analytikjena

# WinASPECT / WinASPECT PLUS

## Method Programming Language

analytikjena

Service: Analytik Jena AG
Customer Services
Konrad-Zuse-Str. 1
07745 Jena

Germany

Phone: Hotline: + 49 (0) 3641 / 77-7407
Fax: + 49 (0) 3641 / 77-7449
Email: service@analytik-jena.de

General information about **Analytik Jena AG**
on the internet: http://www.analytik-jena.de

**Copyrights and Trademarks**



Documentation number: 211:413.23

Edition – February 2011
Implementation of the Technical Documentation:
Analytik Jena AG

# Contents

## Contents

# 1 Method programming for WinASPECT

The macro-programmable software offers the following options:

- Automated scope of basic standard software features for measurement and evaluation
- Extended programming language capabilities (logical branching, variables, etc.)
- Resources extended by special-purpose command groups (I/O, reporting, application, special mathematical evaluation procedures, etc.)

## 1.1 Editing a method

A previously created method file can be edited using an ASCII editor, i.e., an editor which does not insert any formatting or print control characters in text to be edited. The Windows Notepad provides a suitable editor tool for this purpose.

During the edit action, you must observe the structure of a given method file.

Working in WinASPECT® you can use the **Methods / Edit Methods** menu command to open an editor window that allows you to edit several source text files and to translate and launch a desired method for each of these files.

### Structure of a method file

- A method command must have the correct syntax.
- Each method command must be terminated by semicolon.
- Only valid ASCII characters may be used. (Refrain from using special graphic characters of any kind.)

## 1.2 Triggering the execution of a method

A method can be integrated into, and launched from, the Methods menu.

### Creating file links in the Methods menu

1. Use the **Methods / Edit Menu List / Add Method** menu command to open the **Open** standard dialog screen.
2. Select the required method files, multiple selection of a given file is possible. Then confirm your selection with **[OK]**.

For the method files you have selected, links will be created in the **Methods** menu.

## Removing file links from the Methods menu

1. Use the **Methods / Edit Menu List / Remove Method** menu command to call up a selection list of currently valid method links.
2. Mark a link you want to remove and confirm your removal selection with **[OK]**.

If a command is executed incorrectly, the given method sequence will be interrupted with an error message.

# 2 Programming language capabilities

## 2.1 Comments

Each macro line which has an exclamation mark at the beginning will be interpreted as comment and prevented from execution.

## 2.2 Spectrum memory and overlay objects

Internal memory units (buffers) are available for storing spectra.

An Overlay command is provided for showing spectra in overlay view mode.

## 2.3 Variables

There is an option to create variables within a given macro, assign values to these variables or use them in certain terms or when parameters need to be replaced in function calls.

### 2.3.1 Definition of a variable

A variable can be defined by assigning a value. In addition, a unique name must be defined. A variable is both defined and set by way of an initial assignment. It can then be newly set with each further assignment.

### 2.3.2 Syntax of a variable name

A variable name must begin with a letter. Besides letters, it may also contain underscores and numeric characters.

E.g.,: t_meas_number1 = 13;
Incorrect: _1x, 1x; Correct: x1

### 2.3.3 Syntax of a value assignment

A value assignment begins with a variable name which is followed by an equal sign and the value to be assigned. It must be terminated with a semicolon.

E.g.,: varia = 12;

## 2.3.4 Variable types

A difference is made between floating point numbers, integers and string values. Boolean values are interpreted as a partial quantity of the integers.

A variable type is determined by its first assignment.

**Note**:
It is currently not possible to use integers and floating point numbers in an assignment or a term.

**Note**:
In the case of the functions it must be considered whether the numbers to be transferred are integers or –floating point numbers.

| | |
|---|---|
| Integer: | varia = 1; |
| Floating point number | varia = 1.0; |
| String variable | varia = 'ABCD'; |

## 2.3.5 Terms

A term may be composed of constants, previously defined variables, parentheses, unary and binary operators and function calls.

It is possible to assign a term to a variable. Furthermore, a term can be used as a conditional term within a conditional statement.

| | |
|---|---|
| *unary arithmetic operators*: | + \| - |
| *binary arithmetic operators*: | + \| - \| * \| / \| mod |
| *comparison operators*: | < \| > \| <= \| >= \| == \| <> |
| *unary logical operators*: | not |
| *binary logical operators*: | and \| or |
| *parentheses*: | ( \| ) |

E.g.,:
```
varia1 = varia1 * (2 + Cos(7.8));
if (not ((varia1 >= 2) and (3 + varia1 < 7)))
varia1 = 56.0;
```

## 2.3.6 Operators

### Unary arithmetic operators

+ and - prefix operators

(+ only allowed for numerical variable or numerical terms, - also allowed for string variable or string terms)

### Binary arithmetic operators

| | |
|---|---|
| + | Adds two numerical variables or two terms or concatenates two character strings or one character string and one number |
| - | Subtracts two variables or two terms |
| * | Multiplies two variables or two terms |
| / | Divides two numerical variables or two terms with at least one string operand: The result will consist of all left-operand characters before the first right operand occurring in the left operand.<br>In the case of integer division, a / b = q, when a = q * b + r and 0 ≤ r < b. |
| mod | Computes the remainder of two numerical variables or two terms (only makes sense for integer numbers) or with at least one string operand: The result will consist of all left-operand characters after the first right operand occurring in the left operand. |

The standard mathematical rules regarding the priority of operators are applied. In case of doubt, you should use parentheses.

| | | |
|---|---|---|
| E.g.,: | varia1 = 1 + 2 * 3; | corresponds to: varia1 = 7; |
| | varia2 = 1 + '1'; | corresponds to: varia2 = '11'; |
| | varia3 = 6.0 / 4.0; | corresponds to: varia3 = 1.5; |
| | varia4 = 6 mod 4; | corresponds to: varia4 = 2; |
| | varia5 = '6 of 49' / 49; | corresponds to: varia5 = '6 of '; |
| | varia6 = '6 of 49' mod 49; | corresponds to: varia6 = ''; |
| | varia7 = seven mod b; | corresponds to: varia7 = 'en'; |
| | varia8 = '123' - 12; | corresponds to: varia8 = 111; |
| | varia9 = 6 / 4 | corresponds to: varia9 = 1; |

### Comparison operators

The result may take on a True or False logical value. If both operands are of numeric type, the result will be determined by way of comparison of numbers.

If both operators are of string type, a lexical comparison will be performed.

E.g.,: (AToI('11') <> 12) contains the logical value True

('123' > '123b') contains the logical value False

## Logical operators

Can be applied to Boolean and integer terms. For greater clarity, these should be enclosed in parentheses.

E.g.,: not (('a'<'b') and (1<>2)) corresponds to (('a'>='b') or (1==2)) and contains the logical value False.

## Shift operators

Shl and Shr are shift operations. The following rules apply:

$$n\ shl\ k\ = 2^k \times n$$

$$n\ shr\ k = n \div 2^k$$

E.g.,: 1 shl 4 = 16

3 shl 1 = 6

15 shr 1 = 7

0 shr 1 = 0

The shift operations can be used to distinguish between cases:

```
L = 3;
MakeArray(B, L, 2);
Fall = 0;
for (i = 1; i <= L; i = i + 1)
{
 B[i] = YesNoMsg('B[' + i + '] =  0-Nein / 1-Ja');
 Fall = Fall + B[i] shl (i - 1);
}
case (Fall) switch
{
 0: {
      Msg('B = (0, 0, 0)');
     }
```

```
1: {
     Msg('B = (1, 0, 0)');
   }
2: {
     Msg('B = (0, 1, 0)');
   }
3: {
     Msg('B = (1, 1, 0)');
   }
4: {
     Msg('B = (0, 0, 1)');
   }
5: {
     Msg('B = (1, 0, 1)');
   }
6: {
     Msg('B = (0, 1, 1)');
   }
7: {
     Msg('B = (1, 1, 1)');
   }
}
```

### 2.3.7 Interactive assignment of variable values

There is an option to request that the value of a variable should not be provided before the execution of a macro begins:

```
varia = "Prompt[default value]";
```

Where *Prompt* represents the actual command prompt. A *default value* can be specified. It will then be displayed as a proposed value. A default value must be enclosed in square parentheses and the whole command prompt in double quotation marks. This notably implies that double quotation marks (") must not be used in any place within a given command prompt.

E.g.,: varia1 = "Threshold value input: [1.7]";

varia2 = "Enter filespec: ";

# 2.4 Fields

Fields can be created in the macro. Once a field has been created, you may access each individual element of that field using variable assignments.

2-dimensional fields can be displayed and changed interactively.

Access to a given individual field element is provided via indices (for reading and for writing). Any integer-number terms may be used to function as an index.

**Syntax:**

varia = name[index1]..[indexn];

name[index1]..[indexn] = varia;

| | |
|---|---|
| name | The variable name of the field under which the given field was created |
| index1..indexn | Indices |
| varia | The name of any other variable |

**Example:**

field = =

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

varia = field[3][2];

⇨ varia == 8

The *varia* variable is assigned the value of the *field* field, which is contained in the third row and the *second* column of the field.

Other field control commands:

ArrayToSpec

CopyArray

InputArray

MakeArray

OutputArray

SpecToArray

## 2.5 Management of the spectrum buffers

Many macro commands require as parameters indices which refer to datasets in the spectrum memory. A spectrum memory is a related memory area for the storage of spectrum datasets without gaps. A spectrum dataset is also defined as such when only measured values exist for fixed wavelengths (*fixed wavelengths* measurement mode). The requirement that the memory area has a simple relation must always be met. If the memory area is cleaned, i.e., the datasets are deleted using the *Eras* command, it will be necessary to clean the entire area. If the case occurs where datasets are only partially deleted and subsequently new spectrum datasets should be saved, e.g., by loading a spectrum using the *Load* command or by measurement, it will result in error messages and serious exceptions. The macro programmer can directly affect the management of spectrum buffers via the *Load* command, by means of spectrum manipulations in the form

<Befehl> Ergebnispuffer = Argumentpuffer1,...,ArgumentpufferM;

or

<Befehl>(ArgumentenFeld, Ergebnispuffer);

via all spectrum-generating functions such as *ArrayToSpec* and also via the *Eras* deletion command.

The generally permissible situation is one in which a related memory area

(Spektrum1,...,SpektrumN)

already exists. If, under these circumstances, another spectrum is loaded using *Load*, the second parameter buffer must be set to N + 1.

If a manipulation command in the form

<Befehl> Ergebnispuffer = Argumentpuffer1,...,ArgumentpufferM;

or

<Befehl>(ArgumentenFeld, Ergebnispuffer);

is called, then *result buffer* = N + 1 must apply.

If *ArrayToSpec* is called, the second parameter *Puffer* must be set to N + 1. If in general a spectrum-generating command is called, this always requires a parameter which must specify the location where the generated spectrum should be saved. This parameter must be set to N + 1!

If spectra are to be deleted, all N beginning with 1 must be deleted in ascending order:

```
for (puffer = 1; puffer <= N; puffer = puffer + 1)
{
    Eras(puffer);
}
```

Before:

(Spektrum1,...,SpektrumN)

After:

()

If you use the *Meas(ParameterFile, 6)* command to perform a measurement with multiple sample accessories, a spectrum dataset is created for each sample. If n samples are measured, n spectrum datasets will exist accordingly: (sample[1] ,...,sample[n]) (spectrum[1],...,spectrum[n]) i.e., each time the Meas command is called, n spectrum datasets are created. If you delete these n datasets after processing the data, the positions 1...n will be occupied again the next time the command is called. If you do not do this and call the Meas command again, the positions n + 1...2n will be occupied etc. This enables the following alternatives when m * n samples should be measured (provided n samples are measured in one session each time and the first sample is not a reference sample.):

```
ProbenProWechsler = n;
Wiederholungen = m;
for (messung = 1; messung <= m; messung = messung + 1)
{
    Mess(ParameterDatei, 6);
    <Datenverarbeitung>
    for (puffer = 1; puffer <= n; puffer = puffer + 1)
    { Eras(puffer);
    }
}
```

or

```
for (messung = 1; messung <= m; messung = messung + 1)
{
    Mess(ParameterDatei, 6);
    <Datenverarbeitung>
} for (puffer = 1; puffer <= m * n; puffer = puffer + 1)
{
    Eras(puffer);
}
```

If, on the other hand, you use *MeasAccPos*, only one buffer is needed as only one sample is ever measured. This buffer can be released again immediately after the data has been processed. It is assumed that the spectrum memory is empty before the measurements:

```
for (messung = 1; messung <= m * n; messung = messung + 1)
{
    Weiter = Mess(ErgebnisExistiert, messung, 0, False);
    <Datenverarbeitung>
    Eras(1);
```

```
}
```

## 2.6 Conditional statements and the case statement

### 2.6.1 if statement

The execution of a statement can be made contingent on the accuracy of certain conditions:

**Syntax**

```
if (conditional term1)
{
    Statement block1
    }
    else if (conditional term2)
    {
    Statement block2
}
```

The conditional term is subject to evaluation, which needs to be enclosed in parentheses for this purpose. If the conditional term contains the logical value True, the statement block following it will be executed (→ "Terms", p.6).

If this is not the case, statement block1 will be skipped. If an else-branch follows, the statement block2 of the else-branch will be executed.

