Spectroscopic Software

BRUKER

OPUS

Version 4

PROGRAMMING

I 24313

This manual is the original documentation for the OPUS spectroscopic software.

Table of Contents

Introduction

Macros

External Programs

Macro Command Reference

OPUS Command Reference

Client/Server Command Reference

Scripts

1 Programming, Controlling and Communication with OPUS-NT

Data acquisition and processing under OPUS-NT can be automated using self-written programs. These programs can either be written in OPUS' own macro programming language, or in Microsofts VBScript. OPUS-NT provides an interface to third party programs, giving you a link to software designed for purposes other than spectral data acquisition and manipulation. In addition, you can use this interface to access programs you have written yourself.

The first part of this manual covers the interfaces for external programs. The second part references the OPUS functions, macro commands, client/server and VBScript commands.

1.1 Methods of Flow Control

OPUS provides several ways to control program routines. These range from the simple recording of routine commands in a macro to sophisticated evalutaion algorithms, which need OPUS parameters and specral data. And, for example, third party software to control accessories.

This is established by the use of:

- Macros
- VBScript
- external programs

1.2 Interfaces

OPUS comprises of several interfaces for data exchange with other software. Supported are:

- data input and output using pipes.
- an OPUS DDE Server.
- communication with DDE-Servers of other software.
- an OLE interface for VBScripts.

2 Programs running under OPUS-NT

2.1 External Programs

External programs are employed whenever you encounter tasks that cannot be handled by OPUS-NT, or if dedicated programs to solve these tasks already exist.

Basically, there are two categories of external software:

- Programs designed by the user to perform specific calculations or data manipulations, and which are able to communicate with OPUS.
- Any other third-party program, which can be launched and controlled by OPUS.

However, the external software must be written to run on the Windows NT platform, because upon the launch of the program, system functions will be called directly. Therefore, the program must either be a Win32 executable file, a 16 bit Windows, or a DOS program (an OS/2 program without graphical output will also work), This requires, that the respective subsystem has been installed on your computer. In addition, batch files can be used. These files can be identified by their extensions:

1) .exe
2) .com
3) .bat

Currently, the following OPUS commands are supported by the interface:

- Reading and writing spectral data from/to OPUS spectrum files and 3D files, where the frequency range can be selected.
- Loading and unloading OPUS files.
- Running macros and VBScripts, including parameter exchange.
- Access to information of selected files and the possibility to expand OPUS by new functions.
- Reading and writing OPUS parameters from/to OPUS spectrum files.
- Reading data from report blocks.
- Generating and positioning windows.

2.1.1 The External Program Command

Programs are launched from within OPUS by the *External Program* command, located in the *File* menu.

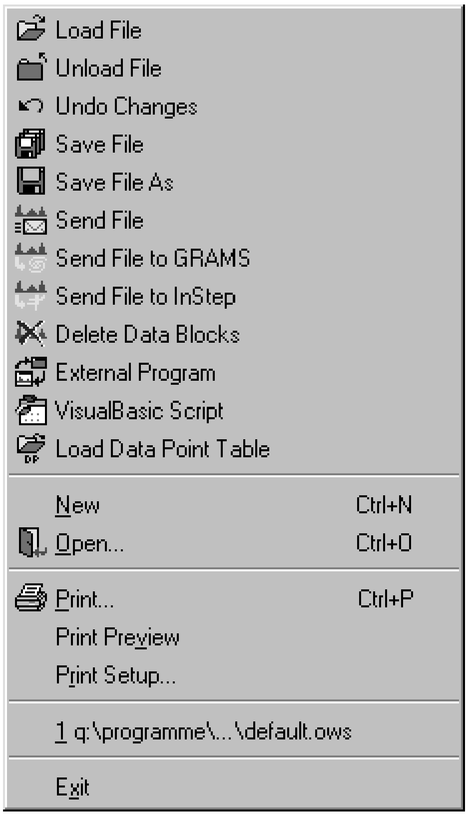


Figure 1: *File* Menu

Upon selecting the command, a dialog box opens. Use this dialog box to specify the name and location of the external program, and any additional parameter or OPUS files you would like to transfer to the program. To launch the program click the *Execute* button.

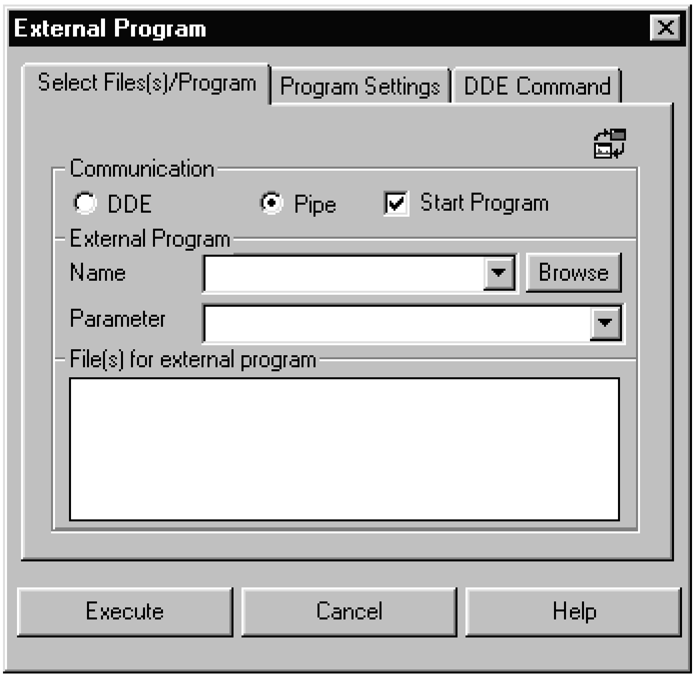


Figure 2: *External Program* Dialog – *Select Files/Program*

2.1.2 The Select File/Program Page

On this page you specify the program you want to start and any additional parameters. You can either:

- use the *Browse* button to locate the program and select it by double-clicking, or
- open the drop-down list to choose a program you ran before, or
- manually type in the programs name.

The external program will be started after clicking the *Execute* button. If you don´t specify a path OPUS will first look in the OPUS directory, next in the active directory (the directory that was used at last), followed by the Windows system directory (SYSTEM/SYSTEM32) and any directory indicated by the path environment variable.

In the following, the parameters of the dialog page will be explained in detail:

Communication

Specifies the type of data communication between OPUS and the external software:

- **D**ynamic **D**ata **E**xchange – OPUS acts as a client for the external program, which in turn must function as a server. Choose this communication type also if no data exchange will take place.
- Named Pipe – is restricted to programs which were especially designed to function as a client to OPUS. A Named Pipe communi-

cation ensure fast data transfer between OPUS and the external program, in combination with the possibility of controlling OPUS by commands.

Start Program Checkbox

If this box is not checked, the program will not be started. This is especially useful, if DDE commands should be repeatedly used with the same server.

External Program – Name

Enter the name (including the path) of the program to be launched. The drop-down list holds the names of the programs which were started during earlier sessions. After the installation of OPUS the list is empty. Alternatively, use the *Browse* button to open a Load File dialog.

External Program – Parameters

Additional parameters to be used with the external programs can be specified in this field. A lot of programs accept parameters like the name of a file which should be processed. These will be forwarded to the external program as command line arguments. The entries can be accessed by the external program using the READ_FROM_ENTRY command.

Files for External Program

Add the data blocks of the spectra to be processed by the external program, if the external program supports this option.

2.1.3 The Program Settings Page

Here, additional parameters to control the external program can be set.

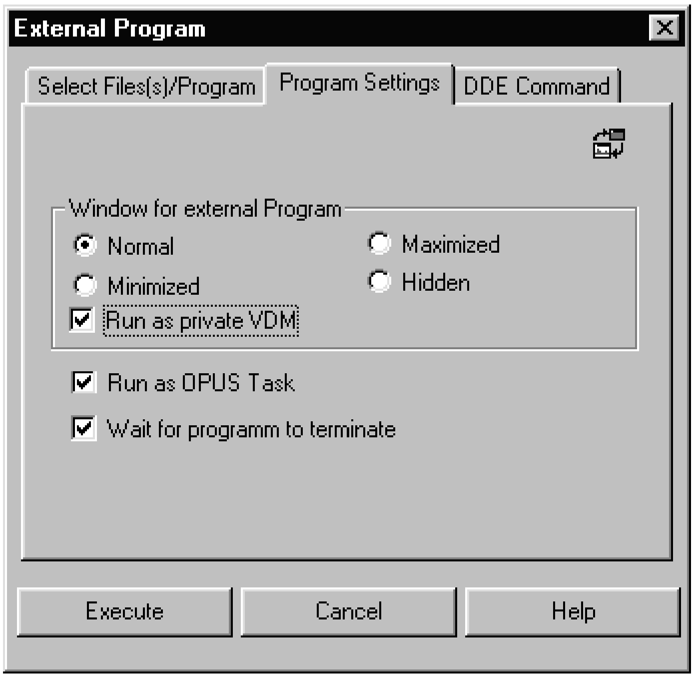


Figure 3: *External Program – Program Settings* Page

Window for External Program

Usually, every program runs in its own window. The size and type of this window can be defined, according to the following options:

- Normal – a window of standard size will be opened.
- Maximized – the program runs in a window of maximum size.
- Minimized – the program runs in a minimized window. The window is not visible, but can be opened by clicking on its icon on the Windows NT Task bar.
- Hidden – the program starts in the background and must be called/ closed using special commands.

Run as Private VDM (16bit executables only)

16 bit applications run in the Windows NT **V**irtual **D**os **M**achine. One advantage of the VDM is its stability; if a program running in a VDM becomes instable, the VDM will shut down without affecting the operating system. Usually, all 16 bit applications run in the same VDM. By checking this box you can force the operating system to open a separate VDM for the external program, resulting in enhanced overall stability at the cost of additional administrative tasks by the operating system.

Run as OPUS Task

If this box is not marked, all connections to the external program will be terminated after the program starts and both, OPUS and the program, will run independently.

Wait for Program to Terminate

OPUS waits until the external program has been terminated. This option is useful while running macros and scripts, which then can make use of a result or parameter obtained from an external program.

2.1.4 The DDE Command Page

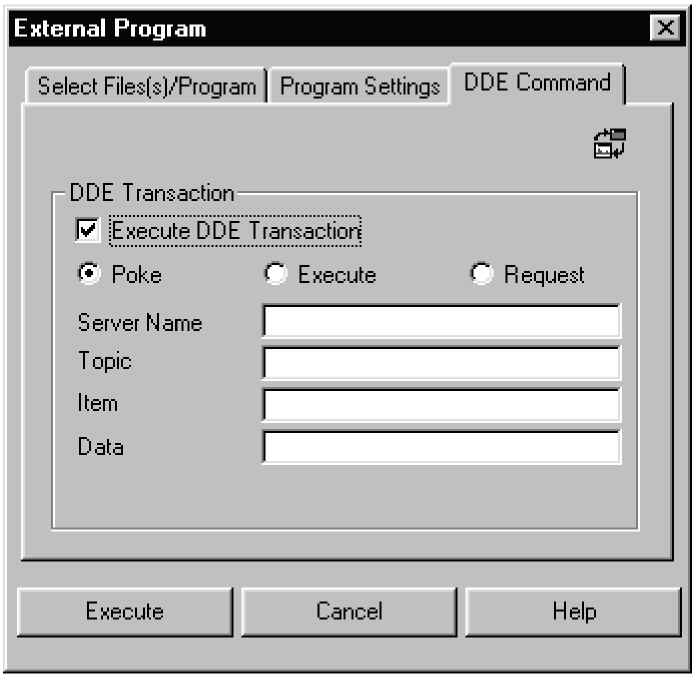


Figure 4: *External Program – DDE Command* Page

The DDE Command Page comprises all DDE interface settings to call a server program.

Execute DDE Transaction

This check box specifies if the DDE server should execute a command sequence. If so, you have to choose which type of DDE transaction should be performed:

Poke

This option sends binary data to the server via the XTYP_POKE transaction.

Execute

A server command will be executed via the XTYP_EXECUTE transaction. The external program does not return any data.

Request

A server command will be executed via the XTYP_REQUEST transaction. The external program returns the result.

Server Name

Defines the name of the server, which was used by the server to register with the system.

Topic

Defines the class of the command. The topic depends on the external program and the command to be executed; consult the documentation of the program used.

Item

Specifies the command which will be executed. The value depends on the program and the performed command; consult the documentation of the program used.

Data

This list is only of importance, if *Poke* was chosen as DDE transaction type. Enter the binary data, which will be forwarded to the server. All data has to be coded in string format, for example *65 0x0A 0x0D*. If the data is a character string, it is to be enclosed in hyphens: *"hallo"*. In this case, the respective ASCII codes including a terminating 0 will be entered in the list. A prefix (i = int, l = long, d = double, f = float) is used to classify the following data to be of the given numerical type. *z* in combination with a figure *n* is used to add *n* zeros.

2.1.5 Writing software

You can write your own client software to communicate with OPUS. These programs can be used to assign tasks to OPUS and read data from OPUS. Named Pipes, which are generated on the host computer can be used to handle the communication between your program and OPUS. Named Pipes can easily be accessed through several programming languages e.g. the language C. In C, these pipes are addressed by commands similar to the commands used for files access, `fopen` and `fclose.` In addition, Named Pipes are supported by many network platforms.

OPUS also provides a DDE server service, that allows the external program to trigger an OPUS function with a DDE command.

To create your program, you can make use of various developing environments. The only restriction that applies is that the resulting executable must run on the Windows NT platform.

2.2 Macros

OPUS offers its own macro system, consisting of an editor, a debugger and a converter to translate macros written with OPUS-OS/2.

Macros are intended to be used, if a command sequence has to be processed repeatedly. Combining these commands in a macro saves the user the trouble of manually calling these commands (and specifying their parameters). Furthermore, frequently used macros can be integrated as command buttons into the OPUS tool bar.

Macros are text files that can be created using any kind of ASCII text editor. However, OPUS provides a comfortable Macro Editor for this purpose. In combination with the OPUS Macro Debugger, macro programming and testing becomes straight forward.

2.3 VBScripts

OPUS provides an interface to the Microsoft Scripting Engine, which is able process several scripting languages. As a result, several programming languages can be used in combination with OPUS. Currently, VBScript is the most commonly used language; furthermore, Java Script can also be used. There are no limitations to the design of the user interface, which is constructed with the help of Microsoft Forms. As usual, a Form is constructed interactively.

Scripts open a way to a nearly unlimited number of applications. They can be used to guide an OPUS user through the software, without the necessity for him to become an OPUS expert. Through their ability to communicate with OPUS directly, scripts can be used to exchange data between OPUS and other software, allowing complex calculations or data manipulation. In addition, by using scripts it is possible to directly access the data stored in OPUS files.

2.3.1 Accessing Scripts

There are two ways to start an existing script:

- Use the *Open* command in the *File* menu (see Fig. 1). Use "*.obs" as default extension. If you load this type of file, the respective user interface will be launched and the script will be processed.

- Use the *VBScript* command in the *File* menu; this command allows to specify parameters to be used when running the script. Furthermore, this *VBScript* command itself can be included in macros and scripts.

2.3.2 The Select Files/Script Page

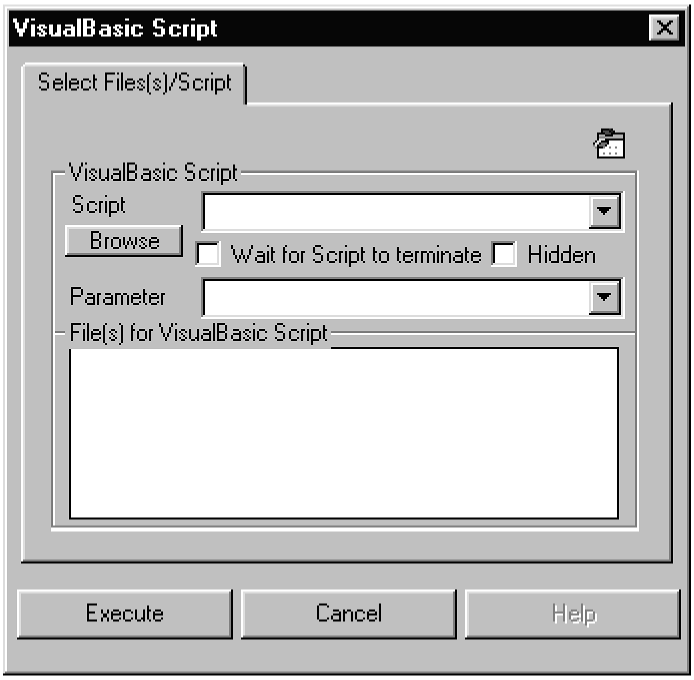


Figure 5: *VBScript* Dialog – *Select Files/Script* Page

Script

Enter the name (including the path) of the script to be launched. The drop-down list holds the names of the scripts which were started during an earlier session. Alternatively, use the *Browse* button to open a *Load File* dialog and locate the script.

Wait for Script to Terminate

OPUS waits for the script to terminate. This option is useful while calling this function from macros and scripts, which in turn make use of a result or parameter obtained from the launched script.

Hidden

The script will run in the background instead of being displayed.

Parameters

Additional parameters to be used in combination with the script can be specified in this field. These will be exchanged in string format. This causes an OpusInform event in the script, with the string as a parameter.

Files for VB Script

Add the data blocks of the spectra to be processed by the script.

2.3.3 Generating a Script

Choose the *New* button in the *File* menu and *VB Script* in the displayed dialog box to generate a new script. This will automatically open the Form Editor window.

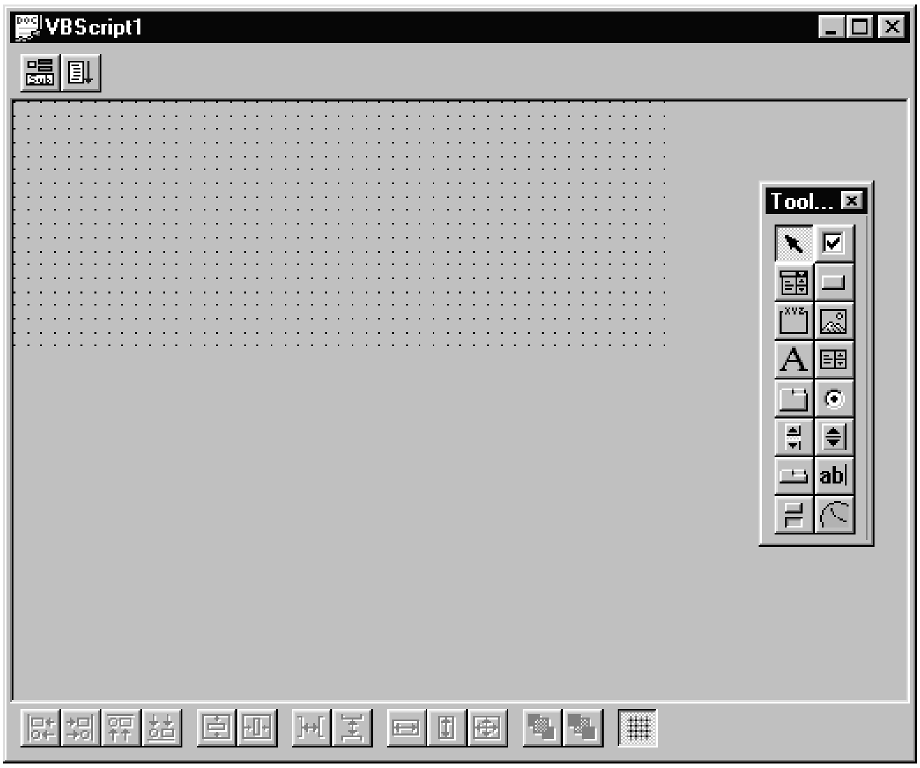


Figure 6: VB Scripting Editor

Use the Form Editor and the Toolbox to include dialog boxes and controls in the script. Select a tool and draw a rectangle to position a new control. The tool bar in the lower part of the window controls the position of the control buttons within the script. Start the script by clicking on the  button. Switch to the text input window by clicking on the  button. Refer to chapter 8 to learn about using the Scripting Editor.

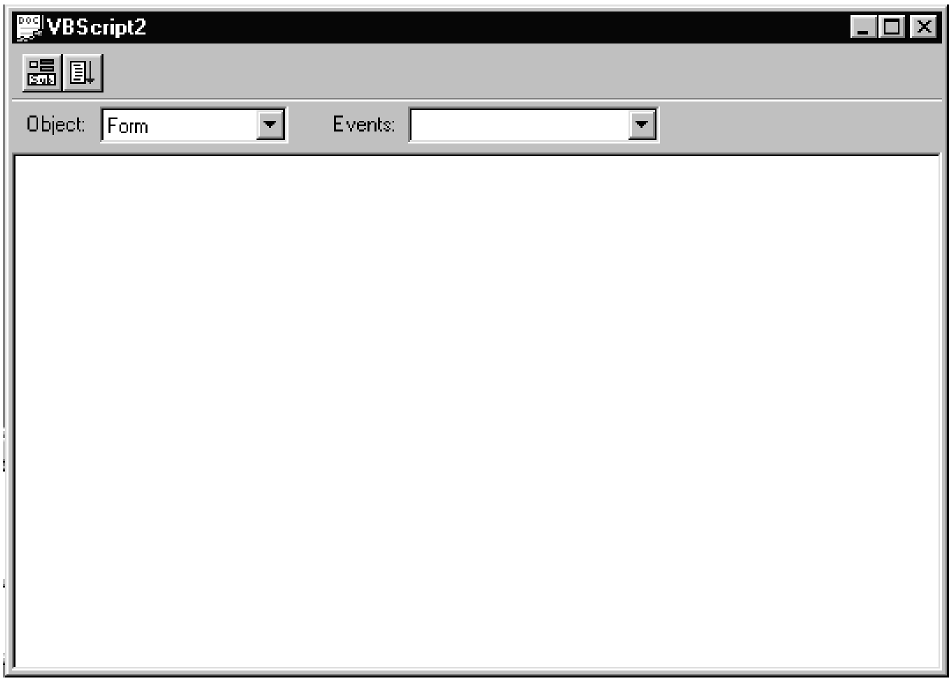


Figure 7: VB Scripting Editor – Text Input Window

2.4 Including Macros and Scripts in the Tool Bar

You can include your scripts and macros in a tool bar to comfortably access the most frequently used programs. OPUS must recognize the macro/script (and the correlated bitmap) upon starting. Therefore, for each macro/script you have to add an entry in the USERMAC.LST file, stored in the OPUS directory.

Path\File@Menunumber@Itemname@Tooltiptext@Statusline

Path\File	The path to the macro/script.
Menunumber	The number of the menu, in which the entry will appear.
Itemname	The name under which the macro/script will be listed in the menu.
Tooltiptext	The text which will be displayed as tooltip for the macro/script.
Statusline	The text which will be displayed in the status line while the macro/script is running.

The numbers of the menus can be taken from the following table:

Menu	Menu Number
Measure	1
Manipulate	2
Evaluate	3
Display	4
Print	5
Macro	6
Edit	7
Validation	8
Setup	9
File	10

In order for the macro/script to be represented by an icon on the tool bar, you have to provide a 16 color bitmap of 16x15 pixels of this icon. This bitmap has to be stored as <*Macro/Scriptname*>.bmp in the same path as the corresponding macro/script.

Example:

C:\OPUS-NT\KALIBRATION.MTX@1@Measure@Standardcalibration
@Macro running

This entry includes the macro CALIBRATION.MTX in the *Measure* menu. “Standard calibration” will be displayed as tooltip. The icon bitmap must be stored as CALIBRATION.BMP in the C:\OPUS-NT\ directory.

2.5 Auto-starting a Script

In some cases it is desired to automatically perform certain tasks upon launching OPUS. In this way, a specifically configured OPUS user interface could be automatically presented to the user. To automatically process a script after the start of OPUS, a parameter has to be included in the Windows NT command line:

Opus.exe /SCRIPT=start.obs

This command automatically loads and processes the script “start.obs” after starting OPUS. The same command can be included in a Windows NT shortcut for OPUS (refer to your Windows NT documentation).

3 Calling OPUS Functions

All OPUS functions can also be called as a text command from:

- the command line
- within macros
- DDE requests of an external client program to the OPUS server
- within scripts

Syntax:

<Function> ([File],{Parameter})

Call from the Command Line

A simple command line interpreter is built into OPUS, but not visible in the default mode. The command line interpreter can be activated with the *Customize Toolbars* command form the *Setup* menu.

For example, if you enter

Baseline (["e:\opus\data\abboe05.0"],{})

at the command line, the file ABBOE05.0 will be baseline corrected. If the file has not been loaded in OPUS, it will load automatically prior to the baseline correction.

The file which is to be processed has to be enclosed by hyphens. The empty braces at the end of the command symbolize, that no parameters have been specified. In that case, the default values will be used. However, parameters can be entered in order to make the command more specific. Parameters consist of a three character code, followed by an equal sign and a value (e.g. BME=1). Parameters have to be separated by a comma.

A file can be loaded more than once at the same time; therefore, a number is added after the file name to identify the version of the file. This number is called clonecount. Furthermore, a data file usually consists of several data blocks, which can be addressed separately:

```
Baseline ["e:\opus\data\abboe05.0" 3:AB]
```

This command processes the absorption data block of the third version (copy) of the file ABBOE05.0.

Call to a Macro from a DDE Client

A command can also be included in a macro. The same syntax as used for the command line entry applies, except that instead of the file name a macro file variable is to be used.

Call from the OPUS DDE Server

If the OPUS DDE server is addressed by another program, the server executes a command, which then in turn allows this program to control OPUS. After a DDE connection has been established between the server (OPUS) and the external program, commands can be exchanged as LinkItem (e.g. using XTYP_EXECUTE). As class name *OPUS/System* has to be entered in the *Topic* field (see chapter 1.1.4). The command syntax is identical to the command line entry, preceded by `COMMAND_LINE` if necessary.

Client/Server Communication via Pipes

Once a Named Pipe connection was established between OPUS and the external program (usually named \\.\PIPE\PROGRAM.EXE), commands can be written to the pipe. Using the language C, this could be achieved with the `fprint` command, the handle of the pipe and the OPUS command syntax.

OPUS Scripting Interface

The OPUS command syntax also applies for scripts. They only differ in the way the commands are transmitted. A special function (member) of the form will be called by the script, which transfers the command to OPUS:

Form.OpusCommand("COMMAND_LINE...")

4 Controlling External Programs

You can also make use of an OPUS interface to control third party software. This allows you for example to control additional laboratory equipment, that you would like to use in combination with your spectrometer.

Command Line Parameters

Use the *External Program* Dialog to forward the command line argument through the *Parameter* field. The software interprets the parameter similar to a command line input. Depending on the program, not all its functionalities may be accessible via the command line options.

Using OPUS as a DDE Client

If a program offers a DDE server interface, OPUS can function as a client to this server. This is again realized through the *External Program* command, which requests DDE commands from the server. It does not matter if the server is already running, or must be started first.

Every server is registered with the operating system, and assigned a unique name by which it can be addressed. Enter the commands in the *Item* field of the *DDE Command* page (see chapter 1.1.4), and select the desired transaction (for example request). Click on the *Execute* button to start the DDE transaction.

Accessing the OLE Interface with Scripts

Scripts are capable to access e.g. Active X documents directly via the OLE interface. You can create for example an Excel file with the CreateObject (“Excel.Sheet”) command. Consult the documentation of the external software to find out about the supported interfaces.

5 OPUS-NT Macro Language

The OPUS macro system uses a special text based macro language. Macros are stored in form of text files, which are interpreted and executed directly within OPUS.

5.1 Creating Macros

OPUS Macros can be written in three different ways:

- Using a text editor.
- Using the interactive Macro Editor.
- Translating OPUS-OS/2 macros into the OPUS-NT format.

If you are familiar with the syntax you can easily generate macros using any type of text editor (e.g. Notepad). Just ensure that you save your macro in plain text format. Make sure you include all three mandatory keywords: “VARIABLES SECTION”, “PROGRAM SECTION” and “PARAMETER SECTION”. A semicolon has to be used as End of Line character.

5.2 General Syntax Rules

A few syntax rules apply for **all** elements of the macro language:

- A macro line (command, declaration) must always be terminated by a semicolon.
- A macro line can be split into several text lines within the text file.
- The three section keywords (VARIABLES SECTION, PROGRAM SECTION, PARAMETER SECTION) must be present in each macro, even if a section is empty.
- The section keywords do not need a semicolon.
- Variable names are always enclosed by < > (e.g. <Index>).
- Strings within a macro line must be enclosed by single quotes (’This is a string’).
- All command lines need brackets after the command name, even if they do not need command arguments.
- A line beginning with the keyword REM will be ignored.

The most common errors in programming macros are missing section keywords and semicolons.

5.3 Macro Keyword REM

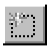
Any line in the three macro sections can be disabled during a macro run by typing REM at the very beginning of a line. This either can be used to temporarily disable lines for testing instead of erasing them, or for adding comment lines within the macro for better readability.

When converting OPUS-OS/2 macros commands which are currently not available will automatically be preceded by the REM keyword.

5.4 The Macro Editor

5.4.1 General

OPUS-NT provides an user-friendly macro editor, which allows you to write and edit your own macros. The macro editor comes with syntax check capability; every time an existing macro is loaded, when a macro line or variable line is edited or when a macro should be stored, a syntax check is automatically performed. In case an error is detected an error message will be displayed and the changes responsible for the error will be revoked. You cannot exit the editor or save a macro unless all errors have been corrected.

The editor consists of two windows, one displays the macro code and the second the macro variables. Attached to each window is a tool bar; on both tool bars you find buttons to insert  and to remove text lines .

Use the *Open Macro* button to load an existing macro. The syntax of macro command lines and variable declarations is checked upon loading a macro. In case an error is detected an error message will be displayed. You have the choice to either start the *Autocorrect* option (see below) or to load the macro and leave all lines unchanged and correct the errors manually. Please note that you cannot save a macro or exit from the macro editor unless you have corrected all syntax errors. By clicking on the *Autocorrect* button all command lines will be scanned and all errors automatically corrected.

Follow these steps to quickly remove syntax errors from your macros:

1) Open the macro editor and load the macro. If your macro contains syntax errors you will see the following error message “Suppress Error Messages and load with Auto Correct Option?”.
2) Click on “Yes”. Now the macro will be loaded and all detected syntax errors will be corrected automatically. The message “Syntax Errors have been corrected automatically ” will be displayed.
3) Confirming this message opens the “Save File” dialog; store the corrected macro.

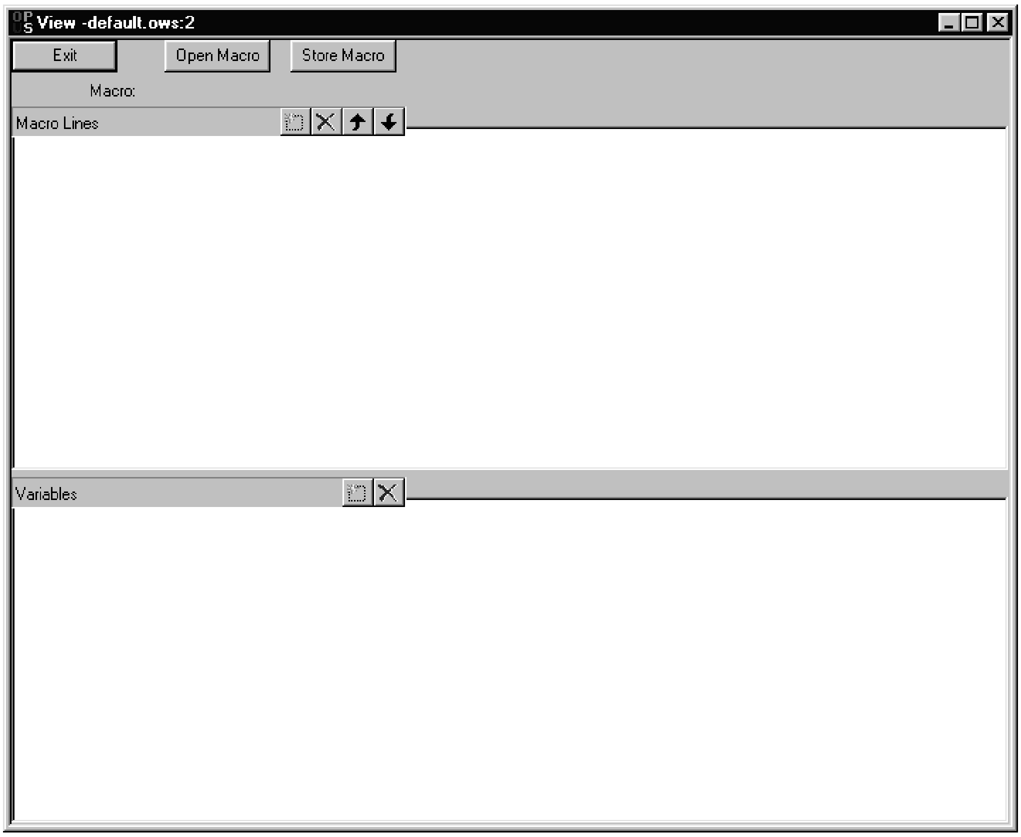
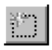


Figure 8: Macro Editor

To insert a line, activate a line in the macro code; click on the  button and a blank line appears below the activated line of code. At the end of the new line another button ... is displayed. This button opens a dialog to assist you during the declaration of special commands (see the following chapter). However, you can also type in the code manually. Edit a line by a double-click, followed by clicking on the ... button.

