer, but needs another frame.

Example for a Frame Operation

As a general layout style for the parameters of frame (and plane) operations the parameter list starts with the input buffer, followed by the input frame, then optional buffer and frame number for a second input buffer,

then followed by buffer and frame number of the output destination. At the end all individual parameters for the operation are given, e.g. a mode.

```
FrameOperation( int theInputBuffer, int theInputFrame,
    int theOutputBuffer, int theOutputFrame, int theMode );
```

Example for a Plane Operation

The general layout style for plane operations inserts the plane index between buffer number and frame index as second parameter:

```
PlaneOperation( int theInputBuffer, int theInputPlane, int theInputFrame,
    int theOutputBuffer, int theOutputPlane, int theOutputFrame,
    int theMode );
```

The number of plane functions is actual very small. The main function for planes of buffers is `CopyBufferPlane`.

5.7.2 Extract and Combine Buffer Frames

The number of frames in buffer can be requested by subroutine `GetBufferNF`.

CopyFrameToBuffer

```
void CopyFrameToBuffer( int theSourceBuffer, int theSourceFrame,
    int theDestBuffer )
```

Copy frame number `theSourceFrame` (0,..,n-1) of buffer `theSourceBuffer` into buffer `theDestBuffer`.

CopyBufferToFrame

```
void CopyBufferToFrame( int theSourceBuffer, int theDestBuffer,
    int theDestFrame )
```

Copy buffer `theSourceBuffer` into a frame (`theDestFrame=0,...,n-1`) of `theDestBuffer`. Buffer `theDestBuffer` may be resized to hold a larger number of frames. If the frame size of the source buffer is larger than the destination frame size, the upper left rectangle is copied out of the source buffer. For smaller frames the outer region is filled with 0-values.

AppendFrameToBuffer

```
void AppendFrameToBuffer( int theSourceBuffer, int theDestBuffer )
```

Append buffer `theSourceBuffer` as a new frame to buffer `theDestBuffer`, which may be empty before this operation. The destination buffer will be resized by this function. Note the same restrictions for different frame sizes as in macro `CopyBufferToFrame`.

CopyBufferPlane

```
void CopyBufferPlane( int inputbuf, int inZ, int inF,
    int outputbuf, int outZ, int outF );
```

Copy a plane of `inputbuf` into `outputbuf`. This subroutine is internally used by the above described macros. The complete plane data is copied, including the plane- and frame-depending buffer attributes.

If `inZ < 0`, the complete frame is copied, otherwise the defined plane.

Value `outZ` defines the first output plane if `inZ < 0`.

Example:

- Copy frame 2 of input buffer into frame 1 of destination:
  `inZ=-1, inF=2 →outZ=0, outF=1`
- Copy all planes of frame 2 into the planes 3,4,5,... of frame 1:
  `inZ=-1, inF=2 →outZ=3, outF=1`
- Copy plane 1 of frame 5 to plane 2 of frame 0:
  `inZ=1, inF=5 →outZ=2, outF=0`

5.8 Buffer Attribute Access

GetBufferName / BufferName

```
string GetBufferName(int p_hBuffer);
```

```
void BufferName(int p_hBuffer, string NewName);
```

Get or set the name (title) of a buffer.

Get / SetVectorGrid

```
int GetVectorGrid (int p_hBuffer);
```

```
void SetVectorGrid (int p_hBuffer, int gridsize);
```

Get or set the grid size of a vector buffer.

Get / SetComment

```
string GetComment ( int p_hBuffer);
```

```
void SetComment ( int p_hBuffer, string comment);
```

Get (or set) the comment string (max. 80 characters) of a buffer. Since the comment is stored in two lines, use \n to specify the separation of the lines.

GetBufferAttributeType

```
int GetBufferAttributeType ( int p_hBuffer, string attr_name);
```

Return 0 if the attribute with name `attr_name` is not existing, 1 for a string type attribute and 2 for a float-array, 3 for a integer-array (32 bit) and 4 for Word arrays (16 bit).

Get / SetBufferAttribute

```
string GetBufferAttribute( int p_hBuffer, string attr_name);
```

```
string SetBufferAttribute( int p_hBuffer, string attr_name, string value );
```

Get or set a string type attribute of a buffer. Use the empty string for parameter `attr_name` to request a list of all stored attributes.

Get / SetBufferAttributeArray

```
line GetBufferAttributeArray (int p_hBuffer, string attr_name );
```

```
void SetBufferAttributeArray (int p_hBuffer, string attr_name, line& theArray);
```

Get or set an array-attribute of a buffer. Example:

```
int array[5] = {5,4,1,2,3};
SetBufferAttributeArray(1,"testInt",array);

int array[10];
array = GetBufferAttributeArray(Buffer,''testInt);
// copy into row 10 of buffer 1, which type can be
// float or Word:
R[1,10] = GetBufferAttributeArray(Buffer,''testInt'');
```

DeleteBufferAttribute

```
void DeleteBufferAttribute (int p_hBuffer, string attr_name);
```

Delete an attribute of the buffer. All attributes can be deleted with `attr_name=""`. All attributes with the same prefix are deleted when give `attr_name="prefix*"`.

CopyBufferAttributes

```
void CopyBufferAttributes( int srcBuf, int srcFrame, int destBuf,
    int destFrame );
```

Append all attributes from `srcBuf` to `destBuf`. If `srcFrame>=0` only the frame-depending attributes are copied to `destFrame`, but the frame indizes of the attribute names will be changed from `srcFrame` to `destFrame`. For example the attribute `FrameName2` includes the name of frame number two. When this frame is copied to a single frame buffer, this subroutine can be used to copy this attribute value into attribute `FrameName0` of the single frame buffer.

Use macro `TransferAllAttributes` to copy the comment and the scaling information together with the attributes.

GetInterFrameTime

```
int GetInterFrameTime( int buffer_a, int frame_a,
    int buffer_b, int frame_b, float& dt );
```

Calculates frame time (differences) for two buffers, the frame numbers for frame_a and frame_b are zero based. The return value is 0 for no error or

a bitwise or'ed error code (see table 3.7 on page 40 about bit operations): attribute AcqTimeSeries was not found (1), attribute AcqTime was not found (2), frame out of range (4), attribute AcqTimeSeries has only one exposure (8), attribute AcqTime has only one exposure (16).

If frame_a = frame_b for the same buffer the autoorrelation dt of the two exposure in this buffer is returned.

If buffer_b = -1 and frame_b = -1 the absolute time for the first laser shot is used.

Calculations with

$$dt(A) = (AcqTimeSeriesA + AcqTimeA)$$

are dangerous due to the limited precision of float numbers and

$$AcqTimeA == AcqTimeSeriesA + AcqTimeA$$

Often $AcqTimeA << AcqTimeSeriesA$ calculating the difference $dt(A) - dt(B) = 0$ for two frames of one buffer. Better calculated dt directly as follows:

$$dt(A - B) = (AcqTimeSeriesA - AcqTimeSeriesB) + (AcqTimeA - AcqTimeB)$$

Here no error occurs as the (usually) huge numbers AcqTimeSeriesA and AcqTimeSeriesB cancel to 0 first, before adding small differences of AcqTimeA and AcqTimeB.

5.8.1 Buffer Attribute Macros

The following macro functions can be used for easier buffer attribute handling.

Get / SetBufferAttributeScale

```
int GetBufferAttributeScale( int theBuffer, string theScaleName,
    float& aa, float& bb, string& unit, string& label);

void SetBufferAttributeScale( int theBuffer, string theScaleName,
    float aa, float bb, string unit, string label);
```

The format of scaling information stores in buffer attributes is a string like "`aa\nbb\nunit\nlabel`" . This attribute type is used to describe scales of analog traces.

GetBufferAttributeValue

```
float GetBufferAttributeValue( int theBuffer, string theAttribute,
    int theElement )
```

Return a single value of a buffer array attribute. For single value attributes (like string attributes) the value will be returned if `theElement` is 0.

TransferAllAttributes

```
void TransferAllAttributes( int fromBuffer, int toBuffer )
```

This macro copies all attributes, scaling information and the comment from one buffer to another.

5.8.2 Frame Buffer Attribute Macros

When a buffer has been acquired via a multi camera readout item and more than one camera has been used, all the camera frames are combined in a single buffer with one frame for every camera image (e.g. double exposure cameras store both images in two frames of the buffer). It is possible to readout different cameras with different image sizes. In this case, the final buffer has a frame size of the maximum width and maximum height of every single images. It is possible to receive the original images from this large buffer later with the help of the following macros and the fact, that the “real” original image size is stored in buffer attribute `RealFrameSizeX` for every frame.

IsProcessedBuffer

```
int IsProcessedBuffer( int theBuffer, int theFrame, int theFlag )
```

Returns TRUE if all processing steps defined by `theFlag` have been executed on this buffer frame. Some flags are defined as constants in file `Constants.cl` and can be or-ed (in fact this are bit-flags): `FRAMEPROC_BACKGROUNDSUB`, `FRAMEPROC_ROTATION` and `FRAMEPROC_CORRECTION`. The flags are stored in buffer attribute `FrameProcessingX` with X as the frame number.

SetProcessedBuffer

```
void SetProcessedBuffer( int theBuffer, int theFrame, int theNewFlag );
```

Add a `FRAMEPROC_x` flag to a buffer frame and change the value of buffer attribute `FrameProcessingX`.

5.9 Buffer Attributes: Scaling

Scales map the pixel positions (width, height, depth and frame) and the pixel intensity from the internal (pixel scale) to a real world scale. Therefore the linear equation is used: `scale = a * x + b`

Every buffer has a general scale, which is used by default for every frame. For every frame an optional frame scale can be defined.

5.9.1 Accessing the General Scales

```
void GetXScale( int p_hBuffer, float& aa, float& bb, string& units,
    string& label);
```

```
void SetXScale( same as above );
```

Get (or set) the scaling information of the X, Y, Z, Intensity or Frame dimension of a buffer. Please change the character of the dimension name in the above named subroutine,.

This subroutines are working on the default scales of a buffer. This scaling information are stored in the first buffer frame. When a frame scale is missing, then the default scale is used for image display and for processing

Use the following macro to copy all scales from a buffer to another buffer. The values of `source_scale` and `dest_scale` are equal to `p_hBuffer` above.

```
TransferScales( int source_scale, int dest_buffer )
```

5.9.2 Accessing the Frame Scale

For the dimensions width (X), height (Y), depth (Z) and for the intensity (I) a different scale can be used for each frame of the buffer. The syntax of

the functions is equal for all dimension, please change the dimension name character only.

```
void SetFrameScaleX( int theBuffer, int theFrame,
    float aa, float bb, string unit, string label )

void GetFrameScaleX( int theBuffer, int theFrame,
    float& aa, float& bb, string& unit, string& label )
```

5.10 Buffer Attributes: Overlays

Overlays are user defined painting structures (lines, polygons, ellipses, texts), which are connected to buffers or cameras and automatically painted above the 2D-image display of a buffer. The overlay display can be enabled or disabled in the **Display Properties** dialog. This overlays can be defined in the **Overlay** dialog, which is available as processing function, or by some macros.

Overlays are connected to a buffer as buffer attributes, and of cause they are stored together with the buffer raw data in file types `IM7`, `IMG`, `IMX`, `VEC` and `VC7`.

The names of this attributes are `Overlay` for a standard overlay and `Overlay0`, `Overlay1`, `Overlay2` and so on for frame overlays. In multi frame buffers the overlay of a single frame display is the sum of the standard overlay (displayed in all frames) and the overlay with the same postfix number as the frame number itself.

For profile display the special overlay attribute `ProfileOverlay` is used the same way as the standard overlay for images.

The overlay attributes use strings to store the complete overlay definition in a simple list syntax. This syntax is not described in this manual, because the easier way to create an overlay in own macro code is the usage of functions in file `System/Display/Overlay.CL` (see below).

More special overlays handle the automatically display of buffer attributes as overlays. In this case the value of a buffer attribute is painted into the 2D image at a defined position with a defined color and size. Again some macros can be used to define this display, and the **Display Properties** dialog includes a card for easy definition by mouse clicks. The names of this attributes is a combination of a constant prefix and the name of the

displayed attribute: `OverlayAttr_[AttrName]`, e.g. for the name of the first camera `OverlayAttr_CameraName0`. Again for profile views another naming convention is used: `ProfileOverlayAttr_[AttrName]`.

5.10.1 Macros for overlay definition

The `OvGAdd`-macros need as parameters the number of a buffer (`bin`), the number of the frame (or -1 for the standard overlay), and an unique name for the overlay item. The `pencolor` if a RGB color value. The `linewidth` must be given in pixel. The coordinates, one or more `x` and `y` parameters, must be given in pixel coordinates, but can be floating point values. All following parameters depend on the item type.

The overlay strings itself can be created with the `OvGCreate`-macros, which use the same parameter syntax as the `OvGAdd`-macros. The `OvGCreate`-macros don't need parameters for buffer and frame number, but they return the definition string for the object. Those strings (objects) can be added to create an overlay definition with different objects.

Add a Line

```
void OvGAddLine (int bin, int frame, string name, int pencolor,
int linewidth, float x0, float y0, float x1, float y1, int options )
```

Add a line object to a buffer overlay. The coordinates are the starting point and the ending point. The `options` define the display of the line end: no arrows (0), start arrow (1), end arrow (2), start amd end arrow (3).

Add a Rectangle

```
void OvGAddRect (int bin, int frame, string name, int pencolor,
int linewidth, float x0, float y0, float x1, float y1)
```

Add a rectangle object to a buffer overlay. The coordinates define the upper left and lower right point of the rectangle.

Add a Circle or Ellipse

```
void OvGAddEllipse (int bin, int frame, string name, int pencolor,
int linewidth, float x0, float y0, float rx, float ry)

void OvGAddRotEllipse (int bin, int frame, string name, int pencolor,
int linewidth, float x0, float y0, float rx, float ry, float angle)
```

Add an ellipse object to a buffer overlay. The coordinates define the center position, while `rx` and `ry` give the radius in both directions. The second function gives an angle in degree for clockwise rotation of the ellipse.

Add a Text

```
void OvGAddText (int bin, int frame, string name, int pencolor, int size,
float x0, float y0, string textStr );

void OvGAddTextFont (int bin, int frame, string name, int pencolor,
    int size, float x0, float y0, string textStr, int fillcolor,
    string font, string style );
```

Add a text (parameter `textStr`) object to a buffer overlay. The position is the lower left corner of the text rectangle. The `size` gives the text height in pixel. The `pencolor` defines the color of the text, while the `fillcolor` defines the color of a filled rectangular background. The `font` must be one of the letters T(imes), H(elvetica), C(ourier), F(ixed) or S(ystem). The `style` can be one of the letters N(ormal), B(old), I(talic) and U(underline). The first macro always creates a normal Times font.

Add a Polygon

```
void OvGAddPolygon (int bin, int frame, string name, int pencolor,
int theLineWidth, line points, int theFillColor, int theMode )
```

Add a polygon object to a buffer overlay. The array `points` is a list of coordinates for the polygon in format format `[x0 y0 x1 y1 x2 y2 ...]`. The object can be filled with a RGB color `theFillColor`, which has to be set to value -1 for no filling.

Read and Write Overlay File

```
void ReadOverlayFile (string theFileName, int theBuffer,
int theFrame);
```

```
void WriteOverlayFile (string theFileName, int theBuffer,
int theFrame);
```

A text file will be read and used as overlay or created from a overlay definition. Please use file extension `OVG`. Both functions are addressing either the frame dependent overlay or with `theFrame=-1` the general overlay. With `theFrame=-2` all overlays will be stored into a multi file, where the frames are divided by `@FRAME=<frame>@` lines.

Move Overlays

```
string MoveOverlayString (string theOverlay,
float theOffsetX, float theOffsetY);
```

Move a string defined overlay by a given offset and return the moved overlay.

Rotate Overlays

```
string RotateOverlayString (string theOverlay,
int theCenterX, int theCenterY, float theAngle);
```

Rotate a string defined overlay around the given pixel position by an angle (in degree, counterclockwise). The angle of a text object will not be changed. A rectangle object will be transfered into a polygon object. Returns the rotated overlay definition.

Delete Objects

```
void OvGDeleteItems (int buf, int frame, string prefix)
```

Delete all overlay items which name starts with the given prefix.

The following subroutines are used internaly by the `OvG`-macros:

```
string OvG_GetNextValue (string listStr, string idStr, int apos)
```

Get the value of identifier idStr from the string defined overlay graphics list, starting at character position apos. If idStr is empty, then return the next value.

```
string OvG_DeleteItems (string listStr, string prefix)
```

Remove all items from the list, which name starts with the given prefix. Return the new list.

5.10.2 Special macros for array defined overlays

A second type of overlay definition is faster than the string definition but not as easy to edit: the float array overlay. Here the complete definition of all objects is stored in a float array. Because of the data type, no text objects are available.