**Examples**

```
if (YesNoMsg('Save spectrum?'))
Save('C:\Programme\WinASPECT\Data\spec.dat', A);

if (YesNoMsg('Save spectrum?'))
{
    Save( 'C:\Programme\WinASPECT\Data\spec.dat', A);
    B = True;
}

if (YesNoMsg('Save spectrum?'))
{
    Save('C:\Programme\WinASPECT\Data\spec.dat', A);
```

```
    B = True;
}
else
    B = False;

if (YesNoMsg('Save spectrum?'))
    Save('C:\Programme\WinASPECT\Data\spec.dat', A);
else if (A = 1)
    B = False;
```

## 2.6.2 case statement

Where the execution of certain statements depends on the value of a discrete variable, working with a case statement may provide a helpful implementation tool.

**Syntax:**

```
case  (integer term/Boolean term) switch
{
    Val1: Statement block1
    ...
    ValN: Statement blockN
}
```

**Example**

```
i = SelMenu('Select', 'Measure', 'Open', 'Save');
case (i) switch
{
 1: Mess(paramfile, 6);
 2: filename = SelFile('Select data file',*.dat);
 3: Save(filename, 1);
}
```

Explanatory notes:

The i-variable may take on only these four values:

1, if the user selects "Measure" in the selection menu

2, if the user selects "Open" in the selection menu

3, if the user selects "Save" in the selection menu

0, if the user selects "Cancel" in the selection dialog

Depending on the value of i, the sequence of statements following the colon after the respective value will be carried out. As zero does not occur, the case block will be skipped.

## 2.7 Loop and branch commands

### 2.7.1 Branch instructions

As part of a macro sequence, branches may be performed. Branching requires that branching marks have been defined.

For greater convenience, a branching mark should be located in a single macro line (of its own). It must be given a unique name. A branching mark must be enclosed in ':' and ';':

**Syntax**

```
:Label_Name;
```

The actual branch is triggered by a goto instruction with the subsequent branch mark name. The goto instruction must be terminated with semicolon:

**Syntax**

```
goto Label_Name;
```

**Example**

```
! Organization of a loop which is carried out ten times.
startindex   = 1;
endindex     = 10;
schrittweite = 1;
:LOOP_BEG;
        ! This may be followed by instructions of any kind
        ...
        startindex = startindex + schrittweite;
        if (startindex <= endindex) goto LOOP_BEG;
! END_LOOP;
```

## 2.7.2 Loop commands

Typically, loop setups are required to implement a great number of instruction block repetitions. An important criterion with such loop setups is a defined truncation point in order to make sure that a given loop will definitely be exited on completion of a finite and defined number of loop runs. In terms of language definition, there are two types of loops:

### For loops

**Syntax**

```
for (initialization statement; conditional term; iteration statement)
{
Statement block
}
```

**Example**

```
MakeArray(Sqrs, 10, 0);
MakeArray(Sqrts, 10, 0);
for (I = 1; I <= 10; I = I + 1)
{
    Sqrs[I]  = I * I;
    Sqrts[I] = Sqrt(I);
}
for (I = 1; I mod > 0; I = I + 2)
    Sqrs[I] = Sqrs[I] * I;
```

Explanatory note

Two fields named *Sqrs* and *Sqrts* are first created with a length of 10. In a next step, the loop run variable I is initialized in the for-loop head with a value of 1. A truncation condition is specified hereafter. If this condition is found to be met, the loop will be passed once. In the example the two assignments are carried out, the iteration statement is then carried out. In the example the run variable I is incremented by one. This is followed by another check whether the condition is met for a further run-through or whether the loop should be exited. In the example the program checks whether the run variable, after it was incremented by one, is still smaller than 10. If this condition is not met, the first statement after the closing curly parenthesis is executed. It is important to ensure that (1) the condition can be met at least once and (2) the loop ends. A loop that will not be carried out is redundant. An endlessly running loop will completely block the executing application. Hence, it represents the more serious programming error.

**Example of violation of rule 1**

```
for (i = 1; i <= -1; i = i + 1)
{
! Will never be performed, because the starting value of I fails to meet this condition
}
```

**Example of violation of rule 2**

```
n = 2;
for (i = 1; i < n; i = i + 1)
n = n + 1;
```

## While loops

**Syntax:**

```
while (condition)
{
  Statement block
}
```

**Example**

```
I = 0;
while ((I < 5) and (A[I] <> Key))
 I = I + 1;
```

Explanatory notes:

A while-loop is equivalent to a for-loop. If its specified condition is met, the next instruction will be carried out (I incremented by 1 in the given case). This is followed by another check whether this condition is met by the new loop situation, etc.

**Example of redundant while-loop:**

```
I = 0;
while (I > 0)     !Condition will never be met
 I = I – 1;
```

**Example of endless loop:**

```
while (True)  !Condition will always be met
 I = I mod 10;
```

# 2.8 Subprograms – functions and procedures

## 2.8.1 Functions

The *function* key word introduces subprogram definitions with a return value.

### Syntax of the function definition

```
function <function designator> <parameter list>: <type designator>
{
 ! Normal macro source text appears here
 <Statement>
  ...
 <Statement>
 <function designator> = <term>;
}
```

The same rules apply to *function designators* as to variable names. Make sure in particular that no reserved words are used. The resulting term is assigned to the *function designator* (see example below).

**Syntax of a parameter list:**

```
<parameter list> = <empty> |
                        (<variable list>;..;<variable list>)
<variable list> =
 (<variable designator>,..,<variable designator>: <type designator>)
```

**Examples of correct parameter lists:**

```
(a: Integer)
(a, b: Integer; s: String)
```

**Examples of incorrect parameter lists**

```
(a; b: Integer; s: String)
(a, b: Integer, s: String)
(a, b: ; s: String)
```

**Examples of function definitions:**

```
! Calculates the sum of a, b. For illustration only,
! Use the "+" operator


function Addiere(a: Integer; b: Integer): Integer;
{
    Addiere = a + b;
}


! Calculates a mod b correctly, if a, b >= 0. For illustration only,
! Use the "mod" operator
! Observe the alternative parameter lists
! (a: Integer; b: Integer) vs. (a, b: Integer) (← shorter)


function Modulo(a, b: Integer): Integer;
{
while (a > b)
  {
    a = a – b;
  }
Modulo = a;
}


! Calculates b^e correctly, if e >= 0. For illustration only,
! Use the "Pow" command


function IntPower(b: Float; e: Integer): Float;
{
  IntPower = 1.0;
  while (e > 0)
  {
      IntPower = IntPower * b;
      e = e – 1;
  }
}
```

Function definitions belong in the macro header. When all the necessary subprograms have been programmed, the main program is compiled, from which the subprograms can be called.

**Example of a small main program:**

```
x = 20;
Summe = Addiere(2x + 1, x * x);
Summe = Modulo(Summe, Summe – x);
y = Modulo(Addiere(Summe, x), x);
Potenz = IntPower(Summe + 1.2345, y);
```

The parameter list can also be empty. In this case the function definition looks as follows:

```
function MMDDYYDatum: String;
{
   DDMMYYYY = Date;
   MMDDYY = Mid(DDMMYYYY, 3, 2) + '/'
        Left(DDMMYYYY, 2) + '/'
        Right(DDMMYYYY, 2);
}
```

The call in the main program is then as follows:

```
MMDDYY = MMDDYYDatum;
```

Use subprograms where a calculation is repeated at several locations in the macro – this saves space and increases clarity. Make sure that no type conflicts occur. The variable type and the function result type must be identical!

The following example demonstrates an error with type conflict:

```
N = 10;
…
N = IntPower(3.4, 2);
```

The first assignment defines N as an integer variable. Later N is assigned the result of a floating-point-valued function. By contrast, the following would be correct:

```
N = 10.0;
…
N = IntPower(3.4, 2);
```

Such an error only occurs during the runtime, and thus remains unnoticed during conversion.

## 2.8.2 Procedures

The *procedure* key word introduces subprogram definitions without a return value.

**Syntax:**

```
procedure <procedure designator> <parameter list>
{
   <Statement>
    ...
   <Statement>
}
```

The same rules apply to *procedure designators* as to variable names. Make sure in particular that no reserved words are used. The same rules apply to the parameter list as to functions.

**Examples of a procedure definition:**

```
! Measures the positions 1 to N
procedure MessePos1BisN(N: Integer);
{
    for (i = 1; i <= N; i = i + 1)
    {
     LOOP = MeasAccPos(ResultExists, i, 0, False);
    }
}
procedure Speichere;
{
    DokumenteVerzeichnis = ReadRegStr('', 'PathData');
    DatDateiPfad = DokumenteVerzeichnis + '\Methods\Data';
    Datum = Date;
    DatDateiName = DatDateiPfad + '\' + Left(Datum, 2) +
                Mid(Datum, 4, 2) + Right(Datum, 2) + '.dat';
    Save(DatDateiName, 1);
}
```

Like function definitions, procedure definitions also belong in the macro header. Here an example of a main program with function and procedure calls:

```
x = 20;
Summe = Addiere(2x + 1, x * x);
Summe = Modulo(Summe, Summe – x);
MessePos1Bis10(Summe);
Speichere;
for (i = 1; i <= Summe; i = i + 1)
{
    Eras(i);
}
```

Use procedures when certain procedures are repeated at several locations in the macro – this saves space and increases clarity.

## 2.9 The Finally block

**Syntax**: Finally;
! Position finalization code here

The *Finally* command introduces the finalization block. The instructions following Finally are also carried out if the macro is exited prematurely, e.g., due to an error, but only if the error does not cause a system crash. If, e.g., you created a temporary file called Temp.dat, you can write:

```
Finally;
Remove(Temp.dat);
```

The Finally block should appear at the end of the macro because all instructions after *Finally* are carried out as soon as the macro is ended. Several Finally blocks are not provided.

If a line is marked with a reserved "finally" label, then all subsequent instructions will be carried out even though a running method may have been stopped by error or following a standard user break. This ensures that necessary clearance actions will be taken once a method execution sequence has reached its final stage.

**Example**

```
finally
Remove(FileCopy);
Eras(1);
```

# 3 Overview of commands

## Data file handling



## File management



## Registration

## Reading and writing ini files/parameter files



## Unary mathematical spectrum operations



## Binary mathematical spectrum operations



## Other mathematical operations

## Spectrum manipulation



## Color evaluation commands (only available with color software)

## Layer thickness calculation

| | | |
|---|---|---|
| Thick | Calculates the thickness of a thin layer | p. 83 |

## Display modes

| | | |
|---|---|---|
| ASca | Autoscaling of abscissa and ordinate with representation | p. 34 |
| Digi | Digit | p. 46 |
| Mran | Same as ASca | p. 62 |
| Over | Overlay representation (2 spectra) | p. 63 |
| Overlay | Multiple-overlay representation (unlimited number of spectra) | p. 64 |
| OverlayZ | Spectrum overlay | p. 64 |
| Para | Shows the relevant parameters of a spectrum | p. 66 |
| Rev | Reverses the abscissa graph | p. 73 |
| Scale | Same as Zoom | p. 74 |
| Tab | Table with measured values (interactive) | p. 82 |
| Tile | Shows all the spectra currently loaded | p. 86 |
| Win1 | Shows one spectrum | p. 88 |
| Win2 | Shows two spectra | p. 89 |
| Win3 | Shows three spectra | p. 89 |
| Zoom | Zoom function with representation | p. 90 |

## Tools for the spectrum management of internal data memory units

| | | |
|---|---|---|
| Copy | Copies the contents of a spectrum buffer | p. 43 |
| Eras | Deletes the contents of a spectrum buffer | p. 48 |
| GetSpecNo | Returns the number of assigned spectrum buffers | p. 53 |

## Output commands

| | | |
|---|---|---|
| GetPrintItem | Determines the code number of a printing element, which is transferred to the *Print* command. | p. 53 |
| Prin | Prints the currently displayed spectra | p. 67 |
| Print | Print command of most generic type | p. 68 |
| PrintPortrait | Switches between portrait and landscape format for printing | p. 69 |

## Array commands

| | | |
|---|---|---|
| ArrayToCycle | Transfers the values stored in the field as cycle information to a cyclic spectrum | p. 34 |

| | | |
|---|---|---|
| ArrayToSpec | Creates a non-cyclic spectrum from field values | p. 34 |
| CopyArray | Copies field elements | p. 43 |
| CycleToArray | Transfers/creates a field containing the cycle information of a cyclical spectrum | p. 45 |
| InputArray | For making changes in a field in interactive mode | p. 55 |
| MakeArray | Creates a field | p. 58 |
| OutputArray | Shows a field | p. 63 |
| SpecToArray | Transfers spectrum values to a field | p. 81 |
| TextFileToArray | Creates a field in which the rows of a given text file are saved | p. 83 |

## Monitoring

| | | |
|---|---|---|
| AddDisplay | Updates the permanent display screen | p. 33 |
| CloseDisplay | Enables an existing display window | p. 37 |
| MakeDisplay | Creates a permanent display window | p. 58 |

## Menu commands

| | | |
|---|---|---|
| SelBuffer | Shows the menu for selecting spectrum buffers | p. 76 |
| SelItems | Shows a multiple-selection menu | p. 77 |
| SelMenu | Shows a single-selection menu | p. 78 |

## Protocol commands

| | | |
|---|---|---|
| ClosProt | Closes an OpenProt/CloseProt parenthesis | p. 37 |
| LdPr | Loads an encrypted protocol | p. 56 |
| OpenProt | Opens an OpenProt/CloseProt parenthesis | p. 63 |
| Prot | Creates a protocol line | p. 70 |
| PrPr | Prints a protocol file | p. 70 |
| ShPr | Shows a protocol file | p. 80 |
| SvPr | Saves a protocol file | p. 82 |