You enter variables in the same manner in the bottom window; instead of the *Special Macro Commands* dialog, a box for the declaration of variables will be displayed after clicking on the ... button.

In addition, the code window has two buttons to shift selected lines up  or  down the text body, in order to simplify restructuring the program. Variables displayed in the lower window cannot be repositioned, but are listed chronologically to their creation.

You can search for any string in the macro command and the variables section. Enter the string you want to search for in the entry field below the two search buttons. Start the search by either clicking on the *Search Command* or *Search Variable* button. Click the button repeatedly to find the next occurance of the search string. After searching the macro is completed the search starts again at the top.

5.4.2 Special Commands

Open the *Special Macro Commands* dialog by clicking on ... . Enter the command in the *Command Name* field or choose a command from the drop-down list. This list contains all special commands.

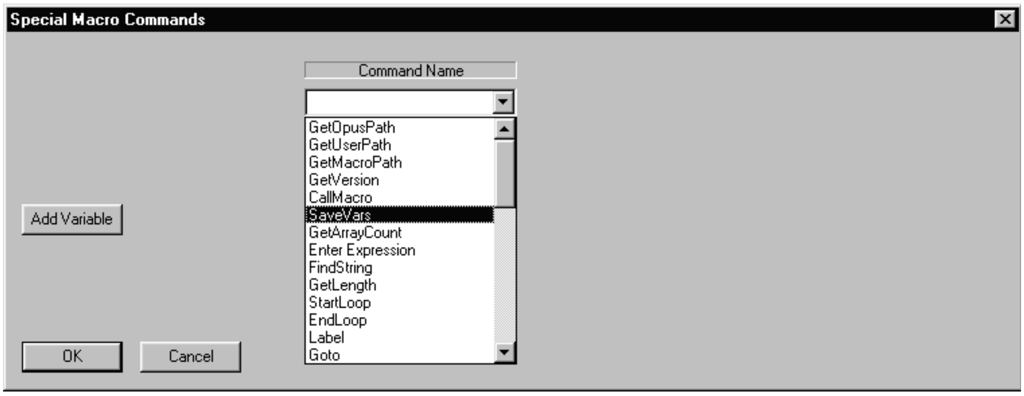


Figure 9: *Special Macro Commands* Dialog

Depending on the command, additional Parameters will be displayed. In the case of Functions, that assign values to variables two fields, *Variable* and *Index*, are shown on the left side of the *Command Name* field.

Some macro commands require parameters which can only be selected from a predefined set of options. In this case the most common used option will be shown automatically in the entry field. For StartLoop and EndLoop instructions the loop index will also be selected automatically.

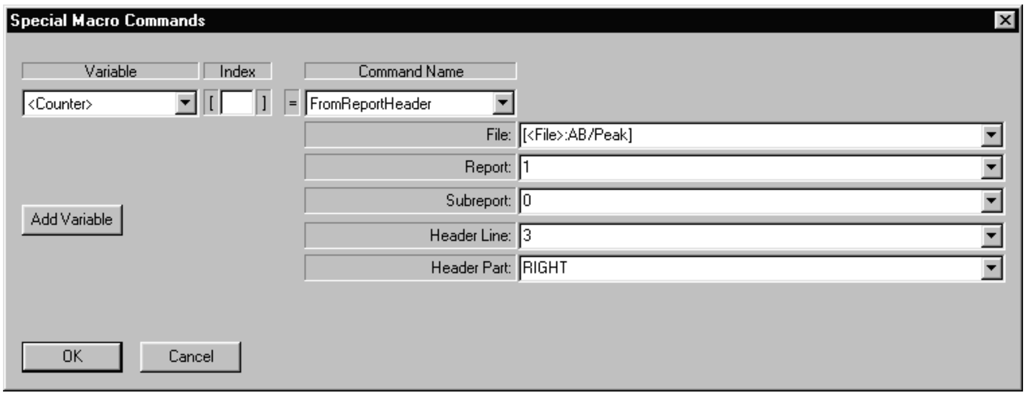


Figure 10: *Special Macro Commands* Dialog – Command Declaration

Drop-down lists provide variables or key words for all fields, depending on the type of the parameter. Each field must contain a value for the command to function properly. The syntax is described in chapter 9. Use the *Add Variable* button (see the next chapter) to define a new variable.

The commands listed in the following open their own dialog box:

CallMacro
UserDialog
StaticMessage

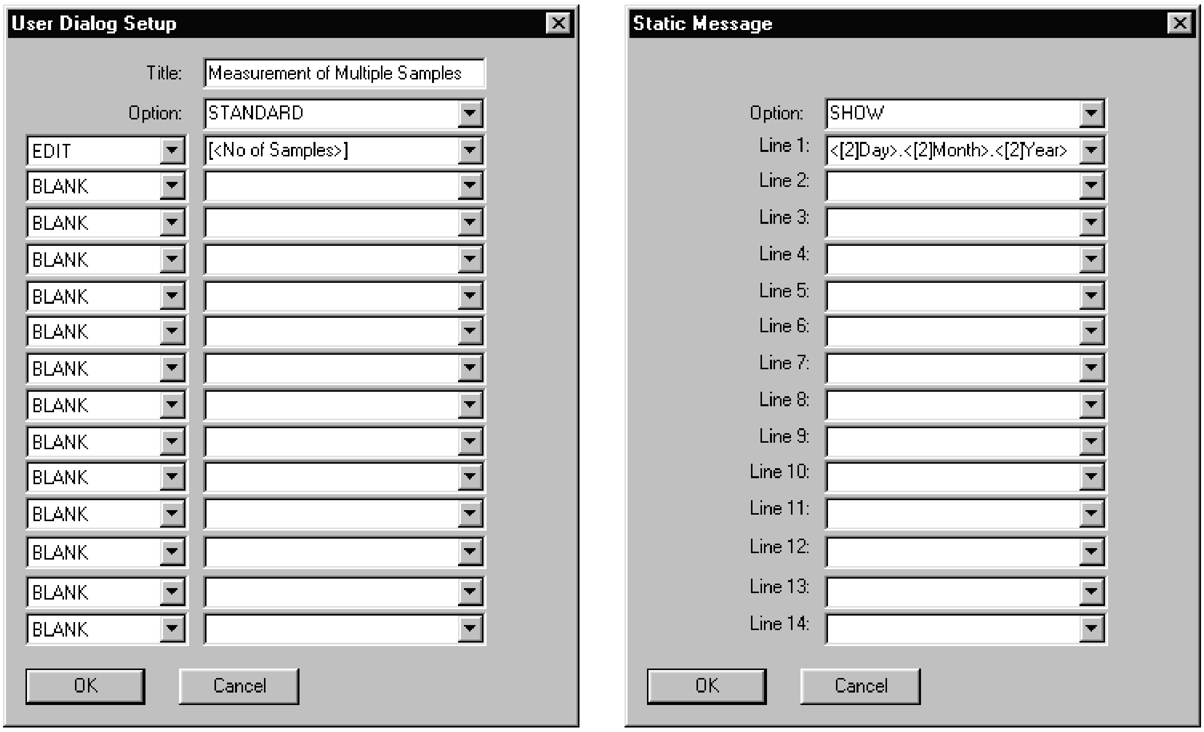


Figure 11: *User Dialog Setup* and *Static Message* Dialog Box

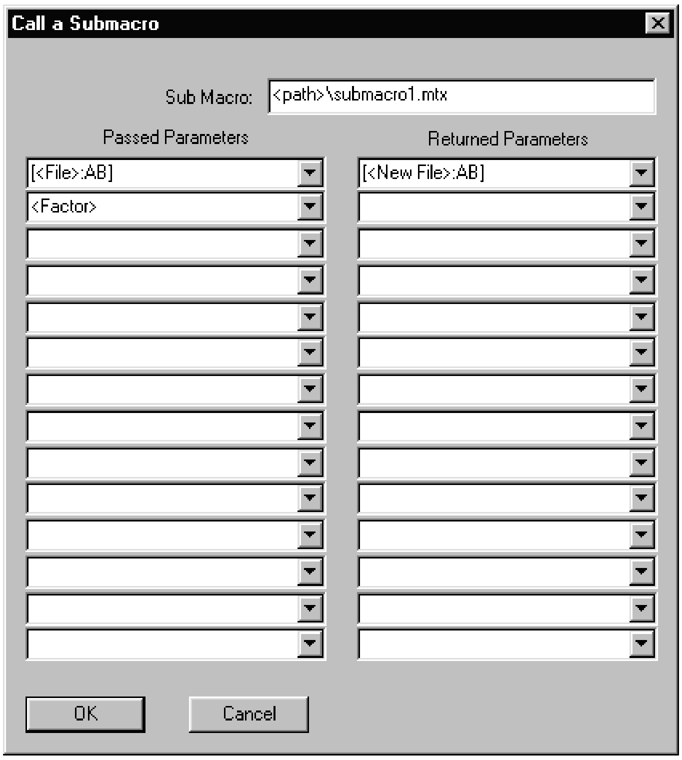


Figure 12: *Call a Sub Macro* Dialog Box

The command *Enter Expression* is an exception in that sense, that instead of the command name an equal “=” sign will be used.

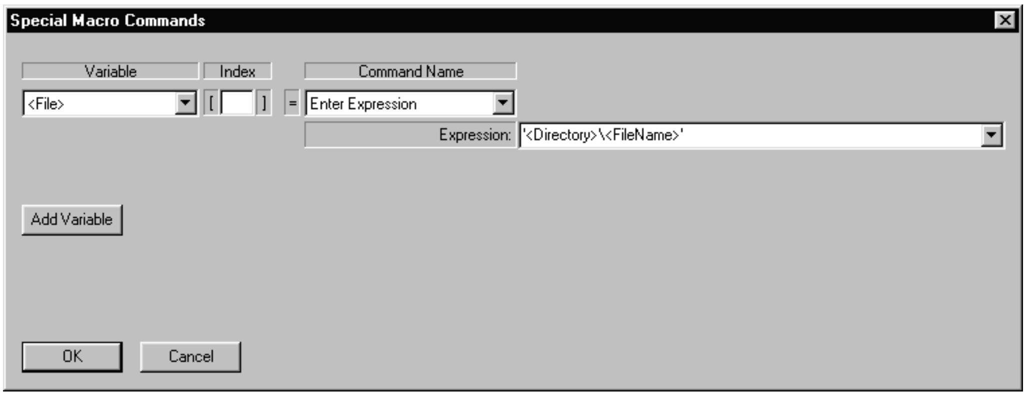


Figure 13: *Special Macro Commands – Enter Expression*

This dialog for example generates the program code

<File> = ’<Directory>\<FileName>’;

5.4.3 The Variable Dialog Box

This dialog box will open after clicking on ... . Choose the type of the variable by checking the respective radio button. Enter a unique name in the *Name* field, and if necessary also a start value. If no start value is stated, 0 will be taken in case of numerical variables and an empty string for all other variables. Check the *Update Automatically* box to have the value of the variable automatically refreshed. These variables are labelled by a preceding asterik “*”.

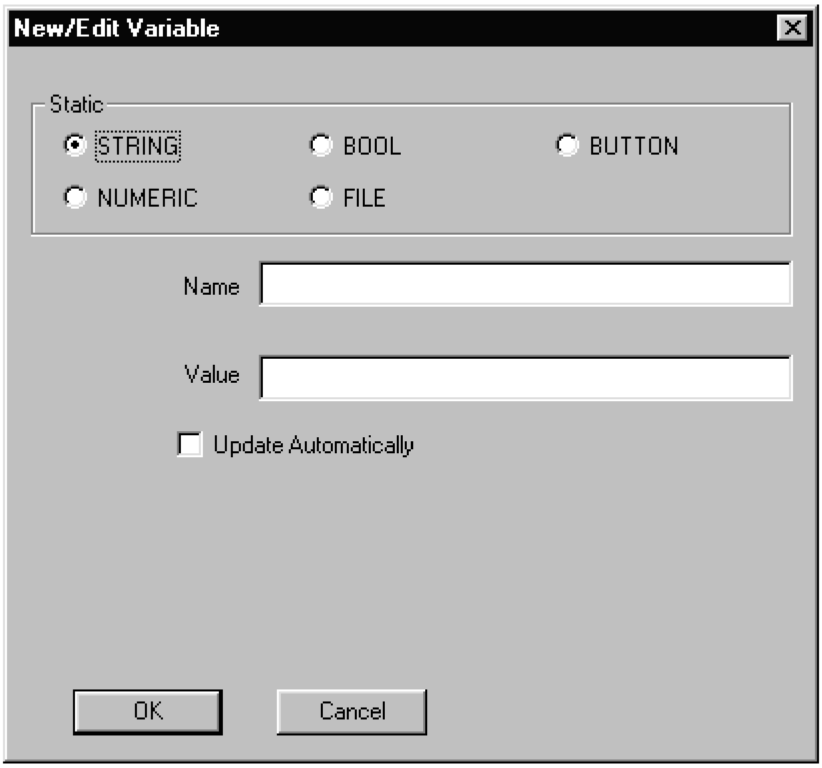


Figure 14: *New/Edit Variable* Dialog

If you select *FILE* as variable type you can specify, besides the name, one or more data blocks in the fields that appear. Choose the desired block type from the three drop-down list and press the *Add* button; they will appear in the list in the lower part of the window.

The first list contains a complete list of spectrum data blocks. The second list consists of derivative blocks, and the last list comprises the rest of the extended blocks. Blocks that are marked by a slash “/” are linked to spectral data. Some blocks can either be linked or not, like for example the Search data block; while the report of a spectrum search is linked to the spectral data block (AB/Search), the report of an information search is not.

5.4.4 Inserting OPUS Commands

OPUS commands are inserted by simply selecting the desired OPUS function from the pull-down menu or the tool bar while you edit a macro. This causes the program code to be inserted below the selected line or to be appended to the macro if no line is selected. You can reposition the code within the program with the up and down buttons. OPUS commands can be edited in the same way as any other macro command. Double-click on the code line and press the ... button. The dialog box of the OPUS command will open and you can alter your settings.

When you select a command its dialog box will be displayed as usual, allowing you to set the function parameters. The list of files to be processed is replaced by a drop-down list comprising all file variables. Instead of a file name you select a variable. This requires that the variable you want to use must be defined beforehand.

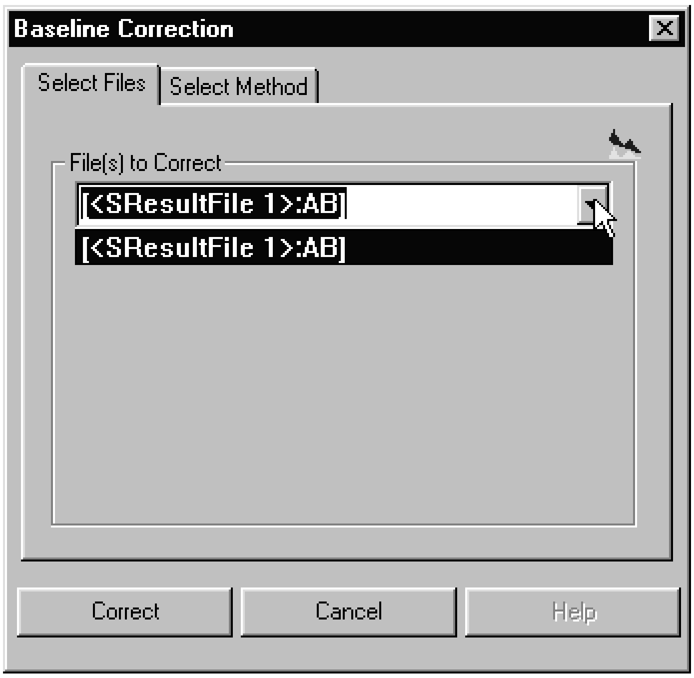


Figure 15: Inserting OPUS Commands

After clicking the command button to execute the command a dialog appears, listing all necessary parameters with their names and current values. In the last column choose a macro variable for any parameter from the list. The parameters will be assigned to these variables during runtime of the macro.

The checkbox at the beginning of each row determines, whether the parameter will be entered into the program line (box checked) or the parameter section. **Parameters which are assigned variables must be included in the program line.**

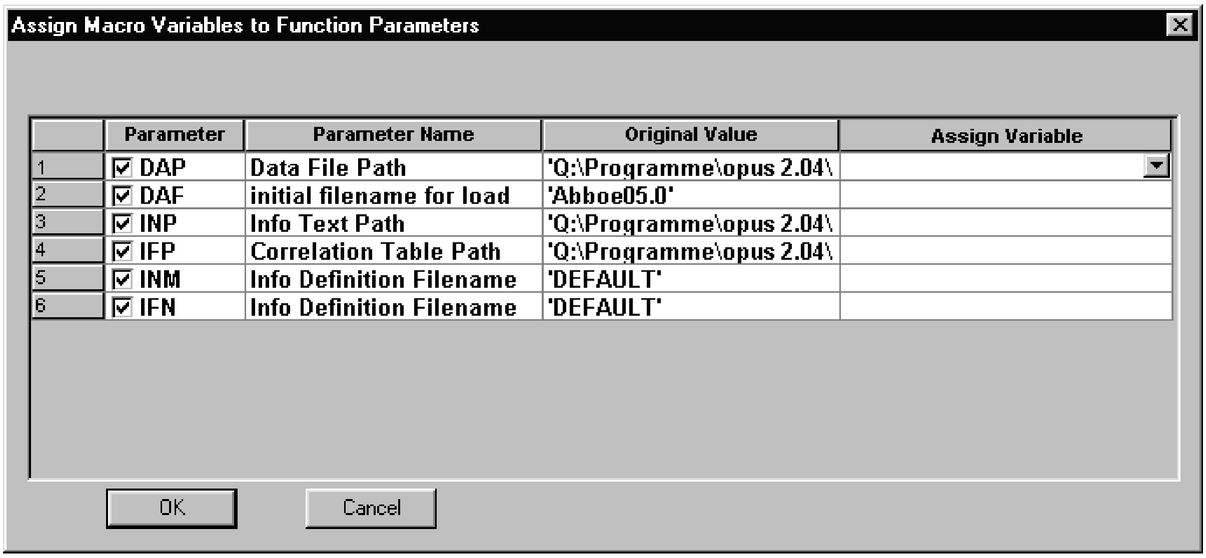


Figure 16: Assigning Parameters

Some OPUS functions are able to return results to the macro. For example a data acquisition generates a file that must be accessed by the macro. In this case the macro editor automatically generates a new FILE variable named <$ResultFile, x>, where x will be incremented automatically.

Example

<$ResultFile 1> = MeasureSample (0,{...

If OPUS functions return text instead (e.g. SendCommand), an additional selection list will be displayed above the parameter list of the parameter dialog. Choose a (already defined) variable from this list.

Example

<Result> = SendCommand (0,{...

5.4.5 Editing OPUS Command Lines

OPUS command lines are edited similar to macro command lines. First select the line by double-clicking it. Clicking on the ... button causes the dialog box of the function to open; the current parameters will be displayed. After clicking on the execution button of the function, the dialog box for parameter selection will appear.

5.5 Debugging Macros

The term “Debugging” means a step-by-step execution of a macro; otherwise the program execution continues until a stop mark is reached. This option greatly facilitates locating and analyzing errors.

You load the macro using the *Debug Macro* command; after you have opened the macro by double-clicking on the file name you see a list containing the first few lines of the macro program. Click on the *Variables* tab to obtain a list of all variables used in the macro and their current value.

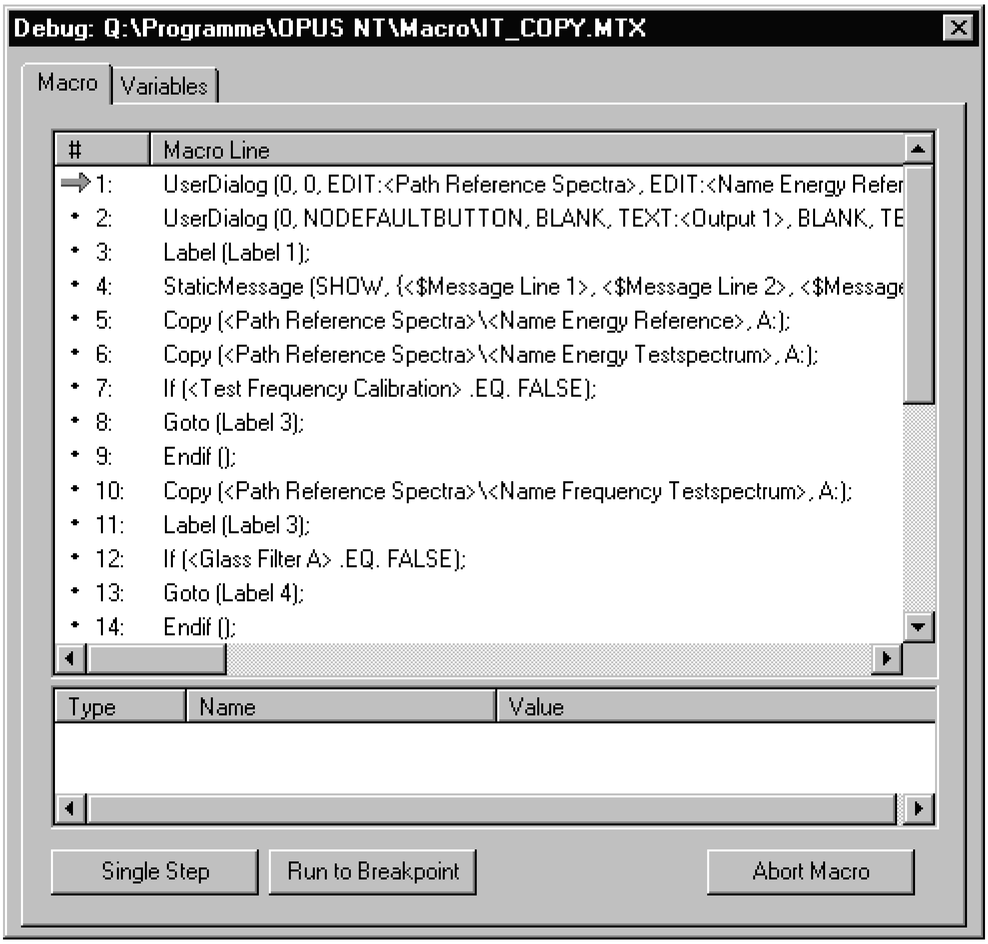


Figure 17: Debugging a Macro

Also you can search a text string within a macro or on the variable page. Enter the string you want to search in the entry field left from the *Search* button. Start the search by clicking on *Search*. The line containing the string will be selected. If you click on *Search* repeatedly the following occurrence of the string is found, until the macro has been completely scanned. After that, the search will begin again at the top of the macro. The line number in which the string was found will be displayed next to the *Search* button.

On the *Variables* page two lines allow separate searches for either the variable name or a variable value. The search run works like searching a text string in a macro. Please note that values of array variables can only be searched if the array variable is expanded by double-clicking on the preceeding plus sign prior to the search run (see below).

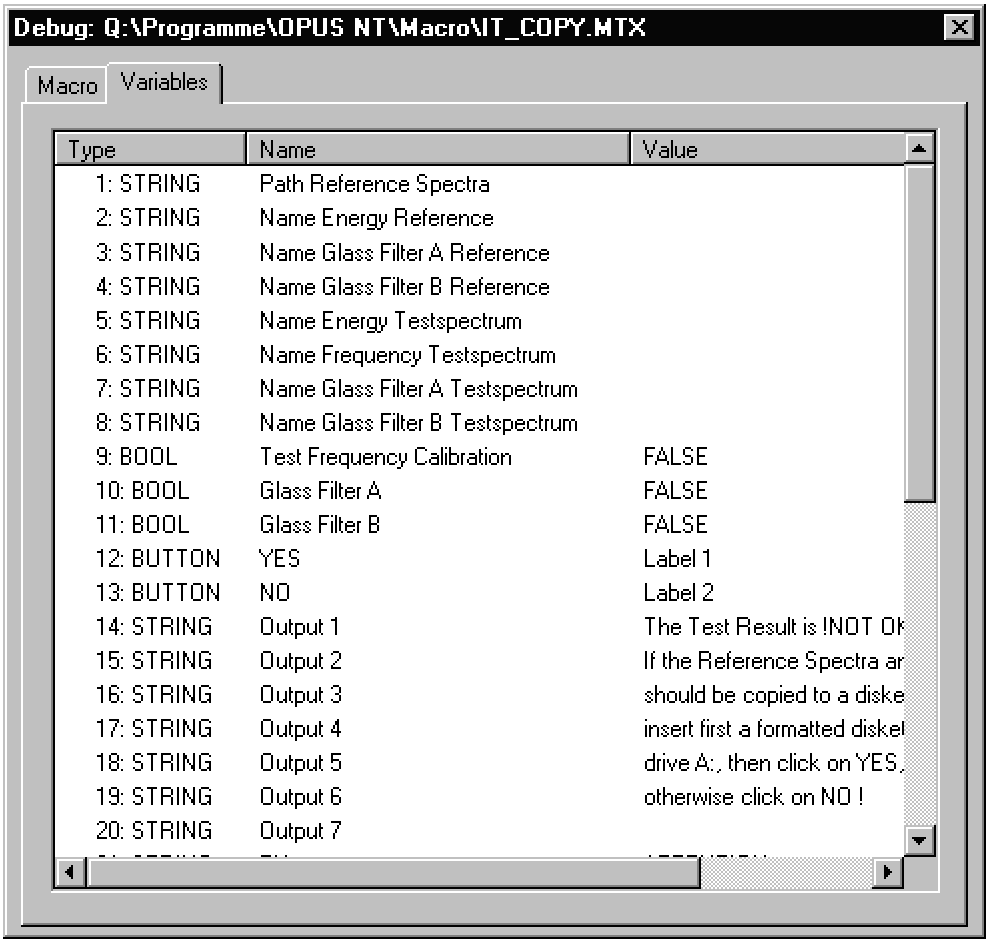


Figure 18: Debugging a Macro – Macro Variables

Arrays are marked by a plus sign in the *Variables* window; the value of an array is the one chosen for example from a selection of a pop-up menu. Double-clicking expands the array, that is every value of the array will be displayed. The index of each value is displayed in the “Name” column.

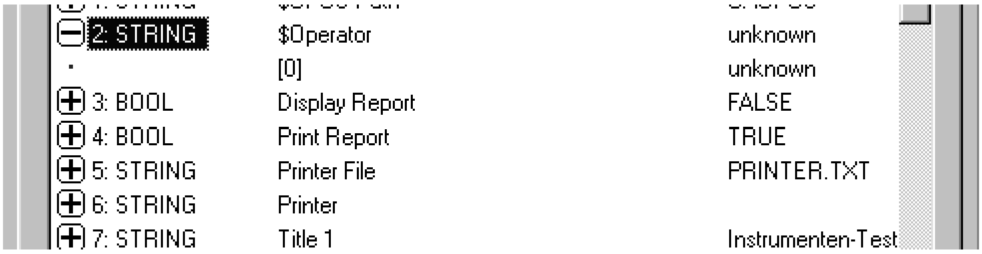


Figure 19: Macro Variables – Collapsed and Expanded Array

5.5.1 Stepping Through a Macro

The first line of the program is marked by a little green arrow to indicate the next line to be executed. This line can be executed by clicking the *Single Step* button. The arrow moves to the next command to be executed (not necessarily the next program line) and stops. As long as a command is being executed you cannot access the window. If the values of variables were changed by a macro command line, they will be displayed with their new values.

Variable values can be changed at run time in the debugger. Select the line containing the value to be changed. Enter the new value in the entry field next to the *Change Value* button and click on the button. For numeric variables only numbers are allowed. For BOOL variables either a number (0 or 1) or TRUE and FALSE are accepted.

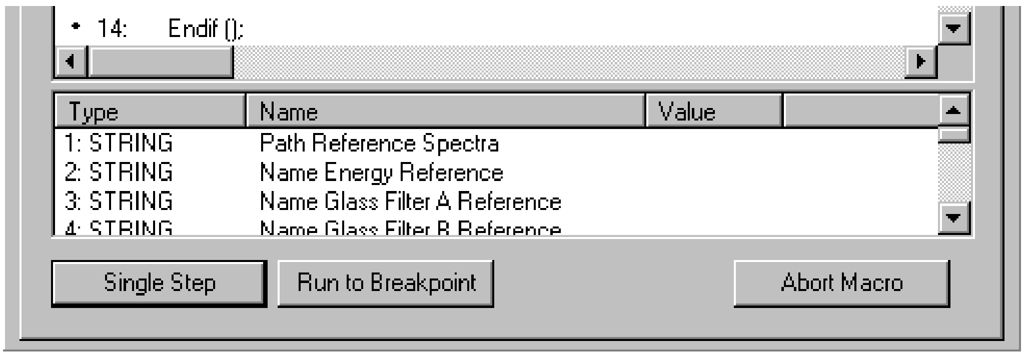


Figure 20: List of Macro Variables with Changed Values

5.5.2 Calling Sub Routines

If the debugger encounters a sub routine call (“Call Macro....”) while stepping through a macro, you have the choice between two options:

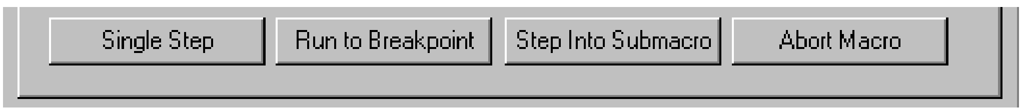


Figure 21: Macro Debugger – Options for Sub Routines

If you continue in *Single Step* mode the sub routine will be processed at once, i.e. the debugger evaluates the routine without explicitly stepping through it. The cursor stops at the next line of the main program. You should use this method only, if you are sure that the sub routine contains no errors.

On *Step into Submacro* an additional debugger window opens and the sub routine will be executed step-by-step. After the sub routine is completed its window will be closed automatically, and the execution of the main program continues. Use this mode if the sub routine is likely to contain errors.

5.5.3 Placing Stop Marks

Double-clicking on a line number in the debugger window will set a stop mark, indicated by a small icon at the beginning of the line. Remove the mark by double–clicking on it. A stop mark causes the debugger to halt at this program line, if *Run to Breakpoint* is used for the execution. This is especially helpful while debugging large macros, in case you are certain that the program code executed before the marked line is free of errors. *Run to Breakpoint* takes you directly to the line you have marked. Be sure that no branch occurs which causes the program to bypass the stop mark. In this case, or if no stop mark has been inserted, the macro will be executed completely without a stop.

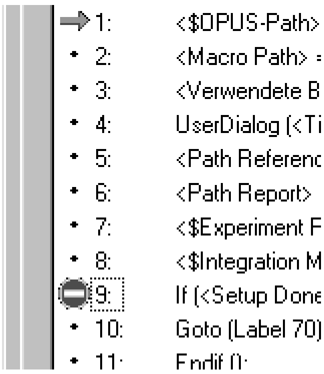


Figure 22: Stop Mark

5.5.4 Aborting a Macro

The *Single Step* mode can be aborted with the *Abort Macro* button. This command is not active, while the debugger is busy evaluating a command.

Breakpoints which are set during debugging can be stored in the macro text file with the *Store Breakpoints* button (see Figure 17). Whenever the macro is started in a debugger session these breakpoints will be activated automatically. When sub macros are called in a debug session, the debugger will automatically stop at the predefined breakpoints in the sub macro.

The breakpoints are stored in a new section starting with the keyword “BREAKPOINTS” (and followed by the line numbers of the breakpoints) which is appended to the macro. Please note that each line numbers requires a separate line and has to be terminated by a semicolon.

Example

```
BREAKPOINTS
3;
```

12;
38;

This will set breakpoints to lines 3, 12 and 38.

5.5.5 Automatic Stop

If a program line cannot be executed due to a programming error, the debugger stops running the macro; the cursor indicates the faulty command line.