The following macros can be found in macro file System/Display/Overlay.CL and are using the same syntax as the OvGAdd-macros. The main difference for the access is the usage of a identifier handle itemID instead of the item name.

```
void OverlayArray_AddLine (int bin, int frame, int itemID,
    int pencolor, int linewidth, float x0, float y0, float x1, float
y1,
    int options);
```

```
void OverlayArray_AddRect (int bin, int frame, int itemID,
    int pencolor, int linewidth, float x0, float y0, float x1, float
y1);
```

```
void OverlayArray_AddRotEllipse (int bin, int frame, int itemID,
    int pencolor, int linewidth, float x0, float y0,
    float rx, float ry, float angle);
```

```
void OverlayArray_AddPolygon (int bin, int frame, int itemID,
    int pencolor, int linewidth, line points, int theFillColor, int
theMode);
```

```
void OverlayArray_DeleteRangeIDs (int buf, int frame, int idFrom, int
idTo);
```

Delete all items with an identifier handle in the range $idFrom \leq id \leq idTo$.

```
void OverlayArray_Reset (int buf, int frame);
```

Delete all items.

5.11 Buffer Attributes: Device Data

DaVis supports a number of non-image devices to be readout during image acquisition, e.g. energy devices or a A/D-converter. Those devices give either single values like an energy value or a complete trace (profile). In the **DaVis** hardware manager the so called **DevData** cards can be used to enable the device data readout. The information are stored as buffer attributes in the image buffers. A *class* of attributes is reserved and uses the prefix `DevData`, see page **??** for the list of attributes. All attribute names should be used with the constant strings `ATTR_DEVDATA_x`, see macro file `cl/System/Hardware/DeviceData.cl`.

5.11.1 Macros for device data access

When a macro programmer wants to access device data information, the attribute names are not needed, because a number of access functions are implemented. All of this macro functions are implemented in macro file `cl/System/Hardware/DeviceData.cl`.

```
int DEVDATA_GetNumber(int theBuffer)
```

Return the number of device data sources in the given buffer. When executing a loop for all device data attributes, start from index 0 and end at the returned value minus one. Use the index as parameter `theDevDataIndex` for the following function.

```
string DEVDATA_GetNameList(int theBuffer)
```

Return a "\n" divided list of device data names.

```
string DEVDATA_GetName(int theBuffer, int theDevDataIndex)
```

Return the name of the given device.

```
int DEVDATA_GetNumberOfData(int theBuffer, int theDevDataIndex)
```

Return the length of the device data's trace.

```
line DEVDATA_GetData(int theBuffer, int theDevDataIndex)
```

Return the trace as an array. The result can be stored in a buffer (row or column) or in a local array, but both buffer and array must have the correct trace size to store the complete trace.

`int DEVDATA_GetScaleI(int theBuffer, int theDevDataIndex, float& aa, float& bb, string& unit, string& label)`

Return the intensity scale of the trace. Same syntax for a function to get the trace's X-scale.

`float DEVDATA_GetOpResult(int theBuffer, int theDevDataIndex, int theOp)`

Calculate the device data (unscaled) value e.g. for image correction and online display. The operation should be one of the constants `DEVDATA_OP_AVG`, `DEVDATA_OP_MIN` or `DEVDATA_OP_MAX`. See macro file `cl/System/Hardware/DeviceData.cl` if more operations are available.

`float DEVDATA_GetOpResultScaled(int theBuffer, int theDevDataIndex, int theOp)`

Same as above, but the returned value is scaled.

`void DEVDATA_SetData(int theBuffer, int theDevDataIndex, line theData)`

Replace the trace by another array. This can be used for own postprocessing functions.

`void DEVDATA_AddData(int theBuffer, string theName, line theData)`

Add new device data to the given buffer. The device gets the name and trace `theData`.

5.12 Drawing into a Buffer

The functions of this group are changing the buffer intensity values. They are drawing no overlay but real intensities.

BufferDrawLine / Circle

```
void BufferDrawLine (int p_hBuffer, int x1, int y1, int x2, int y2,
float value);
```

```
void BufferDrawCircle ( int p_hBuffer, int xcenter, int ycenter,
int xradius, int yradius, float value);
```

Draw a line or circle/ellipse in a buffer. Use intensity `value` as drawing "color".

BufferDrawGeneral

```
void BufferDrawGeneral (int p_hBuffer, int type, float value, line parameters);
```

Draw general object in a buffer. Use intensity `value` as drawing "color". See table 5.2 for descriptions of the different objects and their parameters.

Object	Type	Parameters
Polygon	0	First item includes the number of pixel pairs, followed by the pairs of x- and y-coordinates.
Rectangle	1	x0,y0,x1,y1
Ellipse	2	XCenter, YCenter, XRadius, YRadius

Table 5.2: Object types and their parameters for subroutine `BufferDrawGeneral`.

5.13 Loading and storing a buffer

LoadBuffer

```
void LoadBuffer( string filename, int buf );
```

Load an image from a file on disk into a buffer. Some buffer attributes are automatically created during loading and can be used as additional file information, see table on page **??**.

StoreBuffer

```
void StoreBuffer( string filename, int buf );
```

Store the contents of a buffer to an image file on disk. For all available file types see the *DaVis Software Manual*. If no extension is given in the filename, **DaVis** uses the `IM7` format for image buffers and the `VC7` format for vector buffers.

When storing the **DaVis** standard file types IM7 and VC7, the image and vector data can be compressed to decrease the file size. Those compression parameters are defined in the section about **Storage Parameters** in the *DaVis Software Manual*. The following global variables can be set to the constants IM7_PACKTYPE_x to change the behavior: DefaultIM7PackType for all buffer types and DefaultIM7PackTypeFloat for float, double and vector buffers.

```
int LoadBufferErr( string filename, int buf );

int StoreBufferErr( string filename, int buf );
```

Like LoadBuffer and StoreBuffer, but instead of stopping if an error occures both subroutines will return an error code of table 5.3.

code	Message
0	no error
-1	buffer locked
-2	out of memory
-3	File not found
-4	error while reading file
-9	Other

Table 5.3: Error codes returned by the subroutines LoadBufferErr and StoreBufferErr.

Special parameters for storage of TXT files: Some parameters of the storage of TXT files can be defined in card **Export** of the **Global Options** dialog: The decimal *point* character can be switched between comma and dot, the *separator* between values can be changed between tabulator, space character and carriage return, and the precision of exported float values can be changed between 0 and 15 digits. Programmers are using the variables ExportChars and ExportFloatPrecision.

Buffer_LoadHeader

```
int Buffer_LoadHeader( string theFileName, int theBuffer, int& sizeX,
int& sizeY, int& sizeZ, int& sizeF )
```

Load header and attributes only without loading the complete image or vector data into computer memory. Return 0 if ok, return code < 0 on file reading error (see `LoadBufferErr`).

The buffer size is returned in the given variables. The buffer type is set to `theBuffer`, e.g. call `IsFloat` or `IsSparse` after calling this macro. All attributes are stored in buffer `theBuffer`.

Buffer_LoadFrame

```
int Buffer_LoadFrame( string theFileName, int theBuffer, int theFrame )
```

Load a single frame of a image or vector file into buffer `theBuffer`. Return 0 if ok, return code < 0 on file reading error (see `LoadBufferErr`). Return 1 if frame could not be loaded.

5.14 Buffer Arithmetic

5.14.1 General Functions

Simple Buffer Arithmetics

Simple buffer arithmetics with plus, minus, multiplication and division can be written in a simple way like

```
B[3] = B[1] + B[2]
```

Note: The **data type** of the first buffer defines the resulting data type. This means e.g. when summing up a WORD and a FLOAT buffer, the result will be of data type WORD. On the other hand when summing up a FLOAT and a WORD buffer, then the result will be of data type FLOAT. Additionally the resulting buffer gets all scales and attributes from the first buffer of the operation.

All this simple operations work on the raw intensity and don't take different **intensity scales** into account! When adding two buffers of different intensity scales, then the sum is the sum of the unscaled counts and not of the scales pixel intensities.

Exp / Log / Sqrt / AbsBuffer

```
void ExpBuffer(int inputb, int outputb);

void LogBuffer(int inputb, int outputb);

void SqrtBuffer(int inputb,int outputb);

void AbsBuffer(int inputb, int outputb);
```

Exponentiate every point in a buffer, calculate the natural logarithm, square root or absolute value of every point in a complete buffer. `inputb` and `outputb` may be equal.

The same functions are available for single frames:

```
void ExpBufferFrame(int inputbuf, int inputframe, int outputbuf,
int outputframe);
```

MinMaxBuffer and BuffersMin/Max

```
void MinMaxBufferFrame(int isMax, int inputbuf1, int inputbuf2, int
outputbuf);
```

Calculate the minimum (`isMax=FALSE`) or maximum (`isMax=TRUE`) of every point in two frames.

```
void BuffersMin(int inbuf1, int inbuf2, int outbufMin);

void BuffersMax(int inbuf1, int inbuf2, int outbufMin);
```

Short form for subroutine `MinMaxBufferFrame` concerning the complete buffer.

Mirror / Rotate / FlipBuffer

```
void MirrorLeftRight(int buffer);

void MirrorTopBottom(int buffer);
```

Mirror a buffer: swap left and right or swap top and bottom. Both first function types all mirroring all frames of a multi frame buffer and stores the result in the same buffer.

```
void RotateBuffer(int inputbuf, int outputbuf,
    int center_x, int center_y, float angle);
```

Rotate a complete buffer or a single frame by an arbitrary angle (in degree). If bit 0 of parameter `mode` is set (mode=1), the result is increased to store the complete rotated buffer. By default the buffer is rotated in its own size and the edges will be cut. If bit 1 of parameter `mode` is set (mode=2), the buffers scales are rotated.

Frame overlays of the buffer are rotated, too, see function `RotateOverlayString` on page 107 for details.

```
void FlipBuffer(int buffer, int mode);
```

Flip/rotate a buffer. The first function type flips all frames of a multi frame buffer, while the second works on single frames. See table 5.4 for the available modes.

mode	Function
0	(no change)
1	flipping x and y axis
2	rotation by 90° to the left
3	rotation by 90° to the right
4	vertical mirror (same as `MirrorTopBottom`)
5	horizontal mirror (same as `MirrorLeftRight`)
6	rotate by 180°
7	flipping -x and y axis
+0x40	same operations as before for double frame images (e.g. vector buffers)

Table 5.4: Available `mode` values for subroutine `FlipBuffer`.

MoveBuffer

```
void MoveBuffer(int inputbuf, int outputbuf, float dx, float dy);
```

Copy the contents of inputbuf to outputbuf shifting its points by a specified (x,y)-offset. Fractional shift values are allowed (interpolation among the closest 4 pixels).

ExpandBuffer

```
void ExpandBuffer(int buffer, int x_factor, int y_factor);
```

Expand a buffer by a specified x and y_factor: $1 \leq factor \leq 16$. Every grid rectangle with size **x_factor** * **y_factor** will be filled with the constant source pixel value.

```
void ExpandBufferEx(int inputbuf, int inputframe,
int outputbuf, int outputframe, int x_factor, int y_factor, int mode);
```

Using the given interplator mode = INTERPOLATION_* (*=BILINEAR, BICUBIC, etc.) expand a buffer by a specified x- and y- factor ($1 \leq factor \leq 256$) and do interpolation in between. This function is not equal to an expand buffer followed by a sliding average.

CompressBuffer

```
void CompressBuffer(int buffer, int x_factor, int y_factor);
```

Compress a buffer by a specified x- and y- factor. The average intensity of every source grid rectangle is copied into the destination pixel.

Compress a buffer by a specified x- and y- factor. Set output to `MaskValue`, if `MaxAllowedMaskedPixelRatio` of input pixels have value `MaskValue`. If MaxAllowedMaskedPixelRatio = 0, only create output pixel/vector if all input pixels/vectors had a value $\neq$ MaskValue / were not disabled. If MaxAllowedMaskedPixelRatio = 1, create output pixel/vector even if all input pixels/vectors had a value = MaskValue / were disabled. If MaxAllowedMaskedPixelRatio = 0.3 create output pixel/vector if a maximum of 30% of input pixels/vectors had a value = MaskValue / were disabled.

StretchBuffer

```
void StretchBuffer(int inputbuf, int outputbuf, float x_factor, float
y_factor);
```

Stretch frame by factors for x- and y-direction, use $0.01 \leq factor \leq 1024$.

StretchBufferInteger

```
void StretchBufferInteger(int inputbuf, int outputbuf, int x_factor,
int y_factor);
```

Stretch buffer by `x_factor` and `y_factor`, each $1 \leq factor \leq 1024$.

SetAbove/BelowIntConstant

```
void SetAboveIntConstant( int inputbuf, int outputbuf,
    float refInt, float value);

void SetBelowIntConstant( int inputbuf, int outputbuf,
    float refInt, float value);
```

Set all pixels of the **inputbuf** with intensities higher (or lower) than a reference value **refInt** to a new **value**. Store the result in the **outputbuf**.

ExtractChannel

```
void ExtractChannel( int buffer, int Nchannels, int channel, int outb );
```

Analyze data read in via Cio16-ADC-board. Extracts all columns of the specified **channel** (e.g. with **Nchannel=2** every second column) and stores them in the outputbuffer **outb**. This function is similar to **CompressBuffer** horizontally, but no averaging.

ConvertRGBImage

```
void ConvertRGBImage (int bin, int bout, int mode, float par);
```

Convert image data from a (Imager3) RGB camera to Word image buffer.

- mode 0: convert to a greyscale image
- mode 1: same as mode 0, but half width and height
- mode 10: compute red/green ratio. Each pixel = red*par/green
- mode 11: same as mode 10, but half width and height
- mode 100: extract red component
- mode 110: extract green component
- mode 120: extract blue component

Segmentation

`void Segmentation( int bin, int bout, int threshold, int mode, int& nSegments, line& Xpos, line& Ypos, line& Npixel, line& Xsize, line& Ysize, line& MaxInt, line& SumInt );`

Compute segmentation regions of the input buffer `bin`, which has to be a Word buffer. The detected segments are marked with intensities 10,20,30,... in the output buffer `bout`. The `threshold` defines a segment as a region of pixels all above this intensity.

The following `mode`s are supported:

- 0 = `Xpos,Ypos` are the center of mass of the segments,
- 1 = get the two largest segments with respect to maximum x-size, get left-most point of segment. `Xpos` is the left-most point of the segment, `Ypos` is the y-center of mass of the segment
- 2 = like 0, but a segment is defined as all adjacent pixels which are lower than the highest pixel or pixels already belonging to the segment. In addition only those pixels are taken, which are above 10% of the highest pixel.
- 3 = like 0, but restricted to range defined by `Xpos[0],Ypos[0] +- Xsize[0],Ysize[0]`
- 4 = like 1, but restricted to range defined by `Xpos[0],Ypos[0] +- Xsize[0],Ysize[0]`
- 5 = like 0, but `Xpos,Ypos` are the correct center of the segment rectangle.

The return values give all information about the detected segments. The arrays have to be defined with a large number of items to store all data. If unsure about the number, you can call this subroutine twice, because value `nSegments` gives the real number of detected segments always.

- `nSegments`: number of segments detected
- `Xpos`: array of integers, center x-position of seg.
- `Ypos`: array of integers, center y-position of seg.
- `Npixel`: array of int, number of pixels in each seg.
- `Xsize`: array of int., horizontal segment diameter

- `Ysize`: array of int., vertical segment diameter
- `MaxInt`: array of int., maximum intensity of segment
- `SumInt`: array of in.s, integral intensity of segment

This function can easily be tested by executing `SetupDialogSegmentation(0)`.

Example:

```
int n;
int Xpos[100], Ypos[100], Xsize[100], Ysize[100],
    MaxInt[100],SumInt[100], Npixel[100];
Segmentation( 1, 2, 100, 0, n, Xpos, Ypos, Npixel,
    Xsize, Ysize, MaxInt, SumInt );
InfoText( n + " segments found:" );
```

5.14.2 Image Correction

ApplyImageCorrection

`void ApplyImageCorrection( int bin, int bout, int CorrFunction, int plane, int mode )`

Apply image distortion function for mapping function `CorrFunction` = 0...5 and `plane` = $0...N-1$ corresponding to camera 1 to 6 and the number of one of the N calibration planes (see **DaVis** Manual for the description of the Calibration dialog and the your System Manual FlowMaster, StrainMaster, etc.) for a detailed description of available calibration modes (Stereo calibration, Volume calibration, ...). The available modes are given in table 5.5.

mode	Function
0	apply distortion (buffer `bin` to buffer `bout`)
1	correct distortion (inverse function) (buffer `bin` to buffer `bout`)
16	show distortion in input buffer `bin` (`bout` not used)
17	show correction in input buffer `bin` (`bout` not used)

Table 5.5: The **mode** defines the correction type for subroutine `ApplyImageCorrection`.