## Measuring commands

| | | |
|---|---|---|
| MeasAccPos | Performs motion to a given accessory position and triggers a measurement in that position | p. 60 |
| Mess | Triggers a measurement program | p. 60 |
| PhotoSpecial | Triggers the i-th submenu item of the variable menu section in the measurement menu | p. 67 |

## Accessory commands



## Calibration



## Character string commands



## Data processing



## Type conversion



## Other commands

| | | |
|---|---|---|
| Date | Determines the current date | p. 45 |
| Edit | Calls up "notepad.exe" | P. 47 |
| Exec | Calls up a Windows program (synchronous processing) | p. 48 |
| GetCycleNo | Determines the number of cycles in a given spectrum | p. 52 |
| GetDeviceID | Determines the device type, plus performs device initialization as appropriate | p. 52 |
| GetSpecNo | Determines the number of saved spectra | p. 53 |
| GetUserInfo | Requests user data | p. 54 |
| InitTimer | Sets a time marker for Wait commands (obsolete) | p. 54 |
| Msg | Outputs a message | p. 62 |
| SetDecimalSeparator | Sets the global decimal separator | p. 78 |
| ShFi | Shows a text file | p. 78 |
| ShowNotify | Shows a message | p. 80 |
| Sign | Signs a document | p. 81 |
| Status | Outputs a message to the status line | p. 82 |
| Time | Determines the current time | p. 86 |
| Use4 | Shows the variable assignment | p. 87 |
| Wait | Wait function | p. 88 |
| WaitEx | Wait function | p. 88 |
| WaitLoop | Wait function | p. 88 |
| YesNoMsg | Shows a decision-making question | p. 90 |

# 4 Alphabetic command reference

**Caution!**

You are not allowed to use reserved designators for your own variable names. The following example illustrates a prohibited assignment:

E.g.,: sin = Sin(X);

The previous parenthesis-free notation format of commands can be used:

E.g.,: <Y = Sin Sin X;> is equivalent to <Y = Sin(Sin(X));>

## ?

## 1Dnn

**Syntax** : 1D<nn> dest_buffer = src_buffer;

Computes the 1st derivative of *src_buffer*, relying on *nn* support points in each case, and stores it in *dest_buffer*.

*nn* may represent any of the following numbers:

5, 7, 9, 11, 13, 17, 21, 25

This command may only be applied to equidistant spectra.

**Example**

1D13 2=1;

## 2Dnn

**Syntax** : 2D<nn> dest_buffer = src_buffer;

Computes the 2nd derivative of *src_buffer*, relying on *nn* support points in each case, and stores it in *dest_buffer*.

*nn* may represent any of the following numbers:

5, 7, 9, 11, 13, 17, 21, 25

This command may only be applied to equidistant spectra.

## 3Dnn

**Syntax** : 3D<nn> dest_buffer = src_buffer;

Computes the 3rd derivative of *src_buffer*, relying on *nn* support points in each case, and stores it in *dest_buffer*.

*nn* may represent any of the following numbers:

5, 7, 9, 11, 13, 17, 21, 25

This command may only be applied to equidistant spectra.

## 4Dnn

**Syntax** : 4D<nn> dest_buffer = src_buffer;

Computes the 4th derivative of *src_buffer*, relying on nn support points in each case, and stores it in *dest_buffer*.

*nn* may represent any of the following numbers:

5, 7, 9, 11, 13, 17, 21, 25

This command may only be applied to equidistant spectra.

# A

## Abs

**Syntax** : Y = Abs(X);

Computes the absolute value of the value transmitted in *X* , where *X* may represent a randomly composed numerical term.

**Example**

z = Abs(x – y);

## AccSetPosition

**Syntax** : AccSetPosition(Pos) ;

Moves the sample accessory (e.g., cell changer, autosampler etc.) to the position specified in *Pos* (horizontal x-y movements).

*Pos* stands for any positive integer number from 1 to the accessory's maximum possible sample position.

For working with an APG 53 / 100 autosampler, the following additional values are available:

| | |
|---|---|
| AccSetPosition(0) | The autosampler moves into rinse-vessel position |
| AccSetPosition(-1) | The autosampler raises the aspiration canula out of the sample cup. (Lowering into the sample cup is automated). |

**Example:**

```
AccSetPosition(10);  //move to position 10 and lower aspiration canula
                     //into sample
Meas(SipperParamfile, 6); //measure sample;
AccSetPosition(-1);  //raise aspiration canula up to motion level again
```

## Adapt

**Syntax***:* Adapt dest_buffer = src_buffer, ref_buffer, x1, , xn;

Matches the curve run of *src_buffer* to *ref_buffer*. Parameter analog to the *Norm* command.

## Add

**Syntax:** Add dest_buffer = src_buffer1, src_buffer2;

Adds *src_buffer1* and *src_buffer2* together and saves the result in *dest_buffer*. Both spectra must match each other in terms of abscicca units and ordinate units and contain an identical number of cycles.

## AddDisplay

**Syntax:** AddDisplay(NewRow);

Adds another row to a tabular display that was created via the *MakeDisplay* command. This row must be a one-dimensional field which was created with *MakeArray* (1, column number) .

**Example:** See "MakeDisplay" p. 58

## AddK

**Syntax :** AddK dest_buffer = src_buffer, const;

Adds the *const* constant to all spectrum ordinate values of *src_buffer* and saves the result in *dest_buffer*.

## Append

**Syntax:** Append(dest_file, append_file);

Appends the *append_file* text file to the *dest_file* text file.

## ArrayToCycle

**Syntax :** ArrayToCycle (field, buffer);

Transfers the values stored in the *field* field as cycle information to the cyclic spectrum stored in *buffer*. Both the *field* field and the spectrum in *buffer* must already exist. In the *field* parameter a field should be transferred, which was created with the *CycleToArray* command on the basis of the cyclical spectrum to which the *buffer* parameter refers. The values of these field may have changed in the meantime. The values in the *field* field should be strictly monotonically increasing. The number of rows in the field must be equal to the number of cycles. The number of columns must be 1.

## ArrayToSpec

**Syntax:** ArrayToSpec (field, buffer [, col]);

Creates a non-cyclic spectrum from the values of the *field* field. The first-column values of *field* are used as abscissa values for the new spectrum. *col* indicates the number of the column in *field* (numbering begins with 1), the values of which are interpreted as the ordinate values of the new spectrum. The default value for *col* is 2. The new spectrum will be saved in *buffer*. Use an *EditDim* command to set the desired axis units.

## ASca

**Syntax:** ASca(buffer);

Designates the spectrum which is contained in *buffer*, auto-scaling of the abscissa and the ordinate is based on the minimal and the maximal spectrum values.

## AsciiExport

**Syntax:** AsciiExport(file, buffer);

Saves the spectrum that is contained in *buffer* in a file named *file* in CSV format. A semicolon is used as separator.

## AsciiImport

**Syntax:** AsciiImport(file, field, cols);

Imports the CSV file that was specified with *file*. As part of this process, a field is generated with the name indicated in *field* (not as a character string). *cols* must be used for defining a desired number of columns in a dataset. A semicolon is used as separator.

**Example**

```
AsciiImport('C:\Programme\WinASPECT\Import\Import1.csv',
        Importdaten, 5);
```

## AToF

**Syntax:** X = AToF(S)

Converts an *S* character string into a float-point value and saves this value in *X*. *S* may represent a randomly configurable character string term.

## AToI

**Syntax:** N = AToI(S)

Converts an *S* character string into an integer-number value and saves this value in *N*. *S* may represent a randomly configurable character string term.

# B

## BasI

**Syntax:** BasI dest_buffer = src_buffer[, x1, y1[,...,xn, yn]];

Defines a polygonal chain through these points: *(x1,y1), ... , (xn,yn).* This polygonal chain will represent the new baseline of the spectrum that is contained in *src_buffer*. The resulting spectrum is saved in *dest_buffer*. A maximum of twenty points may be specified. If no points were defined, the points can be selected interactively. If only a single point is transmitted, the line running through this point in a direction parallel with that of the abscissa will be regarded as the baseline.

## Beep

**Syntax:** Beep;

Sounds a short acoustic signal.

# C

## CfImport

**Syntax:** reg = CfImport(CalFileName);

Imports regression coefficients from the WinASPECT calibration file (in cf format) that was specified in the *CalFileName* parameter and stores them in the *reg* variable. This variable is addressed in exactly the same way as the function's return value *RegrKorr*.

**Example:**

```
Calfile = Selfile('Please select a calibration file...',
          cal_file_path + '*.cf');
reg = CFImport(Calfile);
```

## Clip

**Syntax:** Clip dest_buffer = src_buffer [,x1, x2 [z1, z2]];

Saves those spectrum values of *src_buffer*, which are found to be within an abscissa range of (*x1,x2*), in *dest_buffer*. If *z1* and *z2* were also transmitted, a

similar restriction will be made for (*z1, z2*) according to cycle values. If no range limits were transmitted, limits are selected interactively.

## CloseDisplay

**Syntax:** CloseDisplay;

Closes a display window that was created with *MakeDisplay*.

Since no parameters are transmitted in this case, only one display window may be created at a time. There is no option available for multiple-window display at any given time!

## CloseProt

**Syntax**: CloseProt;

Closes an *OpenProt/CloseProt* parenthesis. Within such a parenthesis, *SvPr* and *Prot* can be used to save recorded text rows in encrypted format.

It is only possible to open a protocol stored in encrypted format again using *LdPr*.

**Example**:

```
ProtFile = 'C:\Programme\WinASPECT\Methods\prot\LdPrTest.prot';

!**************************************!
!* Create encrypted protocol with two rows and save*!
!* with the OpenProt/CloseProt parenthesis    *!
!**************************************!
OpenProt;
Prot('Ich kann...');
Prot('...das nicht lesen.');
SvPr(ProtFile);
CloseProt;
!**************************************!
!* Load encrypted protocol with the OpenProt/CloseProt parenthesis *!
!**************************************!
OpenProt;
LdPr(ProtFile);
CloseProt;
```

## Col2XYZ

**Syntax:** Col2XYZ(buffer, XYZ, obs [, verteilung]);

| | |
|---|---|
| *buffer* | Measured values |
| *XYZ* | Calculated tristimulus values for X, Y, Z |
| *obs* | Viewer: 0 / 1 -> 2 degrees / 10 degrees |
| *distribution* | Spectral distribution |

Calculates the X,Y,Z tristimulus values for a spectrum in *buffer* for a viewer *obs.* The spectrum in *buffer* is required to have a spectral range not smaller than 400 nm to 750 nm. Calculation includes the values from 380 nm to 780 nm. If the spectral distribution function of the light source is transmitted, the function must be available as a ".dat" format data file with the name *distribution*. This file must contain the corresponding values within a range of 380 nm to 780 nm with a step size of 5 nm. The result will be saved in a 3x1 vector under the name *XYZ*.

If *distribution* was not transmitted, the data in *buffer* will be interpreted as the values measured for a self-luminous body.

*Note:*
The spectrum in *buffer* will be limited to a range of 380 nm to 780 nm or it will be enlarged (filled with zeros) and interpolated to a step size of 5 nm!

## ColCIEL

**Syntax:** ColCIEL(XYZ, L, light, obs);

| | |
|---|---|
| *XYZ* | Color coordinates X,Y,Z of illuminant mode *light* of viewer *obs* |
| *L* | Calculated remaining color coordinates |
| *light* | Illuminant mode: 0 / 1 / 2 -> A / C / D65 |
| *obs* | Viewer: 0 / 1 -> 2 degrees / 10 degrees |

Transmits the XYZ color coordinates as a 3x1 field. The remaining color coordinates which need to be calculated are stored as a 12x1 field under the name *L*. The following coordinates are calculated and stored in exactly this order in the field:

L, a, b, u, v, x, y, Cab, Cuv, hab, huv, Suv

## ColCount

**Syntax:** ColorCount = ColCount(buffer, type, f);

*buffer* is the parameter which contains the spectrum index of a transmission or absorption spectrum of liquid, for which a color count has to be determined. *type* designates the type of the color code to be determined. Depending on

which value was selected for *type f* (parameter) will take on a meaning as described below:

| Type value | Meaning of type | Meaning of f |
|---|---|---|
| 0 | Platinum cobalt color code | Must be transmitted, although it has no meaning (e.g., 0,0) |
| 1 | Iodine color count | Tube layer thickness in cm |
| 2 | Gardner color count | Tube layer thickness in cm |
| 3 | Permanganate index | Reference absorption value E0 (compare with ISO 8660) |

*f* parameter must be of float-point type.

**Example:**

```
PtCo = 0;
Iod  = 1;
Gardner = 2;
MnO4 = 3;
Schichtdicke = AtoF("Please define layer thickness: [1.0]");
for (probe = 1; probe <= Probenanzahl; probe = probe + 1)
{
    for (row = PtCo + 1; row <= 4; row = row + 1)
        {
        if (row < MnO4 + 1)
        {
            Farbzahlen[1][row] = ColCount(probe, row - 1, Schichtdicke);
        }
        else
        {
            Farbzahlen[1][row] = ColCount(probe, row - 1, 0.0);
        }
    }
}
```

## ColCountShow

**Syntax:** ColCountShow(Items);

Shows iodine and Gardner color counts for samples defined via *Items* as numerical values and RGB colors.
The *Items* field must have two columns. There is no restriction on the number of rows. The first column contains the various sample names and the second column their assigned spectrum indices as character strings. Only continuous transmission or absorption spectra are allowed.