5.5.6 Error Messages

While a macro runs only fatal error messages are displayed while other errors have to be handled with “If (MACROERROR, .EQ., TRUE);” constructs. When a macro is run in debug mode, all error messages will be shown to facilitate debugging and help locating critical sections in a macro.

5.6 Compiling Macros

Macros are generally written and stored as text files. During execution the macro text file is interpreted i.e. the text format is converted (compiled) into a binary format which can then be executed. The *Compile* function performs this step separately and generates a macro file with a binary format which can be executed directly.

Reasons to use compiled binary macros:

- a larger binary macro starts faster
- a binary macro cannot be modified by an unauthorized user

Please note that binary macros cannot be changed directly. If modifications are necessary you need to modify the original text based macro and compile it again. Compiled macros will also not run in the macro debugger.

To prevent permanent changes of macros using the CallMacro or RunMacro functions you need no longer specify the file extension for the sub-macro. If no extension is specified the system automatically uses the file type which is present. If both types are found the system will use the text version.

5.7 Macro Converter

Use the Macro Converter to translate macros written under OPUS-OS/2 into the OPUS-NT format. The conversion may require some changes as a result of the different macro syntax.

OS/2 macros all have the file extension ".MAC", OPUS-NT macros the extension ".MTX". The Macro Converter generates a text file with name of the OS/2 macro and the extension ".MTX" as well as a log file with name of the OS/2 macro and the extension ".LOG".

The Macro Converter was designed as an assistant, who guides you through the necessary steps. Calling up the Converter first displays some general information about the use of the program. On the second page, you specify the name of the OS/2 macro to be processed, either by typing it or use the *Change Macro* function to browse the directory. You can only switch to the next page, if you entered a valid macro name.

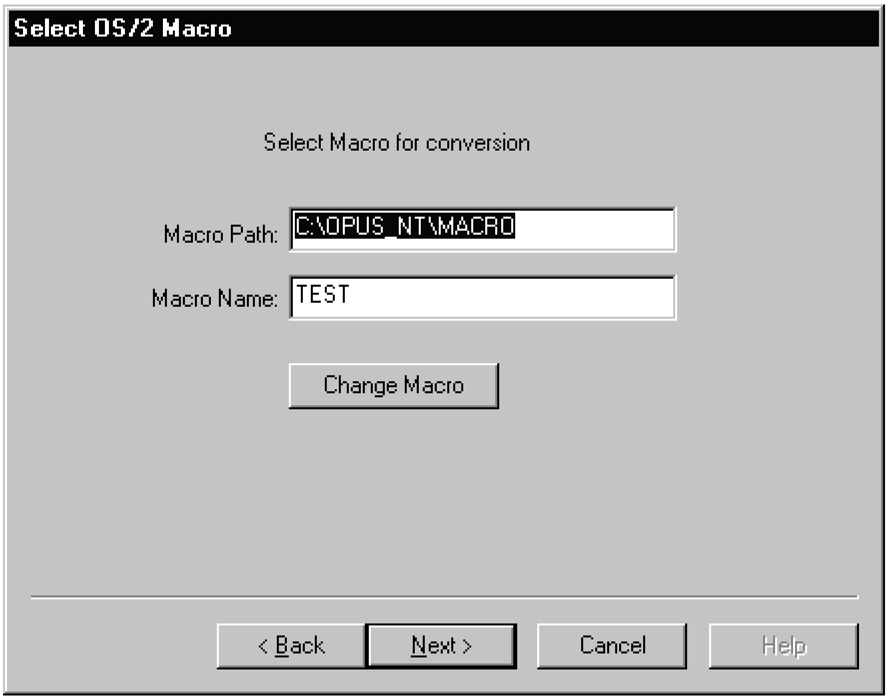


Figure 23: Select Macro for Conversion

The next page contains the settings for the destination directory, and the name of the resulting OPUS-NT macro. The destination directory is by default the same as the one containing the OS/2 macro. The default directory and macro name can be changed either manually, or by navigating to another directory using the *Change Output File* button. Start the conversion by clicking on *Finish*.

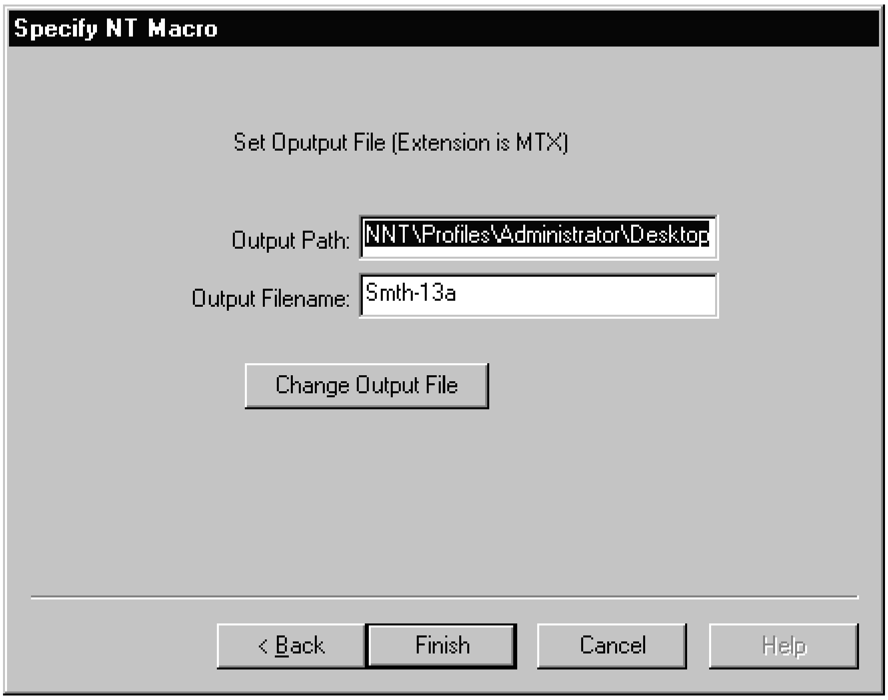


Figure 24: Define New Macro Name

Log Files

During the conversion of an OS/2 macro a log file will be generated containing

- indications which part of the macro are to be altered manually.
- indications about added code.
- indications about removed code.
- warnings for sub routine calls.

You should check the log file before you attempt to run the converted macro.

System Variables

If the OS/2 macro contains system variables (e.g. directories), code will be added during the macro conversion, that ensures the correct initialization of these variables. Information about which lines have been changed are available in the log file.

Example

STRING <$OPUS-Path>; (system variable of the main OPUS directory)

This variable will be initialized by:

<$OPUS-Path> = GetOpusPath();

Example

STRING <$Data File Path>(system variable of the OPUS data directory)

The variable will be initialized by:

```
<$Data File Path> = GetOpusPath();
<$Data File Path> = '<$Data File Path>\DATA';
```

Functions Not Implemented in OPUS-NT

Functions that are currently not implemented in OPUS-NT will be “commented out” by adding a REM command before the function. Information about which lines have been changed are available in the log file.

Measurement Commands

Only those parameters will be added in the command line of the Macro Editor, which were assigned variables. All other parameters are included in the PARAMETER SECTION. If the parameters XPP (experiment path) and EXP (experiment name) are not assigned, a warning will be included in the log file.

Calling Sub Routines

Sub routine calls always include a fixed path statement. A warning is entered in the log file.

Obsolete Parameters

Parameters that are obsolete in OPUS-NT will be handled like commands that are not implemented i.e. a REM command will precede the parameter and render it inactive.

5.7.1 Variables

The types of variables and their handling in OPUS-NT differ from their use in OS/2.

5.7.1.1 Variable Conversion

The following table indicates how variables are mapped.

OPUS-OS/2	**OPUS-NT**
NUMERIC	NUMERIC
TEXT FOR EDIT	STRING
TEXT FOR OUTPUT	STRING
CHECKBOX	BOOL
COMBOBOX	STRING (Array)
BUTTON	BUTTON
FILE	FILE
LABEL	no variable

5.7.1.2 Combobox Variables

COMBOBOX variables common in OPUS-OS/2 will automatically be converted to STRING variables in OPUS-NT. The values assigned to the COMBOBOX variable will be transformed to an array. Depending on the definition of the COMBOBOX variable, the STRING will be initialized in the PROGRAM section of the OPUS-NT macro as follows:

COMBOBOX with user-defined text:

```
<Combo>[0] = ’abc’;
<Combo>[1] = ’xyz’;
.....
```

COMBOBOX containing data of a text file:

```
<Combo> = ReadTextFile (D:\OPUS\PRINTER.TXT);
```

COMBOBOX containing the value of an Enum parameter:

```
<Combo> = GetEnumList (DXU);
```

5.7.1.3 Selecting Variables

Variables can be marked by an preceding “*” to cause the variable value to be refreshed, as it was practise in OPUS-OS/2. The new variable values will be entered directly in the VARIABLES section of the text file.

Example:

```
*STRING <Text> = ’Old Text’;
.....
<Text> = ’New Text’;
```

After the macro has been started once, the declaration line will change to:

```
*STRING <Text> = ’New Text’;
```

5.7.2 Differences in File Handling

The handling of spectrum files in OPUS-NT while executing OPUS functions, differs significantly from its OS/2 counterpart. In OPUS-NT the *Overwrite files* and *Create new files* options for functions from the *Manipulation* menu no longer exist. Instead of the original data file, an internal copy of the file is generated and used for data manipulation. The same procedure is applied by functions of the *Evaluation* menu (e.g. peak table generation, integration). The result is then appended to the copy of the data file in form of a new data block. These modifications are indicated by a red rectangle next to the file name in the OPUS Browser, while a blue rectangle symbolizes an unmodified data file.

The result of a manipulation can be stored by saving the file. Therefore, such an OPUS command will be translated in different ways, depending on the options chosen. The following examples illustrate this using the *Baseline Correction* command.

Example – *Overwrite Files*

```
VARIABLES SECTION
FILE <File> = AB;

PROGRAM SECTION
Baseline([<File>:AB] , {BME=1, BCO=0, BPO=64});
```

Note that the modified file is **not** saved by the macro. If you want to save the result, you need to include the *Save* command or save the file manually.

Example – *Create new files*

```
VARIABLES SECTION
FILE <F1> = AB;
FILE <@result401>;

PROGRAM SECTION
Baseline([<F1>:AB] , {BME=1, BCO=0, BPO=64});
SaveAs([<F1>:AB], {OEX='1', COF=2, SAN=WORK.301,
DAP=E:\opus\WORK});
Restore ([<F1>:AB],{});
<@result401> = LoadFile (E:\opus\WORK\WORK.301, WARNING | ABORT);
```

If the *Create new files* option was chosen, the converter expands the OPUS-OS/2 command automatically to guaranty compatibility.

line 1: Baseline correction of the original file.

line 2: The result of the correction is saved as a work file.

line 3: The original file is restored.

line 4: The work file is loaded for further access by the macro.
The work file is stored in the directory indicated in the macro.

5.7.3 System Directories

The system path variables of OPUS-OS/2 are no longer supported. These variables will be initialized by the appropriate command lines.

5.7.4 Function Parameters and Parameter Assignment

The command lines of a converted macro contain all necessary parameters of OPUS functions (except the measurement functions) in a parameter list. New parameters which did not exist in OPUS-OS/2 will be taken from the active standard parameter set during conversion.

Variables for parameter values that were used in OPUS-OS/2 in the form of an assignment, are now included directly in the parameter list of the command (e.g. ..., DAP = <data path>, ...). They will be replaced by their current value prior to the execution of the command.

5.7.5 Time-behavior of Macros

OPUS-OS/2 macros are executed asynchronous, i.e. commands are sent to the OPUS task manager, who then decides when to execute the commands. Only a few functions (e.g. dialog boxes, wait function) were able to pause a macro until all commands forwarded to the task manager were processed. In some cases a necessary synchronization of the macro had to be achieved by including wait functions with the wait time set to 0.

OPUS-NT macros are executed synchronous, i.e. the next command will only be executed after the termination of the currently processed command. This makes the use of wait functions as described above obsolete. These commands are removed upon macro conversion.

5.7.6 Print Functions

Currently, the *Print* function has not been implemented. OPUS-OS/2 print commands will be converted to PrintToFile function. Make sure, that the output directory and file name was set properly.

5.7.7 Calculations with Variables

So far, calculations that contained only variables (and no spectral data) had to be performed with the spectrum calculator. OPUS-NT supports the direct use of mathematical expressions. For example

```
<Number> = <Number> + 1;
```

is a valid statement in an OPUS-NT macro. This change in syntax considered by the macro conversion routine.

5.7.8 Jump Instructions

OPUS-OS/2 jump instructions are replaced by “Goto (Label)” statements. Conditional Jumps are expanded to three lines:

```
OS/2 Macro:Jump to Label if Expression;
NT Macro:If (Expression);
                    Goto Label;
                    Endif;
```

Please note, that jump statements from OPUS-OS/2 macros can be simplified by using the new If ... Else ... Endif structure. However, this is not done automatically; in case of macros containing several conditional jumps we recommend to manually replace the If ... Endif structures by If ... Else ... Endif structures.

5.7.9 Start Loop with For Each Option

The “Start Loop” instruction in combination with the option “For each File” is now identical to all other “StartLoop” instruction. Instead of the counter or a NUMERIC variable, just state the FILE variable as a counter.

Example:

```
FILE <File> = AB;
.....
StartLoop ([<File>], 0);loop count = number of files selected
```

5.7.10 Load Multiple Files

The “Load” function with the “Load multiple” option is replaced by the general “LoadFile” macro function, followed by a “StartLoop” instruction, with the FILE variable specified for the loop count.

Example:

```
<File> = LoadFile (’D:\OPUS\DATA\*.0’,  WARNING | ABORT);
StartLoop([<File>], 0);
.....
EndLoop(0);
```

5.7.11 User Dialogs

In OPUS-OS/2 macros, the appearance of lines in a user-defined dialog box was determined by the variable type. Now, the appearance of a dialog box line is almost independent of the variable type. It is controlled by keywords added to the “UserDialog” command.

Examples:

```
STRING <Text>;
```

The variable <Text> can be used for an “Edit” control as well as for comment lines or a combo box.

EDIT:<Text> shows an “Edit” control.

TEXT:<Text> shows a comment line.

COMBOBOX:<Text>shows a Combobox.

5.7.12 Client/Server Calls

The Client/Server function of OPUS-OS/2 is also implemented in OPUS-NT.

Concerning the operating system, the OPUS-NT function *External Program* differs slightly from the OPUS-OS/2 Client/Server function. If the external program runs in the Windows NT environment, it can be started in the same way as in OPUS-OS/2.

Not supported are OS/2 programs with a graphical user interface and Rexx scipts unique for OS/2. Simple DOS based software is supported by OPUS-NT.

For Named Pipes there apply certain restriction in Windows NT. While in OPUS-OS/2 “\PIPE\OPUS\PROGRAM.EXE” was used as default name, Windows NT expects a pipe name of the form “\\.\PIPE\PROGRAM.EXE”. Self-written software that is supposed to exchange data with OPUS-NT in that manner has to be adjusted if you want to use it in Windows NT.

In case the macro should wait until the external program is terminated the *Wait for program to terminate* box has to be checked.

The function External Program supplies the return code of the external software or the result of a DDE command as a parameter RS1. The return code is saved as a STRING variable.

5.7.13 Conversion Functions

Conversion functions (like JCAMP, data point table) no longer exist in OPUS-NT. Non-OPUS files are converted automatically upon loading the file, if OPUS recognizes the file format. The converted file will then be saved in a different format, if the appropriate switches are set when using the *SaveFile* command. When converting OPUS-OS/2 macros these changes have to be done manually.

5.8 Writing Portable Macros

A portable macro allows to copy and run a macro written on a specific system on any other system. This will be straightforward if both systems have an identical directory structure which is expected by the macro.

In general, there are several ways to write such macros:

1) All drive and path specifications are stored in variables, which then can be set via a user dialog. If these variables are marked for update, they must only be set during the initial run of the macro. If only a few paths are included in the macro, this may be acceptable, but with an increasing number of paths involved it can become tedious work.
2) A much better and preferable solution is to use a variable home directory with a fixed subdirectory structure. Not only the single variable, which has to be set, but also the transparency of this macro to the user is guar-

anteed, because results are located in the same subdirectories on all systems.

3) The best solution uses either the OPUS path or a User path as home directory, with a fixed sub directory structure. In this case, the first command line in a macro must be <Path> = GetOpusPath (); or <Path> = GetUserPath (); which sets the variable to the current path. There is no need to set the path manually.

During the installation of OPUS-NT, some sub directories are already created. These directories are accessible by all users. If access to some of the sub directories should be restricted, it is still recommended to maintain the OPUS sub directory structure.

OPUS	the main path for the OPUS program and all files necessary to run OPUS.
DATA	sample data.
MEAS	measured spectra .
XPM	experiment files.
METHODS	method files for integration, QUANT, IDENT etc.
MACRO	macros.
SEARCH	intermediate search reports and method files.
PRINTS	output from printing into a file.
SCRIPTS	plot layout scripts.

6 How to Write Macros

In the following you will find various examples of macros used in every-day laboratory work. All of these examples are written with the OPUS-NT Macro Editor. To help you understand how the Macro Editor works, we will explain every step in detail for the first few macros. If you are not familiar with the Editor, we recommend that you work through these macros step by step as they are listed here. After finishing this chapter you should be able to design complex macros by yourself. All of the examples are also available as files on the OPUS-NT CD.

The following chapters are divided into several sections:

- **Task**: the purpose of the macro
- **Macro functions**: explanation of the macro commands used in the macro
- **OPUS functions**: explanation of the OPUS commands used in the macro
- **Generating the macro**: the generation of the macro code is explained step by step
- **Listing**: a listing of the macro code
- **Running the macro**: the specific features of the macro are emphasized

Before you start create a directory where you will save your macros. Keep in mind that sometimes macros build on macros written earlier in this chapter.

6.1 General Remarks

6.1.1 Syntax

A detailed description of the macro syntax is given in chapter 9. Since we are exclusively using the Macro Editor, all code will automatically be generated according to the syntactical rules. However, if you like to try the direct command entry using a text editor, you should read the corresponding chapter first.

6.1.2 The Use of Variables

A macro has to fulfill various tasks, like data handling and data processing, or making decisions depending on the outcome of an OPUS report. These tasks can only be performed effectively, if the macro operates with variables instead of constants like file names, dates etc. Therefore, the OPUS macro systems offers several types of variables.

If a numerical value should be changed during run time of the macro, a variable of the type NUMERIC has to be employed. For file names on the other hand, which can be supplied by the user during run time, a variable of type TEXT is to be used.

6.1.3 Variable Names

A variable is identified by its unique name. The name will be displayed in dialog boxes, and should therefore express the purpose of the variable. For example, the use of the name “x start frequency” provides more information that just the name “x”.

6.1.4 Variable Types

Several types of variables are available in the OPUS-NT macro language. They are summarized in the following; details about the syntax are given in chapter 9

Text Variables

Text variables are used to save text, like file names or results.

Values can either be entered through dialog boxes, using statements or read from parameters or reports.

Numerical Variables

Numerical variables are used to save numbers.

Values can either be entered through dialog boxes, using statements or read from parameters or reports.

Boolean Variables

Boolean variables are used to save the values TRUE or FALSE.

Values can either be entered through dialog boxes, using statements or read from parameters or reports.

File Variables

File variables are used to save spectra.

Spectra that have already been loaded into the OPUS Browser are assigned to the variable using a dialog box. Spectra that were generated by a macro are assigned via statements.

Command Buttons

Command buttons are used to control the flow of a macro.

Command buttons cannot be changed during run time, and therefore are not variables in the true sense. However, they are declared in the variable section, and are always linked to a Goto instruction condition.

Except command buttons all variables can also be used as arrays, i.e. they can hold more than one value. The different values are addressed using index numbers.

6.1.5 Variable Type Conversion

Text and numerical variables (and boolean variables under certain conditions) can be converted into each other. Detail are given in the Macro Reference section.

6.2 Measure 1 – A Simple Macro

Task

Acquire a spectrum using a macro.

Macro Functions

This example uses no special macro functions.

OPUS Functions

Measurement

The measurement command has a special status compared to normal OPUS functions; data acquisition requires a multitude of parameters, some of which are linked to each other. Before you use the measurement command in a macro, you need to define an experiment file, which you will use in combination with the macro. For the use with this example macro, you **must** generate an experiment file “DEFAULT.XPM”, which we save in the directory C:\OPUS_NT\XPM. The type of data acquisition should be set to absorption spectrum.

Generating the Macro

1) Create an experiment file in case you haven’t done so already.
2) From the *Macro* pull-down menu, open the *Macro Editor.* The Editor opens with two empty windows, one for program code and the other to

display variables. The functionality of the Macro Editor was explained in chapter 5.4.

3) From the OPUS *Measure* pull-down menu, select the *Measurement* command; this will open the *Measurement* dialog box.

4) If the experiment file you created is not listed, you have to load it by clicking the *Load* button on the first page of the dialog.

5) Now click on *Collect Sample*. A dialog with all parameters of the *Measurement* command is displayed.

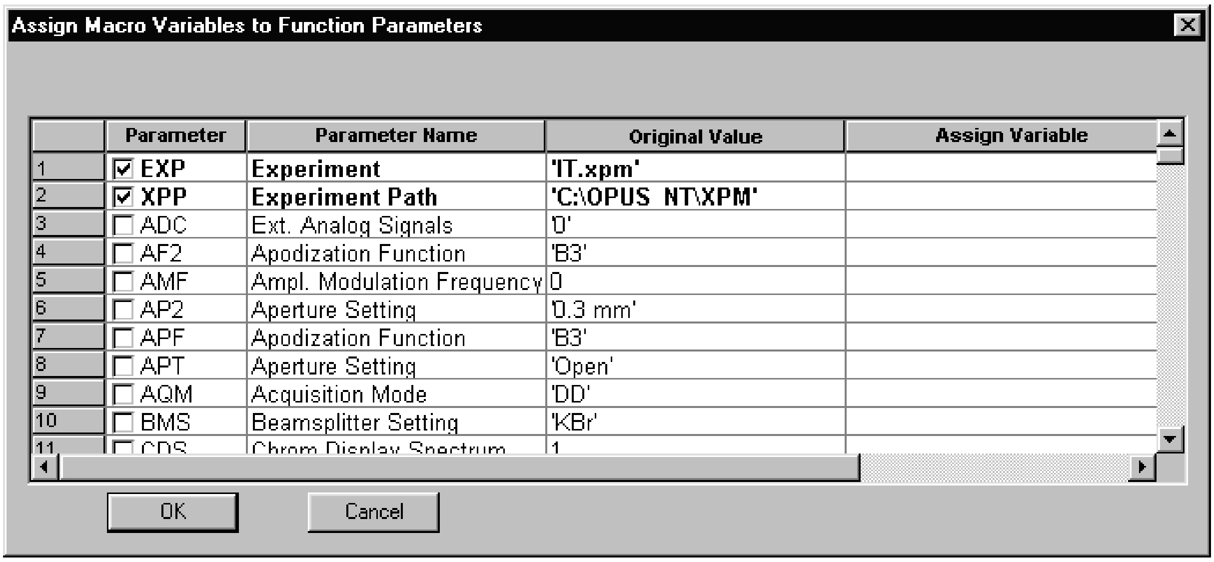


Figure 25: Assigning Macro Variables

The first column list the Parameter abbreviations that are used by OPUS. In addition, a check box controls, whether the parameter will be entered in the command line or the parameter section. While for all other OPUS commands these check boxes are selected by default, this is not the case for the *Measurement* command. Only “Experiment” and “Experiment Path” are checked.

In the second column, the parameter name is displayed followed by the current value of the parameter listed in the third column. The values for “Experiment” and “Experiment Path” are the ones you entered in the *Measurement* dialog box.

The last column is reserved for macro variables. In our case all table cells in this column are empty.

6) Click on *OK*. The following line will appear:

```
<$ResultFile 1> = MeasureSample (0, {EXP=’it.xpm’, XPP=’C:\OPUS_NT\XPM’});
 1          2         3        4  5                 6                      7
```

1: <$ResultFile 1> is a FILE variable automatically generated by the Macro Editor, which represents the acquired spectrum. This variable appears also in the variable list, together with the name and the spectrum data block as specified in the experiment file.

2: The equal sign indicates that the function returns a file or a value, which will be assigned to the variable.

3: The name of the *Measurement* command is MeasureSample. You find a list of all OPUS command names in the OPUS Command Reference section.

4: For all OPUS commands the file list follows the parentheses. Because no spectrum file is processed during the measurement, the value is 0. Note that a file list is mandatory for OPUS functions.

5: Separated by a comma and enclosed in braces is the parameter list of the function. This list usually consists of several parameters separated by commas. If no parameters are used the braces will be empty.

6: A parameter declaration consists of the abbreviation of the parameter, an equal sign and the parameter value. In case the value are characters (text), they must be hyphenated.

7: A close bracket followed by a semicolon indicates the end of the command line.

7) The newly generated variable will be declared in the variable list:

FILE <$ResultFile 1> = AB;

1 2 3 4 5

1: The variable type is FILE.

2: The name of the variable always has to be enclosed by the “smaller as” and “greater than” signs. Variables, that are automatically generated, are preceded by a Dollar sign to distinguish them from user-defined variables.

3: The equal sign is mandatory in the case of FILE variables.

4: The equal sign is followed by the list of data blocks of the selected file (only in the case of FILE variables). In our example, the file consists only of one spectral data block, the absorbance spectrum AB. If a file consists of several data blocks, these are separated by commas.

5: The end of the line is indicated by a semicolon.

8) Now save the macro using the *Store Macro* button. In the Save File dialog, navigate to the directory you generated before and enter “Measure 1” as file name. Click on *Save*.

9) Exit the Macro Editor by clicking on *Exit*.

10) You can display the macro using any kind of text editor (e.g. Notepad).

Listing (MEASURE 1.MTX)

```
VARIABLES SECTION
FILE <$ResultFile 1> = AB;

PROGRAM SECTION
<$ResultFile 1> = MeasureSample (0, {EXP='it.xpm',
 XPP='C:\OPUS_NT\XPM'});
PARAMETER SECTION
```

Note that the parameter section contains no entry; this also is a specific feature of the *Measurement* function.

Running the Macro

- Before you run the macro, first collect a background spectrum using the experiment file you created to be used with the macro.

- Run the macro with the *Run Macro* command from the *Macro* pull-down menu.

6.3 Measure 2 – A Macro Including Data Manipulation

Task

The macro should acquire a spectrum and perform a baseline correction. A peak table should be generated from the result spectrum.

We will built on the macro “Measure 1”, which we wrote in chapter 6.2.

Macro Functions

This example uses no special macro functions.

OPUS Functions

Measurement, Baseline Correction, Peak Picking

Contrary to the *Measurement* command, the OPUS functions responsible for the data processing need far less parameters. We recommend to always include all parameters, as shown in our example, into the command line.

Generating the Macro

1) From the *Macro* pull-down menu, open the *Macro Editor*. Load “Measure 1” by clicking on *Open Macro*. Path and name are shown in the line below the buttons.
2) Appending code to an existing macro is simple; just apply the OPUS function you would like to include. Select *Baseline Correction* from the *Manipulation* pull-down menu.
3) Instead of the file selection box on the first page of the dialog box, a list appears. Click on the triangle button to open the drop-down list, and select `[<$ResultFile 1>:AB]` from the list. The list contains all variables declared in a macro, in this example only one, instead of file names. Parameters are selected as usual on the second page of the dialog. Select *Rubber Band* correction.
4) Now click on *Correct*. The dialog box containing the functions’ parameters will be displayed. By default, all parameter check boxes have been selected.
5) After clicking OK, the function will be appended to the existing macro code in the command window:
```
Baseline ([<$ResultFile>:AB], {BME=2, BCO=0,
BPO=64});
```

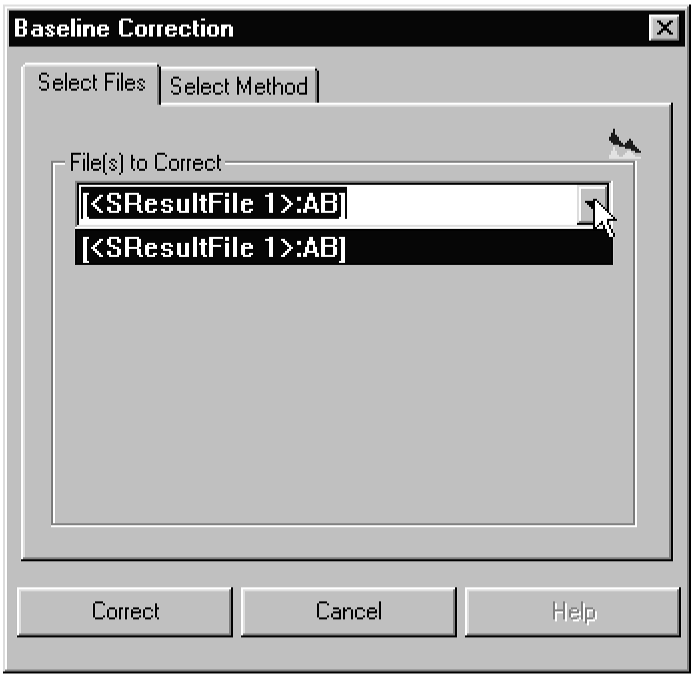


Figure 26: Selecting the FILE Variable

6) From the *Evaluation* pull-down menu, choose *Peak Picking*. Similar to step 3 select the FILE variable from the list that is displayed on the first page. On the remaining pages define the parameters as usual, and click on *Peak Picking*.
7) Again, the functions' parameter box will open. You don't need to make any changes, just click on *OK* to close the box. As a result, the following line will be appended to the macro:

```
PeakPick ([<$ResultFile 1>:AB], {NSP=9, PSM=1,
WHR=0, LXP=400.000000, FXP=4000.000000, QP8='NO',
QP9=0.200000, PTR=20.000000, QP4='NO', QP7=
0.800000, QP6='NO', QP5=80.000000, PPM=1, QP0='NO',
QP3=4});
```

   The result of the *Peak Picking* function is stored in an extra data block (peak report). This block is automatically added to the FILE variable:

```
FILE <$ResultFile 1> = AB, AB/Peak;
```

8) Save the macro by clicking on *Store Macro*. Contrary to the first example, the macro name has already been entered in the *File Name* field, because we edited an existing macro. Change the file name to "Measure 2" and click on *Save*. Exit the Macro Editor.

Listing (MEASURE 2.MTX)

```
VARIABLES SECTION
FILE <$ResultFile 1> = AB, AB/Peak;

PROGRAM SECTION
<$ResultFile 1> = MeasureSample (0,{EXP='Default.xpm',
XPP='C:\OPUS_NT\XPM'});
Baseline ([<$ResultFile 1>:AB], {BME=2, BCO=0,
```

```
BPO=64});
PeakPick ([<$ResultFile 1>:AB], {NSP=9, PSM=1, WHR=0,
LXP=400.000000, FXP=4000.000000, QP8='NO',
QP9=0.200000, PTR=20.000000, QP4='NO', QP7=0.800000,
QP6='NO', QP5=80.000000, PPM=1, QP0='NO', QP3=4});

PARAMETER SECTION
```

Running the Macro

Upon running the macro, a spectrum will be acquired and post-acquisition data processing is performed. The result is a baseline corrected spectrum with a peak table attached.

6.4 Measure 3 – Repeated Data Acquisition Using a Loop

Task

The macro “Measure 2” should be expanded to perform three concurrent sample measurements. In addition, a background spectrum should be measured prior to the data acquisition.

Macro Functions

StartLoop, EndLoop

If a sequence of functions is to be repeated several times, the problem could be tackled by repeatedly including the code responsible for the function. However, there is a more elegant solution. The StartLoop and EndLoop commands define a loop with a counter (Loop Count), that repeats the code enclosed by these commands as often as indicated by the counter.

OPUS Functions

Measurement, Baseline Correction, Peak Picking

Only a background spectrum acquisition is added. The same conditions as for the sample measurement apply.

Generating the Macro

1) From the *Macro* pull-down menu, open the *Macro Editor*. Load “Measure 2” by clicking on *Open Macro*.
2) Select the first line with a single left click of the mouse. From the OPUS *Measure* pull-down menu, select the *Measurement* command. If necessary, load your experiment file and click on *Collect Background*. Exit

the dialog box that appears next without changing any of the parameters by clicking on *OK*. The following code will be appended to the macro:

```
MeasureReference (0, {EXP='Default.xpm',
XPP='C:\OPUS_NT\XPM'});
```

3) Select the line as usual (one left-click with the mouse). Now move the line to the beginning of the macro by clicking on the button. Alternatively, double-click on the command line, and drag it to the top of the list while holding the left mouse button down.