About the Polynomial Correction Function

Up to now the only function implemented to describe the image distortion is a polynomial of 3rd order that links x and y pixel coordinates to corrected positions $x´$ and $y´$. The numerical fields shows the coefficients a_0 to a_9 and b_0 to b_9 of the mapping function $\vec{f}(\vec{pos}') = \vec{pos}$ that maps the raw image position $\vec{pos}$ to the corrected position $\vec{pos}'$ [1]

$$\begin{pmatrix} x \\ y \end{pmatrix} = \vec{f}\begin{pmatrix} x´ \\ y´ \end{pmatrix} = \begin{pmatrix} x´ + dx(x´, y´) \\ y´ + dy(x´, y´) \end{pmatrix}$$

With the normalized coordinates $s = \frac{2\cdot(x´-x_o)}{nx}$ and $t = \frac{2\cdot(y´-y_o)}{ny}$ defined by the image size nx , ny and the origin (x_o / y_o) the offsets dx and dy are calculated by:

$$\begin{pmatrix} dx \\ dy \end{pmatrix} = \begin{pmatrix} a_0 + a_1 s + a_2 s^2 + a_3 s^3 + a_4 t + a_5 t^2 + a_6 t^3 + a_7 st + a_8 s^2 t + a_9 st^2 \\ b_0 + b_1 s + b_2 s^2 + b_3 s^3 + b_4 t + b_5 t^2 + b_6 t^3 + b_7 st + b_8 s^2 t + b_9 st^2 \end{pmatrix}$$

To get an idea of how the different parameters affect the image correction let us set all parameters to 0. Let nx and ny be the image size and (x_o / y_o) the center of the image which will play a particular role for higher order distortions. Then the parameters affect the image correction in the following way:

- a_0 describes a simple shift of the whole image in x direction (b_0 in y direction)
- a_1 leads to a linear stretching into x-direction (b_4 into y-direction) where the origin (x_o / y_o) will stay at its place
- a_4 produces a shear in x-direction where the x position passing through the origin will stay unaffected (b_1 does the same in y-direction)
- a_2 produces a quadratic warping in x-direction (as b_5 will do in y-direction). Once again the origin remains untouched.

and so on. Similarly all other higher order terms $a_{\geq 1}$ and $b_{\geq 1}$ do not change the "origin" (x_o / y_o) – it will always stay at its place. It will only undergo a translation defined by a_o and b_o.

[1] Please note that the functions calculates the *raw* image position (result) from the *corrected* image position (input).

Calculate2x2DVectorsFrom3DVectors

`Calculate2x2DVectorsFrom3DVectors(int in, int out, line pars)`

Calculation of a 2x 2D vector field from a stereo vector field. See table 5.6 for a description of the parameters.

in	source 3D vector buffer
out	buffer with calculated 2x 2D vector field
pars[0]	cameraA (1..6)
pars[1]	cameraB (1..6)

Table 5.6: Parameters for subroutine `Calculate2x2DVectorsFrom3DVectors`

TransformImageWithVectorField

```
void TransformImageWithVectorFieldExt(int in, int inframe, int out,
int ref, int refframe, int mode, int interpolationMode);
```

Transforms an image with a reference vector field so that all pixels are shifted to the position of the (interpolated) vector arrow head. See table 5.7 for a description of the parameters.

ImageReconstructionWithVectorAndImageCorr

```
ImageReconstructionWithVectorAndImageCorr( int inbuf, int inz1, int inframe1,
int inz2, int inframe2, int outbuf, int outz1, int outframe1, int outz2,
int outframe2, int vecbuf, int vecz, int vecframe, int windowsize,
int asymmetric, int useimagecorr, int useheight, int maskbuffer,
float defaultvalue );
```

Transforms 1 or 2 frame(s) of the image `inbuf` with a reference vector field and/or image correction depending on the sign of the given input frames and z planes.

- If (`inz1`,`inframe1`≥ 0)
  transform `inbuf1/inz1/inframe1` to `outbuf1/outz1/outframe1`
- If (`inz2`,`inframe2`≥ 0)
  transform `inbuf2/inz2/inframe2` to `outbuf2/outz2/outframe2`
- If (`outz1`,`outframe1`≥ 0)
  transform `inbuf1/inz1/inframe1` to `outbuf1/outz1/outframe1`

in	scalar input buffer
out	scalar output buffer
ref	reference vector buffer
inframe	frame number of input image
refframe	frame number of vector field
mode	bitmask transformation mode: 1 = internally perform "fill up empty spaces" first 2 = internally perform "fill all" first 32 = respect FramProcessingX attributes to correct image if necessary
Interpolation mode	one of the constants INTERPOLATION_* (e.g. *=BILINEAR or BICUBIC)

Table 5.7: Parameters for subroutine `TransformImageWithVectorFieldExt`

- If (`outz2,outframe2`≥ 0)
  transform `inbuf2/inz2/inframe2` to `outbuf2/outz2/outframe2`

For the other variables see table 5.8.

5.15 Line Arithmetic

5.15.1 Scalar result from lines

LineSum/Rms/Max/MinComponents

```
float LineSumComponents(line theLine);
```

Sum up all components of the line.

```
float LineRmsComponents(line theLine);
```

Calculate rms of the line's components.

vecbuf	if $>$ 0 transform with reference vector field in `vecbuf/vecz/vecframe` else assume $V = (0, 0, 0)$
windowsize	if $>$ 0 use whittaker reconstruction with this window size else use bilinear interpolation
asymmetric	if $\neq$ 0 asymmetric transformation (only transform 2nd frame with -V). else symmetric tranformation: -V/2 for 1st AND +V/2 for 2nd frame
useimagecorr	if $\neq$ 0 use image correction for all frame (CameraNameX attribute necessary !!!) else work on pixel positions in raw image
useheight	if = 0 use no height information (plane=0) for image correction else use internal scalar plane "Height"
maskbuffer	if = 0 transform whole image with Whittaker / bilinear interpolation else transform only inside mask with Whittaker, else use simple bilinear interpolation
defaultvalue	fill pixels with given value if no valid image information is available (usually = 0)

Table 5.8: Parameters for subroutine `ImageReconstructionWithVectorAndImageCorr`

```
float LineMaxComponents(line theLine);
```

```
float LineMinComponents(line theLine);
```

Return maximum or minimum value of all components of the line. Store position of maximum or minimum in global CL-variable `XPos`.

5.15.2 Operate line(s) and return a line result

One or two lines are used as source data for an operation. The result is a line with the minimum length of the source line(s). Both the source(s) and the destination can be rows or columns of buffers or variable arrays.

LineSum

```
line LineSum(line theFirstLine, line theSecondLine);
```

Sum up both lines component by component and return a new line.

Exp / LogLine

```
line ExpLine(line aline);

line LogLine(line aline);
```

Calculate the exponential (logarithm) values of a line.

Examples:

```
P[0] = ExpLine(R[5,10])

P[0] = LogLine(R[5,10])
```

Profile 0 includes the exponential (natural logarithm) values of row 10 from buffer 5 ("exp[row10]", "ln[row10]").

GetLine

```
line GetLine(int p_hBuffer, int x1, int y1, int x2, int y2, int npoints);
```

Get a line from `x1/y1` to `x2/y2`. All pixels of the line from outside of the valid image area, e.g. negative coordinates, are set to 0.

Compose a line from specified buffer along the line from `x1/y1` to `x2/y2` with specified number `npoints` of data points. If `npoints=0`, the number of data points is automatically computed from the length between `x1/y1` and `x2/y2` (in pixels).

Example:

```
P[1]=GetLine(1,100,100,200,200,384);
```

Line 100/100-200/200 (normal length 141 pixels) is expanded to 384 pixels and stored in Profile 1. The pixel intensities of the output line are interpolated in-between the four closest surrounding pixels (not just simply the closest pixel taken).

GetCircle

```
line GetCircle(int p_hBuffer, int xcenter, int ycenter,
    int radius, int npoints);
```

Compose a circle around `xcenter/ycenter` with radius and store the pixel intensities on this circle in a line (array variable, row or column of a buffer or profile). The pixel intensities of the output line are interpolated in-between the four closest surrounding pixels (not just simply the closest pixel taken). If `npoints=0`, the number of data points is automatically computed from the radius.

PlotXY

```
void PlotXY( int profnr, line xvalues, string xdescr, string xunits,
line zvalues, string zdescr, string zunits, int autoscale);
```

Plot a profile from an array of (x,z) value pairs.

`autoscale = 0`: use the x-scale that is already set for the profile buffer

`autoscale = 1`: set the x-scale optimally

SortLine

```
line SortLine(line theInputLine, line& oldIndexList);
```

Sort the input array (float, word or string), return the sorted list and the index table of the old list.

Example:

```
int oldlist[1000];
R[1,1] = SortLine(R[1,0],oldlist);
R[1,2] = oldlist;
```

This example sorts all values of row 0 in buffer 1, stores the sorted list in row 1 and the index of the original order in row 2.

MoveLine

```
void MoveLine( line& inLine, int fromX, int size, line& outLine, int toX );
```

Move part of a line to another line: The item with smallest index is given together with the number of items to be copied into the destination, starting at the given index. Both lines may overlap!

5.15.3 Functions for ranges of lines

This special line functions operate on **lines of the same buffer**, either on columns `C[]` or on rows `R[]`. All rows/columns in in the given range from `line_a` to `line_b` are taken into account.

Sum / AvgLine

```
line SumLine(line line_a, line line_b);
```

```
line AvgLine(line line_a, line line_b);
```

Calculate a sum (average) profile of a range of lines in **one** buffer.

Examples:

```
P[0] = SumLine(R[5,0],R[5,19])
```

```
P[0] = AvgLine(R[5,0],R[5,19])
```

Profile 0 includes the sum (average) of rows 0 to 19 from buffer 5.

Min / MaxLine

```
line MinLine(line line_a, line line_b);
```

```
line MaxLine(line line_a, line line_b);
```

Calculate a minimum (maximum) profile of a range of buffer lines.

Examples:

```
P[0] = MinLine(R[5,0],R[5,19])
```

```
P[0] = MaxLine(R[5,0],R[5,19])
```

Profile 0 includes the minimum (maximum) of rows 0 to 19 from buffer 5. The minimum (maximum) is calculated for each point "x" in profile 0 by taking into account the points (columns) "x" of all 20 rows, then going to the next point (column) "x+1" and so on.

5.16 Rectangles

DaVis uses eight rectangular areas, which are free for the user. The coordinates of this rectangles are stored in the CL-variable

```
int Rectangles[4*(NumRect+1)]
```

The active rectangle is defined by variable `int ActiveRect`, the last available rectangle, which should be used by macros only, is stored as `int NumRect = 9` by default. This value can be increased if wanted, but remember to increase the size of array `Rectangles`! There is no definition for rectangle 0, which is used to describe the complete buffer.

5.16.1 Rectangle Macros

Please use the following macros for rectangle handling:

Get / SetRect

```
int SetRect( int RectNum, int x1, int y1, int x2, int y2);
```

```
int GetRect( int RectNum, int& x1, int& y1, int& x2, int& y2);
```

Both macros return the size of the rectangular region (= width x height).

5.16.2 Moving Rectangles and Volumes

Buffer_ExtractVolume

```
void Buffer_ExtractVolume( int sourcebuffer, int destbuffer,
    line theVolume );
```

Extract volume from source buffer and copy to destination buffer. The destination is completely renewed.

The rectangle or the volume is given with coordinate pairs (from,to) in line of integer values in order x0, x1, y0, y1, z0, z1, f0, f1. Coordinate pair (-1,-1) means *for all*, e.g. f0=f1=-1 works on all frames. If the volume array is shorter, the missing coordinate pairs are used as *for all*.

```
void CutoutRect( int theSrcBuf, int theSrcFrm,
    int x1, int y1, int x2, int y2,
    int theDestBuf, int theDestFrm );
```

Cutout rectangular buffer region and copy to theDestBuf. If theSrcFrm and theDestFrm = -1 then cutout for all frames. If theSrcFrm and theDestFrm != -1 then cutout theSrcFrm and copy to theDestFrm.

The main difference to `Buffer_ExtractVolume` is the possibility to copy a rectangular region from a defined frame to another defined frame. The destination buffer will not loose data of other frames.

MoveRectangle

```
int MoveRectangle( int sourcebuffer, int rectangle,
    int destbuf, int destRow, int destCol );
```

Copy a rectangular region from the source buffer to the destination buffer. Size and type of the destination are not changed, if the destination is valid. If the destination is empty, then it is created with the same type as the source buffer and with the size of the given rectangle. The upper left corner of the rectangle is placed at the specified coordinates `destRow / destCol (x/y)` in the destination.

5.16.3 Statistics on Rectangles and Volumes

Buffer_Statistics

```
line Buffer_Statistics( int theBuffer, int theMode, line theVolume );
```

Calculate statistics on rectangles and volumes for image and vector buffers. In `theMode=0` the complete buffer is taken into account. In `theMode=1` the rectangle or volume is given with coordinate pairs (from,to) in a float line in order x0, x1, y0, y1, z0, z1, f0, f1. Coordinate pair (-1,-1) means *for all*, e.g. f0=f1=-1 works on all frames. If the volume array is shorter, the missing coordinate pairs are used as *for all*. Add `theMode+=0x0100` to return scaled statistics, which is important for buffers with different frame scales.

Return a line with float values in the order average, rms, sum, total number of valid pixel or vectors, minimum intensity, maximum intensity, X-position

of minimum, Y-pos, Z-pos, F-pos, X-position of maximum, Y-pos, Z-pos, F-pos.

For vector buffers the statistics of length are returned as average, rms, min, max. The statistics for each component are starting at array position 20: avgX, rmsX, minX, maxX, avgY, rmsY, minY, maxY, avgZ, rmsZ, minZ, maxZ, rmsTotal.

The total RMS `rmsTotal` is calculated as $R_t = \sqrt{R_x^2 + R_y^2 + R_z^2}$ with R_x as RMS in X-direction.

RectStatistics

```
void RectStatistics( int buf, int rect,
    float& avg, float& rms, float& min, float& max);

void RectStatisticsScaled( int buf, int rect,
    float& avg, float& rms, float& min, float& max);
```

Get the sum, average, minimum and maximum of the intensities of all points in a rectangular buffer region. Use `rect=0` for the whole image. Both functions are taking the valid flags into account if the buffer has this special type of information for each pixel.

The location of the maximum is stored in variables `XPos` and `YPos`. The resulting intensities are in counts for the first function and in intensity scales for the second function.

Sum / Avg / Rms / Max / MinRect

All statistic properties can be obtained by single functions also. Use `rect=0` for the whole image.

```
float SumRect(int buf, int rect);

float AvgRect(int buf, int rect);

float RmsRect(int buf, int rect);

float MaxRect(int buf, int rect);

float MinRect(int buf, int rect);
```

Get the sum, average, root mean square, maximum or minimum of the intensities of all points in a rectangular buffer region. The location of the maximum or minimum is stored in the global variables `XPos` and `YPos`.

SetInside/OutsideRectConstant

```
void SetInsideRectConstant( int inputbuf, int outputbuf,
    int rectangle, float value );

void SetOutsideRectConstant( int inputbuf, int outputbuf,
    int rectangle, float value );
```

Set all pixels inside (or outside) a rectangle to a specified value. This operation should not be used on multi frame images.

BinarizeRectLevel

```
void BinarizeRectLevel (int inputbuf, int outputbuf, int rectangle,
float minvalue, float maxvalue)
```

Set all pixels inside the rectangle to 1 count, if `minvalue <= Pix < maxvalue`. Otherwise the pixel is set to 0. All pixels outside the rectangle are set to 0, too.

CountRect

```
int CountInsideRectPixel( int buf, int rect);
```

Get the number of valid pixels in a rectangular buffer region

5.17 Filters

LinearFilter

```
void LinearFilter( int source_buffer, int dest_buffer, int to_float,
int filter_type, string coeffs, int divide, int rim );
```

Apply a linear filter of type filter_type to the source_buffer. See table 5.9 for a detailed description of the parameters. The coefficients are given line wise: ''`a1 a2 a3 b1 b2 b3 c1 c2 c3`'' for matrix

$$\begin{pmatrix} a1 & a2 & a3 \\ b1 & b2 & b3 \\ c1 & c2 & c3 \end{pmatrix} / divisor$$

Vector coefficients are given from top to bottom for vertical vectors and from left to right for horizontal vectors.

A list of predefined coeficients for linear filters is available in macro file `cl/System/Processing/LinearFilter.cl`. To execute those filters please check the index of the wanted filter in the array `FilterNames[]`. Then set `theActiveFilter` to the correct index and call the subroutine

```
LinearFilter( theSourceBuffer, theDestBuffer,
    FilterFloat[theActiveFilter], FilterType[theActiveFilter],
    FilterFactors[theActiveFilter], FilterDivide[theActiveFilter],
    FilterRim[theActiveFilter] );
```

Parameter	**Value**	
source_buffer		source buffer
dest_buffer		destination buffer
to_float	0 1	destination buffer is of the same type as the source buffer always create FLOAT-buffer as output
filter_type	0/1/2/3 4/5/6/7 8-11 12-15	3x3 / 5x5 / 7x7 / 9x9 matrix 3Hx3V / 5Hx5V / 7Hx7V / 9Hx9V separable filter 3H / 5H / 7H / 9H vector 3V / 5V / 7V / 9V vector
coeffs		contains coefficients, e.g. ”1 1 1 1 1 1 1 1 1”
divide		divide resulting sum by this value
rim	0 1	set outer rim to 0 do not change outer rim

Table 5.9: Parameters of subroutine `LinearFilter`

Examples:

```
LinearFilter( 1, 2, 0, 5, "1 2 4 2 1 1 2 4 2 1", 100, 1);

LinearFilter( 1, 2, 0, 0, "1 2 1 2 4 2 1 2 1", 100, 1); // 3x3 Gauss
```

NonlinearFilter

```
void NonlinearFilter(int source_buffer, int dest_buffer, int mode,
int value);
```

Execute 2D-nonlinear filter. See table 5.10 for the available modes. `value = noise_level = 0,1,...` with automatically calculated background.

mode	filter
1	**Median filter:** take of all value pixels the middle one
2	**Erosion filter**
3	**Dilatation filter**
4	**Concentration filter:** moving all counts of the surrounding pixels which are above the (background + noise_level) to the center pixel, if it is the one with the highest intensity.
5	**Compute sliding average**: value = scale-length
6	**Compute sliding minimum:** value = scale-length
7	**Compute sliding maximum**: value = scale-length
8	**Compute strict sliding minimum**: value = scale-length
9	**Compute strict sliding maximum:** value = scale-length
10-14	Same as 5-9, but only horizontally
20	Local RMS

Table 5.10: Modes of subroutine `NonlinearFilter`

fft

```
void fft (int inputBuffer, int amplitudeBuffer, int phaseBuffer, int
shiftFFT);
```

Calculate the Fast Fourier Transformation (FFT). inputBuffer is the input buffer. amplitudeBuffer is the output amplitude buffer. phaseBuffer is the output phase buffer. If shiftFFT is set to TRUE: Shift the FFT to the center, else 0 frequency is top left in the map. Default true

ifft

```
void ifft(int amplitudeBuffer, int phaseBuffer, int outputBuffer, int
shiftFFT);
```

Calculate Inverse Fast Fourier Transformation (IFFT). amplitudeBuffer is the input amplitude buffer. phaseBuffer is the input phase buffer. outputBuffer The output buffer. If shiftFFT is set to TRUE, then FFT was shifted to the center, must be the same as by the fft function.