**Example:**

```
MakeArray(Items, Probenanzahl, 2, 1);
for (probe = 1; probe <= Probenanzahl; probe = probe + 1)
{
    Items[probe][1] = Probennamen[probe][1];
    Items[probe][2] = '' + probe;
}
ColCountShow(Items);
```

## ColDiff

**Syntax:** ColDiff(L1, L2, diff);

| | |
|---|---|
| *L1* | Color coordinates of 1st sample |
| *L2* | Color coordinates of 2nd sample |
| *diff* | Color differences Eab, Hab, Euv, Huv |

Calculates the color differences of two samples, using lab and luv values as input. *L1* and *L2* are two 5x1 vectors which must contain the following values in this order:

L, a, b, u, v

Once calculated, the color differences are stored as 4x1 vector under the name *diff*. They contain the following values in this order:

ΔEab, ΔHab, ΔEuv, ΔHuv

For *L1* and *L2* you may use the vectors that were created with a *ColCIEL* command (irrelevant values will be ignored).

## ColMetamerie

**Syntax:** ColMetamerie(LProbePrim, LProbeSek, LRefPrim, LRefSek, index);

| | |
|---|---|
| *LProbPrim* | Lab values of sample in primary light |
| *LProbeSek* | Lab values of sample in secondary light |
| *LRefPrim* | Lab values of reference sample in primary light |
| *LRefSek* | Lab values of reference sample in secondary light |
| *index* | Metamerism index |

Calculates the metamerism index of two samples and saves the result under the name *index*.

*LProbPrim*, *LProbSek*, *LRefPrim* and *LRefSek* represent 3x1 vectors with the corresponding lab values.

For the *lab vectors*, you may use the vectors that were created using a ColCIEL command (irrelevant values will be ignored).

## ColShow

**Syntax:** ColShow(light, obs, Items);

This window shows the color locations of sample data that was transmitted via Items in an xy-color space view. The *light*, *obs* parameters code the illuminant mode and the viewer angle. Compare with help for the *ColCIEL* command. The transfer modalities for the *Items* field also apply to *ColCountShow*. Even the meaning of this parameter is identical.

## ColWY

**Syntax:** ColWY(Y, Z, white, yellow);

| | |
|---|---|
| *Y* | Y-value (for 2-degree viewer, illuminant mode C) |
| *Z* | Z-value (for 2-degree viewer, illuminant mode C) |
| *white* | Calculated white portion |
| *yellow* | Calculated yellow portion |

Saves the calculated white and yellow portions under the name *white* and *yellow* respectively.

## ColXYZ

**Syntax:** ColXYZ(buffer, XYZ, light, obs);

| | |
|---|---|
| *buffer* | Spectrum values |
| *XYZ* | Calculated X, Y, Z tristimulus values |
| *light* | Illuminant mode: 0 / 1 / 2 -> A / C / D65 |
| *obs* | Viewer: 0 / 1 -> 2 degrees / 10 degrees |

Calculates the X,Y,Z tristimulus values for a spectrum contained in *buffer* for a *light* illuminant mode and an *obs* viewer. The spectrum in *buffer* must be a transmission-type or reflection-type spectrum (%T or %R) with a minimal spectrum range of 400 nm to 750 nm. Calculation includes the values from 380 nm to 780 nm. For cyclic spectra, only the tristimulus values of the first cycle are calculated (there is no error message!). The result will be saved in a 3x1 vector under the name *XYZ*.

*Note:*
The spectrum in *buffer* will be limited to a range of 380 nm to 780 nm or it will be enlarged (filled with zeros) and interpolated to a step size of 5 nm!

## Compose

**Syntax:** Compose(dest_file, append_file);

Creates a cyclic spectrum file from the spectra contained in *dest_file* and *append_file*. This is accomplished by appending the measurement cycles of *append_file* to those of *dest_file*. The resulting spectrum is then saved under the name *dest_file*. The cycle number of the resulting spectrum is equal to the sum of the cycle numbers of *dest_file* and *append_file*. If *dest_file* does not exist at the moment, a Compose command is triggered. This command sequence will amount to the copying of *append_file* to *dest_file*. The spectra in *dest_file* and *append_file* are required to have identical abscissa ranges (range limits, point number, step size, orientation) and their axis designations for abscissa, ordinate and cycle must be in agreement with each other. If one of the two initial spectra is found to include no cycle values, the cycle values of the resulting spectrum will correspond to the cycle numbers (beginning with 1).

## Conn

**Syntax:** Conn dest_buffer = src_buffer1, src_buffer2 [, x];

Combines *src_buffer1* and *src_buffer2* into a new spectrum which is saved in *dest_buffer*. The spectra which are contained in *src_buffer1* and *src_buffer2* are required to have an overlapping abscissa range. Their abscissa ranges must have the same orientation. *X* must represent a value within their common abscissa range. The values of *dest_buffer* positioned on the left of *x* correspond

with *src_buffer1* and those on the right of *x* correspond with *src_buffer2*. If no x was transmitted, this abscissa value is selected interactively.

## Copy

**Syntax:** Copy dest_buffer = src_buffer;

Copies the spectrum contained in *src_buffer* to *dest_buffer*.

## CopyArray

**Syntax:** CopyArray(src_field, dest_field [, mod [, line_count [, col_count [,src_start_line [, src_start_col [, dest_start_line [,dest_start_colbuffer ]]]]]]);

Copies the field elements of *src_field* to *dest_field*. Both fields must exist. *dest_field* must provide enough space for all *src_field* field elements to be copied. The number, original and target position of the field elements to be copied can be specified in more detail:

| | |
|---|---|
| *line_count* | Number of rows to be copied (default = -1, i.e., all rows of *src_field*) |
| *col_count* | Number of columns to be copied (default = -1, i.e., all columns of *src_field*) |
| *src_start_line, src_start_col* | Starting position of the range being copied within *src_field* (default: 1, 1) |
| *dest_start_line, dest_start_col* | Starting position of the target range within *dest_field* (default: 1, 1) |

Various copying modes can be specified via *mod*:

| | |
|---|---|
| 0 | Performs simple value copying from *src_field* to *dest_field* |
| 1 | Adds the values of *src_field* to those of *dest_field* |
| 2 | Subtracts the values of *src_field* from those in *dest_field* |
| 3 | Multiplies the values of *src_field* with those of *dest_field* |

The default mode setting is 0.

## Cos

**Syntax:** y = Cos(x);

Forms the cosine of a value transmitted in *x* , where *x* may represent a randomly composed numerical term.

**Example:**

y = Cos(Cos(x));

## CosM

**Syntax:** CosM dest_buffer = src_buffer [, x1, y1 [,...,xn, yn]];

Replaces the spectrum values of the spectrum contained in *src_buffer* with a polygonal chain that runs through the points *(x1,y1), ...* , (*xn,yn)*. The resulting spectrum is saved in *dest_buffer*. Up to twenty points may be specified.

## CsvExport

**Syntax:** CsvExport(file, field);

Saves the values which are maintained in a *field* field in a file named *file* in CSV format. A semicolon is used as separator.

## CurvPara

**Syntax:** curv = CurvPara(buffer);

Once this assignment was made, you may access the following variables:

| | |
|---|---|
| curv.start | Beginning of range of measurement |
| curv.end | End of range of measurement |
| curv.step | Step size |
| curv.numb | Number of measurement points |
| curv.min | Minimum measured value |
| curv.max | Maximum measured value |
| curv.note | Remarks |
| curv.absdim | Abscissa dimension |
| curv.orddim | Ordinate dimension |
| curv.cycldim | Cycle dimension |

| | |
|---|---|
| curv.cycl | Number of cycles measured |
| curv.cyclstart | Cycle starting value |
| curv.cyclend | Cycle end value |
| curv.path | Assigned full path |
| curv.orig | Original spectrum (True or False) |
| curv.messpara | Measurement parameter file (only valid for original spectrum) |
| curv.date | Compilation date of data file |

## CycleToArray

**Syntax :** ArrayToCycle (field, buffer);

Automatically creates a field under the name field in which the cycle information of a spectrum contained in *buffer* will be saved. The number of rows in this field is equal to the number of cycles in the spectrum. The number of columns is one. It is possible to change the values in the field and to save them back into the spectrum with the help of the *ArrayToCycle* command.

# D

## Date

**Syntax:** datum = Date;

Saves the date in 'dd.mm.yy' format.

## DefaultPath

**Syntax:** oldpath = DefaultPath(Pathtype [, newpath ]);

| | |
|---|---|
| *PathType* | Type of path (default: 0) |
| 0 | -> Data path (reads data) |
| 1 | -> Data path (saves data) |
| 2 | -> Measurement parameter path |
| 3 | -> Method path |
| 4 | -> Installation directory of WinASPECT® |
| 5 | -> Data path (import) |
| 6 | -> Document path in the general (All Users) profile directory |

| | |
|---|---|
| 7 | -> Application data path in the general (All Users) profile directory |
| *newpath* | Specified path (without a closing '\') |

*Pathtype* 4, 6 and 7 are available for reading only.

If *newpath* was transmitted, it will be set. A path to be set must actually exist! *oldpath* contains the old path, an empty string in the event of an error ('').

## Digi

**Syntax:** Digi(buffer [, x1 [, ... , xn]]);

Outputs the spectrum values that relate to the abscissa values for *x1, ... , xn* in *buffer* to the display screen. If no abscissa values were transmitted, they can be selected interactively.

## DigPoint

**Syntax:** point = DigPoint(buffer, x);

Determines the value measured for the spectrum which is contained in the *buffer* data memory for a given abscissa position *x* (regarding only cycle one). The following two variables can then be accessed:

| | |
|---|---|
| point.x | Abscissa value |
| point.y | Assigned measured value |

## DigPointDec

**Syntax:** point = DigPointDec(buffer, y);

Determines the first abscissa value of a spectrum contained in the *buffer* data memory for a given ordinate position y (regarding cycle one only).

The spectrum must be monotonically decreasing, i.e., ordinate values are required to decrease or remain constant in the case of increasing abscissa values.

The following variables can then be accessed:

| | |
|---|---|
| point.x | First related abscissa value |
| point.y | Ordinate value |

## DigPointY

**Syntax:** point = DigPointY(buffer, y);

Determines the first abscissa value of a spectrum contained in the *buffer* data memory for a given ordinate position y (regarding cycle one only). The following two variables can then be accessed:

| | |
|---|---|
| point.x | First related abscissa value |
| point.y | Ordinate value |

## DivT

**Syntax:** DivT dest_buffer = src_buffer1, src_buffer2;

Divides *src_buffer1* by *src_buffer2* and saves the result in *dest_buffer*. Both spectra must match each other in terms of abscicca units and ordinate units and contain an identical number of cycles. If the initial spectra are of transmission or reflection type, the resulting values will automatically be multiplied by 100 (corresponds to subtraction of related extinction spectra). This command is not available for absorption spectra.

# E

## Edit

**Syntax:** Edit(textfile);

Calls up the 'notepad.exe' Windows editor for editing of *textfile*.

## EditDim

**Syntax:** olddim = EditDim(buf, mod [, newdim]);

For changing or reading the axis designations that belong to a spectrum contained in the *buf* spectrum memory. If *newdim* was transmitted, the new axis designation will be set. The *mod* parameter is set to 1. *newdim* may take on the following values:

| | | |
|---|---|---|
| 1 | -> | %T (transmission) |
| 2 | -> | A (absorption) |
| 3 | -> | %R (reflection) |
| 4, 5 | -> | E (emission) |

## Eras

**Syntax:** Eras(buffer);

Deletes a spectrum saved in *buffer*.

## EsmImport

**Syntax:** EsmImport(file, field1, field2);

Imports the ESM file that was specified via *file*. As part of this process, two fields with the names indicated in *field1, field2* (no character strings) are generated. Field1 contains the imported data in a table. Field2 contains the column headings of the imported table.

**Example:**

```
EsmImport('C:\EsmFiles\Import1.esm', Importdaten, Spalten, 5);
Str = Columns[2];
```

*Note:*
The *Columns* field may not be accessed other than by using **an** index as shown in row 2 of the example.

## Exec

**Syntax:** Exec(program);

Triggers a Windows *program* session. On completion of this session, the particular method will continue to be carried out.

## Exp

**Syntax:** y = Exp (x);

Applies the e-function to the value that was transmitted via *x* , where *x* may stand for any numerical term.

# F

## FCopy

**Syntax:** FCopy(src_file, dest_file);

Copies *src_file* files to *dest_file* (wildcards are allowed).

## FFirst / FNext

**Syntax:** file = FFirst(filemask);

file = FNext;

FFirst must have been called up before the first FNext. In the event of an error, “file” is an empty string. No system, directory or hidden files are returned. A returned file name will contain the full path specification.