4) Next, we will include a command which is not part of the OPUS pull-down menu. Clicking on inserts a blank line in the command window just below the first line

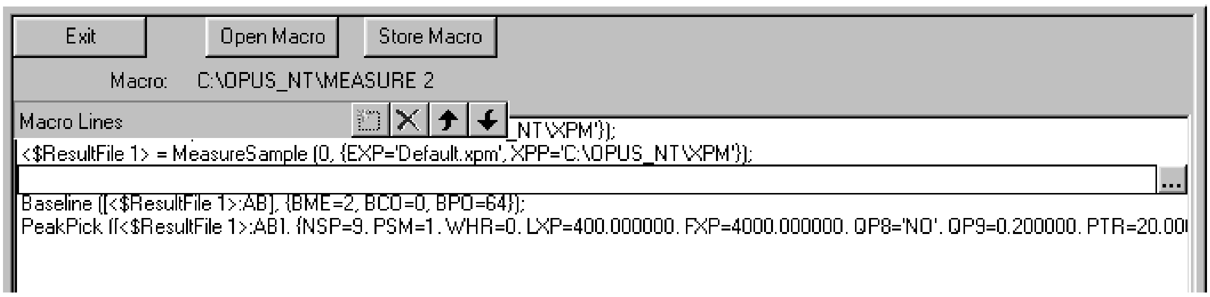


Figure 27: Manually Inserting a Command

5) Press on the button to open the *Special Macro Commands* dialog box.

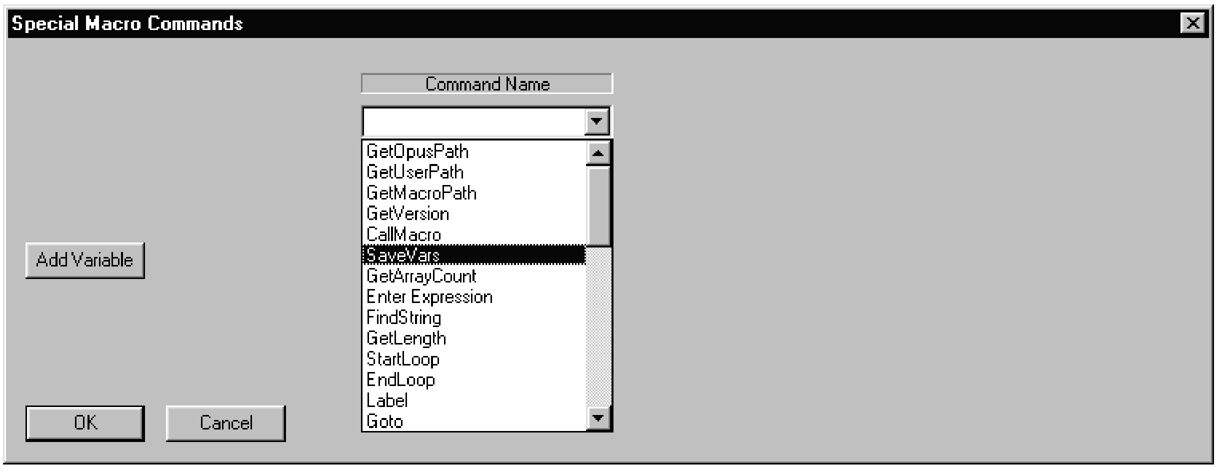


Figure 28: *Special Macro Commands* Dialog

6) From the drop-down list choose the StartLoop command. Two additional parameter fields are displayed.

7) As Loop Count enter “3” and “0” in the Loop Index field. Both fields are empty because no matching variables have been declared so far. Click on *OK*.

`StartLoop (3,0);` is inserted.

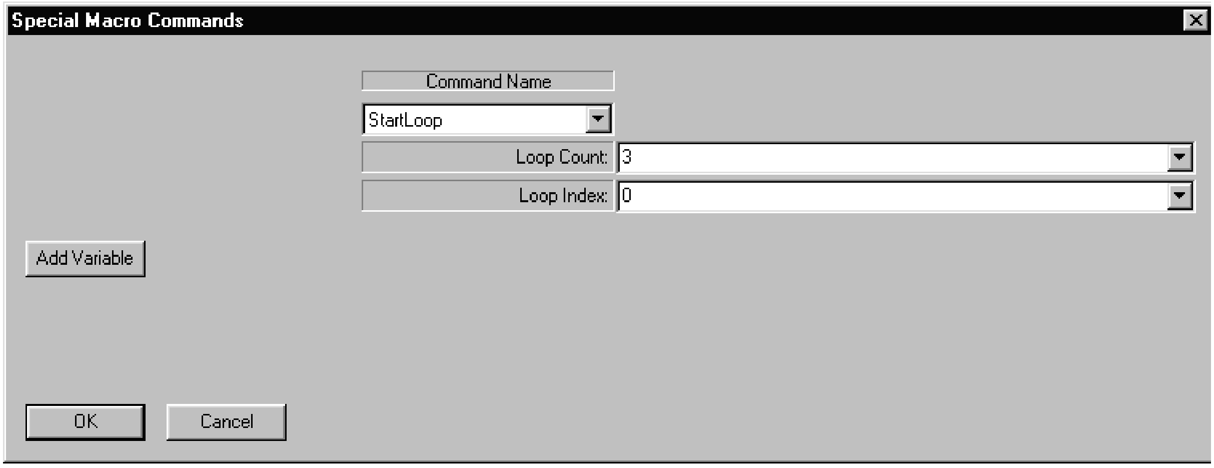


Figure 29: Defining StartLoop Parameters

8) Create another blank line in the command window. Call up the *Special Macro Commands* dialog and choose the EndLoop command. This command requires only the Loop Index as a parameter. The Loop Index links the EndLoop command to the correct StartLoop command; therefore, enter “0”. Especially when several loops are used make sure the correct StartLoop and EndLoop commands are linked.
9) Save the macro as “Measure 3” and exit the Macro Editor.

Listing (MEASURE 3.MTX)

```
VARIABLES SECTION
FILE <$ResultFile 1> = AB, AB/Peak;

PROGRAM SECTION
MeasureReference (0, {EXP=’Default.xpm’,
XPP=’C:\OPUS_NT\XPM’});
StartLoop (3, 0);
<$ResultFile 1> = MeasureSample (0,
{EXP=’Default.xpm’,
 XPP=’C:\OPUS_NT\XPM’});
Baseline ([<$ResultFile 1>:AB], {BME=2, BCO=0,
BPO=64});
PeakPick ([<$ResultFile 1>:AB], {NSP=9, PSM=1, WHR=0,
LXP=400.000000, FXP=4000.000000, QP8=’NO’,
QP9=0.200000, PTR=20.000000, QP4=’NO’, QP7=0.800000,
QP6=’NO’, QP5=80.000000, PPM=1, QP0=’NO’, QP3=4});
EndLoop (0);

PARAMETER SECTION
```

Running the Macro

Upon running the macro, a background spectrum will be acquired and subsequently three sample measurements. However, there is no pause between the sample measurements to change or manipulate the sample. This will be part of the next example.

6.5 Measure 4 – Interacting with the User

Task

Modify “Measure 3” to pause between the repeated data acquisitions. Prompt the user to insert a new sample before the next measurement.

Macro Functions

StartLoop, EndLoop, Message

The Message function displays a message in a dialog box either for a defined time or until the user closes the dialog box.

Note: the option ON_PRINTER is not available in OPUS-NT 2.0.

OPUS Functions

Measurement, Baseline Correction, Peak Picking

This example introduces no new OPUS functions.

Generating the Macro

1) Open the *Macro Editor* and load “Measure 3”.
2) Insert a blank line in the command window just below the “StartLoop” instruction, and open the *Special Macro Commands* dialog.
3) From the list, choose the Message command; quick-select by typing the first letter “M” of the command name after opening the drop-down list.
4) Three parameters are required for the Message function. Enter “Please insert sample.” into the text field. Don’t forget to enclose the text in single quotes.
5) “Option” – the second parameter – is a so called keyword. Open the list and select “ON_SCREEN”, which causes the message to be displayed on the computer screen.
6) “Time” specifies how long (in seconds) the message will be displayed. Also, the HH:MM:SS format (hours, minutes, seconds) is accepted. If you would like the user to confirm the message, choose NO_TIMEOUT as keyword. The message will be displayed, until the user clicks the *OK* button.

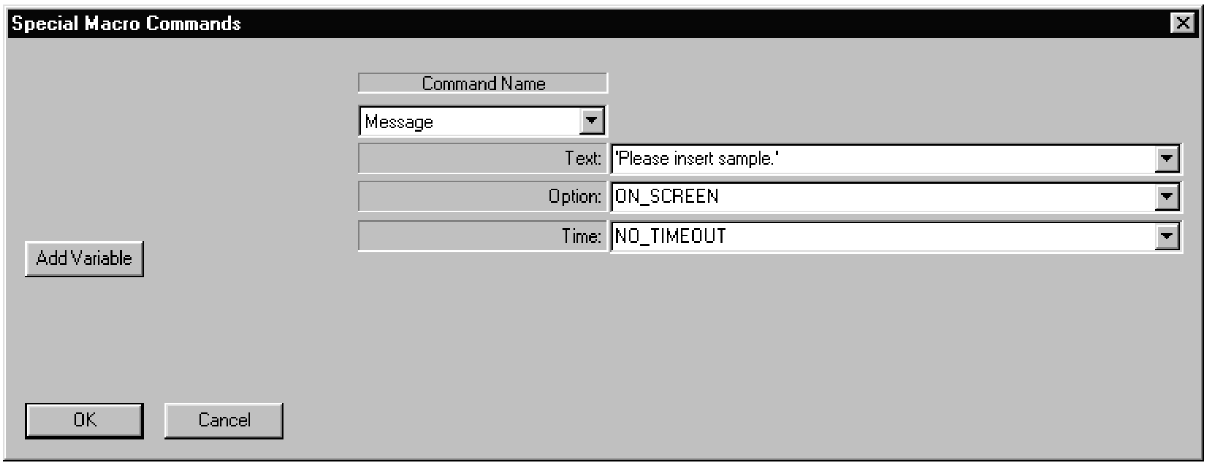


Figure 30: Defining StartLoop Parameters

7) After defining all parameters, close the dialog by clicking *OK*.
8) Save the macro as “Measure 4” and exit the Macro Editor. If you click *Exit* without saving the file, you will be asked to save your work before closing the Macro Editor.

Listing (MEASURE 4.MTX)

```
VARIABLES SECTION
FILE <$ResultFile 1> = AB, AB/Peak;

PROGRAM SECTION
MeasureReference (0, {EXP='Default.xpm',
XPP='C:\OPUS_NT\XPM'});
StartLoop (3, 0);
Message ('Please insert sample.', ON_SCREEN,
NO_TIMEOUT);
<$ResultFile 1> = MeasureSample (0,
{EXP='Default.xpm',
XPP='C:\OPUS_NT\XPM'});
Baseline ([<$ResultFile 1>:AB], {BME=2, BCO=0,
BPO=64});
PeakPick ([<$ResultFile 1>:AB], {NSP=9, PSM=1, WHR=0,
LXP=400.000000, FXP=4000.000000,
QP8='NO', QP9=0.200000, PTR=20.000000, QP4='NO',
QP7=0.800000,
QP6='NO',
QP5=80.000000, PPM=1, QP0='NO', QP3=4});
EndLoop (0);

PARAMETER SECTION
```

Running the Macro

Start the macro. After the background spectrum acquisition has finished, a message “Please insert the sample.” will be displayed before starting each sample measurement. The macro pauses until you confirmed the dialog by clicking *Continue*.

6.6 Measure 5 – Variable Loop Counters

Task

So far we used a fixed Loop Counter for the command repetition. In your every day work you might find it more suitable to let the user choose how many repetitions he actually needs upon launching the macro. This requires a variable Loop Counter.

Macro Functions

StartLoop, EndLoop, Message, UserDialog

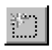
A numerical variable is used for the Loop Counter. The variable value will be entered in a dialog box.

OPUS Functions

Measurement, Baseline Correction, Peak Picking

This example introduces no new OPUS functions.

Generating the Macro

1) Open the *Macro Editor* and load "Measure 4".
2) Insert a blank line in the **variable window** by clicking on the  button on top of the variable window. Open the *New/Edit Variable* dialog by clicking on .
3) In the top section of the dialog box you specify the variable type: select "NUMERIC".

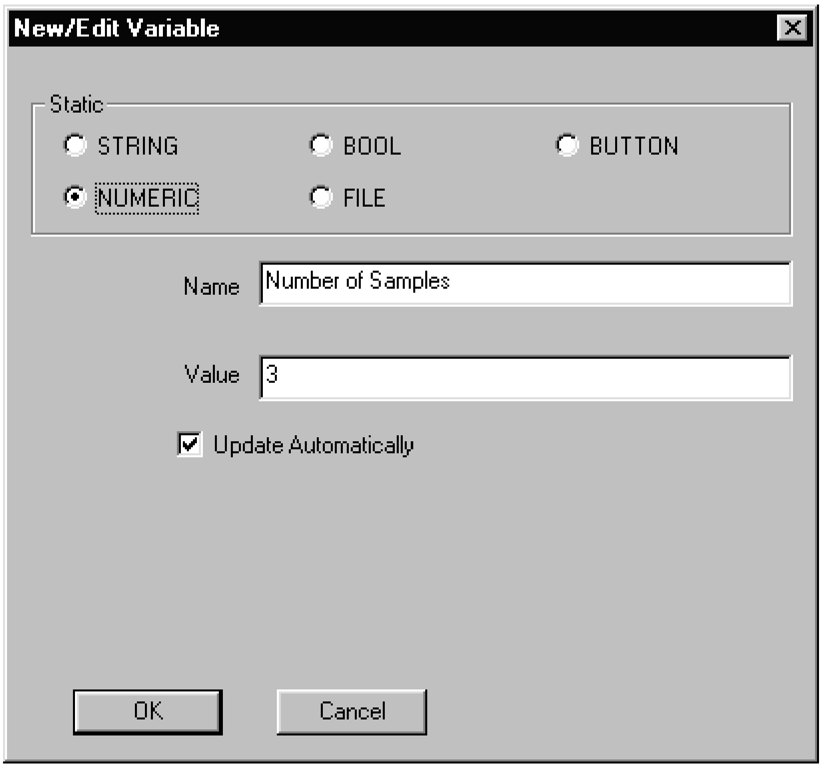


Figure 31: *New/Edit Variable* Dialog Box

4) Enter “Number of Samples” in the *Name* field.
5) Change the default value “0” of the *Value* field to “3”. This defines the starting value which will later be displayed in the dialog box.
6) Check the *Update Automatically* box. This causes the last input made by the user to be saved and displayed during the next run of the macro. Otherwise, the default value “3” will always be used.
7) Close the dialog box by clicking on *OK*. This inserts a new line in the variable window:

```
*NUMERIC <Number of Samples> = 3;
```

   The asterisk indicates, that the variable will be updated automatically.
8) Now you have to replace the Loop Counter with the variable you just defined. Double-click on the StartLoop command line. Open the *Special Macro Commands* dialog box. Change the value of the *Loop Count* field to `<Number of Samples>` by selecting this variable from the drop-down list. Close the dialog by clicking on *OK*. The code changes to:

```
StartLoop (<Number of Samples>, 0);
```

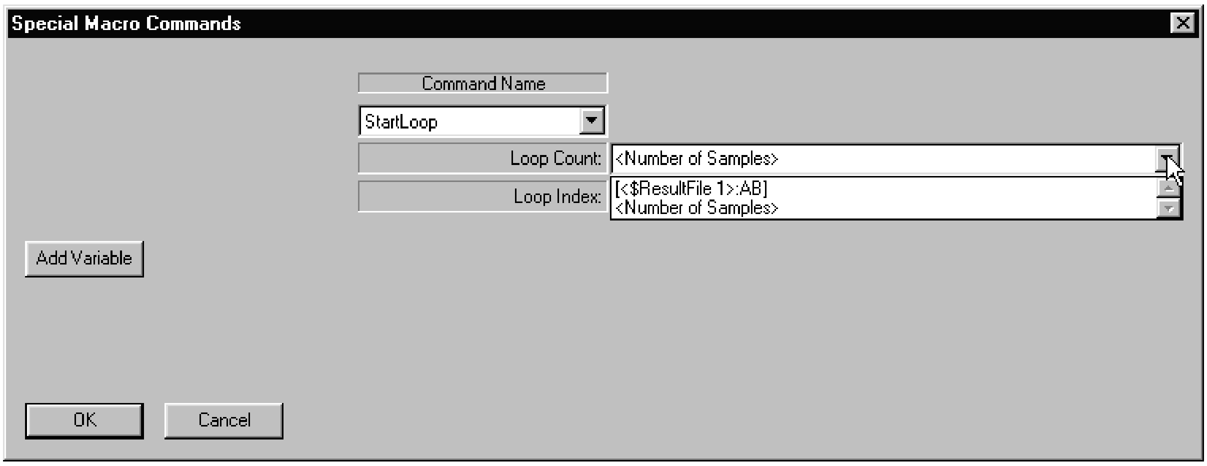


Figure 32: Defining StartLoop Parameters

Alternatively to changing the code via the *Special Macro Commands* dialog, you could have manually edited the code after double-clicking on the StartLoop line. Change the old values in parentheses after the StartLoop (3, 0) against the variable name.

9) Next, you have to create a user dialog box. Insert a blank line in the command window, and open the *Special Macro Commands dialog.* Open the drop-down list of the *Command Name* field, and quick-select the UserDialog command by typing “u”.

10) A new dialog box (shown in Figure 11) opens. Enter “Multiple Sample Acquisition” in the *Title* field. Leave the *Option* set to “STANDARD”.

11) Use the remaining fields to define the text of the user dialog box. A drop-down list is provided for all fields. In the first field on the left side, select the format that will be used to display the text in the user dialog. Since we want the user to edit the value of the Loop Counter variable, we select “EDIT”.

12) The drop-down list of the next field in the same row shows all variables that are suited for the format chosen before. In our example, only <Number of Samples> is shown. Select this variable and exit the dialog box by clicking on *OK*. This generates a new command line in the macro.

13) Reposition the UserDialog command line to the beginning of the PROGRAM SECTION. Store the macro as “Measure 5” and close the Macro Editor.

Listing (MEASURE 5.MTX)

```
VARIABLES SECTION
FILE <$ResultFile 1> = AB, AB/Peak;
*NUMERIC <Number of Samples> = 1.000000000000000;

PROGRAM SECTION
UserDialog (Multiple Sample Acquisition, STANDARD,
EDIT:<Number of Samples>, BLANK, BLANK,
BLANK, BLANK, BLANK, BLANK, BLANK, BLANK, BLANK,
BLANK, BLANK, BLANK, BLANK);
```

```
MeasureReference (0, {EXP='Default.xpm',
XPP='C:\OPUS_NT\XPM'});
StartLoop (<Number of Samples>, 0);
Message ('Please insert sample.', ON_SCREEN,
NO_TIMEOUT);
<$ResultFile 1> = MeasureSample (0,
{EXP='Default.xpm', XPP='C:\OPUS_NT\XPM'});
Baseline ([<$ResultFile 1>:AB], {BME=2, BCO=0,
BPO=64});
PeakPick ([<$ResultFile 1>:AB], {NSP=9, PSM=1, WHR=0,
LXP=400.000000, FXP=4000.000000,
QP8='NO', QP9=0.200000, PTR=20.000000, QP4='NO',
QP7=0.800000,QP6='NO',
QP5=80.000000, PPM=1, QP0='NO', QP3=4});
EndLoop (0);

PARAMETER SECTION
```

Running the Macro

When you start the macro, a user dialog with the title “Multiple Sample Acquisition”, and a field for numerical input will be displayed. The input field is labelled *Number of Samples,* i.e. the variables’ name. Change its value from “3” to “2” and click on *Continue*. Subsequently, two sample measurements will be performed. When you start the macro again, the default value of the input field now is “2”.

6.7 Load 1 – Loading and Processing a Spectrum

Task

So far data processing was only performed on spectra acquired with the same macro. The following examples show, how to load spectra from disk. The first example loads a spectrum and performs a baseline correction, followed by a normalization.

Macro Functions

In OPUS-OS/2, only a special macro command was available for loading spectra. This is also supported by OPUS-NT for reasons of compatibility. In addtiion, the *Load File* command of OPUS-NT can be used in a macro. In the following we will use both commands.

OPUS Functions

Load, Baseline Correction, Normalization

In this example the OPUS function *Load File* will be used.

Generating the Macro

1) Open the *Macro Editor* and select the *Load File* command from the OPUS *File* pull-down menu.
2) Change to the \OPUS_NT\DATA directory, and select the file ABBOE05.0. Click on *Open*.
3) Close the parameter dialog box without any further changes.
4) The following command line is generated:

```
<$ResultFile 1> = Load (0, {DAP='C:\OPUS_NT\DATA',
DAF='Abboe05.0', INP='D:\OPUS\DEBUG\METHODS',
IFP='C:\OPUS_NT\METHODS',INM='DEFAULT.TXD',
IFN='DEFAULT'});
```

   Similar to the *Measurement* command, a new FILE variable is generated for the file that was loaded. However, the data block type (Spec) is different from the blocks of acquired spectra, and only used in combination with macros. This block type allows to write macros, that can handle any type of spectra.

```
FILE <$ResultFile 1> = Spec;
```

   Similar to the first example, select the *Baseline Correction* command and then *Normalization* from the OPUS *Manipulate* menu. Use in both cases the `[<$ResultFile 1>:Spec]` variable.

5) Save the macro as “Load 1”.

Listing (LOAD 1.MTX)

```
VARIABLES SECTION
FILE <$ResultFile 1> = Spec;

PROGRAM SECTION
<$ResultFile 1> = Load (0, {DAP='C:\OPUS_NT\DATA',
DAF='Abboe05.0',
INP='D:\OPUS\DEBUG\METHODS',
IFP='D:\OPUS\Release\METHODS',
INM='DEFAULT.TXD', IFN='DEFAULT'});
Baseline ([<$ResultFile 1>:Spec], {BME=2, BCO=0,
BPO=64});
Normalize ([<$ResultFile 1>:Spec], {NME=1,
NFX=4000.000000, NLX=400.000000, NWR=1});

PARAMETER SECTION
```

Running the Macro

The macro loads the file ABBOE05.0 and subtracts a baseline. Afterwards, the spectrum is normalized. Note that if you start the macro for a second time, the file ABBOE05.0 is loaded and processed again.

6.8 Load 2 – Loading and Processing Several Spectra

Task

In the previous example only the file ABBOE05.0 was processed. To be able to process several files the file name and its path must be stored in variables. We define \OPUS_NT\DATA as the default directory.

Macro Functions

GetOpusPath, UserDialog

Two STRING variables are used to store the path and file name of the spectrum. A user dialog will be used to allow the user to enter name and path. The GetOpusPath function is used to determine the path of the OPUS main directory, which is the extended by the string “\DATA”.

OPUS Functions

Load, Baseline Correction, Normalization

We replace the file name and path statements of the Load function against variables.

Generating the Macro

1) Open the *Macro Editor* and load “Load 1”.
2) Append a blank line to the variable window, and open the *New/Edit Variable* dialog box.
3) Since STRING is the default type, just enter “Path” as name in the *Name* field. The *Value* field remains empty. Exit the dialog box.
4) Generate a second variable labelled “File Name”.
5) Insert a blank line to become the first line in the command window, and open the *Special Macro Commands* dialog box. Select the function GetOpusPath from the *Command* list. This function does not require any parameters. The result will be returned to the STRING variable. Choose `<Path>` from the *Variable* list. Close the dialog box.
6) Insert a blank line below the first line, and enter `<Directory> = '<Path>\DATA'`. Don’t forget the backslash and remember to enclose

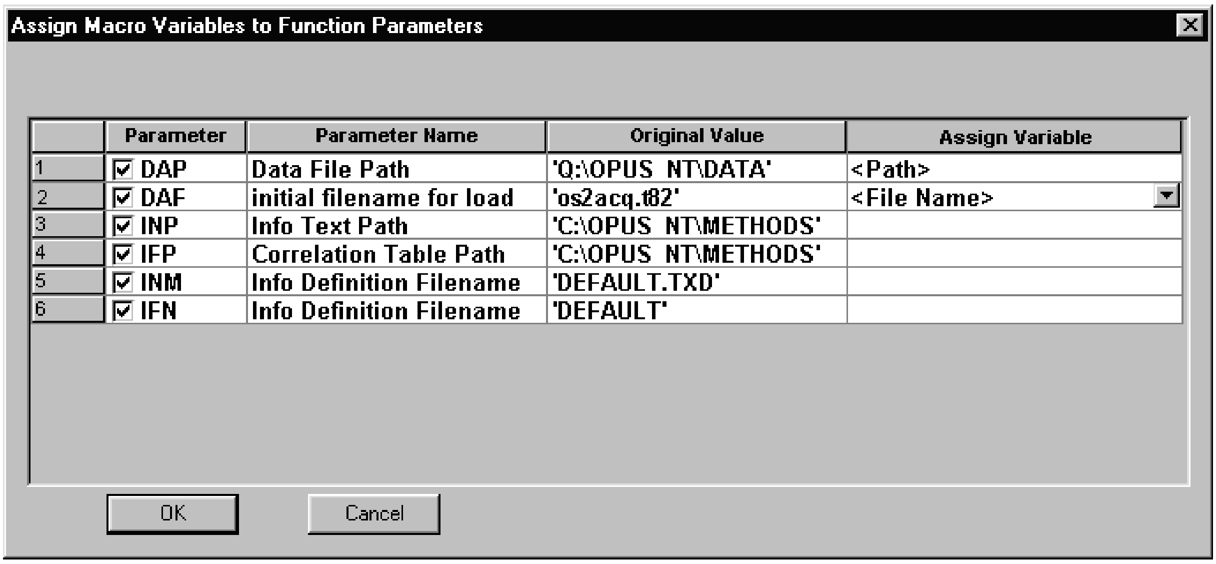
the expression by single quotes.

7) Now, we will add a user dialog box below line 2. Repeat steps 9 to 12 from the example “Measure 5”. This time you will need two lines of the format EDIT, to which you assign the variables `<File Name>` and `<Path>`. As a result the following line should be appended to the macro:

```
UserDialog (Load, STANDARD, EDIT:<Path>,
EDIT:<File Name>, BLANK, BLANK, BLANK, BLANK,
BLANK, BLANK, BLANK, BLANK, BLANK, BLANK, BLANK,
BLANK);
```

8) Replace the file name and directory statements in the Load command line by the variables. Double-click on this line and then on the button at the end of the line. This will open the OPUS *Load File* dialog.

9) Load any file from the directory. The parameter dialog box will be displayed again, listing the path and file name as the first two entries. Click on the topmost field in the *Assign Variable* column. A drop-down list replaces the entry field; choose the variable `<Path>` from this list.


Assign Macro Variables to Function Parameters

	Parameter	Parameter Name	Original Value	Assign Variable
1	☑ DAP	Data File Path	'Q:\OPUS_NT\DATA'	<Path>
2	☑ DAF	initial filename for load	'os2acq.t82'	<File Name>
3	☑ INP	Info Text Path	'C:\OPUS_NT\METHODS'	
4	☑ IFP	Correlation Table Path	'C:\OPUS_NT\METHODS'	
5	☑ INM	Info Definition Filename	'DEFAULT.TXD'	
6	☑ IFN	Info Definition Filename	'DEFAULT'	

OK Cancel

Figure 33: Assigning Variable Values

10) Click on the cell below and select `<File Name>` as described above. Closing the dialog box yields the following program line:

```
<$ResultFile 1> = Load (0, {DAP='<Path>',
DAF='<File Name>', INP='D:\OPUS\DEBUG\METHODS',
IFP='C:\OPUS_NT\\METHODS', INM='DEFAULT.TXD',
IFN='DEFAULT'});
```

11) Store the macro as “Load 2”.

Listing (LOAD 2.MTX)

```
VARIABLES SECTION
FILE <$ResultFile 1> = Spec;
STRING <Path> = 'C:\OPUS_NT\DATA';
STRING <File Name> = '';
```

```
PROGRAM SECTION
UserDialog (Load, STANDARD, EDIT:<Path>, EDIT:<File
Name>, BLANK, BLANK, BLANK,
BLANK, BLANK, BLANK, BLANK, BLANK, BLANK, BLANK,
BLANK, BLANK);
<$ResultFile 1> = Load (0, {DAP='<Path>', DAF='<File
Name>',
INP='D:\OPUS\DEBUG\METHODS',
IFP='D:\OPUS\Release\METHODS',
INM='DEFAULT.TXD', IFN='DEFAULT'});
Baseline ([<$ResultFile 1>:Spec], {BME=2, BCO=0,
BPO=64});
Normalize ([<$ResultFile 1>:Spec], {NME=1,
NFX=4000.000000, NLX=400.000000, NWR=1});

PARAMETER SECTION
```

Running the Macro

A dialog will be displayed, in which you have to enter the values for both variables. In the field "Directory", the path to your directory is displayed. If you enter "GLY.0" as file name you will see, that the macro also handles transmission spectra.

6.9 Load 3 – Multiple File Processing

Task

The previous example will be expanded to be able to load and process multiple files.

Macro Functions

GetOpusPath, LoadFile, UserDialog, StartLoop, EndLoop

Instead of the OPUS *Load File* command, we will use the equivalent macro function LoadFile. This function is able to load one or several files.

OPUS Functions

Baseline Correction, Normalization

This example introduces no new OPUS functions.

Generating the Macro

1) Open the *Macro Editor* and define three STRING variables `<Path>`, `<File Name>` and `<File>`. Assign the value "C:\OPUS_NT\DATA" to `<Path>` and "ABB*.0" to `<File Name>`.

2) Append a user dialog box for <Path> and <File Name>.
3) In contrary to the OPUS *Load File* command, the LoadFile macro function uses a parameter combining path and file name. Therefore, we will use the third variable <File> to combine the values of the remaining two variables. Insert a blank line into the command window, and open the *Special Macro Commands* dialog.
4) Select the Enter Expression function from the *Command* List. This functions allows you to enter a variable assignment; in our example, we want to assign a new variable value using two STRING variables.
5) Choose <File> from the *Variable* list. This is the variable that will contain the result.
6) In the *Expression* field, select <Directory>.
7) Position the cursor in the Expression field after <Directory>. Add "\<File Name>" and enclose the line in single quotes.

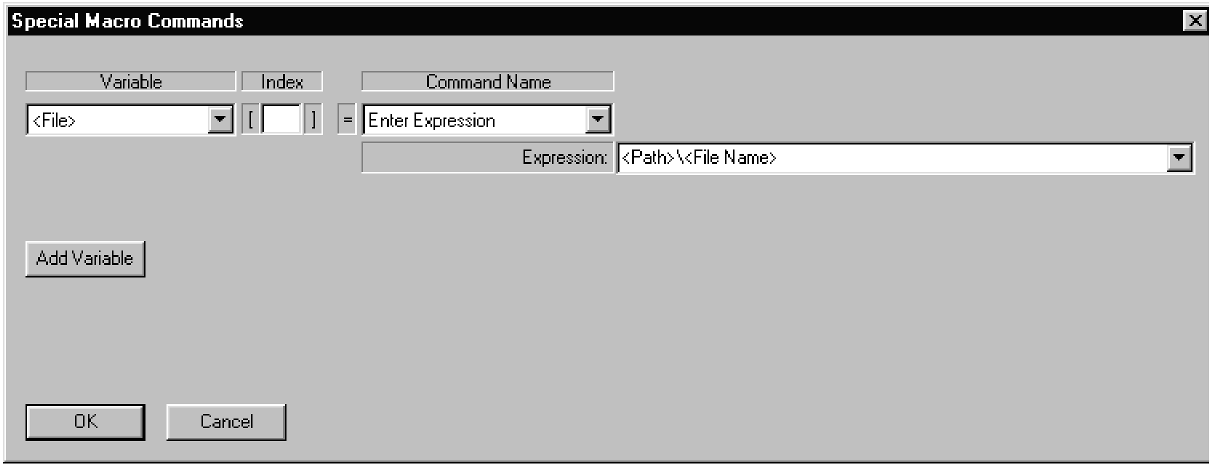


Figure 34: Assigning Variable Values Using Enter Expression

Exit the dialog box. The following line will be appended to the macro:

```
<File> = '<Path>\<File Name>';
```

8) Insert a blank line into the command window and open the *Special Macro Commands* dialog. Select LoadFile.
9) In case of the LoadFile macro command, you have to define a FILE variable for the data file intended to be loaded by yourself. Open the variable window by clicking on the *Add Variable* button.
10) Define a FILE variable "Result". From the first *Value* list, choose the spectrum block "AB". Don't change the remaining two lists. Copy the variable to the list by clicking *Add* and close the dialog box to get back to the *Special Macro Commands* dialog.