5.18 Fitting Functions

FitLine

```
int FitLine( line XX, line YY, line PP, int theFunc, int theMask);
```

Fitting the function $F(X) = A * X + B$ to the given samples (XX,YY). Return the error code or 0, if fine. The parameters are:

- `XX,YY`: X,Y components of sample points
- `PP`: Initial and result model parameter
  - Input : PP[0]=A0, PP[1]=B0, PP[2]=max. iterations
  - Output: PP[0]=A , PP[1]=B , PP[2]=correlation, PP[3]=χ^2, PP[4]=used iterations
- theMask = bitmask to enable single parameter to be fitted

Note: theFunc should be 0. Other values are ignored. Use theMask=3 for a Linear Regression.

5.19 Operations on vector buffers

VectorOperation

```
int VectorOperation( int inbuf, int outbuf, int mode, int parameter );
```

Operations on complete vector fields. See table 5.11 for descriptions of the `mode` parameter.

VectorProcessing

```
void VectorProcessing( int inbuf, int outbuf, int mode, float value );
```

Vector buffer processing. Post processing routines on vectorfields, normally used after PIV or PTV evaluations. See table 5.12 for the `mode` parameter.

VectorDisableIfOutsideInterval

```
void VectorDisableIfOutsideInterval(int buffer, int AVG, int RMS, float
factor);
```

This function allows to disable all vector outside a specified interval where two reference buffers define the thresholds for each grid position. This is used for example in the avergae /RMS calculation to exclude all vectors outside the interval $V_{avg} \pm factor \cdot V_{RMS}$. The parameters of the function are the buffer in question and the two reference buffer *AVG* holding the vectors V_{avg} and *RMS* holding the vectors V_{RMS}. V_{avg} is the central vector for each grid position, V_{RMS} is the allowed interval width and *factor* a factor so that it is easy to relax the threshhold for all three directions x, y and z at once.

GetVector

```
int GetVector( int buffer, int x, int y, float& vx, float& vy, int header );
```

Get vector at position x/y. The vector length is stored in vx and vy. Returns 0 if the vector is disabled, 1 to 4 for the number of the best vector, 5 if the vector is preprocessed or 6 if the vector is smoothed.

If the vector field format is extended (as used for PIV evaluations) you can get different vectors of one position (x/y) selected by `header`: use 1 to 4 for the index of the vector choice.

CONSTANT	**value**	**(S)= scalar result, (V)=vector reult**	**Operation (for scalar results each pixel holds the mentioned value for the vector at that position)**
V_OP_CONVERT_TO_ EXTENDED	6	V	converts vector field to 4-choice vector field
V_OP_CONVERT_TO_ STANDARD	7	v	converts vector field to single-choice vector field
V_OP_MAKE_MASK_ PERMANENT	11	v	sets invalid (masked) vectors to disabled with value (0,0,0)
V_OP_MAKE_ MASKPLANES_PERMANENT	12	v	sets invalid (masked) vectors to disabled with value (0,0,0) and removes mask plane
VECTOR_ TAKEOUTROTATION	0x4000	flag	DEPRECATED (old code might still use this constant, but has no effect)
VECTOR_IN_PERCENT	0x8000	flag	DEPRECATED (old code might still use this constant, but has no effect)
V_OP_PEAK_RATIO	99	S	Each pixel holds the peak ratio for the corresponding vector
V_OP_LENGTH	98	S	vector length (2D/3D)
V_OP_VX	97	S	Vx
V_OP_VY	96	S	Vy
V_OP_VZ	95	S	Vz
V_OP_DIFF4	94	S	local deviation from average of 4 neighbors
V_OP_DIFF8	93	S	local deviation from average of 8 neighbors
V_OP_HEADER	92	S	0=off, 1-5, choice
V_OP_ROT_Z	89	S	xy-2D-vorticity = Vorticity-z = Eyx - Exy
V_OP_MINUS_ROT_Z	88	S	-xy-2D-vorticity = -Vorticity-z = -(Eyx - Exy)
V_OP_ROT_Y	87	S	rot-y = Exz - Ezx
V_OP_ROT_X	86	S	rot-x = Ezy - Eyz

CONSTANT	value	(S)= scalar result, (V)=vector reult	Operation (for scalar results each pixel holds the mentioned value for the vector at that position)
V_OP_ROT_3D	85	V	(Vorticity-x,Vorticity-y,Vorticity-z) as vector (multi plane vector buffer needed)
V_OP_2D_DIVERGENCE_XY	84	S	xy-divergence -(Exx + Eyy)
V_OP_2D_DIVERGENCE_XZ	83	S	-(Exx + Ezz)
V_OP_2D_DIVERGENCE_YZ	82	S	-(Eyy + Ezz)
V_OP_3D_DIVERGENCE	81	S	-(Exx + Eyy + Ezz) (multi plane vector buffer needed)
V_OP_LAMB_VECTOR	80	V	= (Rot-3D) x (V) (multi plane vector buffer needed)
V_OP_DUDX_PLUS_DVDY	79	S	- xy-divergence
V_OP_ABS_DUDX_PLUS_DVDY	78	S	abs(xy-divergence)
V_OP_SHEAR_STRENGTH	75	S	shearing strength
V_OP_SWIRLING_STRENGTH	74	S	swirling strength
V_OP_SWIRL_PLUS_SHEAR	73	S	swirl + shear
V_OP_MAX_3DNORM_STRAIN	72	S	maximum normal strain in 3D (multi plane vector buffer needed)
V_OP_MEDIAN_3DNORM_STRAIN	71	S	medium normal strain in 3D (multi plane vector buffer needed)
V_OP_MIN_3DNORM_STRAIN	70	S	maximum normal strain in 3D (multi plane vector buffer needed)
V_OP_EXX	69	S	Exx
V_OP_EXY	68	S	Exy
V_OP_EXZ	67	S	Exz (multi plane vector buffer needed)
V_OP_EYX	66	S	Eyx
V_OP_EYY	65	S	Eyy
V_OP_EYZ	64	S	Eyz (multi plane vector buffer needed)

CONSTANT	**value**	**(S)= scalar result, (V)=vector reult**	**Operation (for scalar results each pixel holds the mentioned value for the vector at that position)**
V_OP_EZX	63	S	Ezx
V_OP_EZY	62	S	Ezy
V_OP_EZZ	61	S	Ezz (multi plane vector buffer needed)
V_OP_ACCELERATION	58	S	calculate acceleration, parameter = dt between frames
V_OP_ADD_VECBUF	57	V	adds 2 vector buffers $V_x = V_{x1} + V_x; V_y = V_{y1} + V_{y1}; V_z = V_{z1} + V_{z2}$ where abs(parameter) is the number of 2nd (vector) buffer for this operations. Other than simply adding 2 vector buffers via $B[1] + B[2]$ special care is taken of disabled vectors and (mixed) 2D and 3D vector buffer operations, size and format checks. Furthermore all global attributes will be taken from 2nd vector buffer (important for time/ADC values). If parameter > 0 disabled vectors are treated as (0,0,0) as used for Average and RMS calculations if parameter < 0 disabled vector in one of input buffers disables vector in output buffer as used for successive deformation
V_OP_SUB_VECBUF	56	V	subtract 2 vector buffers $V_x = V_{x1} - V_{x2}; V_y = V_{y1} - V_{y2}; V_z = V_{z1} - V_{z2}$ See description of mode 57

CONSTANT	**value**	**(S)= scalar result, (V)=vector reult**	**Operation (for scalar results each pixel holds the mentioned value for the vector at that position)**
V_OP_ADDSQUARED_ VECBUF	55	V	subtract 2 vector buffers $V_x = V_{x1} - V_{x2}; V_y = V_{y1} - V_{y2}; V_z = V_{z1} - V_{z2}$ See description of mode 57
V_OP_SUBSQUARED_ VECBUF	54	V	subtract 2 vector buffers $V_x = V_{x1} - V_{x2}; V_y = V_{y1} - V_{y2}; V_z = V_{z1} - V_{z2}$ See description of mode 57
V_OP_DIV_VECBUF	53	V	divide 2 vector buffers $V_x = V_{x1} = V_x; V_y = V_{y1} = V_{y2}; V_z = V_{z1} = V_{z2}$ See description of mode 57
V_OP_SQRT_VECBUF	52	V	takes square root of vector components of input buffer: $V_x = \sqrt{\lvert V_{x1} \rvert}; V_y = \sqrt{\lvert V_{y1} \rvert}; V_z = \sqrt{\lvert V_{z1} \rvert}$
V_OP_ADD1_IFEXISTS_ VECBUF	51	V	adds 1 to input buffer, if vector exists in vector buffer specified by parameter (as used for Avg, RMS calculations) $V_x + +; V_y + +; V_z + +;$ if vector in buffer parameter is not disabled.
V_OP_ADDLAGRANGE_ VECBUF	50	V	adds 2 vector buffers in the Lagrangian reference frame: $V_x = V_{x1} + V'_{x2}$ with $V_{x2}(x,y,z)' = V_{x2}(x + V_{x1}, y + V_{y1}, z + V_{z1})$ $V_y = V_{y1} + V'_{y2}$ with $V_{y2}(x,y,z)' = V_{y2}(x + V_{x1}, y + V_{y1}, z + V_{z1})$ $V_z = V_{z1} + V'_{z2}$ with $V_{z2}(x,y,z)' = V_{z2}(x + V_{x1}, y + V_{y1}, z + V_{z1})$ See description of mode 57

CONSTANT	**value**	**(S)= scalar result, (V)=vector reult**	**Operation (for scalar results each pixel holds the mentioned value for the vector at that position)**
V_OP_SET_TO_0_BELOW_INT	49	V	disable vector component if it is below the value specified by parameter If $V_x \leq$ parameter set $V_x = 0$ If V $y \leq$ parameter set $V_y = 0$ If $V_z \leq$ parameter set $V_z = 0$
V_OP_ADD_VECBUF_DISABLED_IS_ZERO	157	V	adds up vector fields, disabled vectors will not disable result, but are treated as (0,0,0) vectors
V_OP_SUB_VECBUF_DISABLED_IS_ZERO	156	V	subtracts vector fields, disabled vectors will not disable result, but are treated as (0,0,0) vectors
V_OP_ADDLAGRANGE_VECBUF_DISABLED_IS_ZERO	150	V	adds 2 vector buffers in the Lagrangian reference frame, disabled vectors will not disable result, but are treated as (0,0,0) vectors: $V_x = V_{x1} + V'_{x2}$ with $V_{x2}(x,y,z)' = V_{x2}(x + V_{x1},$ $y + V_{y1}, z + V_{z1})$ $V_y = V_{y1} + V'_{y2}$ with $V_{y2}(x,y,z)' = V_{y2}(x + V_{x1},$ $y + V_{y1}, z + V_{z1})$ $V_z = V_{z1} + V'_{z2}$ with $V_{z2}(x,y,z)' = V_{z2}(x + V_{x1},$ $y + V_{y1}, z + V_{z1})$ See description of mode 57

CONSTANT	**value**	**(S)= scalar result, (V)=vector reult**	**Operation (for scalar results each pixel holds the mentioned value for the vector at that position)**
V_OP_MAX_NORM_STRAIN	48	S	maximum normal strain
V_OP_MIN_NORM_STRAIN	47	S	minimum normal strain
V_OP_MAX_SHEAR_STRAIN	46	S	maximum shear strain
V_OP_MAX_STRAIN_ANGLE	45	S	angle for the maximum strain direction
V_OP_MAX_SHEAR_ANGLE	44	S	angle for the maximum shear direction
V_OP_ADD_VECBUF_DIFFGRID	43	V	Adds vector buffers with different grids. The second vector field is interpolated and the result will have the same grid as the input vector field.
V_OP_EXY_PLUS_EYX_DIV_2	42	S	(Exy + Eyx) / 2
V_OP_MINUS_STRAINMAX_DIV_MIN	41	S	-(maximum normal strain/ minimum normal strain)
V_OP_MINUS_EYY_DIV_EXX	40	S	-Eyy/Exx
V_OP_MINUS_STRAINMIN_DIV_MAX	39	S	-(minimum normal strain / maximum normal strain)
V_OP_ANGLE	38	S	angle for the vector direction
V_OP_3D_DIVERGENCE_MINMAXMIN	37	S	minimum normal strain^2*maximum normal strain
V_OP_3D_DIVERGENCE_MINMAXMAX	36	S	minimum normal strain*maximum normal strain^2

CONSTANT	**value**	**(S)= scalar result, (V)=vector reult**	**Operation (for scalar results each pixel holds the mentioned value for the vector at that position)**
V_OP_3DSHEAR	32	S	Calculates the value 'maximum eigenvalue - minimum eigenvalue' of the strain tensor (only works for multi-z vector fields as derived in Tomo-PIV, where Ezz exists)
V_OP_3DSWIRL_STRENGTH	31	S	Calculates the value lambda-2 in terms of the eigenvalues of the symmetric tensor $S^2 + \Omega^2$ where S and Ω are the symmetric and the anti symmetric parts of the velocity gradient tensor rV (refer to J.Fluid.Mech. (1995), vol 285, pp69-94 : 'On the Identification of a vortex' by Jinhee Jeong and Fazle Hussain) (only works for multi-z vector fields as derived in Tomo-PIV, where Ezz exists)
V_OP_CURVATUREMAX	29	S	max. curvature of the Height scalar field
V_OP_CURVATUREAVG	28	S	average curvature of the Height scalar field
V_OP_SURF_ANGLE_TO_Z	25	S	inclination of the Height scalar field to the z-axis
V_OP_VON_MISES_STRAIN	102	S	von Mises strain (for details refer to StrainMaster manual)
V_OP_TRESCA_STRAIN	103	S	Tresca strain (for details refer to StrainMaster manual)

CONSTANT	**value**	**(S)= scalar result, (V)=vector reult**	**Operation (for scalar results each pixel holds the mentioned value for the vector at that position)**
V_OP_EZZ_PLANE_ STRESS_STATE	104	S	Ezz estimation for strain vector fields
V_OP_DUDX	105	S	derivative dVx/dx
V_OP_DUDY	106	S	derivative dVx/dy
V_OP_DUDZ	107	S	derivative dVx/dz (multi plane vector buffer needed)
V_OP_DVDX	108	S	derivative dVy/dx
V_OP_DVDY	109	S	derivative dVy/dy
V_OP_DVDZ	110	S	derivative dVy/dz (multi plane vector buffer needed)
V_OP_DWDX	111	S	derivative dVz/dx
V_OP_DWDY	112	S	derivative dVz/dy
V_OP_DWDZ	113	S	derivative dVz/dz (multi plane vector buffer needed)

Table 5.11: Parameter `mode` of subroutine `VectorOperation`. (u,v,w) is the length of vector component (Vx,Vy,Vz) and (x,y,z) is the position. The parameter mode selects the different operations. For mode 1 to 5 the result is a simple 1-D float buffer (image). **Hint:** Use constants V_OP_* when needing mode values inside a macro. The definition can be found in file `CL/Constants.CL`

mode	Action
0	no action
1	Local Median Test. Comparison between a vector and its surrounding using the RMS. The parameter `value` specifies the allowed RMS range (intern: value * RMS) If the vector field format is extended the routine searches for another calculated vector corresponding to the condition. If no vector is found the vector will be disabled (the header entry of the vector buffer is set to 0). Vector data stays untouched
2	Fills the empty spaces in a vector field with the mean of the surrounding. If the vector field format is extended the new vector is written to position 4 and the header entry is set to 5.
4	Smooth the complete vector field by a 3x3 mean filter. If the vector field format is extended the new vectors are written to position 4 and the header entry is set to 6.

Table 5.12: Parameter `mode` of subroutine `VectorProcessing`.