**Example:**

Changing the integration time in several parameter files

```
DokumenteVerzeichnis = ReadRegStr('', 'PathData');
WurzelVerzeichnis = DokumenteVerzeichnis + '\Methods\Enzymatik';
ParameterdateiVerzeichnis = WurzelVerzeichnis + '\para';
WildCard = '*.par';
Parameterdatei = FFirst(ParameterdateiVerzeichnis + '\' + WildCard);
while (Parameterdatei <> '')
{
    Parameterdatei = ParameterdateiVerzeichnis + '\' +
    Parameterdatei;
    WriteProfile(Parameterdatei, 'MESS', 'INTEG_TIME', 200);
    Parameterdatei = FNext;
}
```

**Explanatory notes**:

*FFirst* is initially used to search for the first parameter file in the directory. This leads to the internal definition of the Find structure with the wildcard '*.par' and the directory as the current search directory. *FNext* then continues to search for other '.par' files and transfers them as a complete path to the *parameter file* variable until *FNext* returns an empty string. This indicates that the search for all the parameter files to be found in the search directory has been successful. The

internal Find structure is not visible and there is no option available to access it directly. Access is only possible via the *FFirst* / *FNext* functions.

## Fft

**Syntax:** Fft(RealParts, ImaginaryParts, Spectrum);

Calculates the Fourier-transformed spectrum for a given spectrum buffer specified via the corresponding index in the spectrum parameter. Since it assigns complex values to real abscicca units, these are divided into real and imaginary parts, resulting in two real spectra. Both the following should apply:

*real parts* = *spectrum* + 1 and

*imaginary parts* = *real parts* + 1.

*Note*:
*real parts* and *imaginary parts* describe spectrum buffer indices and are therefore natural numbers.

**Example**:

```
Loc = DefaultPath(4);
Pi = 3.14159265;
N = 1024;
A = 2.0;
k = 3.0;

! Create signal I, I[j] = Sqr(cos(k * x[j]))

MakeArray(I, N, 2, 0);
for (j = 0; j < N ; j = j + 1)
{
    X = j * 2 * Pi / N;
    I[j + 1][1] = X;
    I[j + 1][2] = A * cos(k * X);
}

! Save signal in spectrum buffer 1
```

```
ArrayToSpec(I, 1);

! Y-unit is Abs

olddim = EditDim(1, 1, 2);

! Transformation: real parts in buffer 2, imaginary parts in buffer 3

Fft(2, 3, 1);

! Show real parts. x = k is expected to produce value A

Win1(2);

! Empty buffers 1 to 3

Eras(1); Eras(2); Eras(3);
```

## FileAge

**Syntax:** Age = FileAge(FileName);

Returns the age of a file that was specified via *FileName* in units of a second. Age means both the time since a file was created and the time since the most recent change was made to this file. If a file was left unchanged since the time of its creation, the period of time since its creation will be returned. Otherwise, the period of time since the latest change is returned.

## FileExist

**Syntax:** exist = FileExist(FileName);

If a *FileName* file exists, the *exist* variable will have the value True. Otherwise, it will be False. The variable is a Boolean variable that can be used in logical and conditional terms.

*file* must not be a directory or a hidden file.

### FormatRealString

**Syntax:** str = FormatRealString(value, width, precision);

Transforms a *value* number into a character string and stores the result in the *str* variable. *width* indicates the minimum number of characters to be output; *precision* indicates the number of digits after the decimal point.

## G

### GetCycleNo

**Syntax:** CycleNo = GetCycleNo(buffer);

Returns the number of cycles of a spectrum which is contained in *buffer*.

### GetDeviceID

**Syntax:** DeviceTyp = GetDeviceID;

Returns the device type as a character string. For example, if a SPECORD 200 is set (WinASPECT: "Measurement Configuration" menu → "Device Selection"), *DeviceType* will receive the value 'SPECORD 200'. If the device has not yet been initialized, initialization will be performed on selection of this function.

### GetPath

**Syntax:** Path = GetPath(FileName);

Returns the path of a file that was specified via *FileName* with its full file name.

**Example:**

Path = GetPath('C:\Programme\WinASPECT\WinASPECT.exe');

*Path* receives the value 'C:\Programme\WinASPECT\'.

## GetPrintItem

**Syntax** : PrintItem = GetPrintItem(sPrintItem);

Determines for a print element (table, graph etc.) the code number which is transmitted to the *Print* command. The *sPrintItem* parameter formally stands for a character string and can take on the following values:

| | |
|---|---|
| 'NoteS' | Note lines |
| 'TranS' | Transmission spectrum |
| 'Abs' | Absorption spectrum |
| 'REF' | Reflection spectrum |
| 'ENER' | Energy spectrum |
| 'EMIS' | Emission spectrum |
| 'SIGNATURE' | Signature |
| 'GRAPHIC' | Calibration curve |
| 'VALTabLE' | User defined table |
| 'AUDITLIST' | Audit list |
| 'HEADLINE' | Headline |

**Example:**

```
ItemAnzahl = 3;
MakeArray(sPrintItems, ItemAnzahl, 1);
sPrintItems[1] = 'NoteS';
sPrintItems[2] = 'VALTabLE';
sPrintItems[3] = 'HEADLINE';
MakeArray(PrintItems, 1, ItemAnzahl, 2);
for (item = 1; item <= ItemAnzahl; item = item + 1)
{
    PrintItems[1][item] = GetPrintItem(sPrintItems[item]);
}
```

## GetSpecNo

**Syntax:** SpectrumNumber = GetSpecNo;

Returns the number of assigned spectrum buffers. This number corresponds to the index of the last created spectrum buffer.

## GetUserInfo

**Syntax:** userinfo = GetUserInfo;

Having performed this assignment, the following variables can be accessed:

| | |
|---|---|
| *userinfo.level* | User rights |
| *userinfo.login* | Password |
| *userinfo.name* | User name |

# I

## InitTimer

**Syntax:** InitTimer(mod);

Sets (*mod*=1) or cancels (*mod*=0) a time marker, to which all subsequent *Wait* commands will refer. Where a time marker was set, the counting scale of a length of time transmitted to a *Wait* command will refer to this marked (memorized) point in time.

## InpL

**Syntax:** InpL dest_buffer = src_buffer, step;

Computes a new spectrum graph with a step size of *step*, based on the spectrum that is contained in *src_buffer*. Saves the new spectrum in *dest_buffer* (linear interpolation).

## Inpo

**Syntax:** Inpo dest_buffer = src_buffer, step;

Computes a new spectrum graph with a step size of *step*, based on the spectrum that is contained in *src_buffer*. Saves the new spectrum in *dest_buffer* (cubic interpolation).

## InputArray

**Syntax:** InputArray( [rowtitle, columntitle], field);

Changes the values in *field* interactively. Inputs are made in tabular form. *columntitle* contains the column headings, *rowtitle* designates the row titles and *field* represents the input field in the table. Each of the three variables must represent a field that was created using *MakeArray*. In more specific terms, this means:

- Row count (*rowtitle*) = Row count (*field*)
- Column count (*columntitle*) = Column count (*field*)
- Column count (*rowtitle*) = Row count (*columntitle*) = 1

*rowtitle* and *columntitle* must be fields of type STRING. If either *rowtitle* or *columntitle* was not transmitted, the corresponding default values will be set.

If you exit the tables dialog by clicking on 'Cancel', your value changes will not be adopted!

*InputArray* may not be applied to fields other than two-dimensional type fields.

## Int

**Syntax:** S = Int(buffer, x1, y1, x2, y2 [,mode]);

Derives and displays the integral or the area between baseline and spectrum graph. The baseline is defined by the two points *(x1,y1)* and *(x2,y2)*. Subject to integralization is the *x1* and *x2* segment of the spectrum which is contained in *buffer*.

| | |
|---|---|
| *mode* = 0 | Calculation of the integer |
| mode = 1 | Calculation of the area enclosed between graph and abscissa |

# J

## JSave

**Syntax:** JSave(file, buffer);

Saves the spectrum that is contained in *buffer*in a file named *file* in Jcamp-DX format.

# L

## LdPr

**Syntax:**

```
OpenProt;
...
LdPr(ProtFile);
...
CloseProt;
```

Used within an *OpenProt/CloseProt* parenthesis. A protocol stored in encrypted format is loaded. The protocol can now be printed using *PrPr*, displayed using *ShPr* and saved in unencrypted format using *SvPr*.

An example is provided for the *CloseProt* command.

## Left

**Syntax:** lft = Left(str, numb);

Stores the first *numb* characters of a *str* character string in the *lft* variable. *str* may represent a randomly configurable character string term.

**Example**

```
lft = Left(filename mod '\', 10);
```

**Note**:

If numb > Length (str), Left(str,numb) == strg applies.

**Example**:

Left ('ABCD',8) == ABCD

## Length

**Syntax:** len = Length(str);

Determines the number of characters in a *str* character string and stores the value in a *len* variable. *str* may also represent a randomly configurable character string term.

**Note**:
Length ('') == 0

## Ln

**Syntax:** y = Ln(x);

Condition: x > 0

Forms the natural logarithm of a value transmitted in *x* , where *x* may represent a randomly composed numerical term.

## Load

**Syntax:** Load(file, buffer[, mod [ ,mask]]);

Loads a WinASPECT data file with the name *file* to *buffer.* The *mod* and *mask* parameters are helpful for loading cyclic spectra:

| | |
|---|---|
| mod = 0 | Automated loading of selected cycles |
| mod = 1 | Interactive selection of cycles to be loaded |
| mask | Mask of spectra to be loaded (default: '*', i.e., all). |

## Log

**Syntax:** Log dest_buffer = src_buffer;

Forms the common logarithm of a spectrum which is contained in *src_buffer* and saves it in *dest_buffer*.

## Log10

**Syntax:** y = Log10(x);

Condition: x > 0

Forms the common logarithm of a value transmitted in *x*, where *x* may stand for any numerical term.

# M

## MakeArray

**Syntax:** MakeArray(field, dim1 [, dim2,.., dimN [, type]];

Creates a dim1 x ... x dimN field. *type* designates one of the following field types:

| | | |
|---|---|---|
| 0 | ⇨ | Floating point values |
| 1 | ⇨ | Character strings (strings) |
| 2 | ⇨ | Integers, True=1; False=0 |

Default values are:

| | | |
|---|---|---|
| for *cols* | ⇨ | 1 |
| for *type* | ⇨ | 0 |

The row and column number of a field can be called up via the *field.line* and *field.column* variables, once this field has been created.

The various field elements can then be accessed via variable assignments (→ "Fields", p.10).

**Example:**

MakeArray(A, 3, 3, 3, 0);

A ==>

| | | |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 0 | 0 |

| | | |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 0 | 0 |

| | | |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 0 | 0 |

## MakeDir

**Syntax:** MakeDir(FilePath);

Creates a directory with a name as specified in *FilePath*.

## MakeDisplay

**Syntax:** MakeDisplay(Cols, ColCount);

Opens a display window that can be updated with the help of function "AddDisplay" p. 33. *Cols* is a parameter for a single-line array, the column number of which must be transmitted in the *ColCount* parameter. This array contains the column headings.

**Example:**

```
SpaltenAnzahl = 2;
MakeArray(Spaltenueberschrift, 1, SpaltenAnzahl, 1);
Spaltenueberschrift[1][1] = 'Probenbezeichnung';
Spaltenueberschrift[1][2] = 'Absorption';
MakeDisplay(Spaltenueberschrift, SpaltenAnzahl);
MakeArray(NeueZeile, 1, SpaltenAnzahl, 1);
for (probe = 1; probe <= Probenanzahl; probe = probe + 1)
{
    NeueZeile[1][1] = Probennamen[probe][1];
    MeasAccPos(probe, 0, False);

    ! MeasAccPos: command - see further below

    SpecToArray(1, Abs, 1);
    NeueZeile[1][2] = FormatRealString(Abs[1], 6, 4);
    AddDisplay(NeueZeile);
    Eras(1);
}
CloseDisplay;
```

## Mean

**Syntax:** Mean dest_buffer = src_buffer;

Averages the cyclic spectrum contained in *src_buffer* for each abscissa value and saves the result in *dest_buffer*.

## MeanDisp

**Syntax:** meandisp = MeanDisp(buffer [, lft , rght]);

Calculates the mean value and the standard deviation of spectrum values contained in *buffer* (only first cycle, normal distribution provided). *lft* and *rght* can be used to restrict the spectrum range being processed (default: complete range).

On completion thereof, the following variables are available:

| | |
|---|---|
| *meandisp.mean* | Mean value of spectrum values |
| *meandisp.disp* | Spread of spectrum values |

## MeasAccPos

**Syntax:** Next = MeasAccPos(ResultExists, SamplePosition, MeasurementType, False);

Performs motion to an accessory position specified via *SamplePos* and triggers a particular action in that position. The type of action is determined by the *MeasurementType* parameter. Valid conventions are as follows:

| Value of *MeasurementType* | Meaning |
|---|---|
| 0 | Triggers no measurement. |
| 10 | Triggers a reference measurement. |
| 20 | Triggers rinsing (APG): |

Returns False if the user stopped the measurement, otherwise True. The *ResultExists* return parameter returns the value True if measured values exist, even though the measurement was stopped. If the measurement was not stopped, *ResultExists* has the value True. This allows the macro program to continue to run even if the measurement was stopped.

**Example**

Compare with example "MakeDisplay" p. 58.