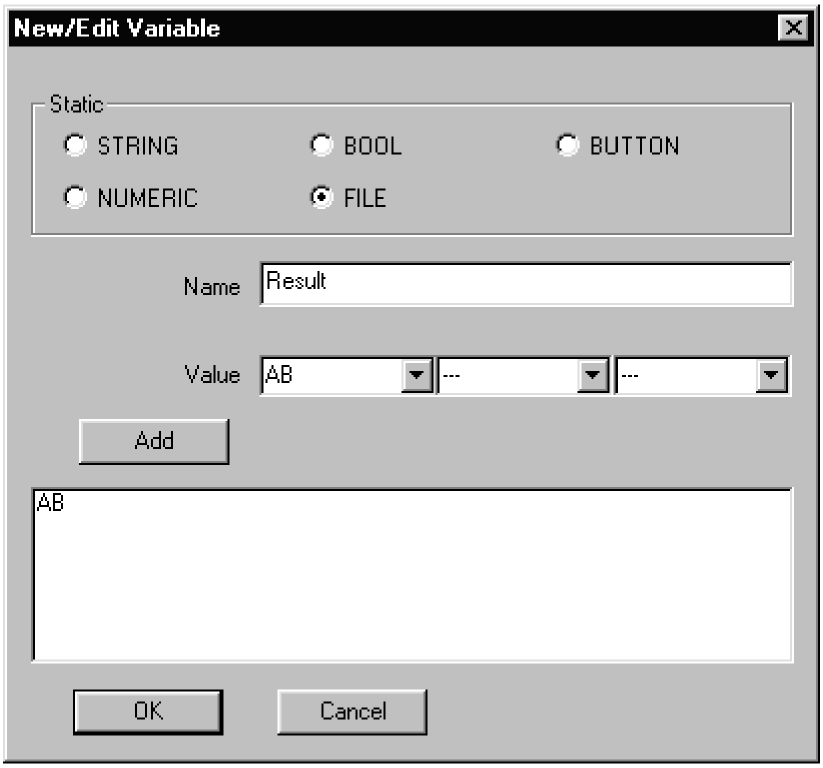


Figure 35: Defining a FILE Variable

11) From the *Variable* list, select the newly generated variable [<Result>:AB]. Choose <File> from the *File Name* field and “WARNING” from the *Option* list. This will add the following line to the macro:

```
[<Result>:AB] = LoadFile (<File>, WARNING);
```

You see that a File expression (including the data block type) can be used as result file in a command line as well.

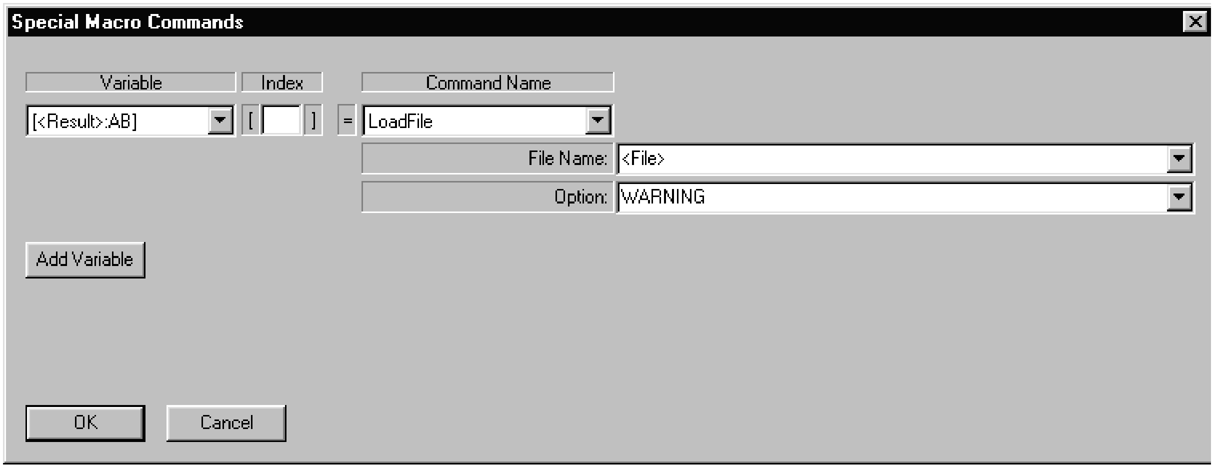


Figure 36: Defining [<Result>:AB]

12) By using wildcards (“*” or “?”) for the file name, we are able to load more than one file that match the preselection. Hence, ABB*.0 will load ABBOE05.0, ABBOE08.0 and ABBOE12.0. But we can state only one variable for all files. In this case, the variable will automatically be

expanded to an array variable, holding more that one value (in our example files). Each value can be addressed using an index number (in square brackets). The first index number is [0]. <xyz>[3] therefore addresses the fourth value of the variable <xyz>.

13) There is an elegant and simple way to address the values of a FILE variable array: use the StartLoop command with the variable name as Loop Counter. Open the *Special Macro Commands* dialog box and select StartLoop.
14) The `[<Result>:AB]` variable is also included in the *Loop Count* list. Select this entry and set the Loop Index to “0”.
15) Add the OPUS functions *Baseline Correction* and *Normalization*, using the `[<Result>:AB]` variable.
16) Finally, add the EndLoop command with a Loop Index “0” as the last line, and save the macro as “Load 3”.

Listing (LOAD 3.MTX)

```
VARIABLES SECTION
STRING <Path> = 'C:\OPUS_NT\DATA';
STRING <File Name> = 'ABB*.0';
STRING <File> = '';
FILE <Result> = AB;

PROGRAM SECTION
UserDialog (Load multiple files, STANDARD,
EDIT:<Path>, EDIT:<File Name>, BLANK, BLANK, BLANK,
BLANK, BLANK, BLANK, BLANK, BLANK, BLANK, BLANK,
BLANK, BLANK);
<File> = '<Path>\<File Name>';
[<Result>:AB] = LoadFile (<File>, WARNING);
StartLoop ([<Result>:AB], 0);
Baseline ([<Result>:AB], {BME=2, BCO=0, BPO=64});
Normalize ([<Result>:AB], {NME=1, NFX=4000.000000,
NLX=400.000000, NWR=1});
EndLoop (0);

PARAMETER SECTION
```

Running the Macro

This time we will use the Macro Debugger to test our macro.

1) From the OPUS *Macro* pull-down menu, select the *Debug Macro* command. Load the macro.
2) A dialog box containing the macros’ code will be displayed, the first line is indicated by a small green arrow. This arrow is a pointer to indicate the command that will be executed next.
3) Click on the *Single Step* button. The user dialog box will be displayed. Since you already defined valid start values you can continue by closing the box.

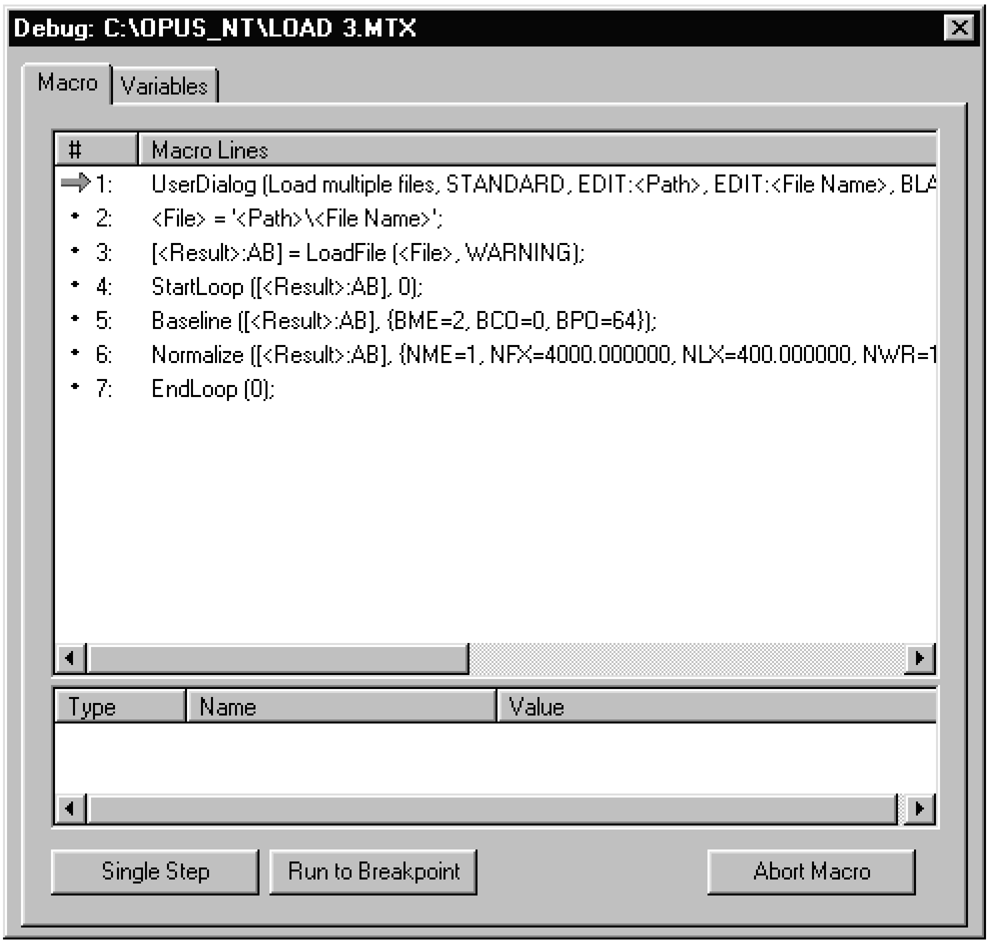


Figure 37: Debugging Load 3

4) Note that the pointer now is in line two. The window at the bottom shows which variables have been changed by the command executed before and their current values.

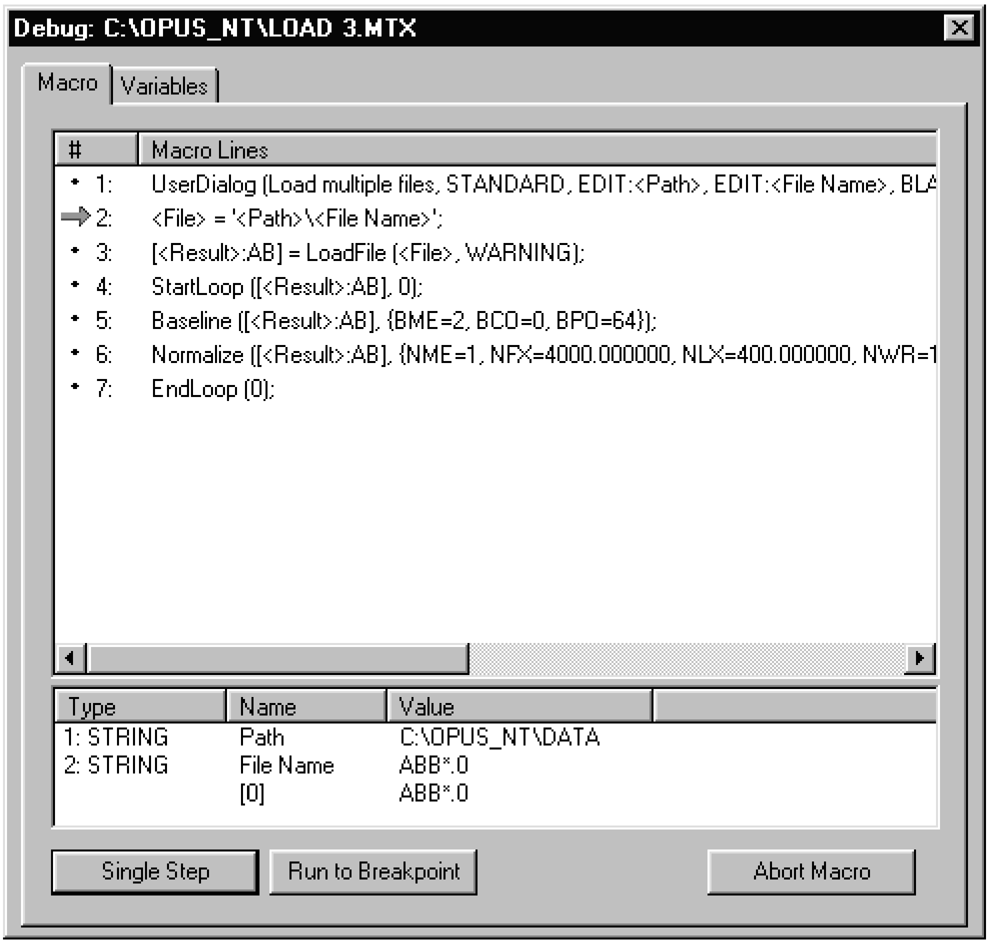


Figure 38: Debugging Load 3 – Single Step Mode

5) A complete list of the macros’ variables is given on the *Variables* page of the Debugger, including the variable type, the name and current value. As you can see, `<File>` and `<Result>` have no value assigned at this point.

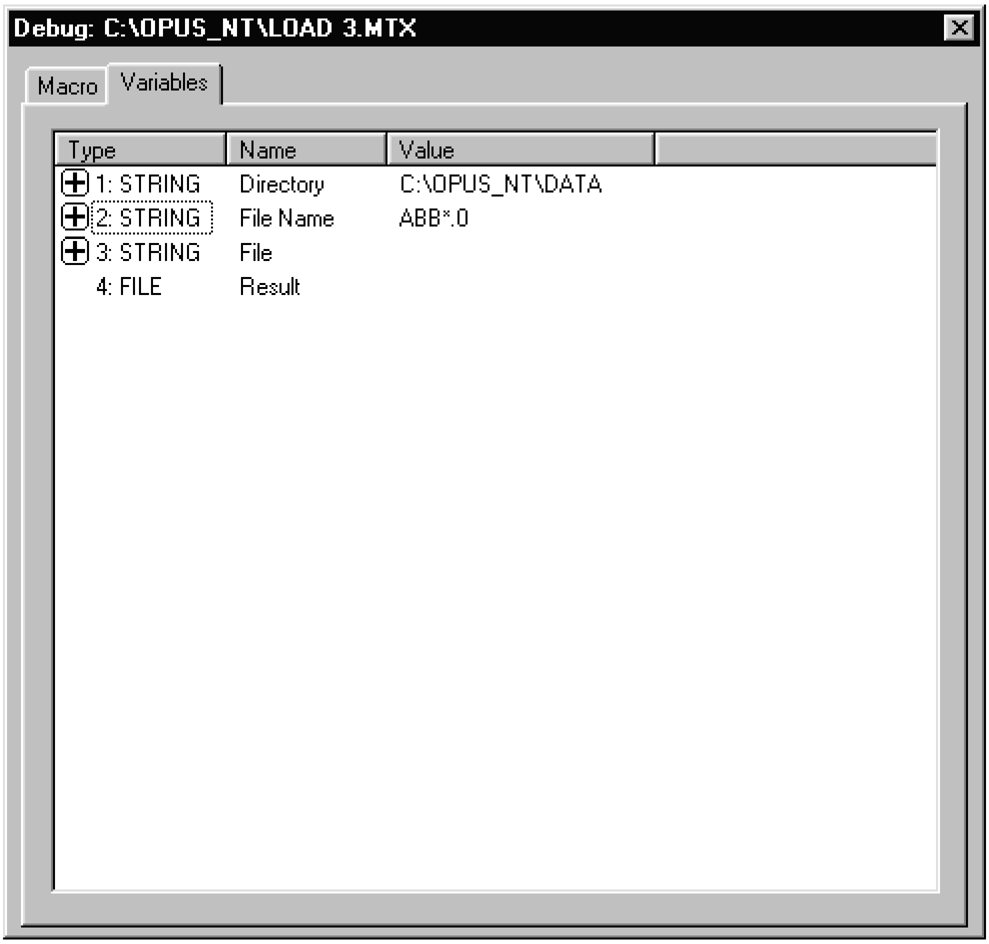


Figure 39: Debugging Load 3 – *Variables* Page

6) Return to the Macro page and execute the next command. The result is shown in the window at the bottom: <File> now contains the result of the combination of both STRING variables.
7) The next step will execute the LoadFile function. All three spectra will be loaded, the variable <Result> holds three different files. The first file is listed twice for reasons that will be explained later.
8) The next step, StartLoop does not seem to perform any action. However, the command initiates the triple repetition of the following commands. You can watch this when you continue to step through the macro.

6.10 Load 4 – Multiple File Processing

Task

This example shows you an alternative route to load and process several files. This time we will make use of the OPUS *Load File* command.

Macro Functions

GetArrayCount, UserDialog, StartLoop, EndLoop, ScanPath

ScanPath reads the content of a directory. You have to specify a directory and a file name using wildcards (e.g. C:\OPUS_NT\DATA\ABB*.0). All matching file names are stored as array in a STRING variable.

GetArrayCount evaluates the number of elements stored in an array variable, which then can be used as Loop Counter to address these elements.

OPUS Functions

Load, Baseline Correction, Normalization

This example introduces no new OPUS functions.

Generating the Macro

1) Open the *Macro Editor* and define the following STRING variables:

```
STRING <File List> = '';
STRING <Path> = 'C:\OPUS_NT\DATA';
STRING <Name> = 'ABB*.0';
NUMERIC <Count> = 0;
NUMERIC <Index> = 0;
```

2) Create a user dialog box for `<Path>` and `<Name>` to be able to test the macro under different prerequisites.
3) Add a blank line, call up the Special Macro Commands dialog box, and select the ScanPath function from the *Commands* list. This command requires a variable to store its result, choose the `<File List>` variable from the *Varible* list.
4) This function uses only the file name and path where to look for the requested file as a parameter. Select `<Path>` from the *Variable* list and add "`\<Name>`". Exit the dialog box.
5) Add a blank line and insert the GetArrayCount command; this command evaluates, how many entries are stored in `<File list>`. The number of entries will be stored in `<Count>`.

```
<Count> = GetArrayCount (<File list>);
```

6) `<Count>` will now be used to define the number of cycles of a loop. Insert a StartLoop command with <Count> as Loop Counter and a Loop Index of 0.
7) From the OPUS pull-down menu select the *Load File* command. Select any file regardless of the directory because in the following we will change the file name and path against variables. Click on *Load*. The dialog box for parameter assignment will open automatically. Assign `<Path>` to the first parameter "DAP", and `<File List>` to the second parameter (DAF). Since `<File List>` is an array variable you also have to specify an index number. In our example we use `[<Index>]` to read the complete file list. The line now reads

<File list>[<Index>].

8) In the next line, `<Index>` must increase by 1 to read the next list element during the next cycle of the loop. Add a blank line to the macro and enter:

```
<Index> = <Index> + 1;
```

9) Add the OPUS Baseline Correction function.
10) Add the EndLoop command.
11) Save the macro as “Load 4”.

Listing (LOAD 4.MTX)

```
VARIABLES SECTION
STRING <File List> = '';
STRING <Path> = 'C:\OPUS_NT\DATA';
STRING <Name> = 'ABB*.0';
NUMERIC <Count> = 0;
NUMERIC <Index> = 0;
FILE <$ResultFile 1> = Spec;
PROGRAM SECTION
UserDialog (Load multiple files, STANDARD,
EDIT:<Path>, EDIT:<Name>, BLANK,
BLANK, BLANK, BLANK, BLANK, BLANK, BLANK, BLANK,
BLANK, BLANK, BLANK,
BLANK);
<File List> = ScanPath (<Path>\<Name>);
<Count> = GetArrayCount (<File List>);
StartLoop (<Count>, 0);
<$ResultFile 1> = Load (0, {DAP='<Path>', DAF='
<File List>[<Index>]',
INP='D:\OPUS\DEBUG\METHODS',
IFP='D:\OPUS\Release\METHODS',
INM='DEFAULT.TXD', IFN='DEFAULT'});
<Index> = <Index>+1;
Baseline ([<$ResultFile 1>:Spec], {BME=2, BCO=0,
BPO=64});
EndLoop (0);

PARAMETER SECTION
```

Running the Macro

As in the last example, we use the Macro Debugger to execute the macro.

1) After the user dialog box ScanPath is executed. Note that `<File List>` now contains three file names.
2) As expected, the GetArrayCount function returns a value of 3 to `<Count>`.
3) In line 5, the first file of the list is loaded.
4) In line 6, the value of `<Index>` is increased by 1 (current value is 1).
5) In line 7, the loaded file is baseline corrected.

6) We won't execute the following two loop cycles step by step. Therefore, by double-clicking on the line number of line 7 (Baseline), a break point will be set, indicated by a small stop sign.
7) Now click on *Run to Breakpoint.* As you can see, the second spectrum will be loaded; during this operation the Debugger is grayed. Also, `<Index>` now has a value of 2.
8) Repeat the *Run to Breakpoint* cycle. The last file is loaded and `<Index>` now has a value of 3.
9) A third click on *Run to Breakpoint* performs the baseline correction and ends the macro.

The main difference to the last example is, that the macro command LoadFile loads all three spectra before processing them in the loop. In this example, the files will be loaded subsequently, while the macro goes through the loop. In the case of only three files, this may seem of minor importance. However, if you process a great number of files you will notice, that loading the files turns out to be quite time consuming. In this case, the latter method is the method of choice.

6.11 Manipulation 1 – Processing of Files Already Loaded

Task

So far all example macros either loaded or acquired the data prior to data processing. However, there is a multitude of applications in which you may want to process data that was already loaded. In this example, we will demonstrate a general route to this type of data processing.

Macro Functions

This example introduces no new macro functions.

OPUS Functions

Baseline Correction

This example introduces no new OPUS functions.

Generating the Macro

1) Define a FILE variable `<File>` with an absorption data block.
2) Select the *Baseline Correction* function from the OPUS pull-down menu and choose the `<File>` variable for processing.
3) Save the macro as “Manipulation 1”.

Listing (MANIPULATION 1.MTX)

```
VARIABLES SECTION
FILE <File> = AB;

PROGRAM SECTION
Baseline ([<File>:AB], {BME=2, BCO=0, BPO=64});

PARAMETER SECTION
```

Running the Macro

1) Load an absorption spectrum (e.g. ABBOE05.0) and start the macro.
2) A file selection box will open. Select the file you loaded before. Clicking on *Continue* will perform a baseline correction.

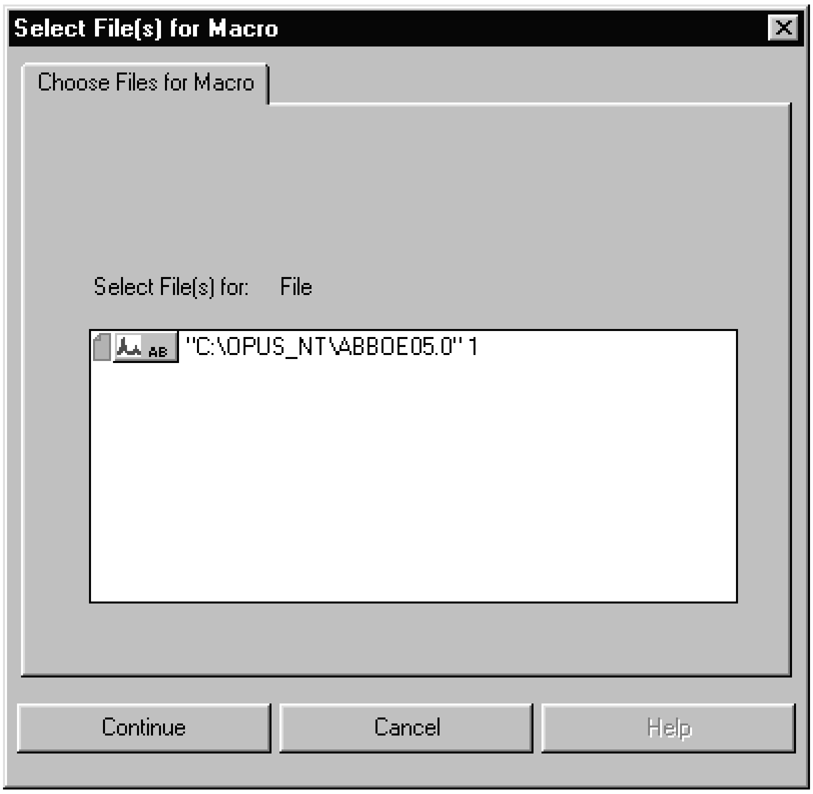


Figure 40: File Selection Box for Macros

6.12 Manipulation 2 – Processing of Files Already Loaded

Task

The previous macro will now be expanded by a user dialog box, from which you can select the spectrum you want to process.

Macro Functions

UserDialog

This example introduces no new macro functions.

OPUS Functions

Baseline Correction

This example introduces no new OPUS functions.

Generating the Macro

1) Load “Manipulation 1”.
2) Append a UserDialog command line to the macro, selecting FILE as the variable type and `[<File>:AB]` as the variable.
3) Move this line to the top of the macro and save it as “Manipulation 2”.

Listing (MANIPULATION 2.MTX)

```
VARIABLES SECTION
FILE <File> = AB;

PROGRAM SECTION
UserDialog (0, STANDARD, FILE:[<File>:AB], BLANK,
BLANK, BLANK, BLANK, BLANK, BLANK,
BLANK, BLANK, BLANK, BLANK, BLANK, BLANK, BLANK);
Baseline ([<File>:AB], {BME=2, BCO=0, BPO=64});

PARAMETER SECTION
```

Running the Macro

In contrary to “Manipulation 1”, a user dialog is now displayed instead of the file selection box.

6.13 Manipulation 2a – Saving Processed Files

Task

OPUS-NT manipulates copies instead of the original file during data processing. Therefore, the result of a data manipulation has to be stored explicitly. We will demonstrate in this example, how files can be saved using a macro. In addition, we will use the OPUS *Unload File* command, which is used to reduce the numbers of files loaded.

Macro Functions

UserDialog

This example introduces no new macro functions.

OPUS Functions

Baseline Correction, Save As, Save, Unload

We will make use of both *Save* and *Save As* commands to store the data. Afterwards the file will be unloaded.

Generating the Macro

1) Load “Manipulation 2”.
2) Select *Save File As* from the OPUS *File* pull-down menu. Use `[<File>:AB]` as the file to be saved and enter “Macrotest.0” as file name. With the help of the *Change Path* button, navigate to the “OPUS\WORK” subdirectory.
3) Ensure that on the Mode page the following options are selected:
   - *OPUS format*
   - *Save All*
   - *Remove All Copies*

   Close the dialog box by clicking on *Save*.
4) No changes are needed concerning the parameters; exit the parameter dialog box.
5) Select the *Save* function from the OPUS pull-down menu. This function replaces the original file, therefore you only have to select the file variable. Here you also don’t need to change any parameters.
6) Select *Unload File* from the OPUS *File* pull-down menu. Again, you only have to select the file variable.
7) Save the macro as “Manipulation 2a”.

Listing (MANIPULATION 2a.MTX)

```
VARIABLES SECTION
FILE <File> = AB;

PROGRAM SECTION
UserDialog (0, STANDARD, FILE:[<File>:AB], BLANK,
BLANK, BLANK, BLANK, BLANK, BLANK,
BLANK, BLANK, BLANK, BLANK, BLANK, BLANK, BLANK);
Baseline ([<File>:AB], {BME=2, BCO=0, BPO=64});
SaveAs ([<File>:AB], {DAP='C:\OPUS_NT\WORK\', OEX='1',
SAN='Macrotest.0', COF=18,
INP='D:\OPUS\DEBUG\METHODS',
IFP='D:\OPUS\Release\METHODS',
INM='DEFAULT.TXD', IFN='DEFAULT'});
```

```
Save ([<File>:AB], {DAP='C:\OPUS_NT\DATA', OEX='1',
SAN='calcul.0',
COF=146,
INP='D:\OPUS\DEBUG\METHODS',
IFP='D:\OPUS\Release\METHODS',
INM='DEFAULT.TXD', IFN='DEFAULT'});
Unload ([<File>:AB], {});

PARAMETER SECTION
```

Running the Macro

As in the last example, load a file first before starting the macro. Select this file in the user dialog and click on *Continue*. You get a glimpse of the baseline-corrected spectrum before the file is unloaded again. However, if you now open the file you will see that it has been baseline-corrected. Also, check the file “Macrotest.0” from the “WORK” directory; both files must be identical.

6.14 Manipulation 3 – Processing of Multiple Files Already Loaded

Task

If you try to process more than one spectrum at a time with the previous example by dropping several files in the file selection field, you will note that only the last spectrum of the file list will be processed. In the following macro we will show how to handle more than one file.

Macro Functions

UserDialog, StartLoop, EndLoop

We use a loop to repeatedly process an array of files. As we have seen in “Load 3” the StartLoop function is able to directly use an array as a Loop Counter.

OPUS Functions

Baseline Correction

This example introduces no new OPUS functions.

Generating the Macro

1) Load “Manipulation 2”.
2) Insert a StartLoop command using `<File>` as Loop Counter and a Loop Index of “0” just below the user dialog command.
3) Append an EndLoop command to the macro and save it as “Manipulate 3”.

Listing (MANIPULATION 3.MTX)

```
VARIABLES SECTION
FILE <File> = AB;

PROGRAM SECTION
UserDialog (0, STANDARD, FILE:[<File>:AB], BLANK,
BLANK, BLANK, BLANK, BLANK, BLANK,
BLANK, BLANK, BLANK, BLANK, BLANK, BLANK, BLANK);
StartLoop ([<File>:AB], 0);
Baseline ([<File>:AB], {BME=2, BCO=0, BPO=64});
EndLoop (0);

PARAMETER SECTION
```

Running the Macro

Load several absorption spectra and run the macro. Select all spectra in the user dialog and click on *Continue*. All selected spectra will be baseline-corrected.

6.15 Manipulation 4 – Multiple File Processing Using Variable Parameters

Task

We expand the last macro to perform a peak pick on the baseline-corrected spectrum. The frequency limits and the peak sensitivity should be kept variable.

Macro Functions

UserDialog, StartLoop, EndLoop

This example introduces no new macro functions.

OPUS Functions

Baseline Correction, Peak Picking

This example introduces no new OPUS functions.

Generating the Macro

1) Load “Manipulation 3”.
2) Add three numerical variables:
   - `<x-Start Frequency>` with a value of “1000”
   - `<x-End Frequency>` with a value of “500”
   - `<Sensitivity>` with a value of “10”

3) Select the “Baseline” command line and choose *Peak Picking* from the OPUS *Evaluate* pull-down menu. Select <File> and make sure that *Use File Limits* on the *Frequency Range* page is **not** selected.
4) After clicking on *Peak pick,* the parameter dialog opens. Assign the following variables:
   - <x-Start Frequency> to “FXP”
   - <x-End Frequency> to “LXP”
   - <Sensitivity> to “PTR”
5) Edit this user dialog box command in the first line. Add three Edit lines for the numerical parameters.
6) Save the macro as “Manipulation 4”.

Listing (MANIPULATION 4.MTX)

```
VARIABLES SECTION

FILE <File> = AB, AB/Peak;
NUMERIC <x-Start Frequency> = 1000;
NUMERIC <x-End Frequency> = 500;
NUMERIC <Sensitivity> = 10;

PROGRAM SECTION
UserDialog (0, STANDARD, FILE:[<File>:AB], EDIT:<x-
Start Frequency>, EDIT:<x-End Frequency>,
EDIT:<Sensitivity>, BLANK, BLANK, BLANK, BLANK, BLANK,
BLANK, BLANK, BLANK,
BLANK, BLANK);
StartLoop ([<File>:AB], 0);
Baseline ([<File>:AB], {BME=2, BCO=0, BPO=64});
PeakPick ([<File>:AB], {NSP=9, PSM=1, WHR=0, LXP=<x-
End Frequency>, FXP=<x-Start Frequency>,
QP8=’NO’, QP9=0.200000, PTR=<Sensitivity>, QP4=’NO’,
QP7=0.800000, QP6=’NO’,
QP5=80.000000, PPM=1, QP0=’NO’, QP3=4});
EndLoop (0);

PARAMETER SECTION
```

Running the Macro

The user dialog shows the file list as before. First slelect the file to be processed and then switch to the *Parameter* page. As you continue, you will see from the peak labels on thee display that the specified limits have been used.

6.16 Average 1 – Averaging Spectra

Task

While most OPUS functions can be integrated into macros without any problems some functions require special consideration. One of these functions is the *Averaging* function from the OPUS *Manipulation* pull-down menu which we will use in the following two examples.