SetVector

```
void SetVector( int buffer, int x, int y, float vx, float vy, int header );
```

Set vector at (x,y) to (vx,vy). If the vector field format is extended (as used for PIV evaluations) the parameter `header` (see table 5.13) selects the position of the vector.

Get / Set4DVector

```
int Get4DVector( int buffer, int x, int y, int z, int frame, float&
vx, float& vy, float& vz, int header );
```

Get vector at position x/y/z in a frame. The vector length is stored in vx, vy and vz. Only for vector fields with out-of-plane component, otherwise same as `GetVector()`.

```
void Set4DVector( int buffer, int x, int y, float vx, float vy, float
vz, int header );
```

header	Description
< 0	Set the header entry to abs(header). The vector is untouched.
0	Vector is disabled.
> 0	Set the header entry and the vector.
5	Vector is preprocessed, the components are stored in vector entry 4.
6	Vector is smoothed, the components are stored in vector entry 4.

Table 5.13: Modes for parameter `header` of subroutines `SetVector` and `Set4DVector`.

Set vector at (x,y,z,frame) to (vx,vy,vz). Only for vector fields with out-of-plane component, otherwise same as `SetVector()`. For parameter `header` see table 5.13.

Get / Set3DVector

```
int Get3DVector( int buffer, int x, int y, float& vx, float& vy, float&
vz, int header );
```

```
void Set3DVector( int buffer, int x, int y, float vx, float vy, float
vz, int header );
```

Get or set a vector at position x/y. The vector length is stored in vx, vy and vz. This function are implemented as macros and internally call subroutines `Get4DVector` and `Set4DVector`.

GetInterpolated3DVector

```
void GetInterpolated3DVector( int buffer, float x, float y, int frame,
float& vx, float& vy, float& vz);
```

Get vector at (float) pixel position x/y from selected frame.

Vector components are stored in `vx`, `vy` and `vz`. The `header` is automatically "currently used" for vector fields with no out-of-plane component `vz=0`.

GetPeakRatio

```
float GetPeakRatio( int buffer, float x, float y, int frame );
```

Get peak ratio of vector at given position.

VectorMinMax / AvgRms

```
void VectorMinMax( int buffer, float& minX, float& minY, float& minL,
float& maxX, float& maxY, float& maxL )
```

Compute length of minimum and maximum vector (`minL` and `maxL`). Compute minimum and maximum of vector-components (`minX`, `minY` and `maxX`, `maxY`).

```
void VectorAvgRms( int buffer, float& avgX, float& rmsX, float& avgY,
float& rmsY );
```

Compute average and r.m.s. of vector field.

Vector3DStatistics

```
void Vector3DStatistics( int buffer, float& avgZ, float& rmsZ, float&
minZ, float& maxZ, float& minL, float& maxL );
```

Compute statistics for 3D-vector field as addition to `VectorMinMax()` and `VectorAvgRms()`.

VectorOP_VxyzPlane

```
void VectorOP_VxyzPlane( int inputbuf, int inputframe, int outputbuf,
int outputframe, int theMode );
```

Extract vector component as image from a vector buffer or use image as component for a vector buffer. Parameter `theMode` defines the component to be worked with: Vx (0), Vy (1), Vz (2).

6 General Macros

6.1 About General Macros

In this chapter important macros are described. They are either loaded by default or have to be loaded manualy or by a `LoadMacroFile()` at the beginning of a user macro file.

6.1.1 Temporary Buffers

For buffer arithmetics often temporary buffers are used to hold results of single calculation steps. The calculation function should not use constant buffer numbers for temporary buffers, because this buffer numbers could be used by other functions. Function `GetTempBuffer()` always returns the handle of an unused buffer or 0 on error. The buffer should be destroyed at the end of calculation by `FreeTempBuffer( int p_hBuffer )` :

```
// calculate logarithm of buffer 1:  B1 = Log(B1)
int hTempBuf = GetTempBuffer();
LogBuffer( 1, hTempBuf );
B1 = B[hTempBuf];
FreeTempBuffer(hTempBuf);
```

Function `GetTempBuffer()` initialize the buffer with a size of 2 times 2 pixel. It is possible to get a free temporary 2D buffer with initialized size and type (word=0 or float=1) by function `GetTempBufferSetSize( int p_nWidth, int p_nHeight, int p_eType )`

The next macro creates a temporary buffer as a copy of another buffer: `GetTempBufferCopy( int p_hBuffer )`

The index of a temporary buffer is above the buffer range for normal usage, so the buffer is not included in the buffer list window.

At the end of a macro run, when the STOP button disappears, all temporary buffers are destroyed by the automatically called function `DeleteTempBuffer()`.

So these buffers can not be used to show results of a macro. For this case please use the reserved buffers, which are described in the following section.

6.1.2 Reserved Buffers

The reserved buffers are used e.g. by the movie dialog to hold the data to be displayed on screen. A macro dialog should call for the number of such a buffer at its first execution during one run of **DaVis**. Then the macro can use this buffer until **DaVis** is finished. The reserved buffer must not be freed and given back to the system for further usage by another macro. To avoid memory leaks you can set the size of the reserved buffer to 0 (call `FreeTempBuffer`).

`int GetReservedBuffer();` returns the buffer index of the reserved buffer. The index is above the buffer range for normal usage, so the buffer is not included in the buffer list window.

`int GetReservedBuffers( int theBuffersN );` returns the buffer index of the first buffer of a consecutive range of the given number of reserved buffers.

`void FreeReservedBuffer( int theBuffer );` frees a reserved buffer for further usage by other macros. This function should be called by macros which use reserved buffers for a short time only, e.g. used for the life time of a dialog.

Example:

```
// Global variable to hold the index of my buffer.
// Set to 0 and receive the index at the first call.
int myReservedBuffer = 0;

void SetupDialogMyFunction()
{
    if (myReservedBuffer==0)
        myReservedBuffer = GetReservedBuffer();
    // now I can use the buffer...
}
```

6.1.3 Stop Action and Error Action

At every end of a macro run a special macro is executed as **stop action**. The default stop action is defined by CL-variable `DefaultStopAction`:

```
string DefaultStopAction =
    "Close(-1); FreeAllTempBuffers();
        Message(\"CL macro stopped\",max(1,pause/2))";
```

This will close all manually opened files, delete all temporary buffers and show the message `"CL macro stopped"` for a short time.

The user should not change this default action when programming own macros. Instead the predefined CL-variable `StopAction` can be used like

```
StopAction = "MyOwnStopAction();" + StopAction;
```

Now at the end of a macro run the user defined macro `MyOwnStopAction()` is executed before the default stop action. **DaVis** deletes the `StopAction` string at every macro start, so the user has to redefine the value at every macro start, if an action should be executed.

The strings `DefaultErrorAction` and `ErrorAction` are working the same way as the stop actions but are executed when an error occures and before the **Error Dialog** appeares.

For easier access to both action strings the following macros can be used to add a new command as first action.

`void AddCommandToEndActions(string theCommand)`

Macro `theCommand` will be executed whenever an error occures or when the user presses the Stop button.

`void AddCommandToStopAction(string theCommand)`

Macro `theCommand` will be executed when the user presses the Stop button.

`void AddCommandToErrorAction(string theCommand)`

Macro `theCommand` will be executed whenever an error occures.

6.1.4 Additional Macros for Dialogs

Additional useful dialog macros are described in chapter **Additional Item Macros** on page 170.

6.1.5 Additional Mathematical Macros

Some mathematical functions are defined in macro file `cl/System/Utilities/Math.cl`. See chapter **Mathematical Functions** on page 67 for descriptions.

6.2 Macros for File and Directory Handling

6.2.1 GetTempDirectory

When buffers or other data must be temporary stored in files, the macro should not use the **DaVis** directory or a special predefined subdirectory. E.g. when the software is running on a read-only drive, no files can be stored in the **DaVis** directory. Or when **DaVis** is installed on a harddisc without much space left, the drive could run out of space and the macro breaks because no more files can be stored.

To avoid the write access problems please call macro function `GetTempDirectory()`, which returns a path for temporary files. The macro should create a subdirectory for its own files and at the end clean up the subdirectory and delete all files.

The default temporary directory can be selected in the **Global Options** dialog and is stored in variable `DefaultPathTemp`.

Of cause no guarantee is given here for the space problem. To avoid this, the macro should call `GetDriveInformation` (see page 53) with the drive name as first parameter, e.g. to get the free space on drive D:

```
GetDriveInformation("D:",2)
```

7 Dialogs

7.1 About Dialogs

Dialogs are used in custom processing operations to present a simple interface to the user to configure parameter settings.

Every dialog needs different so-called **items**: a simple text to describe e.g. an edit item and an item to edit value (numbers or strings) side by side, buttons, lists and more.

In **DaVis** the dialog handling is splitted into three parts: the initialization of variables to be used by the dialog, the creation of the dialog itself and the event handler, which reacts on user activities in the dialog.

A naming convention used for most dialogs is given in the following examples for a dialog called `Test`. The name of the dialog must be an unique string.

- `Test_SetupDialog()` is called from foreign functions, e.g. from the menu or tool bar, to display the dialog on screen. This function initialized variables for the dialog, shows the dialog on top of all windows if it is open, or creates the dialog.

- `Test_Dialog()` should be called only from `Test_SetupDialog()`. This macro creates the dialog and all items.

- `Test_Event( int theItemId )` is the standard event handler, which is called from the internal dialog handler every time the user presses a button in the dialog, selects a list item or presses the `Enter`-key at the end of a parameter input. This macro must not be implemented, because sometimes a dialog includes an OK button or an Execute button, and the buttons can be defined to execute a free CL-command as replacement of the standard event handler.

7.1.1 Dialog and Item Creation

A dialog box is created by subroutine `Dialog`, which returns an unique handle. This handle must be used to create the items of this dialog by subroutine `AddItem` (one call for each item).

Most items are connected to global CL-variables (to store user input) or to CL-commands (to execute when a button is pressed). The direct user input of parameters can be stored in variables of type string, int and float. Selections in a list are stored in integer variable with values 0,1,...(n-1) for lists with n items.

Dialog items in **DaVis** are always working on **copies** of the variables. During the creation of an item, the item reads out the value of the variable and displays the value on screen. When the user changes the value (enter a new value or select an entry of a list item), the internal value of the item is changed only! The value of the variable is not changed by this action. It will be set to the entered value with the next call to subroutine `ApplyDialog` or `ApplyItem`, or by pressing the **Apply** button.

On the other hand, if a macro changes the value of a variable, the value displayed in a connected dialog item is not changed! A macro must call the subroutines `UpdateDialog` or `UpdateItem` to copy the new value from the variable to the dialog item.

Dialog

```
int Dialog(string theDialogName, int x1, int y1,
        int width, int height, string title );
```

Define a new dialog and use the `DialogAttributes` for special behaviour or view parameters (see page 172). Return an unique integer handle used by all other dialog or item functions.

Parameter `theDialogName` must be a string without spaces or special characters. The name must be unique, it is impossible to create two dialogs with the same name. The name is also a way to retrieve the `DialogId` again later with function `GetDialogId()`.

Parameters `x1, y1, width` and `height` describe the position and size of this dialog. The position in used for the first opening of the dialog only! **DaVis** stores the position while closing a dialog and reopens at the same position.

The position is even stored at the shutdown of **DaVis** and is used again after the next start. The values are stored in textfile `DaVis-Dialog.pos` in the **DaVis** directory.

The text `title` appears in the title bar of the dialog box. If the text is too long, the last characters are cut and not displayed.

Note: A dialog cannot be opened during the startup or shutdown of **DaVis**, because the main window is not visible at that time. Please call subroutines `Message` or `Question` instead for special communication with the user.

AddItem

```
void AddItem( int theDialogId, int theItemId, int type,
        int x1, int y1, int x_width, int y_height,
        string text, string variable );
```

Add item to dialog. The item-id must be a unique number within the dialog box. The position is relative to the upper left corner of the dialog window.

In most cases a dialog box is used to enter some values and then start some action. Sometimes it is useful to change parameters while a macro is running, e.g. to change exposure time while continuous grabbing. Use a "!" as first character of `variable` to execute a command or to apply a value while a macro is running. Note not to call a function twice (a button calls the EventHandler while the EventHandler is doing some action), because **DaVis** is not able to handle recursive macros! The described function is available for item types 4, 6, 7, 8, 9, 10, 12 and 15.

In the section starting on page 156 all available items are described with respect to the parameters of subroutine `AddItem`. The `type` can be either the real constant value or a predefined CL constant like `DITEM_OK`. See macro file `CL/System/Utilities/Dialog.cl` or the following sections for a complete list.

Note: It is not useful (and may cause strange behavior) to use the same variable in more than one dialog item. After changing the value in the dialog a call to `ApplyDialog()` will set the variable to an undefined value, which is either the value of the first or of the second item! A call to `ApplyItem()` does not apply all changed valued but only the selected one.

7.1.2 Dialog Event Handler

An **event handler** macro is called by the dialog when certain items are activated. It has the name

```
Event<theDialogName>(int theItemId)
```

or since **DaVis** 7.2 the object orientated style

```
<theDialogName>_Event(int theItemId)
```

E.g. if `theDialogName` is `LaserPower`, then

```
void LaserPower_Event(int theItemId)
```

is the event handler. Parameter `theItemId` is the identifier of the activated dialog item. Such an event handler macro must be defined, if catching of those events for fancy dynamic dialog changes is needed. For simple dialogs it is not required.

The event handler is called only when certain types of dialog items are activated. Types of items supported are Check boxes, Radio button groups, List buttons, List boxes and Scrollbars. It is called with `theItemId = EVENT_RESIZE = -2` when a sizeable dialog has changed its size.

Note use function `ApplyItem()` or `ApplyDialog()` at first to save all screen settings into CL-variables before doing any further action.

The event handler is also activated when the return key is pressed. Here it is called with index `theItemId = EVENT_RETURN = -1`. This can be used when entering parameters in an edit control item, because the edit control item does not produce any event itself. If some action after entering some data is needed, press return and catch the event with the event handler (`theItemId = EVENT_RETURN`). Note that this event is not specific to a single item, so all items must be checked manually. Use attribute DLGA_EVENT to receive the correct item number.

7.2 Dialog Properties

The dialog properties are the handle, size, position and title. For changing or requesting the dialog's title please use functions `Get / SetItemText`. The handle is given by the system and can not be changed by the user.

GetDialogId

```
int GetDialogId( string theDialogName );
```

Get corresponding unique id of dialog. This handle is 0 for a not existing dialog.

Get / SetDialogSize

```
void SetDialogSize (int theDialogId, int x1, int y1, int x_width, int
y_height);
```

If x1 ≤ -10000, current x-position of dialog will not be changed, only a resizing is done. Same for y1.

```
int GetDialogSize (int theDialogId, int& x1, int& y1, int& x_width,
int& y_height );
```

Get size and position of a dialog. If the dialog exists the return value is theDialogId, otherwise 0.

ShowDialog

```
void ShowDialog( int theDialogId );
```

Display and bring to front dialog.

HideDialog

```
void HideDialog( int theDialogId );
```

Apply and close dialog without deleting it.

EnableDialog

```
void EnableDialog( int theDialogId, int enable );
```

Enable or disable (`enable=0`) a dialog. A disable dialog is visible on screen but can not be used. This mode is useful during time intensive calculations or dialog updates.

CloseDialog

```
void CloseDialog( int theDialogId );
```

Apply and close dialog, then delete it.

CancelDialog

```
void CancelDialog( int theDialogId );
```

Close dialog without apply, then delete it.

ApplyDialog

```
void ApplyDialog( int theDialogId );
```

Store edited values back into CL-variables and update display.

UpdateDialog

```
void UpdateDialog( int theDialogId );
```

Reloads CL-variables and displays them on screen.

Note: All changes made by the user on screen are lost!

GetDialogStatus

```
int GetDialogStatus( string theDialogName );
```

Check if dialog is not defined (return 0) or defined and visible (1) or hidden (2). For embedded dialogs bit 2 is set (value 4 is added to the return value).

7.3 Dialog Items

DaVis supports most standard Windows® dialog item types, e.g buttons. To create a new item in an existing dialog use subroutine `AddItem`.

7.3.1 Standard Buttons

If the attribute DLGA_DEFAULTBUTTON is defined and this button is the first defined button (in the order of all `AddItem` calls), the button gets a thick black border and its command is executed when pressing the Return key.

OK button: `type = 1 = DITEM_OK`

Executing CL-command `variable` when button is pressed or `CloseDialog()` is called or the Close-item in the title bar is pressed. `variable` can be empty (= ""). Button title is `text` or "OK" if empty.

Apply button: `type = 2 = DITEM_APPLY`

Executing CL-command `variable` when button is pressed or `ApplyDialog()` if empty. Button title is `text` or "Apply" if empty.

Cancel button: `type = 3 = DITEM_CANCEL`

Executing CL-command `variable` when button is pressed or `CancelDialog()` if empty. Button title is `text` or "Cancel" if empty.

7.3.2 User Defined Button

`type = 4 = DITEM_BUTTON`

Executing CL-command `variable` when pressed. If `variable` is empty, the standard event handler is called with the identification number of this item as id-parameter `theItemId`. This button can be the default button.