## Mess

**Syntax:** Mess(file, mod);

Depending on the content of *mod* the following measurement menu functions will be carried out:

| | |
|---|---|
| *mod* = 0 | Initialization of device |
| *mod* = 1 | Parameter menu call |
| *mod* = 3 | Performance of a dark measurement process (simultaneous spectrometer) |
| *mod* = 4 | Performance of a reference measurement |
| *mod* = 6 | Performance of a sample measurement |
| *mod* = 7 | Performance of a wavelength correction |
| *mod* = 8 | Performance of a standard correction |

| | |
|---|---|
| *mod* = 9 | Performance of a special correction |
| mod=10 | Loading of measurement parameters contained in ‘file’ |

All of these functions will use the data contained in the *file* parameter file. If *file* is found to represent an empty string, the given function will revert to those measurement parameters which had been most recently used or loaded since WinASPECT started.

With each selection of *Mess* a spectrum buffer is created. Its purpose is to save the measured results. For example, after ten measurement sequences run successively without removal of a spectrum buffer in between (with the help of an *Eras* command), ten spectrum buffers will have been created. These can then be addressed via a given number from 1 to 10 in the same order in which they were created by measurement.

## Mid

**Syntax:** md = Mid(str, first, numb);

Applies *numb* characters of a *str* character string beginning with the *first* position (1st character corresponds to position 1) and stores these characters in the *md* variable. *str* may represent a randomly configurable character string term.

**Note**
If the requested characters are not available, an empty string is returned.

**Example:**

Mid('ABCD',2,2) == 'BC'

Mid ('ABCD',8,3) == ''
Mid ('ABCD',2,5) == 'BCD'

## MinMax

**Syntax:** extr = MinMax(buffer [, lft [, rght]]);

Determines the minimum and maximum of all spectrum values contained in *buffer* whose relevant abscissa values fall within the range of *lft* and *rght* (all cycles are included).

On completion thereof, the following variables are available:

| | |
|---|---|
| *extr.min* | Minimum of spectrum values |
| *extr.minx* | Relevant abscissa value |
| *extr.max* | Maximum of spectrum values |
| *extr.maxx* | Relevant abscissa value |

## Mran

**Syntax:** Mran(buffer);

Performs autoscaling of abscissa and ordinate and shows the corresponding spectrum contained in *buffer*.

## Msg

**Syntax:** Msg(message);

Outputs a *message* message on the screen. Execution of a currently running method will halt until this message has been acknowledged.

## MULK

**Syntax:** MULK dest_buffer = src_buffer, const;

Multiplies all spectrum values of *src_buffer* with a *const* constant and saves the result in *dest_buffer*.

## Mult

**Syntax:** Mult dest_buffer = src_buffer1, src_buffer2;

Multiplies *src_buffer1* and *src_buffer2* with each other and stores the result in *dest_buffer*. Both spectra must match each other in terms of abscicca units and ordinate units and contain an identical number of cycles. If the initial spectra are transmission spectra or reflection spectra, the resulting values will automatically be divided by 100 (corresponds to addition of related absorption spectra). This command is not available for absorption spectra.

## Norm

**Syntax:** Norm dest_buffer = src_buffer, ref_buffer, x1, ... , xn;

Computes normalization factors for the bands with *x1, ... , xn* abscissa values (measured value of reference spectrum divided by measured value of source spectrum). The values of the spectrum which is contained in *src_buffer* are then multiplied by the mean value of normalization factors and the result is saved in *dest_buffer*. Up to twenty abscissa values can be specified. This command may only be applied to absorption spectra with positive values.

## Note

**Syntax:** Note(buffer [, note]);

Edits the note pertaining to a spectrum contained in *buffer*. *note* represents a note to be automatically inserted. The note will also be displayed on printing of a spectrum and saved as part of the file header when the given spectrum is saved.

# O

## OpenProt

**Syntax**: OpenProt;

Opens an *OpenProt/CloseProt* parenthesis. Refer to the documentation for the *CloseProt* and *LdPr* commands.

## OutputArray

**Syntax:** OutputArray([rowtitle] , [columntitle] , field);

Shows a field in tabular form. Its parameters are similar to those of InputArray. *OutputArray* may only be applied to two-dimensional fields.

## Over

**Syntax**: Over set = buffer1[, ... , buffern];

Generates an overlay display screen showing the spectra contained in *buffer1, ... , buffern*. This display can be transmitted using the *set* parameter to the *Win1, Win2, Win3* and *Prin* commands.

*Note:*

This command is no longer maintained. Use the Overlay command!

## Overlay

**Syntax:** Overlay(Buffers, ResultBuffer);

Generates an overlay display screen showing the spectrum numbers contained in the *Buffers* field. Stores the result in *ResultBuffer*.

**Examples:**

```
MakeArray(Buffers, 1, N);
Buffers[1][1] = 1;
…
Buffers[1][N] = N;
Overlay(Buffers, N + 1);
```

## OverlayZ

**Syntax**:

```
OverlayZ(SrcBuffers, ZData, DestBuffer, ZUnit, Title);
```

*OverlayZ* overlays all spectra transmitted to the *SrcBuffers* integer field based on their index and stores the overlay in the spectrum memory with the number specified in *DestBuffer*.

The next free memory must be specified as the *DestBuffer*. If the highest assigned spectrum memory is the one with number N, *DestBuffer* must be set to N + 1.
*ZData* is a floating point field, which contains a cycle value for each of the spectra to be overlaid, e.g., a temperature or a time.
*ZUnit* designates the unit of the cycle dimension as a character string. For example, '°C' for temperature cycles or 's' for time cycles.
*Title* contains the desired title for the cyclic overlay spectrum.

**Example**

Program example of a 5-temperature cycle. A parameter file must still be specified:

```
ParameterDatei = ...;
Position = 1;
TemperaturAnzahl = 5;
MakeArray(Temperaturen, TemperaturAnzahl, 1);
Temperaturen[1] = '30.0';
Temperaturen[2] = '40.0';
Temperaturen[3] = '55.0';
```

```
Temperaturen[4] = '65.0';
Temperaturen[5] = '90.0';
! Create SourceBuffer integer field for transferring the indexes
! of the spectra to be overlaid
MakeArray(QuellPuffer, TemperaturAnzahl, 2);
! Create TInfos floating point number field
MakeArray(TInfos, TemperaturAnzahl, 0);
! Initialize number of measurements
Gemessen = 0;

for (T = 1; T <= TemperaturAnzahl; T = T + 1)
{
    WriteProfile(ParameterDatei, 'ACCY', 'SOLLTEMP',
                    Temperaturen[T]);
    while (ReadProfile(ParameterDatei, 'ACCY', 'SOLLTEMP', '0.0') <>
            Temperaturen[T])
    {
        Status('Setze T' + T + ': ' + Temperaturen[T]);
    }
    Status('Messung bei T' + T + ': ' + Temperaturen[T]);
    ! Set changed parameter file
    Mess(Parameterdatei, 10);
    ! Measure
    Weiter = MeasAccPos(ErgebnisExistiert, Position, 0, False);
    if (Weiter or ErgebnisExistiert)
    {
        ! If measurement not stopped then
        ! increment number of measurements by one
        Gemessen = Gemessen + 1;
    }
    else if (not Weiter)
    {
        ! If measurement stopped then exit for-loop
        ! set loop variable T one higher than upper limit
        T = TemperaturAnzahl + 1;
    }
    !QuellPuffer = (1, 2,..., TemperaturAnzahl)
```

```
    QuellPuffer[T] = T;
    !TInfos = (30.0, 40.0,..., 90.0)
    TInfos[T] = AtoF(Temperaturen[T]);
}
! The number of spectra that exist corresponds to the measurements
performed
! => The last assigned buffer has the Measured
 index! Normally the following applies: Gemessen == TemperaturAnzahl
ZielPuffer = Gemessen + 1;
! Create overlay and define Y-unit as absorption
OverlayZ(QuellPuffer, TInfos, ZielPuffer, '°C', '5-Temperatur-Zyklen');
AlteEinheit = EditDim(ZielPuffer, 1, 2);
! Show overlay
Win1(ZielPuffer);
! Release all assigned spectrum buffers
for (T = 1; T <= ZielPuffer; T = T + 1)
{
    Eras(T);
}
```

# P

## Para

**Syntax:** Para(buffer);

Shows the parameters that are relevant to the spectrum contained in *buffer*.

## PeakSearch

**Syntax:** PeakSearch(buffer, Peaks, borderMod, limit, border);

Searches for all peaks of a spectrum that is defined by *buffer* and returns a list of peak abscissas in the second parameter - *Peaks*. The length of the *Peaks* field reflects the number of peaks. *border* indicates a threshold, conditional upon *borderMod*, above which an extremum is to be interpreted as a peak. *limit* stands for the minimum amount of variance above which an extremum will be regarded as a peak. If a value differs from its neighboring values by less than was specified via *limit* , it will not be regarded as a peak. If a value is found to be smaller than the amount of the value that was transmitted for *border*, this

value will equally not be interpreted as a peak value. Three values are allowed for *borderMod*:

| | |
|---|---|
| 0 | Searches for maxima and minima. *limit* and *border* will be ignored. |
| 1 | Searches for maxima. |
| 2 | Searches for minima. |

## PhotoSpecial

**Syntax**: PhotoSpecial(i);

Triggers the i-th submenu item of the variable menu section in the measurement menu.

For the *i* parameter the values between 1 and 5 are recommended.

## Pow

**Syntax:** power = Pow(basis, exponent);

Computes the exponentiation power = basis $^{exponent}$.

Where *basis* and *exponent* may each represent a randomly composed numerical term.

*Note:*
The function is only available in WinASPECT PLUS.

## Prin

**Syntax**: Prin(p1, template [, file [, v1 [, p2 [, v2 [, p3 [, v3]]]]]]);

Prints spectrums depending on a transmitted template. Available print template selections (templates) are:

| | |
|---|---|
| 0 | '1 graphic window - full page'; |
| 1 | '1 graphic window - half page'; |
| 7 | '1 object: graphic, note'; |
| 10 | '1 object: graphic, text'; |
| 11 | '2 graphic windows'; |
| 12 | '3 graphic windows'; |

Where a *file* parameter was transmitted, it must contain the full path of a text file or an empty string (''), depending on the content of *template*.

## Print

**Syntax:** Print(PrintItems, PrintRefs);

Parameter: *PrintItems* is a character string field; *PrintRefs* an integer field. Both fields must be of identical length and consist of a single line of several columns each. Entries with identical index belong together. An entry in PrintItems specifies the element to be printed, or a value table has the code *GetPrintItem*('VALTabLE'). The corresponding entry in *PrintRefs* indicates the data source on which the value table is based (in the case of the value table, a two-dimensional array).

**Example**

A term should include the following elements:

- a headline
- a value table
- a spectrum
- a note

1. Begin by defining the *PrintItems* and *PrintRefs* fields for four elements each:

```
ItemAnzahl = 4;
MakeArray(PrintItems, 1, ItemAnzahl, 2); // 2 stands for integer number type
MakeArray(PrintRefs, 1, ItemAnzahl, 1);
```

2. Assign codes for the print contents:

```
MakeArray(sPrintItems, ItemAnzahl, 1);
sPrintItems[1] = 'NoteS';          //code for note
sPrintItems[2] = 'Abs';             //code for absorption spectra
sPrintItems[3] = 'VALTabLE';   //code for value tables
sPrintItems[4] = 'HEADLINE';   //code for headlines
for (item = 1; item <= ItemAnzahl; item = item + 1)
{
    PrintItems[1][item] = GetPrintItem(sPrintItems[item]);
}
```

*Advisory note:*
The order of *PrintItems* is identical with that of printing, except for the headline. The headline can be inserted at a random position, e.g., like here at the end.

3. Data sources for printing:

| | |
|---|---|
| Value table | Named "Results"<br>Generated via MakeArray(results, rows, cols, 1) |
| Absorption spectrum | Spectrum buffer for index 3. |
| Headline | "Sample print" text |
| Note | Via these commands:<br>Prot(Notiz1, 1); //(1 empties the protocol file)<br>…<br>Prot(NoteN);<br>written line by line to the<WinASPECT master path>\prot.tmp file |

Assign values to the PrintRefs field:

```
DokumenteVerzeichnis = ReadRegStr('', 'PathData');

ProtokollDateiName = Dokumenteverzeichnis + '\Methods\' +
// Determine WinASPECT master directory
//and save in Location variable

PrintRefs[1][1] = Location + '\prot.tmp'; // path + name of note file

PrintRefs[1][2] = '3';
//Index of the absorption spectrum in the spectrum buffer

PrintRefs[1][3] = 'Ergebnisse'; // name of field for value table,
                                //must be a character string field!

PrintRefs[1][4] = 'Sample print'; // headline text
```

4. Append print command:

```
Print(PrintItems, PrintRefs);
```

*Note*
For a list containing the codes, refer to the documentation for the *GetPrintItem* command.

## PrintPortrait

**Syntax:** old = PrintPortrait(new);

Switches between portrait and landscape format for printing. *new* must represent one of the two Boolean values True or False. In the case of True, printing is switched to portrait format, otherwise to landscape format. Similarly, the previous setting (True or False) is stored in the *old* variable.

## Prot

**Syntax:** Prot(str);

Stores a *str* character string in the internal protocol file in a new line. Numerical terms are automatically converted into character strings.

The internal protocol file is named 'prot.tmp'. It is created in the installation directory of WinASPECT and deleted on termination of a WinASPECT session. When a method is triggered, the protocol is initialized (the content of 'prot.tmp' is deleted).