Usually, an OPUS function processes only one file at a time which is the reason why these functions must be enclosed by a loop if several files should be processed. The average function uses several files at once to calculate an average spectrum.

Macro Functions

UserDialog, StartLoop, EndLoop

This example introduces no new macro functions.

OPUS Functions

Average

The average function uses several files at once to calculate an average spectrum, which is stored in a new file. To include this function in a macro, we make use of the fact that an existing average spectrum can be updated.

Generating the Macro

1) Open the Macro Editor and define two FILE variables `<First File>` and `<Next File>`, each of them having an absorption data block assigned.
2) Start by generating a user dialog box, in which you include these variables.
3) Now choose the *Averaging* function from the OPUS *Manipulation* pull-down menu. Assign `[<First File>:AB]` as the spectrum used for averaging and set the following parameters:
   - Don't select *Update Average Spectrum*
   - Select *Weighting with Number of Scans*
   - Don't select *Create/Update Standard Deviation Spectrum*
   - Don't select *Compute Average Report*

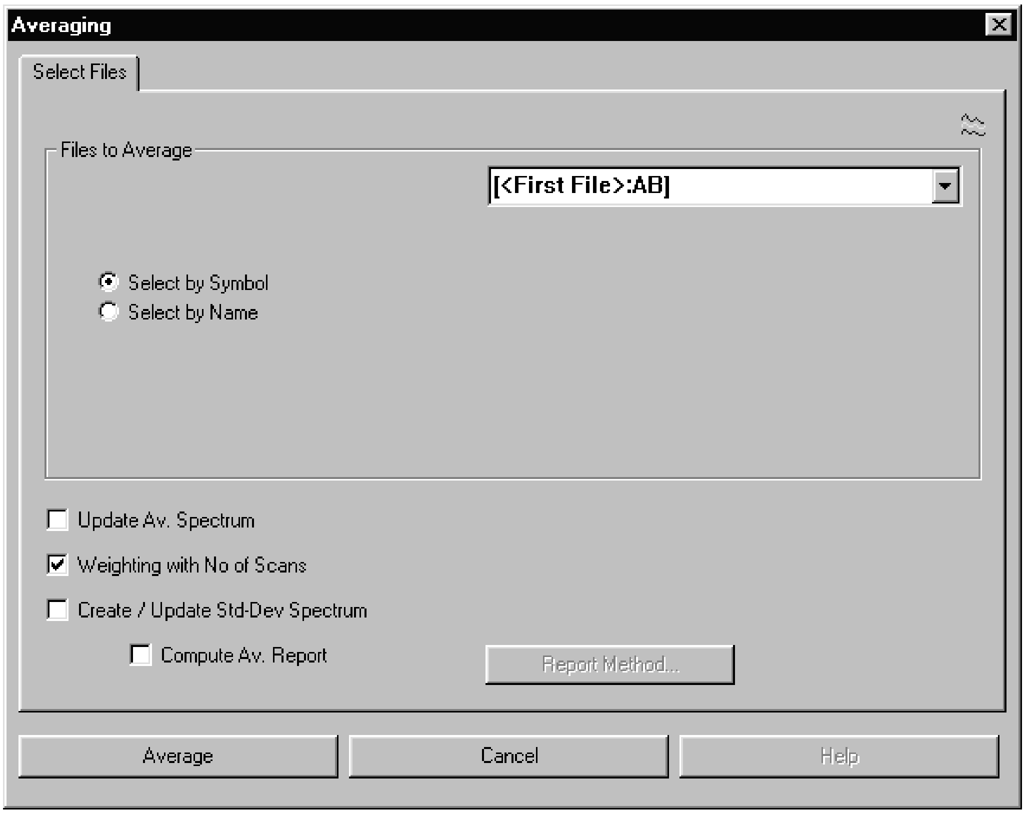


Figure 41: *Averaging* Dialog Box

4) These settings create a new file (the corresponding variable [<$Result File 1>:Spec] is automatically created) containing the average spectrum of the selected file, i.e. only a copy. Click on *Average* and confirm the parameter dialog box by clicking the *OK* button.
5) For the remaining files we need a loop with <Next Files>, acting as Loop Counter:
StartLoop ([<Next Files>:AB], 0);
6) Choose the *Averaging* once more and select this time <Next Files> as the spectrum to be averaged.
7) If you select the *Update Average Spectrum* parameter, you can pick the average spectrum created during the first run ([<$Result File 1>:Spec]) from the list. This time the result is not stored in a new file, but included in the average calculation during every cycle of the loop.

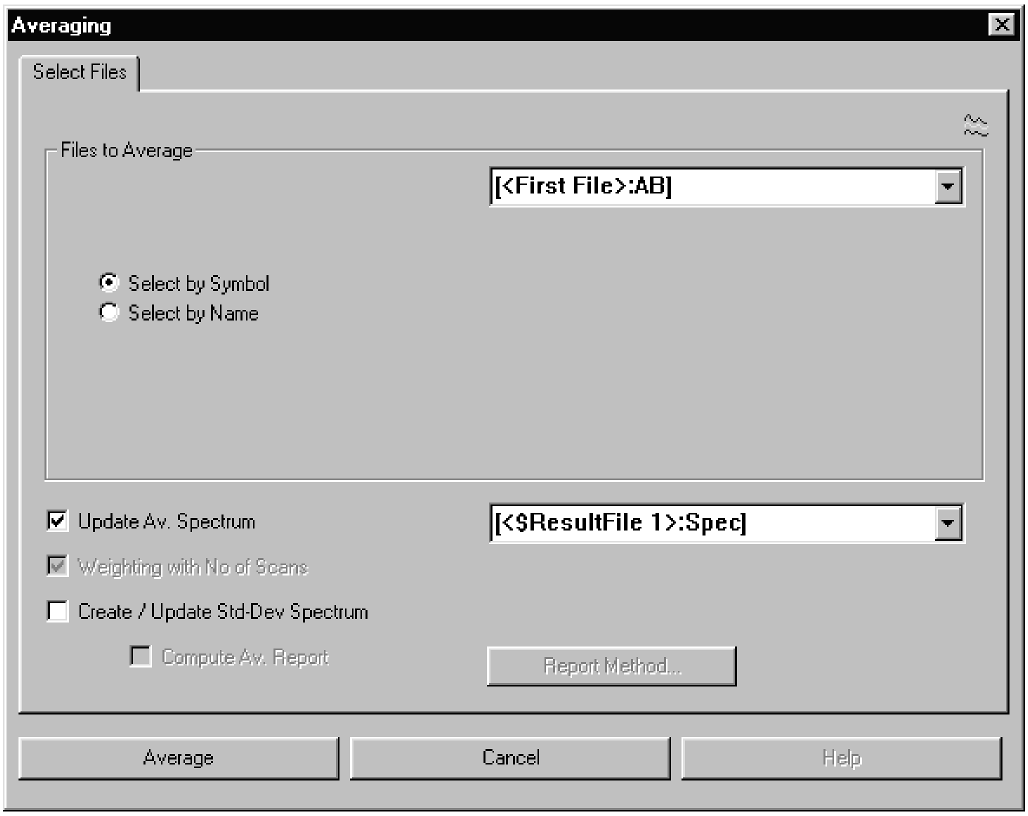


Figure 42: *Averaging* Dialog Box – Update Average Spectrum

8) Close the loop by appending the EndLoop command and save the macro as “Average 1”.

Listing (AVERAGE 1.MTX)

```
VARIABLES SECTION
FILE <First File> = AB;
FILE <Next Files> = AB;
FILE <$ResultFile 1> = Spec;

PROGRAM SECTION
UserDialog (Average, STANDARD, FILE:[<First File>:AB],
FILE:[<Next Files>:AB],
BLANK, BLANK, BLANK, BLANK, BLANK, BLANK, BLANK,
BLANK, BLANK, BLANK,
BLANK, BLANK);
<$ResultFile 1> = Average ([<First File>:AB], 0, 0,
{QA0=1, QA2=0, QAE=’NO’, QAF=’NO’,
QAL=’LIS’, QAM=’D:\OPUS\Debug\’, QAN=’*.*’, QAO=0,
QFB=’’, QFC=’’});
StartLoop ([<Next Files>:AB], 0);
Average ([<Next Files>:AB], [<$ResultFile 1>:Spec], 0,
{QA0=1, QA2=0, QAE=’NO’, QAF=’YES’,
QAL=’LIS’, QAM=’D:\OPUS\Debug\’, QAN=’*.*’, QAO=0,
QFB=’’, QFC=’’});
EndLoop (0);

PARAMETER SECTION
```

Running the Macro

Start by loading three absorption spectra. Open the Macro Debugger and run the macro.

1) After stepping through the first line, the user dialog is displayed, consisting of two input fields. Enter the first spectrum into the upper field (assigned to `<First File>`), and the remaining two spectra into the lower field. Click on *Continue*.
2) The result is a new spectrum identical to the spectrum selected first. Continue for two more steps.
3) After the next averaging operation the average spectrum is the average of the first and second spectrum.
4) Click on *Run to Breakpoint* to end the macro. Verify the result by manually averaging the three spectra, using the OPUS *Averaging* function. Compare the result to the file calculated by the macro. Both files should be identical.

6.17 Average 2 – Averaging Spectra Including the Standard Deviation

Task

In addition to the calculation of an average spectrum, it is possible to generate a standard deviation spectrum and store it in a separate file. Similar to the average spectrum, it is possible to also update the standard deviation spectrum.

Macro Functions

StartLoop, EndLoop

This example introduces no new macro functions.

OPUS Functions

Average

This example introduces no new OPUS functions.

Generating the Macro

1) Load “Average 1”.
2) Edit the second line (the Average command) and select the *Create/ Update Standard Deviation* check box. Exit the *Average* dialog box and the parameter dialog box. This only changes the parameter of the Average command in the macro.

3) However, as a consequence of the parameter change not only one but two files will be generated by the macro. The `[<$ResultFile 1>:Spec]` variable will contain to files, the average spectrum and the standard deviation spectrum.
4) Therefore, we also have to edit the Average command line within the loop. In the Average dialog box, you must add an index to the file selected for the average result: `[<$ResultFile 1>:Spec]`. In addition, check the *Update Standard Deviation* box, select `[<$ResultFile 1>:Spec]` as the result file and add an array index of “1” manually.
`[<$ResultFile 1>[1]:Spec]`
5) Close the functions’ dialog box and save the macro as “Average 2”.

Listing (AVERAGE 2.MTX)

```
VARIABLES SECTION
FILE <First File> = AB;
FILE <Next Files> = AB;
FILE <$ResultFile 1> = Spec;

PROGRAM SECTION
UserDialog (Average, STANDARD, FILE:[<First File>:AB],
FILE:[<Next Files>:AB],
BLANK, BLANK, BLANK, BLANK, BLANK, BLANK, BLANK,
BLANK, BLANK, BLANK,
BLANK, BLANK);
<$ResultFile 1> = Average ([<First File>:AB], 0, 0,
{QA0=1, QA2=0, QAE='YES', QAF='NO',
QAL='LIS', QAM='D:\OPUS\Debug\', QAN='*.*', QAO=0,
QFB='', QFC=''});
StartLoop ([<Next Files>:AB], 0);
Average ([<Next Files>:Spec], [<$ResultFile
1>[0]:Spec], [<$ResultFile 1>[1]:Spec], {QA0=1,
QA2=0, QAE='YES', QAF='YES', QAL='LIS',
QAM='D:\OPUS\Debug\', QAN='*.*', QAO=0,
QFB='', QFC=''});
EndLoop (0);

PARAMETER SECTION
```

Running the Macro

Start the macro and choose the same files as for the last example. In addition to the average spectrum, a standard deviation spectrum is calculated.

6.18 Parameter 1 – Reading Out Spectrum Parameters

Task

Accessing parameters and data of a spectrum file is a common task. In the following we will use macros to read out data from a spectrum file. In our first example we read the sample name information from a file, add a charge number and write the result to the file.

Macro Functions

GetParameter, UserDialog,

To read information from an OPUS file we use the GetParameter command, which returns the parameter in a STRING variable. We use EnterExpression to merge two text variables and plain text.

OPUS Functions

Edit Parameter

Only a few parameters of an OPUS file can be edited, due to security reasons. For editing the parameters, the *Edit Parameter* function from the OPUS *Edit* pull-down menu is available. However, this functions always saves the complete parameter set from the OPUS file. Therefore, we also have to read the complete parameter set, regardless of the number of parameters we want to edit.

Generating the Macro

1) Define a FILE variable named `<File>` and assign it an absorption block. Define two text variables `<Sample Name>` and `<Charge Number>`; you don't need to assign any values to the variables.
2) Create a user dialog box with the variables `<File>` (type FILE) and `<ChargeCharge Number>` (type EDIT).
3) Open the *Special Macro Commands* dialog box and select the GetParameter command from the *Commands* list. Select `<Sample Name>` from the Variable list and enter "0" as index. We will use this variable also for the remaining OPUS parameters. Select the variable `<File>` from the *File* list for the first argument. "SNM" addresses the sample name; choose it from the parameter list of the second argument. After closing the dialog box the following line will be appended to the macro:

```
<Sample Name>[0] = GetParameter ([<File>:AB], SNM);
```

4) We have to write similar statements for the remaining parameters sample form ("SFM"), operator name ("CNM") and sample number ("RSN"). Pay special attention to define the correct index numbers.

```
<Sample Name>[1] = GetParameter ([<File>:AB], SFM);
```

```
<Sample Name>[2] = GetParameter ([<File>:AB], CNM);
<Sample Name>[3] = GetParameter ([<File>:AB], RSN);
```

5) In the next line we change the sample name:

```
<Sample Name>[0] = '<Sample Name>[0] Charge:
<Charge Number>';
```

Either append this line manually or use the *Special Macro Commands* dialog and EnterExpression. Make sure to include the correct index number. As you can see you can combine variables and text in this type of statement.

6) Select the *Edit Parameter* command from the OPUS *Edit* pull-down menu. Choose `[<File>:AB]` from the *Select File* field and leave the remaining fields blank. Click on *Change* and assign the variables in the parameter dialog that appears next.

7) Save the macro as “Parameter 1”.

Listing (PARAMETER 1.MTX)

```
VARIABLES SECTION
STRING <Sample Name> = '';
STRING <Charge Number> = '';
FILE <File> = AB;

PROGRAM SECTION
UserDialog (0, STANDARD, FILE:[<File>:AB],
EDIT:<Charge Number>, BLANK, BLANK, BLANK,
BLANK, BLANK, BLANK, BLANK, BLANK, BLANK, BLANK,
BLANK, BLANK);
<Sample Name>[0] = GetParameter ([<File>:AB], SNM);
<Sample Name>[1] = GetParameter ([<File>:AB], SFM);
<Sample Name>[2] = GetParameter ([<File>:AB], CNM);
<Sample Name>[3] = GetParameter ([<File>:AB], RSN);
<Sample Name>[0] = '<Sample Name>[0]
Charge: <Charge Number>';
ParameterEditor ([<File>:AB], {CNM='<Sample Name>[2]',
SNM='<Sample Name>[0]',
SFM='<Sample Name>[1]', RSN=<Sample Name>[3], XTX='',
YTX='', ZTX='',
XAF=1.000000, YAF=1.000000, ZAF=1.000000});

PARAMETER SECTION
```

Running the Macro

In the OPUS Browser load a file and check its sample parameters by placing the cursor on the sample parameters. A small frame appears, listing the values of the data block. Start the macro, select the file you loaded and enter any text you like as charge number. Click on *Continue* and compare the parameter with its old value.

6.19 Parameter 2 – Generating Info Blocks

Task

If you ever created you own library you know that the files you want to include must have an information block. In general you probably stated the sample name and the preparation method during the sample measurement. We can expand the last macro and have it create an info block in addition.

Macro Functions

GetParameter, UserDialog,

This example introduces no new macro functions.

OPUS Functions

Edit Parameter, Information Input

We will use the *Information Input* function from the OPUS *Edit* pull-down menu to append an information block to a spectrum (assuming that the spectrum does not already include one).

Generating the Macro

1) Load “Parameter 1”.
2) Select the *Information Input* function from the OPUS *Edit* pull-down menu.
3) Select `[<File>:AB]`. The “DEFAULT.TXT” mask should be loaded by now. If this is not the case click on *Load Text Mask* and load “DEFAULT.TXT” from the “OPUS\METHODS” directory.
4) Enter the variables in the text fields:
   - in the *Compound Name* field: `<Sample Name>[0]`
   - in the *Sample Preparation* field: `<Sample Name>[1]`
   - in the *Charge Number* field: `<Charge Number>`

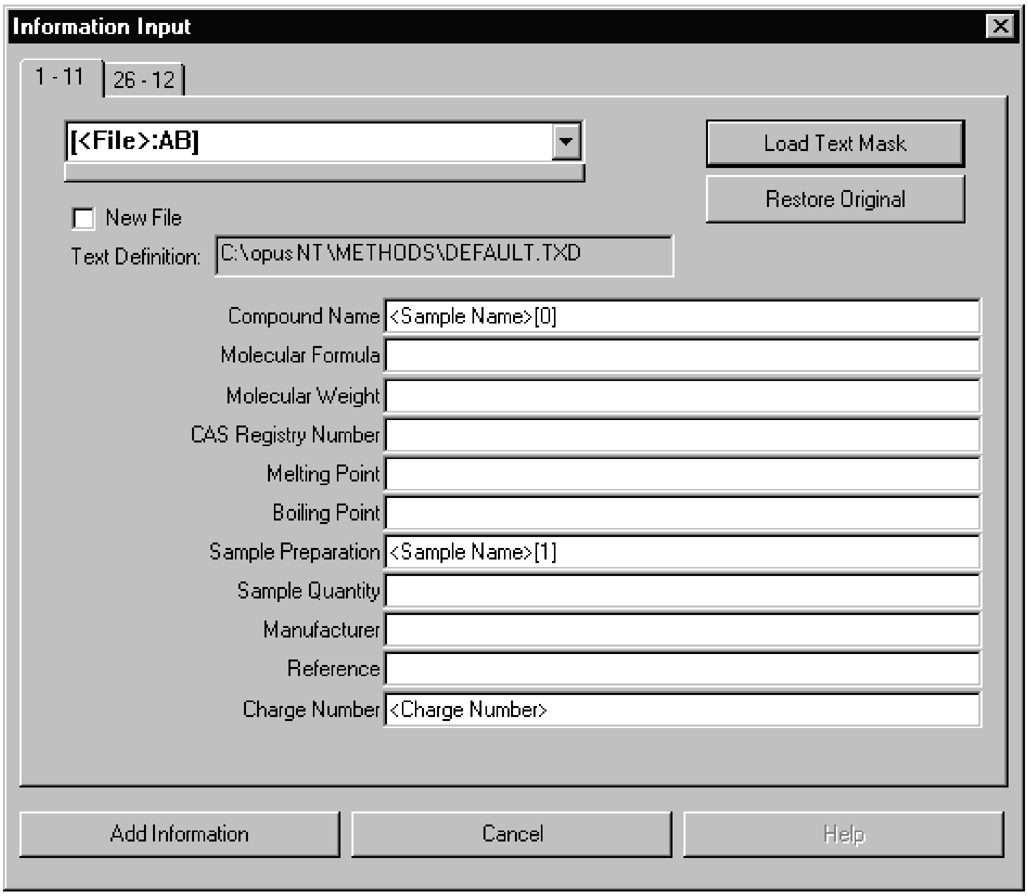


Figure 43: *Information Input* Dialog Box

5) If you prefer to enter these parameters in the parameter dialog box, you must enter a random character in the fields of the ***Information Input*** **dialog** you want to access. Click on *Add Information* to switch to the parameter dialog box and edit the entries.

6) If necessary, enter the variables in the fourth column. Keep in mind, that `<Sample Name>` is an array variable for which you have to specify an index value.

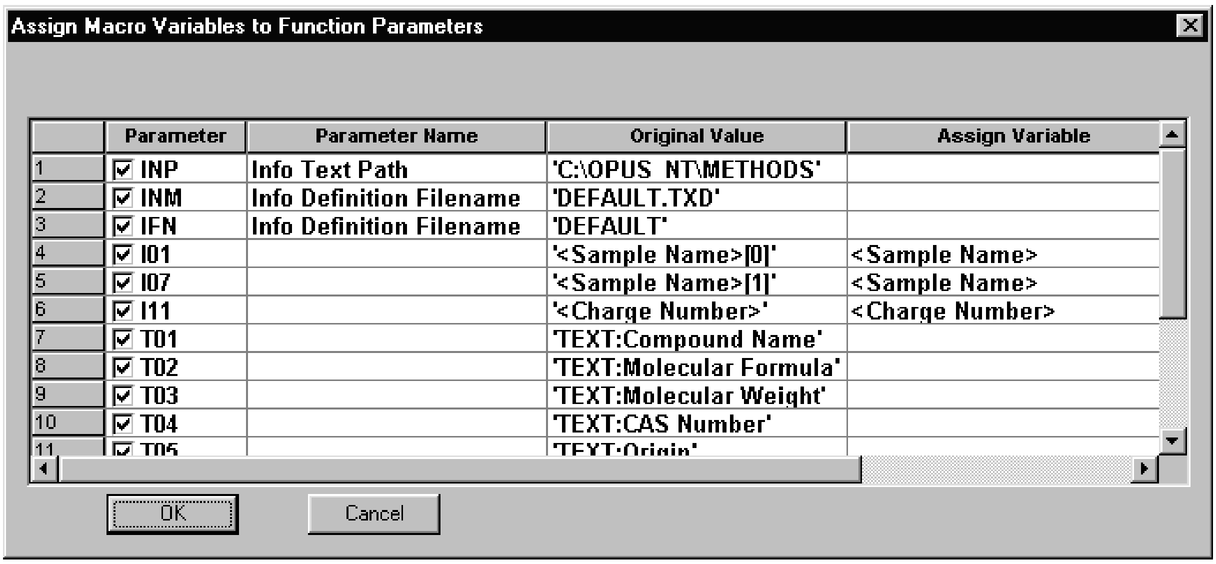


Figure 44: Parameters of the InfoInput command

7) Save the macro as “Parameter 2”.

Listing (PARAMETER 2.MTX)

```
VARIABLES SECTION
STRING <Sample Name> = ’’;
STRING <Charge Number> = ’’;
FILE <File> = AB, Info;

PROGRAM SECTION
UserDialog (0, STANDARD, FILE:[<File>:AB],
EDIT:<Charge Number>, BLANK, BLANK, BLANK,
BLANK, BLANK, BLANK, BLANK, BLANK, BLANK, BLANK,
BLANK, BLANK);
<Sample Name>[0] = GetParameter ([<File>:AB], SNM);
<Sample Name>[1] = GetParameter ([<File>:AB], SFM);
<Sample Name>[2] = GetParameter ([<File>:AB], CNM);
<Sample Name>[3] = GetParameter ([<File>:AB], RSN);
<Sample Name>[0] = ’<Sample Name>[0] Charge: <Charge
Number>’;
ParameterEditor ([<File>:AB], {CNM=’<Sample Name>[2]’,
SNM=’<Sample Name>[0]’,
SFM=’<Sample Name>[1]’, RSN=<Sample Name>[3], XTX=’’,
YTX=’’, ZTX=’’,
XAF=1.000000, YAF=1.000000, ZAF=1.000000});
InfoInput ([<File>:AB], {INP=’D:\OPUS\Debug\Methods’,
INM=’DEFAULT.TXD’, IFN=’DEFAULT’,
I01=’<Sample Name>[0]’, I07=’<Sample Name>[1]’,
I11=’<Charge Number>’,
T01=’TEXT:Compound Name’, T02=’TEXT:Molecular For-
mula’, T03=’TEXT:Molecular Weight’,
T04=’TEXT:CAS Number’, T05=’TEXT:Origin’,
T06=’TEXT:Boiling Point’,
T07=’TEXT:Sample Technique’, T08=’TEXT:Weight’,
T09=’TEXT:Manufacturer’,
T10=’TEXT:Reference’, T11=’TEXT:Charge Number’,
T12=’TEXT:Comment’});

PARAMETER SECTION
```

Running the Macro

The macro generates a new information block. Open a report window and have the information block displayed.

6.20 Parameter 3 – Replacing Info Block Entries

Task

The last macro always generated a new information block or replaced an existing information block. Now we will only change one or a few entries in an existing block, in our example the compound name.

Macro Functions

This example introduces no new macro functions.

OPUS Functions

Information Input

This example introduces no new OPUS functions.

Generating the Macro

1) Define a FILE variable `<File>` with an AB data block associated and a text variable `<Sample Name>` with “New Sample Name” as initial value.
2) Open the *Information Input* dialog box and select `<File>`. Load the “DEFAULT.TXD” mask if necessary and enter `<Sample Name>`. Exit this dialog box as well as the parameter dialog box without any further changes.
3) Add “IRM = ’R’” in the parameter section of the InfoInput command line. The parameter section is the part enclosed in braces:

```
InfoInput ([<File>:AB], {IRM=’R’,
INP=’C:\OPUS........
```

4) Save the macro as “Parameter 3”.

Listing (PARAMETER 3.MTX)

```
VARIABLES SECTION
FILE <File> = AB, Info;
STRING <Sample Name> = ’New Sample Name’;

PROGRAM SECTION
InfoInput ([<File>:AB], {IRM=’R’,
INP=’C:\OPUS_NT\METHODS’, INM=’DEFAULT.TXD’,
IFN=’DEFAULT’, I01=’<Sample Name>’, T01=’TEXT:Compound
Name’, T02=’TEXT:Molecular Formula’, T03=’TEXT:Molecu-
lar Weight’,
T04=’TEXT:CAS Number’, T05=’TEXT:Origin’,
T06=’TEXT:Boiling Point’,
T07=’TEXT:Sample Technique’, T08=’TEXT:Weight’,
```

```
T09='TEXT:Manufacturer',
T10='TEXT:Reference', T11='TEXT:Charge Number',
T12='TEXT:Comment'});

PARAMETER SECTION
```

Running the Macro

Load a file and first run “Parameter 2”. This generates an information block, which you can display in a report window. Now run “Parameter 3” and check the block again. You will see that only the sample name has changed.

6.21 Parameter 4 – Read From a Report

Task

Often it is necessary to read specific data from a report, which contains the results of several data evaluation methods. In this example, we will extract the number of peaks of a peak pick and subsequently use a loop to read the frequencies of the peaks which will be displayed in a message box. In addition, we will format the output.

Macro Functions

StartLoop, EndLoop, Message, FromReportHeader, FromReportMatrix

We will use the FromReportHeader command to extract a value from the header of a report. That requires to ascertain the position of the desired information (here the number of peaks). In our case the number of hits is found in the third row of the header.

Headers always consist of two parts: a title (e.g. number of hits) and the actual value. The command FromReportHeader allows you to select either the title (option: LEFT) or the value (option: RIGHT).

The FromReportMatrix command allows to read data from a matrix if the column number is known. We want to read frequency values which are located in the first column.

Note: Both commands use the report and subreport parameters which should be set to “1” and “0”, respectively. Only a Quant and Ident report may consist of several main reports and/or subreports. Refer to the manual of these software packages for details.

We will demonstrate the use of characters in combination with a message box, which are laos used as control characters in the command lines. Such a control character will be interpreted as a printable character, if it is repeated twice; if you want to enclose the unit cm-1 by brackets, you achieve this by typing ’[[cm-1]]’. The following characters act as control characters:

< > [ ] { } ’ ; , :

OPUS Functions

Peak Picking

This example introduces no new OPUS functions.

Generating the Macro

1) Define a FILE variable <File> with an AB data block associated and three numerical variables <Count>, <Index> and <Peak Position>. Initialize <Index> with “1”.
2) Select the *Peak Picking* function from the OPUS *Evaluate* pull-down menu to generate a report (consisting of a peak table). On the *Select Files* page, choose the variable <File> and set the *Sensitivity* to “20”. On the *Frequency Range* page select *Use File Limits*.
3) From the *Special Macro Commands* dialog box, choose the FromReportHeader command. Enter the following values:
   - *Variable*: <Count>
   - *File*: [<File>:AB/Peak]
   - *Report*: “1”
   - *Subreport*: “0”
   - *Header Line*: “3”
   - *Header Part*: “RIGHT”

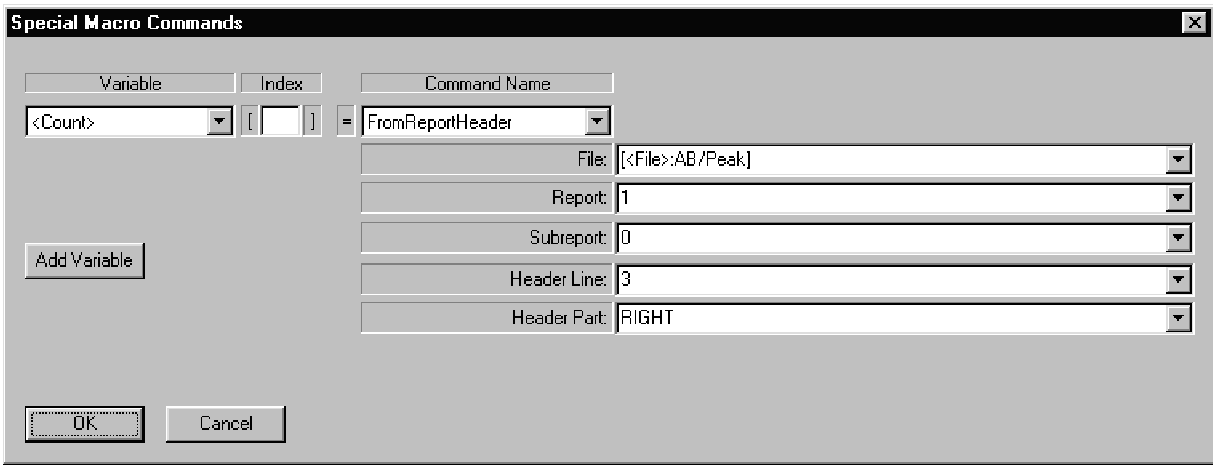


Figure 45: *Special Macro Commands* Dialog – FromReportHeader Definition

4) Insert the StartLoop command using <Count> as variable.
5) From the *Special Macro Commands* dialog box, choose the

FromReportMatrix command. Enter the following values:

- *Variable*: `<Peak Position>`
- *File*: `[<File>:AB/Peak]`
- *Report*: “1”
- *Subreport*: “0”
- *Row*: `<Index>`
- *Column*: “1”

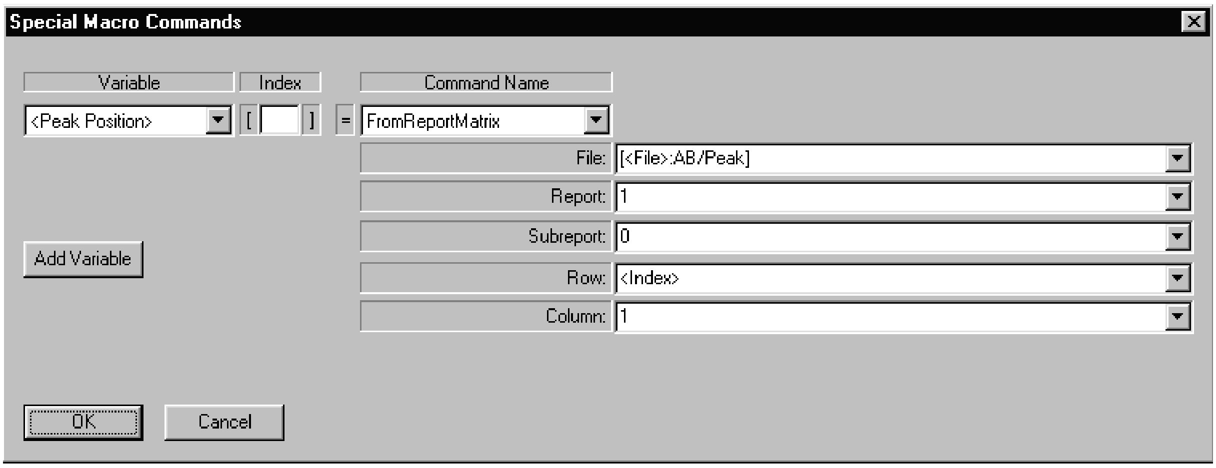


Figure 46: *Special Macro Commands* Dialog – FromReportMatrix Definition

6) We make use of the Message command to display the value, that was read from the report block. Enter the following message text: `’<Index>, Peak at <[,0]Peak Position>[[cm-1]]’`.
   Included in the command line is a statement to format the output; this statement is enclosed in square brackets and defines the number of decimals in our example no digits after the decimal point. [,2] for example would cause an output with two digits after the decimal point. However, these statements only concern the text output and do not change the data. Further information about text formatting can be found in the Macro Reference section.
7) We avoid the necessity to confirm each message by setting the *Time* parameter to “5”. As a result, the message will be displayed for 5 seconds.
8) Next we increment the variable `<Index>` for the line number by 1: `<Index>` = `<Index> +1`
9) Finally, the loop has to be closed by the EndLoop command.
10) Save the macro as “Parameter 4”.