Is is possible to execute the button via shortcut key, e.g. by key combination `Shift` and `F1` with

```
SetItemPar( theDialogId, theItemId, DLGPAR_SHORTCUT, 0, "s~F1" )
```

The alignment of the button's text can be changed by text "<text>\nalign=<placement>" with placement from constants `PLACE_x` (see below).

7.3.3 Native Text

`type = 5 = DITEM_TEXT`

If `x_width<0` the text is right justified. Another way to define the placement is subroutine `SetItemPar` with mode `DLGPAR_PLACEMENT` and the or'ed con-

Value	Color
0	red
1	green
2	blue
3	cyan
4	magenta
5	yellow
6	black
7	darkgray
8	gray
9	lightgray
10	white

Table 7.1: The standard dialog color table, used e.g. by text items.

stants `PLACE_LEFT`, `PLACE_HCENTER`, `PLACE_RIGHT` and the temporary unused vertical modes `PLACE_TOP`, `PLACE_VCENTER`, `PLACE_BOTTOM`.

The `variable` specifies the font as "`<font-name>, <size>, <attributes>, <color>, <background-color>, <angle>`" with

- `fontname`: T(imes), H(elvetica), C(ourier), F(ixed), M(S Sans Serif) or S(ystem), D(efault font)
- `size`: size of font in points
- `attributes`: N(one), B(old), I(talic), U(nderline) and O(utline). Old mode S(hadow) has been disabled meanwhile. A combination of this attributes is allowed.
- `color`: see table 7.1
- `background-color`: see table 7.1, additional transparent (-1) and system background color (-2)
- `angle`: rotation angle in degree

The font type and the font size can be changed later with command `SetItemPar` and modes `DLGPAR_TEXTFONT` and `DLGPAR_TEXTSIZE`. **Note:** Not all combinations of those parameters create the wanted font. Sometimes the system disables settings and creates a *best font*.

7.3.4 Native Check Box

`type = 6 = DITEM_CHECKBOX`

The checkbox uses `variable` a valid integer variable, which is either 0 (off) or 1 (checked). The event handler is called immediately.

7.3.5 Native Radio Group

`type = 7 = DITEM_RADIO`

Uses `variable` as a valid integer variable, which is 0/1/...
Parameter `text` contains the text for the different radio buttons, separated by '\n', for vertical item lines or by '\t' for a horizontal order. In both cases either the given height or width is divided into equal and equidistant spaces for all items.

By default the radio group is surrounded by a group box. This painting can be disabled by including a '\b' into the `text` parameter.

For example the text `"button0\nbutton1\nbutton2"` creates a radio group with three radio items. The event handler is called immediately after selecting an item.

It is possible to disable single items with the following command and value `set = 2*itemN+mode` and `mode=0` to disable and `mode=1` to enable the radio item:

```
SetItemPar( theDialogId, theItemId, DLGPAR_ITEM_ENABLE, set, "" )
```

7.3.6 Edit Control

`type = 8 = DITEM_EDIT`

For user input of parameters with `variable` as a valid CL-variable, either `int`, `float` or `string`. If the `variable` is of type `int` or `float`, `text` can be used as format string, see subroutine `FormatNumber` on page 61 for syntax. The event handler is called immediately when the return key is pressed.

The value can be entered in hidden mode with no displayed text, if `text="P"`. This can be used to enter a password.

A special disabling mode is available, which paints the item like the info item, makes it read only but does not replace some pixels by gray pixels:

`SetItemPar( theDialogId, theItemId, DLGPAR_ABLE_STATE, mode, "" )` with the following float values: disabled (0), enabled (1), info (2).

7.3.7 List Button

`type = 9 = DITEM_LIST_BUTTON`

Use `variable` as a valid integer variable, corresponding to the selected line (0...n). `text` contains the text for the different lines in the list, separated by '\n', **e.g.** `"line0\nline1\nline2"`. The event handler is called immediately.

Sometimes the value of `variable` should be different from the selected line index (0...n). In this case an offset can be stored in string `variable` after the variable name. E.g. "`myVariable +2`" defines the range of the selected lines as 2...(n+2).

Another possibility is a list of values for the different lines, so each selection can easily be mapped to a special value: Therefore the name of the variable must be followed by a comma-separated list of integer values, e.g. "`myVariable,5,7,2,1,9`".

7.3.8 List Box

`type = 10 = DITEM_LIST_BOX`

Same as list button, but a range of lines is visible. The event handler is called immediately. If `variable` is a valid string variable, the user can select multiple lines, which are returned in the variable as a list of the line names, separated by '\n'.

7.3.9 List Edit

`type = 11 = DITEM_LIST_EDIT`

With `variable` as a valid CL-variable, either int, float or string, which is set to the edited or selected value. `text` contains the text for the different lines of text, separated by "\n", e.g. `"line0\nline1\nline2"`. The event handler is called immediately if return is pressed or if a list item is selected. The special disabling mode of the simple **Edit Control** item is available.

7.3.10 Simple Text Editor

`type = 12 = DITEM_TEXT_EDIT`

This editor uses `variable` as a valid string CL-variable, which is set to the edited text. Parameter `text` may contain `'h'` or `'v'` to add horizontal or vertical scroll bar. The event handler is never called.

If the displayed text is larger than the edit item, the programmer can scroll the item and activate a line (use line=-1 for the last line) by a call to

```
SetItemPar( theDialogId, theItemId, DLGPAR_TEXT_LINE, line, "" )
```

The text can be entered in hidden mode, where no text is displayed, if `text` includes a `'P'`. Use this to enter a password.

7.3.11 Horizontal or Vertical Scrollbar

`type = 13 = DITEM_SCROLLBAR`

This scrollbar is attached to an edit control of type int or float. When moving the scrollbar the value in an associated edit control object is changed.
`text` must be in the format of `'flags,min,max,step'`, where `min` and `max` define the allowed range of the value, and `step` is the increment when pressing the small buttons $>$ or $<$. The following `flags` are available:

- 'h' for horizontal or 'v' for vertical scrollbar.
- 'e' call event handler each time it is pressed. (optional)
- 's' call event handler only one time at the end of scrolling. (optional)
- 'n' no update of the 'variable'/'associated edit control' during macro is running. (optional)

`variable` must specifiy the integer id-value of the associated edit control item or a valid integer variable.

7.3.12 Simple Bitmap

`type = 14 = DITEM_BITMAP`

Displays the bitmap defined by `text` as the filename (must be of type `*.bmp`).

If a second bitmap exists with name `filename_.bmp`, this bitmap will be displayed as "disabled bitmap" whenever the button is disabled. By default a disabled bitmap is dithered.

If the size of the bitmap item is set to 0 in the call of `AddItem()`, the size of the bitmap file is used as the size of the item.

7.3.13 Bitmap Button

`type = 15 = DITEM_BITMAP_BUTTON`

This button includes a bitmap instead of the text. `text` is used as filename (must be of type `bmp`) and `variable` defines the executable CL-command. If `variable` is empty, the standard event handler is called with the identification number of this item as id-parameter `theItemId`.

If a second bitmap exists with name `filename_.bmp`, this bitmap will be displayed as "disabled bitmap" whenever the button is disabled. By default a disabled bitmap is dithered.

Is is possible to execute the button via shortcut key by `SetItemPar` with mode `DLGPAR_SHORTCUT` and the key as text parameter.

7.3.14 Toggle Button

`type = 18 = DITEM_TOGGLE`

A toggle button stays pressed after releasing the left mouse button. This item can display a text or a bitmap, which is defined by parameter `text` as any text or as the filename (must have extension `bmp`).

If a second bitmap exists with name `filename_.bmp`, this bitmap will be displayed as "disabled bitmap" whenever the button is disabled. By default a disabled bitmap is dithered.

`variable` must be a valid integer CL-variable, optionally followed by `'= value'`. If no value is specified, the `value` is set to 1. When the button is pressed, the variable is set to `'value'`, when it is released, it is set to 0, and the event handler is called immediately for both actions.

Note that when other toggle buttons also reference the same variable (using different values), those buttons are dynamically updated appropriately.

Note that whenever a toggle is pressed, the associated variable is immediately updated (so that other buttons can change their state also). This is not true for most other items, where the associated variable is only updated when `ApplyItem()` or `ApplyDialog()` is executed.

The following example behaves like a simple checkbox:

```
text="toggle", variable="IsPressed=1"
```

The toggle button can show different texts for the pressed and released mode: `text="normal\npressed"`. Additionally the toggle button is able to show bitmaps:

`text="standard.bmp"` or
`text="normal.bmp\npressed.bmp"`

7.3.15 Spin Button

`type = 20 = DITEM_SPINBUTTON`

A spin button is an edit item for integer or float values with small buttons for counting up and down. The `variable` must name a valid integer variable. Use `test="min-max,<flags>"` as parameter definition and set one or more of the following flags: horizontal (`h`), vertikal (`v`), wrap (`w`), space between every third digit (`t`), buttons right to the edit item (`r`), left (`l`), hexadecimal (`x`), call event-handler every time a button is pressed (`e`), `+offset` to add or subtract the offset every time the user presses a button, `*factor` to multiply or divide the value. The default range is from 0 to 100.

When enabling the event-handler for all actions, the handler is called whenever a key is pressed in the item, e.g. to edit the number manually. The "key event" can be avoided by using one of the offset or multiply parameters.

The range can be changed later by a call to

```
SetItemText(theDialogId,theItemId,"min-max")
```

7.3.16 Slider Control

`type = 21 = DITEM_SLIDER`

A slider with optional marks above and below the slider line to define a single value (variable of type integer) or a range (variable of type string).

Text="min-max,hva<tickfreq>nre" defines the modes: h = horizontal, v = vertikal, a<tickfreq> = auto tick marks with tick frequency (optional, default=*(max-min)/10*), n = no tick marks, r = range, e = event-handler, s = only one event at the end of scrolling when the user leaves the mouse button, b = buddy (not used yet).

7.3.17 Progress Bar

type = 22 = DITEM_PROGRESS

Display a progress bar, e.g. for showing a progress when running a time intensive macro. The variable must be of type integer. The text parameter defines the display mode: Text="*min-max,hvbs*": h=horizontal, v=vertical, b=filled with blocks, s=smoothed filling, p=display percentage text above a smoothed filling

7.3.18 Group Box

type = 23 = DITEM_GROUP

A group box is a titled three-dimensional looking rectangle, which can be arranged around some dialog items. The text is the group box title.

7.3.19 Info Text

type = 25 = DITEM_INFO

A info text item looks like a disabled edit item.

If variable="" then the text is displayed and can be changed by subroutine SetItemText. If the text includes a multi line string (at least one "\n"), the info item is created with a horizontal and a vertical slider. Both sliders can be switched off during the creation with added characters "\n\\h" or "\n\\v". These characters will be removed from the text and not displayed.

If the variable is a valid variable, its value is displayed and can be changed by subroutines UpdateItem and UpdateDialog. In this case the text defines the display mode: Text="mhv" for multi line, horizontal and/or vertical scrollbar. Add a "f" to the text and this item will cut the left part of a filepath and replace it by "...", so the filename itself is visible.

The font type and the font size can be defined with command `SetItemPar` and modes `DLGPAR_TEXTFONT` and `DLGPAR_TEXTSIZE`. If the displayed text is larger than the info item, the programmer can scroll the item and activate a line (use line=-1 for the last line) by a call to

```
SetItemPar( theDialogId, theItemId, DLGPAR_TEXT_LINE, line, "" )
```

A special mode of the Info Text item is available to display the text of the Info Window: Enable this mode by command `SetItemPar(..,DLGPAR_INFO,1,"")`. To create such an item the following macro function must be called:
`void AddItemInfoWin( int theDialogId, int theItemId, int theXPos, int theYPos, int theXSize, int theYSize )`

7.3.20 Embedded Dialog

`type = 27 = DITEM_EMBEDDED`

This item includes a complete dialog with an own dialog-id and own event handler. The event handler of the parent dialog (which includes the embedded dialog) is called after every event of the embedded dialog. If the parent dialogs calls `ApplyDialog`, `ApplyItem` or the corresponding update commands, the command is executed for all items of the embedded dialog.

The embedded dialog is not able to close, cancel, hide or show itself. This actions can be executed with the item functions only (e.g. use `HideItem` instead of `HideDialog`). The close command of an OK-button (type `1 = DITEM_OK`) is executed when closing the parent dialog.

By default the embedding is done without any border surrounding the dialog. If `text="b"` a 3D-border is painted like a group box.

It is useful to create an embedded dialog, if a complete dialog should be used as a sub-dialog. With the embedded dialog no dialog must be programmed twice. The code of the original dialog can be used without changes. Of cause the dialog sizes have to be equal, or the dialog must check variable `DialogEmbedded` to resize itself.

The embedding of dialogs is very easy: At first write a macro function which creates a stand-alone dialog. Then include this macro as a CL-command in parameter `variable` of the `AddItem(..., DITEM_EMBEDDED, ...)`.

An **example** is the following embedding of the **Global Options** dialog:

```
AddItem( parentDialogId, 1001, 27, 10, 40, sizeX-25, 300,
        "b", "DialogGlobalOptions()" );
```

During the sequence, which defines the embedded dialog items, the global variable `DialogEmbedded` is set to the id of the parent dialog. Since *DaVis 6.2* a recursive embedding is allowed.

Later, e.g. during the event handler of the embedded dialog, the macro can ask the dialog for more information about the embedding. Subroutine `GetDialogStatus` returns value 5 if the dialog is embedded and displayed or value 6 if embedded but hidden. If the dialog would not be embedded, the return values would have been 1 or 2.

The parent dialog always gets an event with the item-id of the embedded dialog whenever the event handler of the embedded dialogs has been executed. Additionaly the parent dialog can get an event whenever the embedded dialog is activated by the user (e.g. click into the dialog). Therefore the parent has to execute the command `SetItemPar( theDialogId, theItemId, DLGPAR_CLICKEVENT, 1, "" )` and if a special event handler exists, the macro is called with all parameters set to 0 beside `theItemId` as id of the embedded dialog:

```
void EventClickMyName( int theItemId, int press,
        int key, int x, int y )
```

7.3.21 Changing Item Attributes

DeleteItem

```
void DeleteItem(int theDialogId, int theItemId );
```

Delete item in dialog. For card items the whole card will be deleted, the card button and all items inside this card. The other cards will be renumbered, so the value of the card's variable (see `AddItem`) will change.

ApplyItem

```
void ApplyItem(int theDialogId, int theItemId );
```

Store edited value back into CL-variable.

UpdateItem

```
void UpdateItem(int theDialogId, int theItemId );
```

Reload CL-variable and displays it on screen. Note: Previous changes made by the user on the screen are lost.

Hide / ShowItem

```
void HideItem( int theDialogId, int theItemId );
```

```
void ShowItem( int theDialogId, int theItemId );
```

Hide or show dialog item. For card items the card is opened with `ShowItem()`, a call to `HideItem()` has no effect.

Disable / EnableItem

```
void DisableItem( int theDialogId,int theItemId );
```

```
void EnableItem ( int theDialogId,int theItemId );
```

Disable or enable dialog item. No effect for card items.

Get / SetItemSize

```
void SetItemSize( int theDialogId, int theItemId, int x1, int y1, int
x_width, int y_height );
```

Set new size and position of item. If `x1`$\leq$`-10000` only a resizing is done. Same for `y1`.

```
void GetItemSize( int theDialogId, int theItemId, int& x1, int& y1,
int& x_width, int& y_heigth );
```

Get size and position of an item. If the dialog or item does not exist, all values are set to 0.

Note: Some item types change while creation, so the values may differ from the values defined by `AddItem()` and `SetItemSize()`.

Gluing of dialog items is supported via subroutines `GetItemPar / SetItemPar` (page 168) with parameter `theMode=-8`. An item with glue holds the distances to some borders of the dialog when the dialog is resized. By default

the position of the top left corner and the size are not changed during a resizing of the dialog.

Get / SetItemText

```
void SetItemText( int theDialogId, int theItemId, string text );
```

Replace item text with new text. If the item type is bitmap or bitmap button (types 14 or 15) the new bitmap is loaded. In case **theItemId=0** the dialog's title is changed.

```
string GetItemText( int theDialogId, int theItemId );
```

Returns item's text or an empty string, if item or dialog does not exist. If **theItemId=0** the dialog's title is returned.

SetItemColor

```
void SetItemColor ( int theDialogId, int theItemId,
int theBackgroundColor, int theForegroundColor );
```

Set color of item to RGB color. Works with most item types without buttons. Both colors can be set to value -2 for the system background color. Value -1 for `theBackgroundColor` does not paint a background.

The following RGB-colors are defined as constants: `COLOR_BLACK`, `COLOR_BLUE`, `COLOR_CYAN`, `COLOR_GREEN`, `COLOR_MAGENTA`, `COLOR_RED`, `COLOR_WHITE`, `COLOR_YELLOW`.

Use function `RGB(red,green,blue)` to calculate the RGB-color by its spectral parts (each= 0,..,255).

Get / SetItemPar

```
void SetItemPar( int theDialogId, int theItemId, int theMode, float
theFloatValue, string theStringValue );
```

```
int GetItemPar( int theDialogId, int theItemId, int theMode, float&
theFloatValue, string& theStringValue );
```

Set or get a parameter (float or string or both, depends on `theMode`) of a dialog item. Return 0 if ok or an error code or even a return value (depends on `theMode`).