**New syntax** Prot(line [, mode]);

Where 0 is transmitted to the *Mode* parameter, the character string transmitted to the *Line* parameter is appended to the last protocol line:

| Before | After |
|---|---|
| Line1 | Line1 |
| … | ... |
| LineN | LineN + Line |

If, on the other hand, 1 is transmitted to the *mode* parameter, the current protocol is discarded and a new protocol is initialized with the character string transmitted to the Line parameter as the first line.

| Before | After |
|---|---|
| Line1 | Line |
| … | |
| LineN | |

## PrPr

**Syntax:** PrPr;

Prints all protocol lines that have been stored by Prot commands up to that point (PrintProtocol).

# R

## ReadHeader

**Syntax:** curv = ReadHeader(file);

Reads the header data of the WinASPECT *file* data file. Once this assignment has been made, you may access the following variables:

| | |
|---|---|
| *curv.start* | Beginning of range of measurement |
| *curv.end* | End of range of measurement |
| *curv.numb* | Number of measurement points |
| *curv.note* | Remarks |
| *curv.absdim* | Abscissa dimension |
| *curv.orddim* | Ordinate dimension |
| *curv.cycldim* | Cycle dimension |
| *curv.cycl* | Number of cycles measured |
| *curv.messpara* | Measurement parameter file (only valid for original spectrum) |
| *curv.date* | Compilation date of data file |

## ReadProfile

**Syntax:** string = ReadProfile(file, section, entry, default);

Reads the value of a variable. This value is returned as a character string in the *string* variable. It must have been saved beforehand with the help of a *WriteProfile* command. The *file* text file is accessed in the process. Its structure is identical with that of Windows initialization files. The value is searched for in the *section* section under the *entry* key word. If no such key word is found, the *default* value is returned via a string.

## ReadRegBool

**Syntax:** bool = ReadRegBool(key, name);

Reads the assigned Boolean value of a given *key* key and *name* value name from the registration database and stores this value in *bool*.

## ReadRegInt

**Syntax:** integ = ReadRegInt(key, name);

Reads the assigned integer number value of a given *key* key and *name* value name from the registration database and stores this value in *integ*.

## ReadRegStr

**Syntax:** str = ReadRegStr(key, name);

Reads the assigned character string value of a given *key* key and *name* value name from the registration database and stores this value in *str*.

## RegrKorr

**Syntax:** reg = RegrKorr(buffer [,mod [, lft [, rght [, regr_buf]]]]);

Applies a selected regression to the spectrum contained in *buffer*. If *regr_buf* was transmitted, the command will indicate the spectrum buffer that contains the newly determined regression graph. *mod* is the parameter that indicates the type of regression:

| *mod* | | |
|---|---|---|
| | 0 | y = a1*x |
| | 1 | y = a0 + a1*x |
| | 2 | y = a0 + a1*x + a2*x² |
| | 3 | y = a0 + a1*exp (-a2*x) |
| | 4 | y = a0 + a1*x + a2*exp (-a3*x) |
| | 5 | y = a0*x / (a1 + x) (Michaelis-Menten) |
| | 6 | y = x/a2 + 1/a0 (Linewaever-Burk) |
| | default: | 1 |

The range of regression can be restricted via *lft* and *rght*. The default value for both parameters is -1 (left and right boundary of spectrum).

The result is stored to the 'kinetics.txt' text file in the installation directory of WinASPECT.

On completion thereof, the following variables are available:

reg.a0, reg.a1, reg.a2, reg.a3

Coefficients of regression model

NOT IMPLEMENTED:

reg.e0, reg.e1, reg.e2, reg.e3

Related error tolerances

reg.corr Correlation coefficient (for models 0,1,6) or error sum of squares (for remaining models)

## Remove

**Syntax:** Remove(file);

Deletes the *file* file. Accepts wildcards.

## Rename

**Syntax:** Rename(src_file, dest_file);

Renames the *src_file* file to *dest_file*.

## Rev

**Syntax:** Rev(buffer);

Reverses the abscissa graph of a spectrum contained in *buffer*.

## Right

**Syntax:** rgt = Right(str);

Accepts the last *numb* characters of a *str* character string and stores the result in the *right* variable. *str* may represent a randomly configurable character string term.

## Round

**Syntax:** y = Round(x);

Rounds a value that was transmitted via *x* , where *x* may represent a randomly composed numerical term.

## RunSipper

**Syntax:** RunSipper(PumpT);

Moves a currently selected accessory device into the position for rinsing and causes the sippier to pump for a length of time as specified via *PumpT* in seconds.

# S

## Save

**Syntax:** Save(file, buffer);

Saves the spectrum that is contained in *buffer* in a file named *file*.

## SaveFile

**Syntax:** FileName = SaveFile(Caption, PathMask);

Opens a file saving dialog screen with a headline as defined via *Caption* and with the open initial path or the file type mask, which are both to be specified in the *PathMask* parameter. If a file is selected, the complete file name is stored in the *FileName* character string variable. If the dialog is canceled, *FileName* refers to the empty string ‘’ and the macro is ended. The file name can still be used to save a file.

**Example:**

```
CurveFileName = SaveFile('Save the calibration line under:',
CalibExportPath + '*.cal');
```

## Scale

**Syntax:** Scale(buffer, x1, y1, x2, y2);

Displays the spectrum that is contained in *buffer* within the abscissa limits *x1* and *x2* and within the ordinate limits *y1* and *y2* (same as Zoom command).

## Sect

**Syntax:** Sect dest_buffer = src_buffer, firstx [, interactiv [, mod [, secondx [, scycl [, ecycl [, max [,deriv [, sm]]]]]]]];

Cuts through the cyclic spectrum that is contained in *src_buffer* and stores the result in *dest_buffer*. The other parameters have the following meaning:

| | |
|---|---|
| *interactiv* | Selection of cutting mode (1 - interactive, 0 - the cutting parameters must have been transmitted),.default: 0 |
| *scycl* | First cycle to be considered (cycles beginning with 1), default: 1 |
| *ecycl* | Last cycle to be considered (-1 = all cycles from scycl), default: -1 |
| *deriv* | Degree of derivative of the original spectrum before the cut is applied (0, 1, 2, 3, 4), default: 0 (no derivative) |
| *sm* | Smoothing degree of the original spectrum before the cut is applied (0, 5, 7, 9, 11, 13, 17, 21, 25). See also SMnn command. default: 0 (no smoothing) |

To ensure smoothing or derivative can be applied, the spectrum must have more than 25 measured values for each cycle.

A value is formed from each of the cycles to be considered depending on *mod*:

| | |
|---|---|
| *mod* = 0 | The measured value assigned to the *firstx* abscissa value. |
| *mod* = 1 | Forms the extremum (maximum for *max*=1 or minimum for *max*=0) between the *firstx* and *secondx* abscissa values. |
| *mod* = 2 | Forms the amount of the difference between the *firstx* and *secondx* assigned abscissa values: Value = Abs (M(*firstx*) - M(*secondx*)) |
| *mod* = 3 | Forms the quotient from the *firstx* and *secondx* assigned measured values: Value = M(*firstx*) / M(*secondx*) |
| *mod* = 4 | Determines the area between the *firstx* and *secondx* abscissa values, the abscissa and the graph. |
| *mod* = 5 | a1 of the regression model y = a1 * x |
| *mod* = 6 | a1 of the regression model y = a1 * x + a0 |
| *mod* = 7 | a1 of the regression model y = a2 * $x^2$ + a1 * x + a0 |
| *mod* = 8 | Equivalent to mod = 0 with subsequent application of mod = 5 on the cutting spectrum (for cyclic spectra only!). |
| *mod* = 9 | Equivalent to mod = 0 with subsequent application of mod = 6 on the cutting spectrum (for cyclic spectra only!). |
| *mod* = 10 | Equivalent to mod = 0 with subsequent application of mod = 7 on the cutting spectrum (for cyclic spectra only!). |
| *mod* = 11 | Equivalent to mod = 0 with subsequent formation of mean value of values of the resulting cutting spectrum (for cyclic spectra only!) |
| *mod* = 12 | Equivalent to mod = 1 with corrected baseline (line between the measured values of the spectrum graph on the *firstx* and *secondx* abscissa values). |

| | |
|---|---|
| *mod* = 13 | Equivalent to mod = 4 with corrected baseline (line between the measured values of the spectrum graph on the *firstx* and *secondx* abscissa values). |
| *mod* = 14 | Forms the difference between the *firstx* and *secondx* assigned measured values: Value = M(*firstx*) - M(*secondx*) |

## SelBuffer

**Syntax:** buf = SelBuffer(text);

Shows the menu for selecting a spectrum memory. For more detailed information, *text* is also output.

## SelFile

**Syntax:** file = SelFile(text, filemask);

Shows the menu for selecting a file. The title bar also contains *text*. *filemask* must be a valid selection mask. If *filemask* contains no directory specification, the currently selected data directory will automatically be displayed. The *file* variable contains the selected file.

## SelFiles

**Syntax:** SelFiles(files, text, filemask);

Opens a file opening dialog screen with multiple-selection option. Saves the full file name of selected files in a single-line array file. This array will contain as many columns as files were selected. For the *text*, *filemask* parameters, the same rules apply as in the case of *SelFile*.

**Example:**

```
SelFiles(DatFiles, 'Please select a spectrum ...',
        dat_file_path + '*.dat');
Probenanzahl = DatFiles.Column;
MakeArray(Probennamen, Probenanzahl, 1, 1);
for (probe = 1; probe <= Probenanzahl; probe = probe + 1)
{
    Probennamen[probe][1] = (DatFiles[1][probe] / '.') mod '/';
    Load(DatFiles[1][probe], probe);
}
```

## SelFolder

**Syntax:** Path = SelFolder(DlgCaption)

*SelFolder* shows a directory dialog in which a directory can be selected interactively. The result assigned to the *Path* variable is a character string and contains the complete directory path. The *DlgCaption* parameter dispalys the desired directory dialog designation, e.g., 'Select an export directory...'.

**Example**

```
ExportPfad = SelFolder('Save export files as...');
```

## SelItems

**Syntax:** SelItems(ItemNames, CheckArray, Caption);

Provides a dialog for multiple selection. The *ItemNames* field contains the descriptions of selectable elements (e.g., sample names). *CheckArray* designates the return parameter. On completion of this command, it will contain, for each selectable element, either '0' if the particular element was not selected and a '1' if it was selected, where the i-th element will correspond to the i-th location within the *CheckArray* field. The dialog headline must be transmitted via *Caption* .

**Example:**

```
MakeArray(Probennamen, 1, Probenanzahl, 1);
for (probe = 1; probe <= Probenanzahl; probe = probe + 1)
{
    Probennamen[1][probe] = Probendaten[probe][1];
}
SelItems(Probennamen, RepeatCheck, 'Samples due for new
measurement:');
for (probe = 1; probe <= Probenanzahl; probe = probe + 1)
{
    if (RepeatCheck[1][probe])
    {
        repeatprobenanzahl = repeatprobenanzahl + 1;
    }
}
```

## SelMenu

**Syntax:** item = SelMenu(text, item1, item2 [ , ... ]);

Genereates its own selection list with at least two entries (*item1*, ... ). The title bar also contains *text*. On completion of this command, the 'item' variable will contain the selected entry (1 = item1, ... ).

## SetDecimalSeparator

**Syntax:** SetDecimalSeparator(newsep);

Sets the global decimal separator.

**Example:**

```
SetDecimalSeparator('.');
```

## ShFi

**Syntax:** ShFi(textfile);

Displays a small *textfile* text file.

## Shift

**Syntax:** Shift buffer2 = buffer1, ToShift ;

Shifts the spectrum contained in *buffer1* along the abscissa axis by the amount of *ToShift* and stores the resulting spectrum under *buffer2*.

## ShowCalib

**Syntax:** ShowCalib(RegrCorrCoeffs, buffer, RegrMode, XUnit, YUnit);

Represents the spectrum contained in *buffer* as a regression graph. Its *RegrCorrCoeffs* parameter contains the fitting graph settings and is the result of a (→) RegrKorr function call, where *XUnit* and *YUnit* contain the X and Y units. *RegrMode* encodes the regression model as follows:

1: Model $y = ax + b$

2: Model $y = ax^2 + bx + c$

**Example:**

```
MakeArray(Messwerte, Standardanzahl, 2, 0);
for (std = 1; std <= Standardanzahl; std = std + 1)
{       {
Messwerte[std][1] = Konzentration;
}
…
Mess(Paramdatei, 6);
for (std = 1; std <= Standardanzahl; std = std + 1)
{
SpecToArray(std, MeasData, 1);
Messwerte[std][2] = MeasData[1];
Eras(std);
regrmode = 1;     //Modell y = ax + b
ArrayToSpec(Messwerte, 1);       //Creates a spectrum at 1, with //(x,y) values as contained in the measured values field
olddim = EditDim(1, 1, 2); //Important: Sets the dimension of the //spectrum at 1, refer to (→) EditDim.
reg = RegrKorr(1, regrmode, -1, -1, 2);     //Determines the //regression graph and stores graph parameters in reg
ShowCalib(reg, 1, regrmode, 'c [mmol/l]', 'A'); //Calls up ShowCalib
modelanswer = YesNoMsg('Do you want to change the calibration model?');
while (modelanswer)
{
Eras(1);
Eras(2);
ArrayToSpec(Messwerte, 1);
olddim = EditDim(1, 1, 2);
regrmode = SelMenu('Select a calibration model:',
'y = ax + b',
'y = ax² + bx + c');
reg = RegrKorr(1, regrmode, -1, -1, 2);
ShowCalib(reg, 1, regrmode, 'c [' + Einheit + ']', 'A');
modelanswer = YesNoMsg('Do you want to change the calibration model again?');
}
```

## ShowNotify

**Syntax**: ShowNotify(NotifyCaption, Time);

Shows for the *Time* in seconds a message window with the message text transmitted in *NotifyCaption*. The message closes automatically in contrast to the message windows opened with *Msg* .