Listing (PARAMETER 4.MTX)

```
VARIABLES SECTION

FILE <File> = AB, AB/Peak;
NUMERIC <Count> = 0;
```

```
NUMERIC <Index> = 1;
NUMERIC <Peak Position> = 0;

PROGRAM SECTION

PeakPick ([<File>:AB], {NSP=9, PSM=1, WHR=1,
LXP=400.000000, FXP=4000.000000, QP8='NO',
QP9=0.200000, PTR=20.000000, QP4='NO', QP7=0.800000,
QP6='NO', QP5=80.000000, PPM=1, QP0='NO', QP3=4});
<Count> = FromReportHeader ([<File>:AB/Peak], 1, 0, 3,
RIGHT);
StartLoop (<Count>, 0);
<Peak Position> = FromReportMatrix ([<File>:AB/Peak],
1, 0, <Index>, 1);
Message ('<Index>. Peak at <[,0]Peak Position> [[cm-
1]]', ON_SCREEN, 5);
<Index> = <Index>+1;
EndLoop (0);

PARAMETER SECTION
```

Running the Macro

Load a data file and run “Parameter 4”. After you selected the spectrum to be processed, a message indicating the first peak is displayed. In the bottom part of the message box, a counter shows the remaining display time. You can skip the message box at any time by clicking on *OK* or wait until the timer runs out.

6.22 Control 1 – Controlling a Macro Using Buttons

Task

So far, we mostly wrote linear code, that is all command lines will be processed subsequently. The first exception from a linear progression was introduced with the loop command. Now we will learn how to tweak a macro program, in order to make it flexible and more powerful.

Note that the following examples should only demonstrate the principle of how to control the flow in a macro and therefore will only use Macro functions.

In our first example we will integrate two buttons (Button 1 and Button 2) in a user dialog box to launch different actions. Clicking on the buttons will display different messages, followed by the initial dialog box. The macro will only be terminated by clicking on *Continue*.

Macro Functions

Label, Goto, Message, User Dialog

We will define two BUTTON variables which are linked to different labels. Clicking on the respective button in the user dialog box will then result in a jump to one of these labels. The tweaks will be closed by the Goto command.

The Goto command allows to continue a macro at any line of the code that will be indicated by a label. The label can be placed anywhere in the macro.

OPUS Functions

This example introduces no new OPUS functions.

Generating the Macro

1) Open the *New/Edit Variable* dialog box and define a BUTTON variable; enter “Button 1” in the *Name* field and “first” in the *Goto Label* field. This generates the following line in the variables window:
   `BUTTON <Button 1> = Goto (first)`
   As you can see, the variable is linked to a jump via the Goto command.

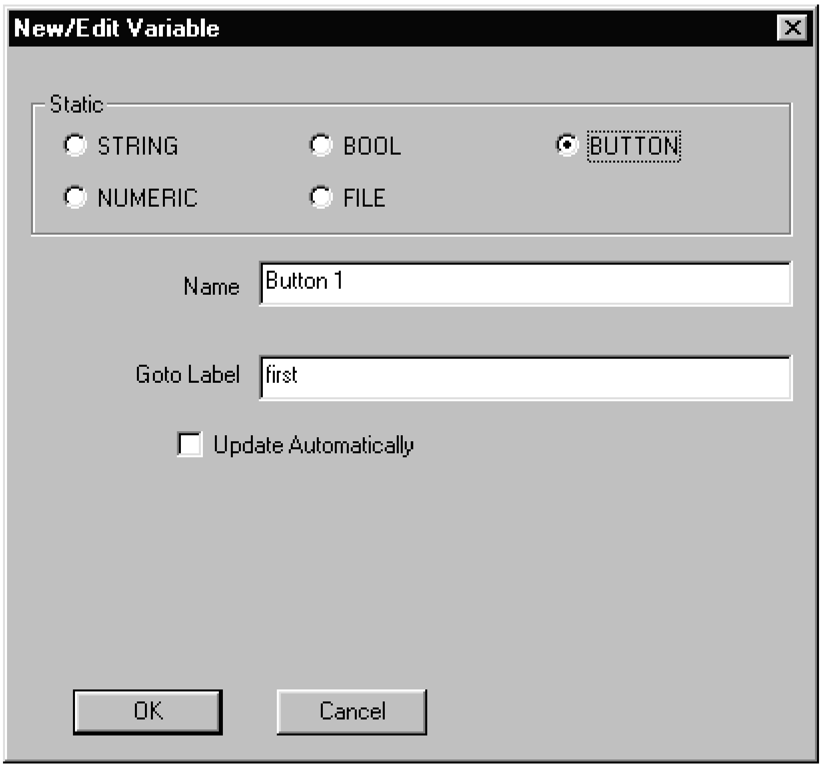


Figure 47: *New/Edit Variable* Dialog Box – Defining a Button Variable

2) Add another BUTTON variable (“Button 2”) and link it to the label “second”.
3) Open the *Special Macro Commands* dialog box, select the UserDialog command and BUTTON as variable type. From the drop-down list

choose `<Button 1>`. To display both buttons in the same line, type "`+<Button 2>`" after the first variable name.

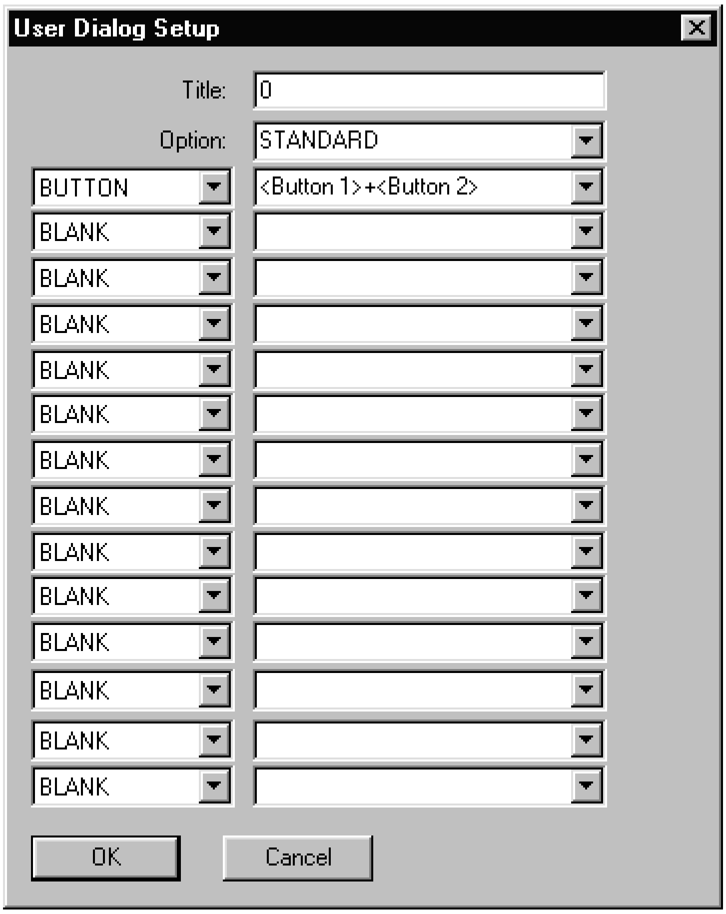


Figure 48: Defining Button Variables

4) Each time you click on *Continue* in a user dialog box, the next program line will be processed. We will redirect the macro to the last line by inserting a Goto statement after the line containing the UserDialog command. Use "end" as label name. You will notice that a label with the same name will automatically be created. We will move this label to the end of the code in the last step.
5) Insert the label for the first jump:
`Label (first)`
6) Append a message that indicates the correct target like "First Button pressed". Set the *Timer* to 5 seconds.
7) After the delay time has expired the user dialog box should be displayed again. Therefore, insert another Goto command with "start" as the label name. Again, the label "start" is automatically generated. Move the label to the top of the PROGRAM SECTION.
8) In a similar manner, add the label for the second button, its message and the Goto statement.
9) Finally, move the line `Label (end)` to the last position of the macro.
10) Save the macro as "Control 1".

Listing (CONTROL 1.MTX)

```
VARIABLES SECTION
BUTTON <Button 1> = Goto (first);
BUTTON <Button 2> = Goto (second);

PROGRAM SECTION
Label (start);
UserDialog (0, STANDARD, BUTTON:<Button 1>+<Button 2>,
BLANK, BLANK, BLANK, BLANK,
BLANK, BLANK, BLANK, BLANK, BLANK, BLANK, BLANK,
BLANK, BLANK);
Goto (end);
Label (first);
Message ('First Button pressed', ON_SCREEN, 5);
Goto (start);
Label (second);
Message ('Second Button pressed', ON_SCREEN, 5);
Goto (start);
Label (end);

PARAMETER SECTION
```

Running the Macro

Run the macro and test both buttons in the user dialog box. Exit the macro by clicking on *Continue*.

6.23 Control 1a – Controlling a Macro Using Buttons

Task

We will modify “Control 1”, so that the default buttons (*Continue*, *Help* and *Cancel*) won’t be displayed in the user dialog box.

Macro Functions

Label, Goto, Message, User Dialog

We will use the option NODEFAULTBUTTON to suppress the default buttons in the user dialog box.

OPUS Functions

This example introduces no new OPUS functions.

Generating the Macro

1) Open "Control 1" and edit the UserDialog command line. Choose NODEFAULTBUTTON from the *Option* list.
2) Save the macro as "Control 1a".

Listing (CONTROL 1a.MTX)

```
VARIABLES SECTION
BUTTON <Button 1> = Goto (first);
BUTTON <Button 2> = Goto (second);

PROGRAM SECTION
Label (start);
UserDialog (0, NODEFAULTBUTTON, BUTTON:<Button
1>+<Button 2>, BLANK, BLANK, BLANK, BLANK,
BLANK, BLANK, BLANK, BLANK, BLANK, BLANK, BLANK,
BLANK, BLANK);
Goto (end);
Label (first);
Message ('First Button pressed', ON_SCREEN, 5);
Goto (start);
Label (second);
Message ('Second Button pressed', ON_SCREEN, 5);
Goto (start);
Label (end);

PARAMETER SECTION
```

Running the Macro

When you run the macro, only the two buttons you defined are displayed. To exit the macro, you now have to click on the small x on the right side of the title bar.

6.24 Control 2 – Controlling a Macro Using If, Else And Elseif

Task

Simple program structures can easily be controlled by the Goto command. The extensive use of Goto statements in longer macros can be confusing. The If statement is a better way of structuring complex programs.

The following example analyzes several parameters entered in a user dialog box:

- Check box: if the check box is selected by the user a message will be displayed.
- Drop-down list showing the options “yes” and “no”: if “yes” is chosen a message will be displayed.
- Numerical field: the input of a numerical field will be compared to a value (here greater than or equals 10) and the result will be displayed.
- Text field: a text search will be performed using the input of the text field on another predefined text and the result will be displayed.

Macro Functions

If, Else, Endif, Message, User Dialog, FindString

The If command compares two variables or values using several numerical or text operators:

Numerical operators:

.EQ.	tests identity
.GT.	tests if value 1 is greater than value 2
.LT.	tests if value 1 is smaller than value 2
.GTEQ.	tests if value 1 is greater than or equal to value 2
.LTEQ.	tests if value 1 is smaller than or equal to value 2
.NE.	tests if value 1 is not equal to value 2

Text operators:

.NOCASE_PARTOF.	tests if text 1 is included in text 2, the text is not case-sensitive
.CASE_PARTOF.	tests if text 1 is included in text 2, the text is case-sensitive

An If statement must be terminated by an Endif statement; Else can be included optionally:

- If statement without Else – if the condition is met, all code enclosed by the If/Endif structure will be processed. Otherwise, the program jumps to the line following the Endif statement. This structure is used if an action should either be processed or not.
- If statement in combination with Else – if the condition is met, all code enclosed by the If/Else structure will be processed. Then a jump to the Endif statement follows and the macro continues with the line following the Endif statement. Otherwise, the code enclosed by the Else/Endif will be processed. This structure is used to process two alternatives.

We will use new line types in the user dialog box: CHECKBOX, COMBOBOX and TEXT.

OPUS Functions

This example introduces no new OPUS functions.

Generating the Macro

1) Define the following variables:

`BOOL <Checkbox> = TRUE;` (the value of BOOL variables can either be “TRUE” or “FALSE”)
`STRING <Yes or No> = '';` (no value defined)
`NUMERIC <Test of Numbers> = 0;`
`STRING <Search in> = 'abcdefghijk';`
`STRING <Search for> = '';` (no value defined)
`NUMERIC <Result> = 0;`

2) Initialize the variable `<Yes or No>` with “Yes” and “No” by entering the following command lines:

```
<Yes or No>[0] = 'Yes';
<Yes or No>[1] = 'No';
```

3) Include a user dialog box of the following type:

Type	Variable	Comment
`CHECKBOX`	`<Checkbox>`	a check box will be displayed
`COMBOBOX`	`<Yes or No>`	a drop-down list consisting of the two values will be displayed
`EDIT`	`<Test of Numbers>`	field for numerical input
`TEXT`	`<Search in>`	displays a text
`EDIT`	`<Search for>`	field for text input

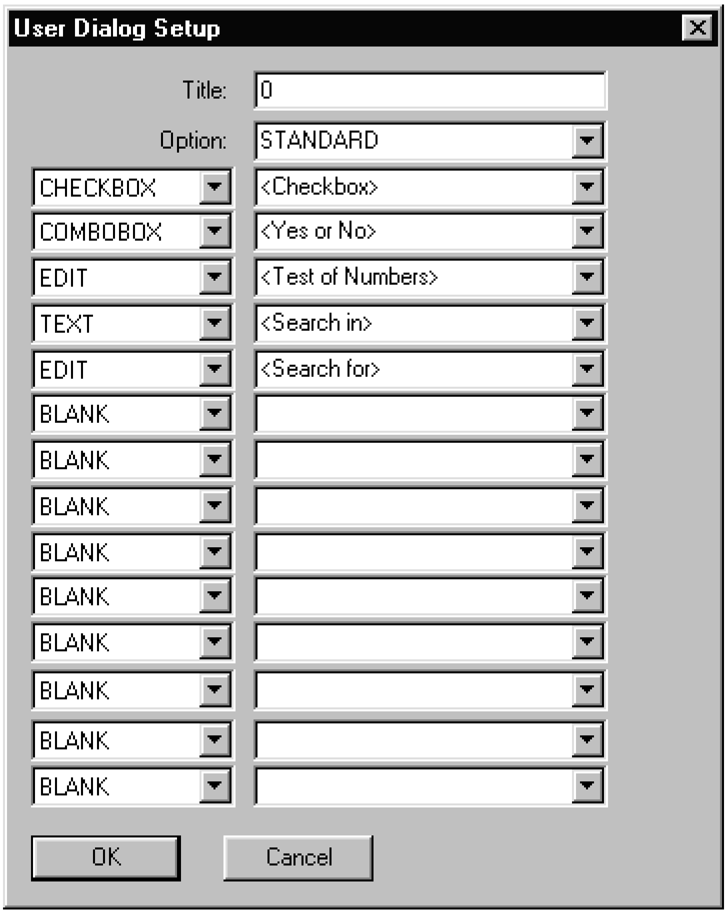


Figure 49: Defining the User Dialog

4) Now the test sequences are included. First we will test if the check box was selected in the user dialog. Include the If command and choose `<Checkbox>` as the first variable. This variable has the value “TRUE” if the check box has been selected, otherwise “FALSE”. Set the *Condition* to “.EQ.” and *Variable 2* to “TRUE”.

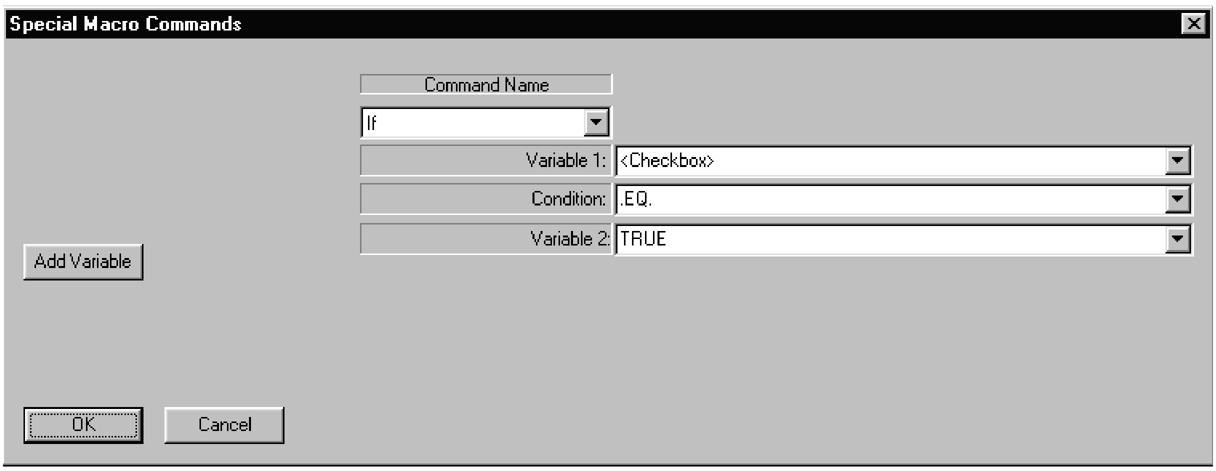


Figure 50: Defining the If Statement

5) In case the check box was selected the following message should be displayed:

`Message ('Check box was checked', ON_SCREEN, NO_TIMEOUT);`

6) The `Endif();` statement closes the first If sequence.
7) Next we will test which value was selected from the drop-down list. This can be done by clicking the (array) type variable without an array index. The variable always returns the value previously chosen from a combo box. Include the following If statement:
Variable 1 `<Yes or No>`
Condition `.CASE_PARTOF.`
Variable 2 "Yes"
8) Again, if the string "Yes" was selected the following message should be displayed:
`Message ('Yes was selected', ON_SCREEN, NO_TIMEOUT);`
9) Close the second If statement with the `Endif();` command.
10) The third test compares the user input (stored in `<Test of Numbers>`) to 10. Include the following If statement:
Variable 1 `<Test of Numbers>`
Condition `.GEQT.`
Variable 2 "10"
11) If the input was greater than 10 or equals 10, the next line is executed and should show the following message:
`Message ('Number is >>= 10', ON_SCREEN, NO_TIMEOUT);`
Note the repeated ">" sign.
12) In order to be able to display another message in case the input is smaller than 10, we include the `Else();` command.
13) The message following the `Else();` command will only be processed if the input was smaller than 10:
`Message ('Number is << 10', ON_SCREEN, NO_TIMEOUT);`
14) We close this If structure with the `Endif();` command.
15) The last test is a text comparison. This time we choose an alternative route. We use the FindString function which searches one text segment in a second text. The result of the search is the position of the text searched for, starting at 0. If the query was unsuccessful -1 will be returned.

    Open the *Special Macro Commands* dialog box and select the FindString command. Use `<Result>` as variable name for the position of the string and set the remaining parameters as follows:
    Search in `<Search in>`
    Search for `<Search for>`
    Search Option "CASE"
16) Now we only need to check whether the returned value differs from -1:
`If (<Result>, .NE., -1);`
17) Again we need to display two messages depending on the outcome of the search. Append a messages stating that the text was found, followed by `Else()`. In the next line include a message stating that the text was not found and terminate the structure by `Endif()`.
18) Save the macro as "Control 2".

Listing (CONTROL 2.MTX)

```
VARIABLES SECTION

BOOL <Checkbox> = TRUE;
STRING <Yes or No> = '';
NUMERIC <Test of Numbers> = 0;
STRING <Search in> = 'abcdefghijk';
STRING <Search for> = '';
NUMERIC <Result> = 0;

PROGRAM SECTION

<Yes or No>[0] = 'Yes';
<Yes or No>[1] = 'No';
UserDialog (0, STANDARD, CHECKBOX:<Checkbox>, COM-
BOBOX:<Yes or No>, EDIT:<Test of Numbers>,
TEXT:<Search in>, EDIT:<Search for>, BLANK, BLANK,
BLANK, BLANK, BLANK, BLANK, BLANK, BLANK, BLANK);
If (<Checkbox>, .EQ., TRUE);
Message ('Checkbox was checked', ON_SCREEN,
NO_TIMEOUT);
Endif ();
If (<Yes or No>, .CASE_PARTOF., 'Yes');
Message ('Yes was selected', ON_SCREEN, NO_TIMEOUT);
Endif ();
If (<Test of Numbers>, .GTEQ., 10);
Message ('Number is >>= 10', ON_SCREEN, NO_TIMEOUT);
Else();
Message ('Number is smaller than 10', ON_SCREEN,
NO_TIMEOUT);
Endif();
<Result> = FindString (<Search in>, <Search for>,
CASE);
If (<Result>, .NE., -1);
Message ('Text was found', ON_SCREEN, NO_TIMEOUT);
Else();
Message ('Text was not found', ON_SCREEN, NO_TIMEOUT);
Endif();

PARAMETER SECTION
```

Running the Macro

Complex programs like these should preferably be tested with the Macro Debugger. Check if all combinations work and if the conditions are met successfully.

6.25 Control 3 – Error Handling

Task

When writing macros it is crucial to know if all functions are executed correctly. Making mistakes while writing your own macros will eventually be unavoidable. Most of the OPUS and Macro functions will return an error code, that can checked within a macro and can be used to change the flow in a macro.

In this example, we will use the OPUS *Load File* command which returns an error message if the indicated spectrum is not found.

Macro Functions

If, Else, Endif, Message, User Dialog,

We use the keyword MACROERROR in combination with the If command to test for errors. The If command must be placed right after the function to be tested.

If MACROERROR is used with the message command, a specific error message will appear on the screen while running the macro.

OPUS Functions

Load, Baseline Correction, Normalize

This example introduces no new OPUS functions.

Generating the Macro

1) We will base this example on the “Load 2” macro.
2) Insert an If statement after the Load function; use “MACROERROR” as *Variable 1*, the .EQ. condition and “TRUE” as *Variable 2*.
   `If (MACROERROR, .EQ., TRUE);`
3) In case of an error, we will make use of the MACROERROR keyword to display an error message.
   `Message (MACROERROR, ON_SCREEN, NO_TIMEOUT);`
4) We would like the macro to proceed normally if no error occurs. Therefore, we include an `Else();` statement.
5) Append an Endif(); statement to become the last line of the macro.
6) Save the macro as “Control 3”.

Listing (CONTROL 3.MTX)

```
VARIABLES SECTION

FILE <$ResultFile 1> = Spec;
```

```
STRING <Path> = 'C:\OPUS_NT\DATA';
STRING <File Name> = 'XYZ';

PROGRAM SECTION

UserDialog (Laden, STANDARD, EDIT:<Path>, EDIT:<File
Name>, BLANK, BLANK, BLANK, BLANK, BLANK, BLANK,
BLANK, BLANK, BLANK, BLANK, BLANK, BLANK);
<$ResultFile 1> = Load (0, {DAP='<Path>', DAF='<File
Name>', INP='D:\OPUS\DEBUG\METHODS',
IFP='D:\OPUS\Release\METHODS', INM='DEFAULT.TXD',
IFN='DEFAULT'});
If (MACROERROR, .EQ., TRUE);
Message (MACROERROR, ON_SCREEN, NO_TIMEOUT);
Else ();
Baseline ([<$ResultFile 1>:Spec], {BME=2, BCO=0,
BPO=64});
Normalize ([<$ResultFile 1>:Spec], {NME=1,
NFX=4000.000000, NLX=400.000000, NWR=1});
Endif ();

PARAMETER SECTION
```

Running the Macro

Run the macro and enter a non-valid file name. After clicking on *Continue*, an error message is shown. Also try an existing file name.

6.26 Timer 1 – Timer Function With Delay Time

Task

In certain situations a time control of the macro is desirable. Examples are certain actions, that should be launched or repeated at a given time or simply the evaluation of the current date and time during run time.

We start by programming a clock that will turn itself off after a delay of one minute.

Macro Functions

StartLoop, EndLoop, GetTime, StaticMessage, Timer

The GetTime function returns the current date and time. Separate variables are used to return the year, month, day, hour, minute and second.

We will bundle the date and time in two text lines with the help of format functions, and display them in a static message box. Contrary to a regular message box, the static message won't interfere with processing the commands of the macro. A repeated call of the static message refreshes the display or can be used to hide the message. We will refresh the display every second. This is done by placing the Timer command in a loop.

We format all variable output to two digits; include [2] as formatting command. The result will be a leading "0" in case of one digit values, while only **the last two digits** will be displayed if the value has more than two digits:

value of the variable: 1	displayed value: 01
value of the variable: 1999	displayed value: 99

OPUS Functions

This example introduces no new OPUS functions.

Generating the Macro

1) Define the following numerical variables: <Year>, <Month>, <Day>, <Hour>, <Minute> and <Second>.
2) The macro will be controlled by a loop with the counter set to 60 (corresponging to a run time of 60 seconds).
```
StartLoop (60, 0);
```
3) Open the *Special Macro Commands* dialog box and select the GetTime function. Due to the large number of return values of this function, the variables are passed as function arguments.
```
GetTime (<Year>, <Month>, <Day>, <Hour>, <Minute>,
<Second>);
```
4) Now select the StaticMessage command and leave the value of Option set to "SHOW". Enter an expression for the date in the first line:
```
<Date> = '<[2]Day>.<[2]Month>.<[2]Year>';
```
In the second line, enter an expression for the time (note the double colons):
```
<Time> = '<[2]Hour>::<[2]Minute>::<[2]Second>';
```

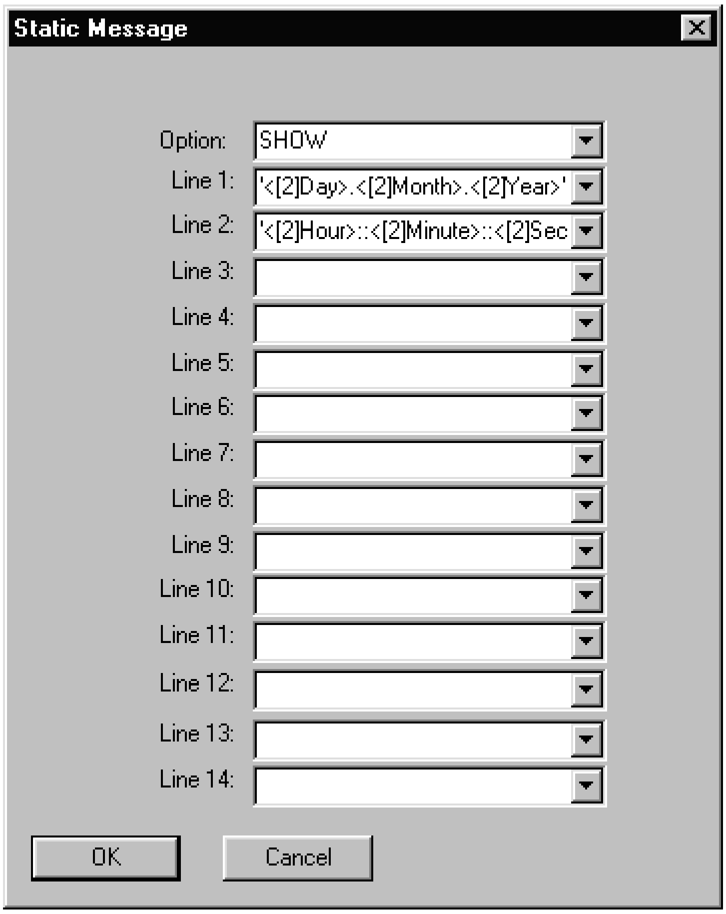


Figure 51: Static Message

5) Append the Timer command with *Option* set to “WAITTIME” and *Time* set to “1”. This causes the macro to wait for one second.
6) Append an EndLoop statement and save the macro as “Timer 1”.

Listing (TIMER 1.MTX)

```
VARIABLES SECTION

NUMERIC <Hour> = 0;
NUMERIC <Minute> = 0;
NUMERIC <Second> = 0;
NUMERIC <Year> = 0;
NUMERIC <Month> = 0;
NUMERIC <Day> = 0;

PROGRAM SECTION

StartLoop (60, 0);
GetTime (<Year>, <Month>, <Day>, <Hour>, <Minute>,
<Second>);
StaticMessage (SHOW, {’<[2]Day>.<[2]Month>.<[2]Year>’,
’<[2]Hour>::<[2]Minute>::<[2]Second>’});
Timer (WAITTIME, 1);
EndLoop (0);
PARAMETER SECTION
```

Running the Macro

A small dialog box is shown in the upper-left part of your screen after you started the macro. The current date and time will be displayed for one minute. The displayed time is refreshed every second.

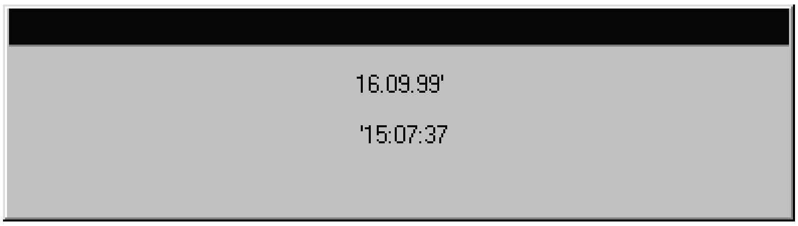


Figure 52: Displaying Date and Time

6.27 Timer 2 – Timer Function Using a Clock

Task

Another option of the Timer function allows to wait until a specified time has been reached. In this example, we will evaluate the current time, add one minute and pause the macro until the clock reaches this time. After one minute has passed a message will be displayed.

Macro Functions

GetTime, Message, Timer

We will use the Timer command in combination with the WAITUNTIL keyword.

OPUS Functions

This example introduces no new OPUS functions.

Generating the Macro

1) Define three numerical variables: `<Hour>`, `<Minute>` and `<Second>`.
2) Open the Special Macro Commands dialog box and select the GetTime function. Since we don't need the date set the parameters for `<Year>`, `<Month>` and `<Day>` to "0". Use `<Hour>`, `<Minute>` and `<Second>` for the remaining parameters.
```
GetTime (0, 0, 0, <Hour>, <Minute>, <Second>);
```
3) Now we add one minute to the current time:
```
<Minute> = <Minute> + 1;
```
4) Choose the Timer command from the *Command Name* list in the *Special Macro Commands* dialog box and select WAITUNTIL in the *Option*

field. Enter the new time in the HH:MM:SS format into the *Time* field:
`<Hour>:<Minute>:<Second>`

5) Append a message box displaying the calculated time:
```
Message (’It is <[2]Hour>::<[2]Minute>::<[2]Second>’
, ON_SCREEN, NO_TIMEOUT);
```
6) Save the macro as “Timer 2”.

Listing (TIMER 2.MTX)

```
VARIABLES SECTION

NUMERIC <Hour> = 0;
NUMERIC <Minute> = 0;
NUMERIC <Second> = 0;

PROGRAM SECTION

GetTime (0, 0, 0, <Hour>, <Minute>, <Second>);
<Minute> = <Minute> + 1;
Timer (WAITUNTIL, <Hour>:<Minute>:<Second>);
Message (’It is <Hour>::<Minute>::<Second>’,
ON_SCREEN, NO_TIMEOUT);

PARAMETER SECTION
```

Running the Macro

After starting the macro, the message “Macro Waiting” will be displayed. The remaining time is shown in the lower part of the box. You can immediately continue the macro by clicking on *Continue*.

6.28 Timer 3 – Timer Function Using the If Statement

Task

“Timer 2” has the disadvantage, that no commands are processed while the macro pauses. The following macro presents an alternative way. As in “Timer 2”, we evaluate the current time and add one minute. But in addition a message is displayed for 2 seconds, after every 10 seconds until the calculated time has passed. Finally, a message showing the current time will be displayed.

Macro Functions

GetTime, Message, Timer, If, Else, Endif, Goto

We will use an If statement in combination with the TIME keyword.

OPUS Functions

This example introduces no new OPUS functions.