Negative values for theMode are used for dialogs (see table 7.3.21). If theItemId=0 the parameter of the dialog itself will be set or get. The items Image (30), Profile (31) and Window (34) accept the theMode = WINPAR_x parameters as used in subroutines GetWinPar / SetWinPar. The other values for theMode are defined as constants DLGPAR_x.

theMode	**Function**
DLGPAR_STATE	Return the or'ed bits of the item state: item exists (1), item is visible (2), item is enabled (4)
DLGPAR_TEXT	Set item text
DLGPAR_TEXT_LINE	Scroll edit or info item to a line, use line -1 to scroll to the end.
DLGPAR_TEXTFONT	Set font for the edit and info item: Times (0), Helvetica (1), Courier (2), Fixed (3), System (4), MS Sans Serif (5)
DLGPAR_TEXTSIZE	Set text size for the edit and info item
DLGPAR_PLACEMENT	Define placement for text item with or'ed constants PLACE_LEFT, PLACE_HCENTER, PLACE_RIGHT and PLACE_TOP, PLACE_VCENTER, PLACE_BOTTOM
DLGPAR_TOOLTIP	Set text for tool tips and enable tool tips. Disable this information with empty text. A tool tip is a small rectangular pop-up window that displays a brief description of a command bar button's purpose. The tool tips are enabled by menu **Window – Tool Tips** and by CL-variable DefaultToolTipState. The time for the mouse cursor to stay untouched on an item until the tool tip opens is defined by CL-variable DefaultToolTipTime in tenth of seconds, or in the **Global Options** dialog.
DLGPAR_SHORTCUT	Define a shortcut key for a button item, use string value for definition.
DLGPAR_ ACTIVE_DIALOG	Returns the index of the active dialog when called with theDialogId=0 and theItemId=0.

continued on next page. . .

theMode	Function
DLGPAR_VISIBLE_ EXEC	By default all button type items will enable the STOP button in the tool bar when the button is pressed. This is useful for all time consuming actions. Therefore select items and lists don't enable the STOP button. With the parameter set to TRUE, those items can be programmed to enable the STOP button.
DLGPAR_BITMAP	Set background bitmap for dialogs (`theItemId=0`); `theStringValue` must be the name of an existing bmp-file, an empty string deletes the background bitmap
DLGPAR_ABLE_STATE	Special modes for the edit items (8 and 12): disabled (0), enabled (1), disabled but not grayed like the info item (2)
DLGPAR_GLUE	Define a glue mode, so during a resizing of the dialog the item will automatically be resized or repositioned. The distance to one or more borders of the dialog will not change. The following glue constants can be or'ed: `GLUE_STICKY_LEFT`, `GLUE_STICKY_RIGHT`, `GLUE_STICKY_TOP` and `GLUE_STICKY_BOTTOM`
DLGPAR_DLGNAME	Return the name of the dialog, only for `theItemId=0`
DLGPAR_INFO	Display info text in an info item.
DLGPAR_ITEM_ENABLE	Disable (0) or enable (1) a single radio item of a radio group: `theFloatValue = 2*itemN+enable`.

7.3.22 Additional Item Macros

The following macros and constants are defined in file `CL/System/Utilities/Dialog.cl`:

AddItemButton

```
void AddItemButton( int theDialogId, int theItemId, int theXPos, int
theYPos, int theWidth, int theHeight, string theLabel, string theCallback,
string theToolTip );
```

Create a simple button and define a tool tip text.

AddItemCheckbox

```
void AddItemCheckbox( int theDialogId, int theItemId, int theXPos,
int theYPos, int theWidth, int theHeight, string theLabel, string theVariable,
string theToolTip );
```

Create a check box item and define a tool tip text.

AddItemBitmapButton

```
void AddItemBitmapButton( int theDialogId, int theItemId, int theXPos,
int theYPos, int theWidth, int theHeight, string theFileName, string
theVariable, string theToolTip );
```

Create a bitmap button item and define a tool tip text.

AddItemLine

```
void AddItemLine( int theDialogId, int theItemId, int theYpos,
int theXwidth );
```

Creates two line items `theItemId` and `theItemId+1` as a horizontal border line, e.g. between groups of items. Both lines have different colors and look 3D-like.

SetAbleItem

```
void SetAbleItem( int set, int theDialogId, int theItemId );
```

The defined item will be enabled if `set=TRUE` or disabled if `set=FALSE`.

MoveItemOffset

```
void MoveItemOffset( int theDialogId, int theItemId,
int theXOffset, int theYOffset );
```

Moves an item without resizing.

ExistsItem

```
int ExistsItem( int theDialogId, int theItemId );
```

Return TRUE if the item exists. Internally calls `GetItemPar(..)`

GetItemState

```
int GetItemState( int theDialogId, int theItemId );
```

Returns flags of the item state: item exists (1), item is visible (2), item is enabled (4). Internally calls `GetItemPar(..)`

7.4 Special Settings

7.4.1 Attributes and Modal Dialogs

Attributes for the dialog can be defined in variable `DialogAttributes`. A call to `Dialog()` will reset `DialogAttributes` to value 0. This default value creates a dialog with the icons "minimize" and "close" at the right side of the title bar. The dialog can not be resized by the user and is not visible until a call to `ShowDialog()`.

Please use the constants of table 7.3 and defined in macro file `DialogConstants.cl`, because the values may change in further versions of **DaVis**. The constants can be or-ed to set the attributes, e.g.

```
DialogAttributes = DLGA_NOTICON | DLGA_NOTCLOSE
```

More about Modal Dialogs

A **modal dialog** gets the focus immediately after a call to `ShowDialog()` and then it's the only object in **DaVis** which can be used for inputs. All

Constant	Function
DLGA_SIZE	The dialog is sizable by the user. If the size changes, the event handler is called with parameter EVENT_RESIZE. For embedded dialogs this event is called when the master dialog executes `SetItemSize`.
DLGA_NOTCLOSE	Not close able (closeable by default), no "close" icon appearers in the titlebar.
DLGA_NOTICON	Not iconizable (iconizable by default), no "minimize" icon appearers in the titlebar.
DLGA_VISIBLE	Initially visible.
DLGA_MODAL	Modal dialog (see note below). No other attributes are allowed here. Internal adding flag for not iconizable.
DLGA_EVENT	Call the event handler with the correct id of the active item when pressing the `Return` key.
DLGA_FOCUS	Call the event handler with index EVENT_FOCUS when the dialog is focused (first click inside a non-active dialog).
DLGA_CENTER	Open the dialog at a center position, depends on the dialog size and screen size.
DLGA_OPENMOUSE	Open dialog centered to mouse position, useful for modal dialogs.
DLGA_DEFAULTBUTTON	The first button becomes the default button (gets a thick black border) and will be executed when pressing the `Return` key..
DLGA_MAXIMIZE	Maximize dialog during opening, gets not titlebar. Internally uses the DLGA_NOTCLOSE flag.

Table 7.3: Constants for `DialogAttributes`

other dialogs, image windows, tool bar icons and menus are "switched off" until the modal dialog is closed. No second call to `ShowDialog()` is allowed.

Example for a Modal Dialog

The following macro code creates a dialog with one radio group of three choices and a button with the label **Execute**. When the dialog appears on screen, it is the only usable object in **DaVis** until the user presses the button. Then the macro returns the number of the selected choice.

```
// global variable to store the choice value:
int TestChoice;

int GetRadioChoice()
{
    // Create a modal dialog and center the dialog in
    // the DaVis main window:
    DialogAttributes = DLGA_MODAL | DLGA_CENTER;
    int did = Dialog( "Test", 0,0 /*center*/, 150,110,
            "Please select!" );
    // Create the radio group with three choices:
    AddItem( did, 1, 7, 10, 10, 130, 60,
        "Choice 1 \nChoice 2 \nChoice 3", "TestChoice" );
    // This button closes the dialog automatically:
    AddItem( did, 2, 1, 30,75, 80,25, "Execute","");
    // With the following command the macro sleeps
    // and waits for a CloseDialog().
    ShowDialog(did); // SLEEP
    // The dialog is closed now and the macro runs on:
    return TestChoice; // return the choice value
}
```

8 Working with SETs

8.1 About SETs

DaVis uses **SETs** to store parameters and data files as *single objects*. For example all acquisition parameters and the acquired images are stored as a SET. Most dialogs are supporting the SET structure to make the usage of large numbers of files easier:

- The **Set view** can display a movie of a complete SET. The user can navigate within a SET as known from common video player programs. All images are displayed with the same display attributes (color palette, color mapping, ...), with the same zoom and at the same position.
- The **Project Manager** organizes large numbers of different image acquisition recordings and their processing. Based on directory structures and SETs in a hierarchical order the user gets an easy overview about the data and can easily access all levels of acquisition and processing.
- The **Processing** dialog uses a SET as source, then executes the selected function for all images, and at the end stores the results in a new SET. This SET includes the processing parameters, so no information about the user's activities are lost.

The following chapter gives some simple examples for storing, loading and working with SETs.

8.1.1 The SET file and its directory structure

Simple SET

The format of simple SETs has not changed in former versions of **DaVis**. Old SETs created by **DaVis** 7 or 8 and can be loaded into **DaVis** 10. The SET

file itself is a simple text file and lists all static CL-variables of the included groups of variable. A subdirectory of the same name as the SET file includes the image or vector files:

```
MyData.set
MyData/B01.im7
MyData/B02.im7
MyData/...
MyData/B17.im7
```

The SET file can look like:

```
// set file <E:/RGB-Bayer-bitshift=12.set> created by DaVis
#GROUP Sets
SetType = 256;
SetGroups = "";
SetTime = "Thu Jan 19 13:17:02 17";
SetComments = "";
SetStart = 1;
SetInc = 1;
SetSourceSet = "";
SetViewCallback = "";
SetLoadCallback = "";

#GROUP MyParameter
MyExposureMode = 3;
MyExposureTime = 1.5;
MyScanAngleStart = 10;
MyScanAngleEnd = 90;
MyScanAngleStep = 20;
```

This example for a SET file includes the CL-variable groups Sets with information about the set itself and about the numbers of the stored buffers. Then other groups of variables follow, in this case a group named MyParameter with some exposure and scan parameters.

The definition of this parameters (static variables) can bee seen in the following examples.

The image or vector files are ordered and named with a starting `B` for **buffer** followed by the index in the set. The example of the SET above contains 17 image files.

Sometimes optional parameters are coded into the file name in a style of name/value pairs:

```
MyData/B01_delay=0.03.im7
MyData/B02_delay=0.08.im7
```

When DaVis is loading such files, the name/value pairs are automatically converted into attributes and can be accessed later by some processing operations or by the display e.g. to show the values as overlays.

Stream SET

The stream SET came into the **DaVis** software with version 8.4. This format uses the same SET file as the simple SET but organizes the image data, scales, attributes and other information in another way. There are different files for all data types. The image data is stored in one file per camera with all images of e.g. a recording instead of one file per image. This decreases the number of files per SET and increases the writing speed dramatically. Therefore the recording dialog stores stream sets instead of simple sets.

The SET programming interface must be used to load images from such a SET (see next section below). Simple SETs can be accessed with subroutines like `LoadBuffer` but those subroutines don't work with stream SETs. Better change old code with SET access to `SET_LoadBuffer`.

All standard dialogs and functions in DaVis are able to read with SETs of all types, e.g. the processing or export dialogs. Processing results are usually stored as simple SETs and not as stream SETs.

Stream SETs can be converted to simple IM7 based SETs by using the context menu in the **Project Browser** dialog. This is useful when a set is recorded with DaVis 10 and should be loaded into a older version of DaVis (up to version 8.3) or by another software like Matlab.

8.1.2 The SET programming interface

All description and details about the SET interface can be found in macro file `CL/System/Utilities/Sets.cl`. This sections gives information about the basic functions.

SET_Create

```
int SET_Create( string p_sSetName, string p_sSetGroups, int p_eSetType,
string p_sComment, int p_nNumBuffer, int p_eMode );
```

Create a new set file with the given path and name `p_sSetName` and optional create the corresponding folder. Store all static variables of the list of groups `p_sSetGroups` in the SET file. Define the SET type as `SET_TYPE_IMAGE` or `SET_TYPE_VECTOR` or another type. A comment can be stored, too, and the number of buffers may be given, but this value is no longer used. The mode `p_eMode` is either `SET_CREATE_SUBDIR` to create the corresponding folder or `SET_CREATE_NOSUBDIR` to create the SET file only. Return 0 on success.

After creating the SET, the storage of buffers is possible.

SET_StoreBuffer

```
int SET_StoreBuffer( string p_sSetName, int p_nIndex, int p_nBuffer );
```

Store buffer number `p_nBuffer` in the given SET with file index `p_nIndex`. Return TRUE if ok, else a macro error will be displayed.

SET_GetInfo

```
void SET_GetInfo( string p_sSetName, int& p_nFiles, int& p_nMin, int& p_nMax );
```

Get information about a SET file information: number of files, minimum and maximum file number.

SET_LoadBuffer

```
int SET_LoadBuffer( string p_sSetName, int& p_nIndex, int p_nBuffer );
```

Load file with given index into a buffer. Return TRUE on success.

SET_GetSourceSet

```
string SET_GetSourceSet( string p_sSetName );
```

Get path and name of source (parent) data SET, which has been used to create this SET during processing. For example after processing with vector calculation the source set of a vector set is the image set.

SET_ReplaceVariable and SET_ReplaceGroup

```
void SET_ReplaceVariable( string p_sSetName, string p_sVariableName,
line p_pValue );
```

```
void SET_ReplaceGroup( string p_sSetName, string p_sGroupName );
```

Replace the value of a variable in the SET file without touching other variables in the file and without loading all variables into **DaVis**. This function may be used e.g. to change the type of a SET file.

Replace the values of a group of variables in the SET file by the active values of the static variables. In this function the variables of other groups are not touched in the file and not loaded into **DaVis**.

8.2 Examples of programming SETs

8.2.1 Storing and Loading a SET

The macro function `StoreMySet` gets the name of the `SET` as parameter and the number of buffers, which should be included into the `SET`. Before calling this macro, the buffers must be copied into the **DaVis** buffers 1,...,`theBuffers`. For example an image acquisition function could acquire a number of images and store them temporary in the PC's memory before writing them into a `SET` on the harddisc.

```
#GROUP MyParameter
static int MyExposureMode;
static float MyExposureTime;
static int MyScanAngleStart;
static int MyScanAngleEnd;
static int MyScanAngleStep;

void StoreMySet( string theSetName, int theBuffers )
{     // create a SET with subdirectory for image file,
    // store variables of group "MyParameter" in the SET
    SET_Create( theSetName, "MyParameter", SET_TYPE_IMAGE,
        "Comment my SET", theBuffers, SET_CREATE_SUBDIR );
    int i; // store buffer 1-5 with file index 1-5
    for ( i = 1; i <= theBuffers; i++ )
        SET_StoreBuffer( theSetName, i, i );
}
```

Later the SET information (the variables) and the images can be loaded into **DaVis** by a function like this:

```
void LoadMySet( string theSetName )
{     // load parameter from <theSetName>.set
    int nFiles, nMin, nMax; // range of image files
    SET_GetInfo( theSetName, nFiles, nMin, nMax );
    SET_LoadGroups( theSetName );
    // print some information
    InfoText( "The set included a scan from "
        + MyScanAngleStart + "° to "
        + MyScanAngleEnd + "° with stepwidth "
        + MyScanAngleStep + "° " );
    // load all image files of the set
    int fileN = 1, buf = 1;
    while ( SET_LoadBuffer(theSetName,fileN,buf) )
    {
        InfoText( "Loaded file "+fileN+" to buffer "+buf );
        buf++;
    }
}
```

9 Processing User Function

9.1 Overview

This chapter describes how to write your own User Function and execute this function with the **Processing Dialog**. There are two possibilities to do this:

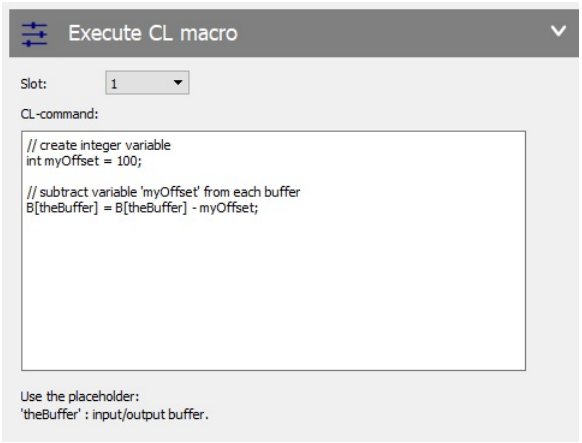
1. **Execute CL macro 1-20:** This is a very fast and easy way to write own functions, but this functions must calculate for each source buffer one result buffer. The simple function can be edited in the embedded editor. No macro file has to be changed. See page 181.

2. **Processing Function Wizard:** The **Processing Function Wizard** can be used to create your own function and add this to the standard operation groups. With this function you have full access to all possibilities of the processing, but this is the most complex way to program own functions. The generated CL-code will be added to a specified CL-file and can be edit. See page 182.