**Example**:

```
ShowNotify ('Automatic printing in progress...', 2);
```

## ShowSamples

**Syntax**: ShowSamples(CalibrationFile, Xvalues);

Shows the points, which belong to the abscissa values transmitted to the Xvalues field, on a calibration curve, which is specified via a ".cal" file contained in the Calibration file parameter, as labeled marks. If n values were transmitted, n marks with the labels Sample1...Sample<n> will be visible. The curve range that was used for calibration is highlighted in bold font.

**Example:**

```
AnzahlXWerte = 5;
MakeArray(XWerte, AnzahlXWerte, 1, 0);
XWerte[1][1] = -1.5;
XWerte[2][1] = -0.5;
XWerte[3][1] = 0.01;
XWerte[4][1] = 0.3;
XWerte[5][1] = 0.6;
DokumenteVerzeichnis = ReadRegStr('', 'PathData');
KalibrierDatei = DokumenteVerzeichnis + '\calibexport\Kal.cal';
ShowSamples(KalibrierDatei, Xwerte);
```

## ShPr

**Syntax:** ShPr;

Shows on the screen all protocol lines that were stored up to that moment with the help of a Prot command. (ShowProtocol).

## Sign

**Syntax:** Sign;

Opens a dialog to electronically sign a protocol.

## Sin

**Syntax:** Y = Sin(x);

Forms the sine of a value transmitted in *x*, where *x* may represent a randomly composed numerical term.

## SMnn

**Syntax:** SM<*nn*> dest_buffer = src_buffer;

Smoothes the spectrum values contained in *src_buffer* including *nn* reference values each time. Saves the result in *dest_buffer*.

*nn* may represent any of the following numbers:

5, 7, 9, 11, 13, 17, 21, 25

The initial spectrum should be one with equidistant reference points.

## SpecToArray

**Syntax:** SpecToArray (buffer, field [, cycl]);

Creates a two-column *field* field from the values of a spectrum contained in *buffer*. The abscissa values of the spectrum are transferred to the first column of the field. *cycl* specifies the number of spectrum cycles (beginning with 1), whose values are transferred to the second column of the field. The default value for *cycl* is 1.

## Sqrt

**Syntax:** Y = Sqrt(x);

Condition: $x \geq 0$

Forms the positive square root of a value transmitted via *x*, where *x* may represent a randomly composed numerical term.

## Status

**Syntax:** Status (text);

Outputs a *text* message to the status line. This message will stay on display until the status line is overwritten again. This notably implies that specific method-related texts must be cleared by outputting an empty string on completion of a method.

## Sub

**Syntax:** Sub dest_buffer = src_buffer1, src_buffer2;

Subtracts *src_buffer2* from *src_buffer1* and saves the result in *dest_buffer*. Both spectrums must match each other in terms of abscissa and ordinate units.

## SvPr

**Syntax:** SvPr (ProtFile);

Saves all protocol lines that were stored up to that moment with the help of a *Prot* command in the *protfile* text file (*SaveProtocol*). This file is automatically given a '.prt' extension. It is a normal ASCII file without control characters and can be processed with any editor (e.g. 'notepad.exe').

If the protocol should be saved so that it cannot be edited (i.e. in encrypted format), all associated *Prot* commands including the *SvPr* command must be included in an *OpenProt/CloseProt* parenthesis. An example is provided for the *OpenProt* command.

The saving of a protocol file is equivalent to the beginning of a new protocol.

# T

## Tab

Function not implemented

**Syntax:** Tab buffer;

Shows a table with measured values (interactively) of the spectrum contained in *buffer*.

## TextFileToArray

**Syntax**: TextfileToArray field, textfile;

Automatically creates a field under the name *field*, in which the lines of the *textfile* text file are saved. The number of lines in this field is equal to the number of lines in the text file. The number of columns is one. A line should not contain more than 250 characters. The permitted number of lines may vary depending on computer and should not exceed 4000. The line number and column number of a field can be called up via the field.line and field.column variables, once this field has been created.

## Thick

**Syntax:**

DickeVerbund = Thick(SrcBuffer, MaterialDateiname, L0, L1, Arc);

DickeVerbund = Thick(SrcBuffer, MaterialDateiname, L0, L1, Arc, DestBuffer);

Determines the thickness of a thin layer from an interference spectrum. The return value is a structure value, i.e. names separated by a point are used to access the actual values:

! Geometric and optical layer thickness

DickeVerbund.geo, DickeVerbund.opt

! Cauchy coefficients for the wavelength-dependent refractive index,
! rel. error sum of squares

DickeVerbund.n0, DickeVerbund.n1, DickeVerbund.n2

Parameters to be specified:

| | |
|---|---|
| *DestBuffer* | Optional,<br>this is transmitted if the Fourier transform of the interference spectrum is needed. |
| *SrcBuffer* | Spectrum buffer index containing the interference spectrum. |
| *MaterialFileName* | Names of the material file which contains data pairs (wavelength, refractive index(wavelength)) as support points.<br>If the refractive index is constant, the material file will only contain one pair, e.g., (190.0, 1.5) with constant refractive index 1.5. The value for the wavelength can be selected as desired, however it must be specified in the file.<br>The file extension is "*.mat". |

| | |
|---|---|
| Arc | Angle in radians, which the incident beam encloses with the optical axis. Generally, this is 0.0 rad. |

**Material file structure:**

<Comment line: The layer material should be specified here>

<Number of support points> N

<Wavelength> <Re(refractive index)> <Im(refractive index)> [1]

...

<Wavelength> <Re(refractive index)> <Im(refractive index)> [N]

The imaginary parts are optional.

**Examples of the material file**

Aluminum / Palik: Handb. Opt. Const. of Solids p.396

11

350.0 0.375 4.24

375.7 0.432 4.56

400.0 0.490 4.86

450.0 0.618 5.47

500.0 0.769 6.08

550.0 0.958 6.69

600.0 1.20 7.26

650.0 1.47 7.79

700.0 1.83 8.31

750.0 2.40 8.62

800.0 2.80 8.45

Spat

4

320.0 1.4

350.0 2.3

400.0 2.8

500.0 8.2

**Example of layer thickness calculation:**

! Pfade

```
DokumenteVerzeichnis = ReadRegStr('', 'PathData');
DatDateiVerzeichnis = DokumenteVerzeichnis + '\Data';
MatDateiVerzeichnis = DokumenteVerzeichnis + '\Thickness';
MaterialDatei = MatDateiVerzeichnis + '\';
! Select spectrum file
DatDatei = SelFile('Select spectrum file...',
                   DatDateiVerzeichnis + '*.dat');
! Select material file
MaterialDatei = MatDateiVerzeichnis + '\Material.mat';
! Define buffer for interference and FFT spectrum
! Interferenz-Spektrum = Quelle
! FFT-Spektrum = Ziel
QuellPuffer = 1;
ZielPuffer  = 2;
Load(DatDatei, QuellPuffer);
! Read out left and right limit
X0 = AtoF(ReadProfile(DatDatei, 'GENERAL', 'FIRSTX', '200.0'));
X1 = AtoF(ReadProfile(DatDatei, 'GENERAL',  'LASTX', '700.0'));
! Determine layer thickness
DickeVerbund = thick(QuellPuffer,
                     MaterialDatei,
                     X0, X1,
                     0.0,
                     ZielPuffer);
! Show interference and FFT spectrum
Win2(QuellPuffer, ZielPuffer);
! Release buffer
Eras(QuellPuffer);
Eras(ZielPuffer);
! Show results
Msg('Geometric thickness: ' + DickeVerbund.geo + CR +
    'Optical thickness    : ' + DickeVerbund.opt + CR +
    'n0: ' + DickeVerbund.n0 + CR +
    'n1: ' + DickeVerbund.n1 + CR +
    'n2: ' + DickeVerbund.n2);
```

## Tile

**Syntax:** Tile;

Displays all currently loaded spectrums and overlay objects on the screen.

## Time

**Syntax:** tim = Time;

Saves the time in 'hh:mm:ss' format.

## ToUpper

**Syntax:** upper = ToUpper(string);

Converts all small letters contained in *string* into capital letters and assigns the result to *upper*.

## Tran

**Syntax:** Tran dest_buffer = src_buffer, mod;

Converts the absorption spectrum contained in *src_buffer* into a transmission spectrum (*mod*'TRA') or vice versa (*mod*'Abs'). Saves the result in *dest_buffer*.

Additional modes:

| | |
|---|---|
| 'cm-1' | nm -> cm-1 |
| 'nm' | cm-1 -> nm |
| 'Kubelka' | Kubelka-Munk transformation (this function is not implemented as yet) |

## Trunc

**Syntax:** Y = Trunc(x);

Truncates the digits after the decimal point of a value transmitted to *x*, where *x* may represent a randomly composed numerical term.

## TxtExport

**Syntax:** TxtExport(ExportFile, ExportFeld);

Exports a text table (two-dimensional character string field) that is contained in *ExportFeld* to a text file specified via *ExportFile*.

**Example:**

```
ExportDateiName = SaveFile('Save export file as:',exportpfad + '*.txt');
//Save dialog; see (→)SaveFile
FirstRow = '';
for (col = 1; col < ResCols; col = col + 1)
{
    FirstRow = FirstRow + Spalten[1][col] + ';';
}
FirstRow = FirstRow + Spalten[1][ResCols];
ExportArray[1][1] = FirstRow;       //Fills the character string field…
for (probe = 1; probe <= DefProbenanzahl; probe = probe + 1)
{
    SecondRow = '';
    for (col = 1; col < ResCols; col = col + 1)
    {
        SecondRow = SecondRow + Ergebnisse[probe][col] + ';';
    }
    SecondRow = SecondRow + Ergebnisse[probe][ResCols];
    ExportArray[probe + 1][1] = SecondRow;
}
TxtExport(exportdateiname, ExportArray);
//Save
```

# U

## Use4

**Syntax:** Use4;

Shows the currently valid variable assignment.

# W

## Wait

**Syntax:** Wait(seconds [, text]);

Interrupts the execution of a running method by *seconds* seconds or until the dialog box is canceled ("Continue >>" key). If *text* was specified, it will be displayed for that time.

## WaitEx

**Syntax**: WaitEx(seconds [, text]);

Interrupts the execution of a method by *seconds* seconds. If *text* was specified, it will be displayed for that time.

## WaitLoop

**Syntax:** waitresult = WaitLoop(deltaT, Caption);

Opens a modal window. A currently running process will wait. If this window is not exited, a waiting time as specified in seconds in *deltaT* will pass before the window is automatically closed again and value 1 returned to *waitresult*. If the window was closed beforehand by an operator command, *waitresult* will be assigned the value 0. A designation is transmitted to *Caption*, e.g., for which event/condition you are waiting.

**Example:**

```
waitresult = WaitLoop(deltaT, 'Pause until renewed E2 measurement...');
while (waitresult)
{
  Mess(Parameterdateiname, 6);
  waitresult = WaitLoop(deltaT, 'Pause until renewed E2 measurement...');
}
```

## Win1

**Syntax:** Win1(buffer);

Represents the spectrum contained in *buffer* on the screen.

## Win2

**Syntax:** Win2( buffer1, buffer2);

Simultaneously displays the spectrums which are contained in *buffer1* and *buffer2* on the screen.

## Win3

**Syntax:** Win3( buffer1, buffer2, buffer3);

Simultaneously displays the spectrums which are contained in *buffer1, buffer2* and *buffer3* on the screen.

## WriteProfile

**Syntax:** WriteProfile(file, section, entry, value);

Saves a *value* value in the *file* text file in the *section* section under the *entry* key word. The *file* text file is required to have the same structure as a Windows initialization file (also refer to *ReadProfile* command). If there is no such file, it will be automatically generated. The *value* value or variable must not represent a field.

## WriteRegBool

**Syntax:** WriteRegBool(key, name, bool);

Stores a Boolean value *bool* under a given *key* key and with a *name* value name to the registry database.

## WriteRegInt

**Syntax:** WriteRegInt(key, name, integ);

Stores an *integ* integer number value under a given *key* key and with a *name* value name to the registry database.

### WriteRegStr

**Syntax:** str = WriteRegStr(key, name, str);

Stores a *str* character string value under a given *key* key and with a *name* value name to the registry database.

## Y

### YesNoMsg

**Syntax:** bool = YesNoMsg(question);

If *question* is answered with 'yes', the ‘bool’ variable will have a True value, otherwise, its value will be False. ‘bool’ is a Boolean variable that can be used in logical and integer-number terms.

## Z

### Zero

**Syntax:** Zero dest_buffer = src_buffer;

Performs an automatic zero-line correction of an absorption spectrum currently contained in *src_buffer* and saves the result in *dest_buffer*. In the case of a transmission spectrum, it will use the 100%T line for reference.

### Zoom

**Syntax:** Zoom(buffer, x1, y1, x2, y2);

Displays the spectrum that is contained in *buffer* within the abscissa limits *x1* and *x2* and within the ordinate limits *y1* and *y2*.

No integer-number terms must be transmitted in the values *x1, x2, y1,* and *y2*.

# 5 Index