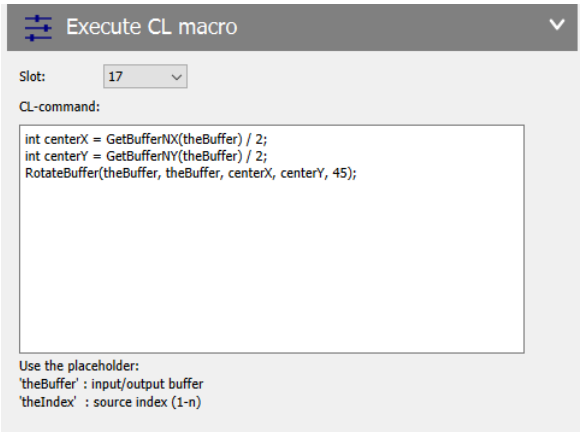
9.2 Execute CL macro 1-20

Open the **Processing** dialog from from the **Project Manager**. Then choose **Group = user functions** and **Operation = execute CL macro n**. Now activate the **Parameter** card of the function and enter your CL-commands in the text item of the dialog.

Use the placeholder `theBuffer` as the handle of the input/output buffer and `theIndex` as running index 1 to n in the source set. This index is not equal to the source file index in case of a free source range. E.g. when using range 3 to 11 with an increment of 2 then `theIndex` is counting from 1 to 5. The operation must not take the real source index into account.



This first example subtracts 200 counts from each pixel of each source buffer.

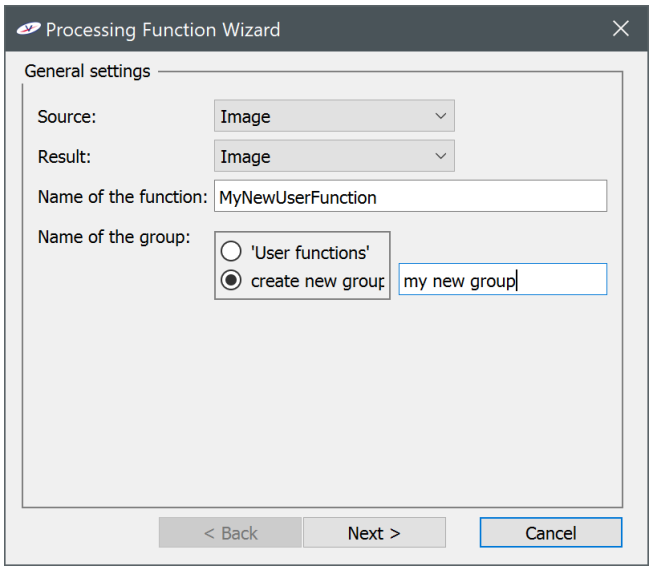


This second example rotates each each source buffer by 45 degree around the center of the image.

9.3 Processing Function Wizard

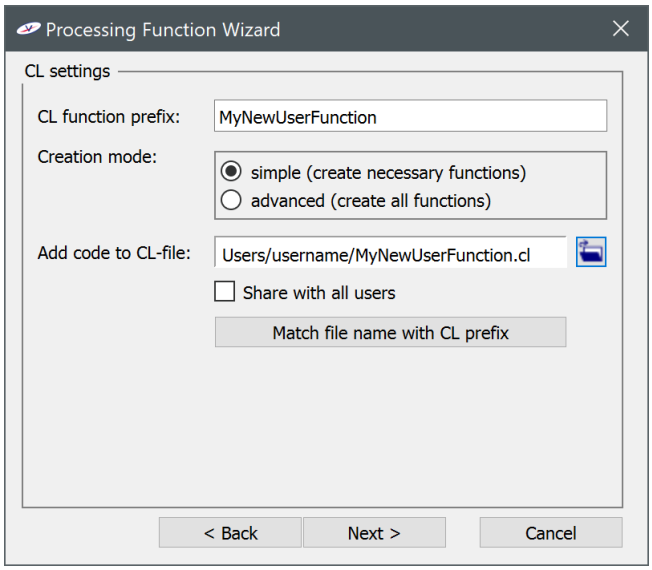
This chapter describes how to use the **Processing Function Wizard** to create a new function and add this to the standard groups/functions of the Processing dialog. The wizard can be reached from menu **Extras - Macro - Create new Processing Function**.

1. **Step: General settings**



First you have to choose if you want to create an **Image** or **Vector** function (**Function type**) and the **Result type** of this function. Then you have to enter the name of the function and the group to which the function is added (standard **user function** or create new one).

2. **Step: CL settings**



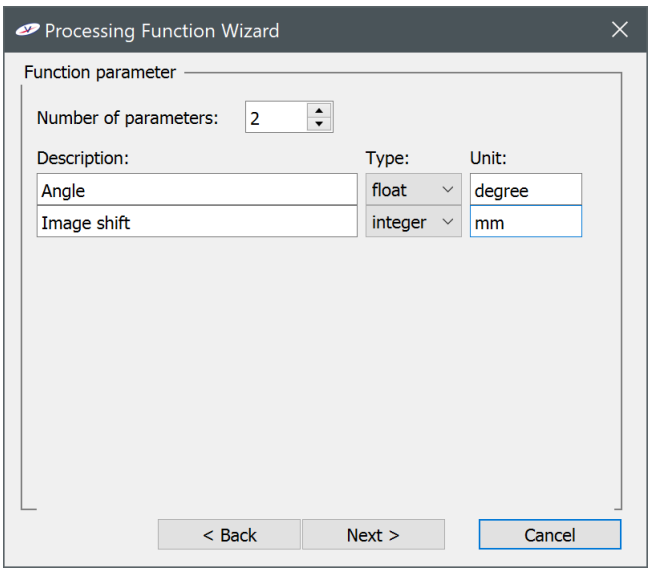
In this dialog you have to specify some CL settings:

- **CL function prefix:** A processing function consists of several callback macros with the same **function prefix** (see page 186),

for example:
MyNewImageFunction_Operation(...)
MyNewImageFunction_Dialog(...)
MyNewImageFunction_ ...
Please enter a well-defined name with at least 5 character, for example: **MyNewImageFunction**, **MyRotateFunction**...

- **Creation mode:** Choose between **simple** (creates only the important and necessary callback functions) and advanced (creates all callback functions). See page 186 for further details.
- **Add code to CL-file:** The Wizard adds the new function to this CL-file (if exists) or create a new one. **Note:** If you specify a file in the **Autostart** folder (`User/<name>/CL_Autostart`) the file is loaded automatically at every startup of **DaVis**.
- **Share with all users**: Choose between creating a user-specific CL file stored in `User/<name>/CL_Autostart` that can only be used by you, or creating a shared CL file stored in `User/All users/CL_Autostart` that can be used by everyone.
- **Match file name with CL prefix**: Click this button to use the CL function prefix as the filename for the CL file.

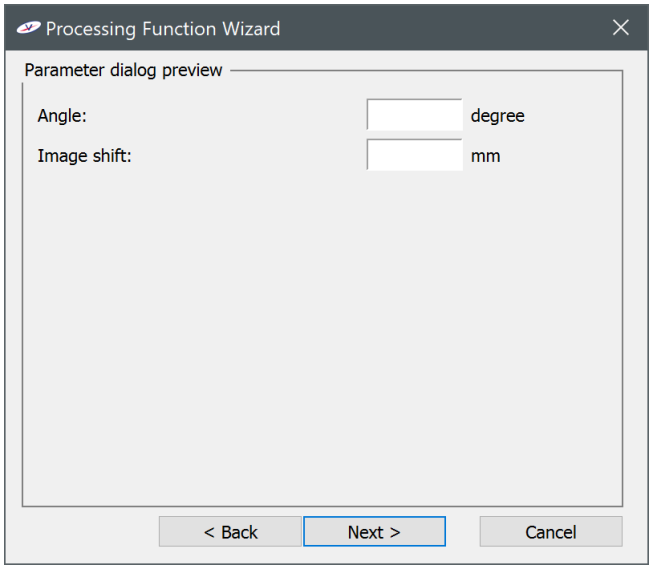
3. **Step: Function Parameter**



Now you can define up to eight function parameters with a short description, type of the parameter and unit. These parameters will be

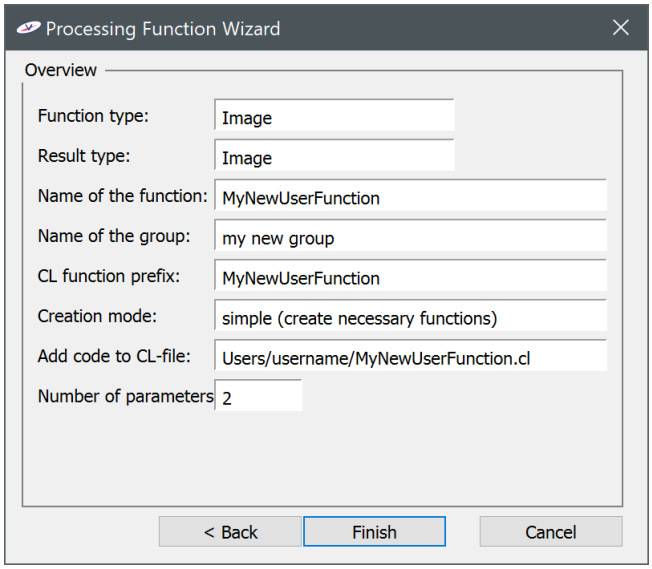
automatically added to the corresponding parameter dialog (see next step).

4. **Step: Parameter dialog preview**



This is a preview of the parameter dialog (if at least one parameter is defined).

5. **Step: Overview**



This is an overview of all entered names and values. Press **Finish** to create the CL-code of the new function in the specified CL-file. After that these CL-file is loaded into **DaVis** and the new function is avail-

able. Furthermore a text editor is started with the CL-file. Now you can edit the new **Processing** function.

9.3.1 Callback functions

The following list shows all available callback macros and their description. Replace <thePrefix> with a well-defined function name. This prefix must be used in the BP_AddUserFunction(..) to add your function to the standard **Processing** operation groups (see page 188 for further details).

```
// ============================================================
int <thePrefix>_ResultType( int theSourceType )
// Return the result type of the operation
// theSourceType = IBP_SRC_IMAGE or IBP_SRC_VECTOR
// Return one of the constants IBP_RESULT_IMG or IBP_RESULT_VEC

string <thePrefix>_GetDefaultFileOrSetName()
// Return the operation specific store name (for default setfiles)

int <thePrefix>_CheckSource( int& theSetFile, int& theMin,
                        int& theMax, int& theInc )
// This function is called just before the <thePrefix>_StartExecute
// macro (if data source = SET). Use this macro to check the source
// or overwrite the dialog settings min, max or increment.
// theSrcMode :  type of the current source:  IBP_IN_*
// theSetFile :  name of source setfile.
// theMin, theMax, theInc :  dialog settings (range and increment)

void <thePrefix>_Dialog()
// Open small parameter dialog.  This dialog will be embedded
// in the middle of the Processing dialog (420 x 50 pixel).

string <thePrefix>_GetInfoString()
// Return information string which is added to the '.set' file.

string <thePrefix>_GetDescription()
// Return description to be displayed in the processing dialog
// as subtitle of the user function.
```


```
// int <thePrefix>_OperationLoopAgain()
// This macro is called after every complete operation loop.
// Return value:  call operation loop again (TRUE,FALSE)

// string <thePrefix>_StoreGroups()
// Return the parameter groups of the function which are stored
// in the result dataset.

int <thePrefix>_StartExecute( int theNImages )
// This function is called just before the processing loop.
// It can be used to initialize the operation or create some temporay
// buffer.
// Return:  0 = ok, -1 = abort operation

int <thePrefix>_Operation( int theBuffer, int theIndex )
// Execute operation on theBuffer
// theIndex :  source image number (1-n)
// Return value:
// IBP_OP_STORE : ok, store buffer
// IBP_OP_NOSTORE : ok, no storage
// IBP_OP_ERROR : error, abort operation

int <thePrefix>_ResultBuffers ()
// return number of additional resulting buffers.  Default return
// value = 0

int <thePrefix>_EndExecute( int theFirstResultingBuffer,
                        int theNImages )
// When calculating additional results:  copy then to theBuffer,
// theBuffer+1, ...
// theNImages :  number of additional resulting buffers
// return value 0 :  ok
```

```
// return value -1 :  error
// ============================================================
```

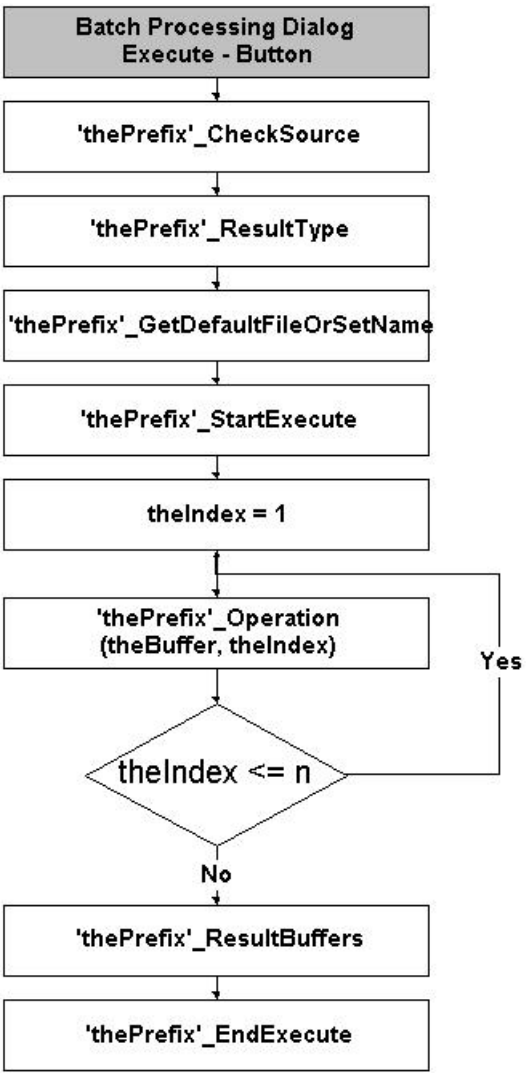
9.3.2 Add User Function

Use the following macro to add your new function to the standard groups/functions of the **Processing** dialog.

```
// ============================================================
int BP_AddUserFunction( int theSrcType,
            string theGroupName,
            string theFuncName,
            string thePrefix)
// Add a new function the the Processing dialog
// theSrcType:  source type of the new function:
//         IBP_SRC_IMAGE or IBP_SRC_VECTOR
// theGroupName:  name of the new group
// theFuncName:  name of the new function
// thePrefix:  function prefix
// Return value:  function handle
// ============================================================
```

9.3.3 Callback flowchart

The following picture shows the flowchart of a **Processing** operation:



A Appendix

A.1 Start Options

When started from command shell or from a link on the desktop, some command line options can be given to **DaVis**. More commands for **Distributed Processing** and **Cluster Script Processing** and **Command Line Processing** are given in manual **1003013_DistributedProcessing_D10.1.pdf**.

- `<filename>.IM7`: The image or vector file is loaded at end of startup into consecutive buffers 1,2,3,... to be used by customized macros starting after the main screen.
- `<filename>.CL`: The macro file is loaded at end of startup and an optional autostart macro is executed.
- `<filename>.SET`: The variables from this SET file are loaded at beginning of startup. No image or vector files are loaded.
- `-nosplash`: During startup a simple start dialog is displayed instead of the splash screen (see figure in chapter about startup in the DaVis software manual). This is useful when having display problems via Remote Desktop.
- `-nowindow`: Don't display startup or shutdown window.
- `-logpath=<path>`: By default the logfiles like `LOG.TXT` are stored in subfolder `Users/<name>/log` with `<name>` as the user name of the active account. When running **DaVis** on a write protected folder or e.g. to check the workers of distributed computing, this start option defined another folder to store the log files. The folder has to exist when starting **DaVis**.
- `-nocls`: Don't touch `CLS` files. Old settings are not read at startup and changed settings are not stored at shutdown. Used for testing.
- `-noruncheck`: Don't check if DaVis is already running.

- `-silent`: Modal dialogs are not waiting for user input. Used for testing.
- `-stdoff`: Disable output from `InfoText` and `StatusText` to standard console. For Linux version only.
- `-exec=<macro command>`: A macro command can be executed at the end of startup procedure, e.g. `-exec=InfoText("Here I am");`
- `-splashstatus`: Shows status text output on the splash screen during startup.

Index

Manual No 1003002 - December 13, 2022

LaVision
FOCUS ON IMAGING

LaVisionUK Ltd
2 Minton Place / Victoria Road
Bicester, Oxon, OX26 6QB / UK
www.lavisionuk.com
Email: sales@lavision.com
Tel.: +44-(0)-870-997-6532
Fax: +44-(0)-870-762-6252

LaVision GmbH
Anna-Vandenhoeck-Ring 19
D-37081 Göttingen, Germany
www.lavision.com
Email: sales@lavision.com
Tel.: +49(0)551-9004-0
Fax: +49(0)551-9004-100

LaVision, Inc.
211 W. Michigan Ave., Suite 100
Ypsilanti, MI 48197, USA
www.lavisioninc.com
Email: sales@lavisioninc.com
Phone: +1(0)734-485-0913
Fax: +1(0)240-465-4306