Generating the Macro

1) Load “Timer 2” and delete all code from the `PROGRAM SECTION` except the first two lines.
2) Insert a label:
   `Label (Action);`
3) Insert an If statement and enter the calculated time (as HH:MM:SS) in the field *Variable 1*. Enter “.GT.” and “TIME” as *Condition* and *Variable 2*. TIME causes the If statement to compare the current time to the value of *Variable 1*.
4) The operator .GT. ensures, that the command lines following the If statement will be processed until the condition is met. We use the following message, visible for 2 seconds, to indicate that the calculated time is not reached.
   `Message (’Macro still working’, ON_SCREEN, 2);`
5) Use the Timer function to wait another 10 seconds:
   `Timer (WAITTIME, 10);`
6) Append `Goto (Action);` to jump to the label after 10 seconds have elapsed.
7) Append an `Else();` statement to execute the following lines after the specified time has been reached.
8) After the predefined time has been reached the following message will be displayed:
   ```
   Message ('It is <Hour>::<Minute>::<Second>',
   ON_SCREEN, NO_TIMEOUT);
   ```
9) Close the If statement with `Endif();`
10) Save the macro as “Timer 3”.

Listing (TIMER 3.MTX)

```
VARIABLES SECTION

NUMERIC <Hour> = 0;
NUMERIC <Minute> = 0;
NUMERIC <Second> = 0;

PROGRAM SECTION

GetTime (0, 0, 0, <Hour>, <Minute>, <Second>);
<Minute> = <Minute> + 1;
Label (Action);
If (<Hour>:<Minute>:<Second>, .GT., TIME);
Message (Macro still working, ON_SCREEN, 2);
Timer (WAITTIME, 10);
Goto (Action);
Else ();
```

```
Message ('It is <Hour>::<Minute>::<Second>',
ON_SCREEN, NO_TIMEOUT);
Endif();

PARAMETER SECTION
```

Running the Macro

When you start the macro, two alternating messages will be displayed: “Macro still working“ and “Macro waiting”. After one minute, the last message will be displayed.

6.29 Main 1 – Calling Sub Routines with RunMacro

Task

The examples so far have been relatively simple. If you are facing a complex task you will notice, that the total length of the macro increases rapidly. To keep macros clearly structured and simple to read they should be divided into small sub routines. These routines can be tested individually and independent from the status of the main macro. The task of the main macro should therefore be restricted to call these sub routines and exert the overall control.

Our first example, will be a main macro calling “Measure 3” as a sub routine. Before and after calling the sub routine messages should be displayed.

Macro Functions

RunMacro, Message

This example introduces no new Macro functions.

OPUS Functions

This example introduces no new OPUS functions.

Generating the Macro

1) Start with displaying a message:
   ```
   Message ('Submacro is started', ON_SCREEN,
   NO_TIMEOUT);
   ```
2) From the OPUS pull-down menu, select the *Run Macro* command. A load file dialog box will be displayed. Choose “Measure 3” from the Macro directory. Exit the parameter dialog box and append another message:

```
Message ('Submacro has finished', ON_SCREEN,
NO_TIMEOUT);
```

3) Save the macro as “Main 1”.

Listing (MAIN 1.MTX)

```
VARIABLES SECTION

PROGRAM SECTION

Message ('Submacro is started', ON_SCREEN,
NO_TIMEOUT);
RunMacro (0, {MPT='C:\OPUS_NT\Macro', MFN='MEASURE
3'});
Message ('Submacro has finished', ON_SCREEN,
NO_TIMEOUT);

PARAMETER SECTION
```

Running the Macro

When you start the macro, the first message will be displayed. After confirming the dialog, a background spectrum and three sample spectra are measured. Finally, the second message will be displayed.

6.30 Main 2 – Calling Sub Routines with CallMacro

Task

The previous example can easily be implemented, but has its limitations. For instance, data measured or loaded in the sub macro are not accessible from the sub macro and vice versa. Also, parameter values cannot be exchanged between both programs. These restrictions can be overcome by using the Call-Macro command.

We will use the macro “Manipulate 2”, which performs a baseline correction on a spectrum, as a sub routine and display two messages in the main macro, prior to and after the data processing.

Macro Functions

CallMacro, GetMacroPath, UserDialog, Message

We will use the CallMacro command to access the sub routine. CallMacro is able to forward variables to the routine via a user dialog box included in the sub routine. This dialog box must contain these variables in the same order and with the same type in which they appear in the CallMacro command line. In our example, we will exchange only one FILE variable.

We determine the path of the sub macro with the GetMacroPath command. This command returns the path to the directory from which the main macro was started (and which also must contain the sub macro). This ensures that macros work independent of a specific directory structure.

OPUS Functions

This example introduces no new OPUS functions.

Generating the Macro

1) Start a new macro and define a FILE variable `<File>` with an absorption block associated and a TEXT variable `<Path>`.
2) Insert the GetMacroPath command and store the result in `<Path>.`
3) Create a user dialog box to be able to assign a spectrum to `<File>.`
4) Display the following message:
   `Message ('Submacro is started', ON_SCREEN, NO_TIMEOUT);`
5) Select the CallMacro command from the *Special Macro Commands* dialog box. A new dialog box will be displayed. In the first field we enter the path and file name of the sub macro to be called:
   `'<Path>\manipulate 2.mtx'`
   You don't have to enter the file name extension ".MTX". This allows you to either run macros in text or binary format without the need to modify a macro.

   The remaining two columns are used to define the parameter exchange. In the left column, enter the parameter to be forwarded; select `[<File>:AB]` from the list. The right column holds the returned parameters. We don't need to make any entries here, exit the dialog by clicking *OK*.

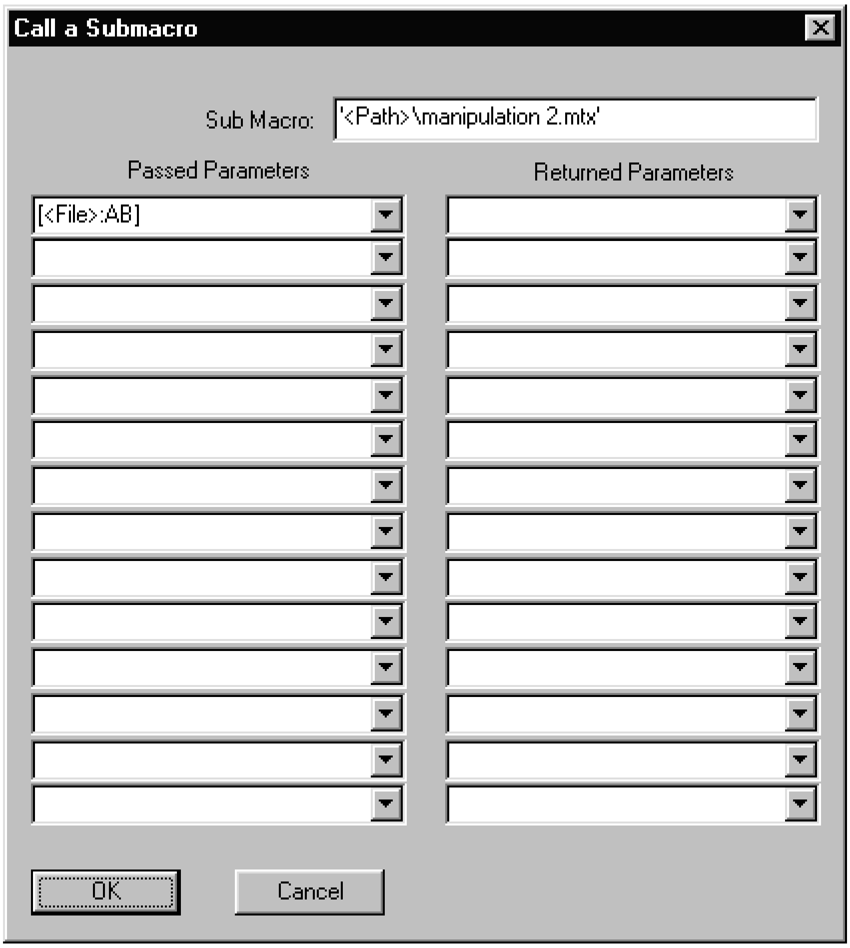


Figure 53: *Call a Submacro* Dialog Box

6) Finally, include a message to indicate that the sub macro has been processed:
```
Message ('Submacro has finished', ON_SCREEN,
NO_TIMEOUT);
```
7) Save the macro as “Main 2”.

Listing (MAIN 2.MTX)

```
VARIABLES SECTION

STRING <Path> = '';
FILE <File> = AB;

PROGRAM SECTION
<Path> = GetMacroPath ();
UserDialog (Main Macro, STANDARD, FILE:[<File>:AB],
BLANK, BLANK, BLANK, BLANK, BLANK, BLANK, BLANK,
BLANK, BLANK, BLANK, BLANK, BLANK, BLANK);
Message ('Submacro is started', ON_SCREEN,
NO_TIMEOUT);
CallMacro ('<Path>\manipulation 2.mtx', {[<File>:AB]},
{});
Message ('Submacro has finished', ON_SCREEN,
NO_TIMEOUT);
PARAMETER SECTION
```

Running the Macro

In OPUS, load a spectrum and run the macro in the Macro Debugger. Step through the first three command lines and select the spectrum previously loaded. When the CallMacro line is reached, an additional button is displayed in the Macro Debugger: *Step Into Submacro*. Upon clicking this button, a new dialog box containing the sub macro is displayed. While this dialog is active, you have no access to the main macro. Now, step through the sub macro; note that the user dialog of the sub macro is **not** displayed. It is used only to assign the values forwarded by the main macro (here the spectral data) to one of its own variables. After the sub macro has been completely processed, the main macro takes control again.

6.31 Main 3 – Returning Values From a Sub Routine

Task

In this example we will call two sub macros. The first sub macro generates a new spectrum by multiplying a spectrum with the Spectrum Calculator. The second sub macro is based on “Parameter 4”, which we will extend and save as “Submacro 2”. This macro creates a peak table with variable frequency limits and sensitivity. All values (the spectrum name, multiplication factor, and the parameters for the peak picking) should be entered in the main macro by the user and then forwarded to the sub macros. The peaks will be read from the peak table and displayed in a message.

Macro Functions

CallMacro, GetMacroPath, UserDialog, Message, StartLoop, EndLoop, FromReportHeader

This example introduces no new macro functions.

OPUS Functions

Spectrum Calculator, Peak Picking

We will use the *Spectrum Calculator* from the OPUS *Manipulate* pull-down menu. This function always generates a new result file.

Generating the Macro

1) Start a new macro for the first sub macro and define a FILE variable `<File>` and a NUMERICAL variable `<Factor>`.
2) Create a user dialog box and insert both variables.

3) Open the OPUS *Manipulate* menu and select the *Spectrum Calculator*. Select the variable `[<File>:AB]` and multiplicate it by 2. Click on the "=" sign. The parameter dialog box opens. We want to use a variable instead of a constant factor, therefore we need to modify the following line for the parameter FOR:
`'[<File>:AB]*<Factor>'`
4) After clicking on *OK*, a new FILE variable `<$ResultFile 1>` is generated.
5) Append a user dialog box, containing only `<$ResultFile 1>`. This causes `<$ResultFile 1>` to be returned to the main macro.
6) Save the macro as "Submacro 1".
7) Load "Parameter 4".
8) Add three NUMERICAL variables `<x-Start>, <x-End>` and `<Sensitivity>`.
9) Add a user dialog containing the variables `<File>`, `<x-Start>`, `<x-End>` and `<Sensitivity>`. Move this line to the top of the macro.

   Edit the PeakPick command and deactivate the option *Use File Limits*. In the parameter dialog box assign the following variables:

FXP	`<x-Start>`
LXP	`<x-End>`
PTR	`<Sensitivity>`

10) Save the macro as "Submacro 2".
11) Load "Main 2" and add the following variables:
```
NUMERIC <x-Start> = 4000;
NUMERIC <x-End> = 3000;
NUMERIC <Sensitivity> = 1;
NUMERIC <Factor> = 0.5;
FILE <New File> = AB;
```
12) Append the four NUMERICAL variables to the user dialog in line two.
13) Change the message text of the messages in line three and five to "Submacro 1..."
14) Edit the CallMacro command and change the name of the sub macro to "Submacro 1.mtx". Add `<Factor>` as second parameter to be exchanged. Choose `[<New File>:AB]` from the *Returned Paramters* list for the `[<File>:AB]` variable.

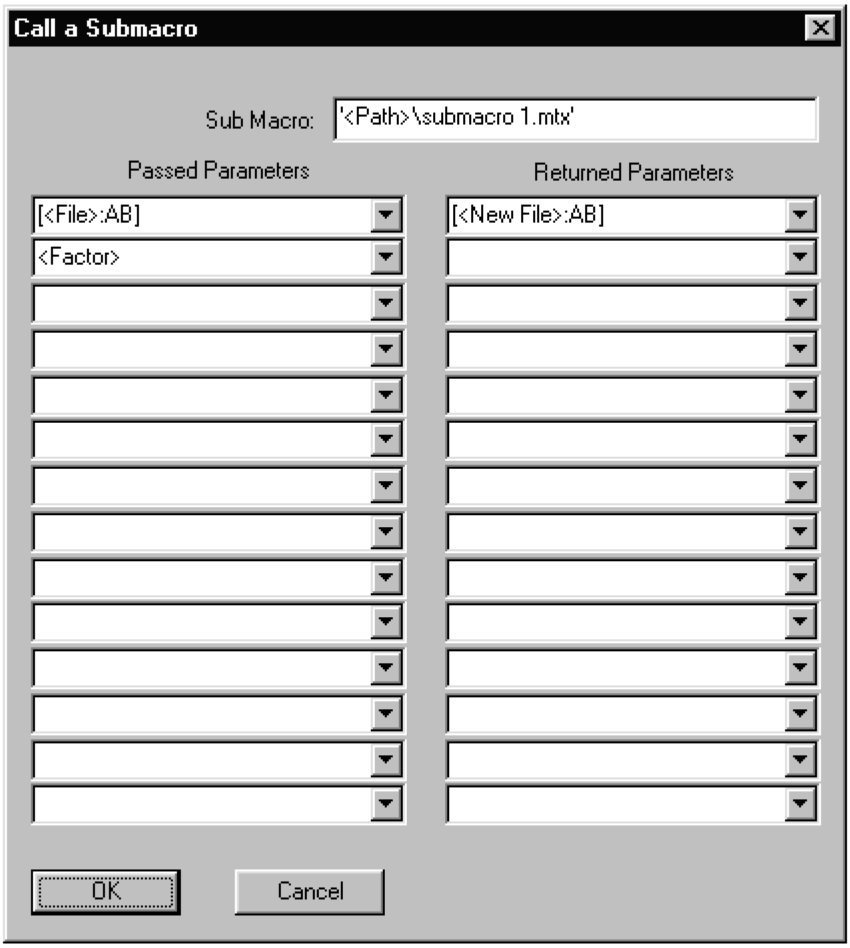


Figure 54: Defining Parameters For Sub Macro 1

15) Add the following message:
`Message (’Submacro 2 is started’, ON_SCREEN, NO_TIMEOUT);`

16) Insert a CallMacro command; enter `’<Path>\submacro 2.mtx’` in the *Sub Macro* field. Select `[<New File>:AB]`, `<x-Start>`, `<x-End>` and `<Sensitivity>` as parameters to be transferred. Sub macro 2 does not return any values.

17) Add the following message:
`Message (’Submacro 2 has finished’, ON_SCREEN, NO_TIMEOUT);`

18) Save the macro as “Main 3”.

Listing (SUBMACRO 1.MTX)

```
VARIABLES SECTION

FILE <File> = AB;
NUMERIC <Factor> = 0;
FILE <$ResultFile 1> = AB;

PROGRAM SECTION
```

```
UserDialog (0, STANDARD, FILE:[<File>:AB], EDIT:<Fac-
tor>, BLANK, BLANK, BLANK, BLANK, BLANK, BLANK, BLANK,
BLANK, BLANK, BLANK, BLANK, BLANK);
<$ResultFile 1> = Calculator ([<File>:AB], {CDI=1,
FOR='[<File>:AB]*<Factor>'});
UserDialog (0, STANDARD, FILE:[<$ResultFile 1>:AB],
BLANK, BLANK, BLANK, BLANK, BLANK, BLANK, BLANK,
BLANK, BLANK, BLANK, BLANK, BLANK, BLANK);

PARAMETER SECTION
```

Listing (SUBMACRO 2.MTX)

```
VARIABLES SECTION

FILE <File> = AB, AB/Peak;
NUMERIC <Count> = 0;
NUMERIC <Index> = 1;
NUMERIC <Peak Position> = 0;
NUMERIC <x-Start> = 0;
NUMERIC <x-End> = 0;
NUMERIC <Sensitivity> = 0;

PROGRAM SECTION

UserDialog (0, STANDARD, FILE:[<File>:AB], EDIT:<x-
Start>, EDIT:<x-End>, EDIT:<Sensitivity>, BLANK,
BLANK, BLANK, BLANK, BLANK, BLANK, BLANK, BLANK,
BLANK, BLANK);
PeakPick ([<File>:AB], {NSP=9, PSM=1, WHR=0, LXP=<x-
End>, FXP=<x-Start>, QP8='NO', QP9=0.200000, PTR=<Sen-
sitivity>, QP4='NO', QP7=0.800000, QP6='NO',
QP5=80.000000, PPM=1, QP0='NO', QP3=4});
<Count> = FromReportHeader ([<File>:AB/Peak], 1, 0, 3,
RIGHT);
StartLoop (<Count>, 0);
<Peak Position> = FromReportMatrix ([<File>:AB/Peak],
1, 0, <Index>, 1);
Message ('<Index>. Peak at <[,0]Peak Position> [[cm-
1]]', ON_SCREEN, 5);
<Index> = <Index>+1;
EndLoop (0);

PARAMETER SECTION
```

Listing (MAIN 3.MTX)

```
VARIABLES SECTION

STRING <Path> = '';
FILE <File> = AB;
NUMERIC <x-Start Frequency> = 4000;
NUMERIC <x-End Frequency> = 3000;
NUMERIC <Sensitivity> = 1;
```

```
FILE <New File> = AB;
NUMERIC <Factor> = 0.5;

PROGRAM SECTION

<Path> = GetMacroPath ();
UserDialog (Main Macro, STANDARD, FILE:[<File>:AB],
EDIT:<x-Start Frequency>, EDIT:<x-End Frequency>,
EDIT:<Sensitivity>, EDIT:<Factor>, BLANK, BLANK,
BLANK, BLANK, BLANK, BLANK, BLANK, BLANK, BLANK);
Message ('Submacro 1 is started', ON_SCREEN,
NO_TIMEOUT);
CallMacro ('<Path>\submacro 1.mtx', {[<File>:AB],
<Factor>}, {[<New File>:AB]});
Message ('Submacro 1 has finished', ON_SCREEN,
NO_TIMEOUT);
Message ('Submacro 2 is started', ON_SCREEN,
NO_TIMEOUT);
CallMacro ('<Path>\submacro 2.mtx', {[<New File>:AB],
<x-Start Frequency>, <x-End Frequency>, <Sensitiv-
ity>}, {});
Message ('Submacro 2 has finished', ON_SCREEN,
NO_TIMEOUT);

PARAMETER SECTION
```

Running the Macro

Similar to the last example, use the Macro Debugger to test the macro.

6.32 Output 1 – Directing Output to a File

Task

The last two examples demonstrate how to handle data output. The first example simply writes three text lines to a file. This file should then be read and its content displayed. Finally, we will delete the file.

Macro Functions

TextToFile, Message, StartLoop, EndLoop, Delete

TextToFile writes text line by line to a specified file. The text will either be appended to an existing file, or a new file can be created to hold the text. The reverse case involves using the ReadTextFile command. This command reads a text file line by line and stores the content in an array.

A set of macro commands exist to copy, rename and delete files. We will use the Delete command to delete the text file created before.

OPUS Functions

This example introduces no new OPUS functions.

Generating the Macro

1) Start a new macro and define the following variables.
```
STRING <Path> = ”;
STRING <Lines> = ”;
STRING <Text> = ”;
NUMERIC <Count> = 0;
NUMERIC <Index> = 0;
```
2) Initialize the first three array elements of <Lines>:
```
<Lines>[0] = ’Line 1’
<Lines>[1] = ’Line 2’
<Lines>[2] = ’Line 3’
```
3) Get the current OPUS path using the GetOpusPath command and save it in <Path>.
4) Expand the <Path> variable by the subdirectory “WORK”.
5) Select the TextToFile command from the *Special Macro Commands* dialog box. Fill in the following text:

Path	<Path>
File Name	“Text.txt”
Text	<Lines>[0]
Output Option	“REPLACE_TEXT”

   REPLACE_TEXT generates a new file or overwrites an existing one with the same name.
6) Repeat these steps twice, each time incrementing the array counter of the <Lines> variable. Instead of “REPLACE_TEXT” use “APPEND_TEXT” to append these lines to the file.
7) Append the ReadTextFile command to read the text file. In the *Text File Name* field, enter ’<Path>\test.txt’. Specify <Text> to hold the return value.
8) Use the GetArrayCount command to determine the number of elements contained in <Text>; save it in <Count>.
9) Start a loop using <Count> as loop counter.
10) Include an array element of <Text> in a message, use <Index> as array index.
11) Increase <Index> by one and close the loop with EndLoop().
12) Append the Delete command. Only the path and file name is required as parameter. State the file to be deleted by entering ’<Path>\test.txt’.
13) Save the macro as “Output 1”.

Listing “OUTPUT 1.MTX”

```
VARIABLES SECTION
```

```
STRING <Path> = '';
STRING <Lines> = '';
STRING <Text> = '';
NUMERIC <Count> = 0;
NUMERIC <Index> = 0;

PROGRAM SECTION

<Lines>[0] = 'Line 1';
<Lines>[1] = 'Line 2';
<Lines>[2] = 'Line 3';
<Path> = GetOpusPath ();
<Path> = '<Path>\WORK';
TextToFile (<Path>, Test.txt, <Lines>[0],
REPLACE_TEXT);
TextToFile (<Path>, Test.txt, <Lines>[1],
APPEND_TEXT);
TextToFile (<Path>, Test.txt, <Lines>[2],
APPEND_TEXT);
<Text> = ReadTextFile ('<Path>\Test.txt');
<Count> = GetArrayCount (<Text>);
StartLoop (<Count>, 0);
Message (<Text>[<Index>], ON_SCREEN, NO_TIMEOUT);
<Index> = <Index>+1;
EndLoop (0);
Delete ('<Path>\test.txt');

PARAMETER SECTION
```

Running the Macro

Run the macro from the Macro Debugger. Watch for the three messages that are displayed.

6.33 Output 2 – Plotting Spectra

Task

We will demonstrate how to plot spectra in different ways. The first command will plot two spectra in one frame. The second command draws two spectra in two different frames.

Macro Functions

UserDialog

This example introduces no new macro functions.

OPUS Functions

Plot

To plot spectra, we use the OPUS *Print Spectra* function from the *Print* menu.

Generating the Macro

1) Start a new macro and define two FILE variables `<File 1>` and `<File 2>`.
2) Add a user dialog containing both variables.
3) From the OPUS *Print* pull-down menu, choose the *Print Spectra* function. Select “FRAME1.PLE” (from the SCRIPT directory) as template and `[<File 1>:AB]` as variable. Close the dialog by clicking on Plot. As you can see in the parameter dialog, the parameter PPA itself consists of a parameter list. You should neither change this parameter nor the template, because they are linked to each other.
4) In contrary to all other OPUS commands, *Print Spectra* allows to print several spectra at once. If you would like to include two or more spectra in the same frame, add their names to the file selection list. Note that the file names are **not separated by commas**.
   `Plot ([<File 1>:AB] [<File 2>:AB], {...`
5) In the same manner, add a second Plot command, this time using “FRAME2.PLE” as template. It contains two frames labelled “OBEN” (upper) and “UNTEN” (lower), between which you switch using the pull down list. Assign `[<File 1>:AB]` to “OBEN” and `[<File 2>:AB]` to “UNTEN”. Exit the dialog by clicking on *Plot*. Now you see, that in the command line generated this time the file names are separated by commas, indicating that they will be plotted in different frames.
   `Plot ([<File 1>:AB], [<File 2>:AB], {...`
6) Save the macro as “Output 2”.

Listing (OUTPUT 2.MTX)

```
VARIABLES SECTION

FILE <File 1> = AB;
FILE <File 2> = AB;

PROGRAM SECTION

UserDialog (0, STANDARD, FILE:[<File 1>:AB],
FILE:[<File 2>:AB], BLANK, BLANK, BLANK, BLANK, BLANK,
BLANK, BLANK, BLANK, BLANK, BLANK, BLANK, BLANK);
Plot ([<File 1>:AB][<File 2>:AB], {PDV='Printer',
SCP='C:\OPUS_NT\Scripts', SCN='frame1.PLE', PUN='CM',
```

```
POP='D:\OPUS\Debug\PRINTS', POF='PRINT.TXT', PDH=0,
PPA='FRM=1,NPL=0,XSP=4000,XEP=400,YMN=0.0,YMX=1.2,ASE=
NO,CWN=NO,CSU=-200.0,,COL=,', PL2=20});
Plot ([<File 1>:AB], [<File 2>:AB], {PDV='Printer',
SCP='C:\OPUS_NT\Scripts', SCN='frame2.PLE', PUN='CM',
POP='D:\OPUS\Debug\PRINTS', POF='PRINT.TXT', PDH=0,
PPA='FRM=2,NPL=0,XSP=4000,XEP=400,YMN=0.0,YMX=1.2,ASE=
NO,CWN=NO,CSU=-
200.0,,COL=,NPL=0,XSP=4000,XEP=400,YMN=0.0,YMX=1.2,ASE
=NO,CWN=NO,CSU=-200.0,,COL=,', PL2=20});

PARAMETER SECTION
```

Running the Macro

In OPUS, load two absorption spectra and start the macro. Choose a spectrum for each FILE variable in the user dialog box and click on *Continue*. Two plots will be printed.

7 Writing External Programs

It is not the intention of this manual to provide a general introduction to programming. Basic knowledge of the fundamentals of writing programs and experience in either Basic or C is required. This chapter demonstrates the design of programs that interact with OPUS, using simple examples (which can also be found on the OPUS CD).

7.1 A Basic Program with DDE Communication Capability

VisualBasic offers an simple approach to establish DDE communication, because DDE functionality has been implemented in certain elements. Furthermore, graphical user interfaces can easily be generated in VisualBasic.

This example can be found on your OPUS CD as Form1.frm (program) and OPUSFrontEnd.vbp (project file).

The form consists of three buttons (Take Reference, Measure Sample and Exit) and a text box called `ddeLink`, which supplies the communication and is also used for text output.

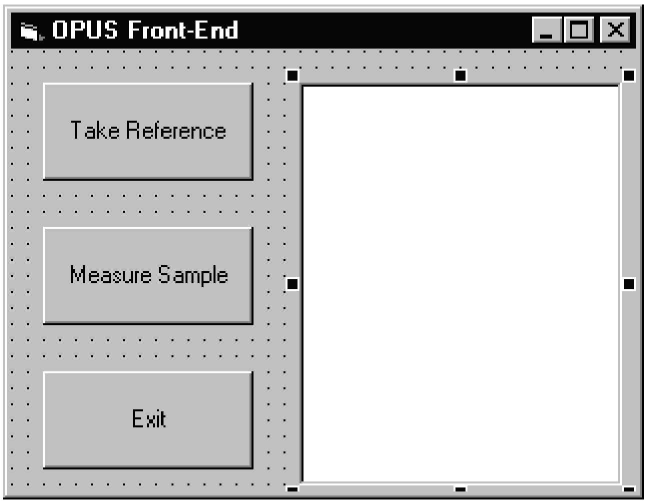


Figure 55: File Form1

7.1.1 Initializing the Connection

The `Load` function (`Form_Load`) serves to interpret a parameter, which contains a text command, as a program that is to be launched. This functionality is used in the Basic program to start OPUS. The function `connectToServer` opens the connection to OPUS.

```
Option Explicit
Dim connected As Integer
Dim timeOut As Integer
Dim serverName As String

Private Sub Form_Load()
Dim BefZl
timeOut = 6000  ’ in ~ tenths of a second
connected = 0
BefZl = Command()
If Len(BefZl) > 0 Then
    Shell (BefZl)
End If
serverName = "OPUS|System"
connectToServer (serverName)
End Sub
```

Its main purpose is to initialize the DDE functionality of the `ddeLink` text box object. First the LinkMode is set to `vbLinkNotify`, which is the asynchronous mode (the Basic program does not pause). As soon as a command was processed by OPUS, the `LinkNotify` event of the `ddeLink` object will be activated. In general, this has proven to be useful, because otherwise the Basic program will wait for a result to be returned and in the meantime will not process any input (e.g. like *Cancel*). By setting the `LinkTopic` to `OPUS|System`, the Basic program rgistered a connection from the system with the given name. If OPUS has been running, it has requested a DDE service using this name and will function as a server. In addition, the `LinkTimeout` is defined.

```
Public Function connectToServer(server As String) As
Integer
connectToServer = 0
On Error GoTo connectToServerErr
ddeLink.LinkMode = vbLinkNotify
ddeLink.LinkTimeout = 100 ’ give time for connection
ddeLink.LinkTopic = server    ’ Set link topic.
ddeLink.LinkTimeout = timeOut
connectToServer = 1
connected = 1
ddeLink = "Connected to " + server
Exit Function
connectToServerErr:
connected = 0
ddeLink = Err.Description
Exit Function
End Function
```

7.1.2 Processing the Commands

Both routines, which are started by pressing one of the buttons have a similar design. Important is the LinkItem, which is used to transmit a command to OPUS as text. In this case it is either TAKE_REFERENCE or MEASURE_SAMPLE. Both commands expect the name of an OPUS experiment file, which defines the type of experiment.

```
Private Sub Reference_Click()
On Error GoTo requestErr

ddeLink.LinkItem = "TAKE_REFERENCE xxx.xpm"
Exit Sub

requestErr:
ddeLink = Err.Description
Exit Sub
End Sub

Private Sub Sample_Click()
On Error GoTo requestErr

ddeLink.LinkItem = "MEASURE_SAMPLE xxx.xpm"
Exit Sub

requestErr:
ddeLink = Err.Description
Exit Sub
End Sub
```

7.1.3 Notification and Result

The `LinkNotify` routine is called as soon as OPUS has processed the command and supplies the result.

The `LinkRequest` call instructs OPUS to transfer the result to the `ddeLink` object. Here, the result will only be displayed in the text box. This would also be the handle for a data processing routine.

If a sample measurement has been started, the spectrum will be sent in the form of a data point table.

```
Private Sub ddeLink_LinkNotify()
On Error GoTo requestErr
ddeLink.LinkRequest
Exit Sub
End Sub
```

7.1.4 Error Handling

`OnErrorGoto` has already been used in the routines described above. If the connection should terminate or if an error occurs, the sub routines for the respective events is called. In our example, the messages will only be displayed.

```
Private Sub ddeLink_LinkClose()
ddeLink = "Connection closed"
connected = 0
End Sub
```

```
Private Sub ddeLink_LinkError(LinkErr As Integer)
Select Case LinkErr
    Case 1
        ddeLink = "Data in wrong format."
    Case 11
        ddeLink = "Out of memory for DDE."
End Select
End Sub
```

7.1.5 Program Termination

Upon termination, the form will be unloaded; in our example the unload function also illustrates the possibility to close OPUS.

```
Private Sub Exit_Click()
Unload Form1
End Sub

Private Sub Form_Unload(cancel As Integer)
On Error GoTo requestErr

ddeLink.LinkExecute "CLOSE_OPUS"
Exit Sub

requestErr:
ddeLink = Err.Description
Exit Sub
End Sub
```

7.2 A C Program Using the Pipe Interface

The ability of OPUS to function as a server can be used by client software to exchange data and parameters or to control macros. One route to exchange data is the use of a Named Pipe, which is a dedicated operation system function for data transfer. The advantage in using a Named Pipe is the fact, that the pipe can be treated like any file system object. Pipes can be opened, closed, read from, and written to similar to files on a hard drive. These functions are embedded in almost any programming language (C: `fopen` and `fclose`, Basic and Fortran: `open` and `close`).

Furthermore, Named Pipes are supported by several network operating systems (like Novell, LAN Server, Windows for Workgroups, Windows NT and OS/2 Warp Connect). In principle, such a client program is able to run on a LAN machine, even if an operating system other than Windows NT is running.

The following program exists also in a similar version for OS/2; a comparison of both versions outlines the Windows NT specific items.

7.2.6 Establishing a Connection

Besides the declaration of variables, the first part of the program mainly serves to establish a Named Pipe connection. First, the name of the Pipe is determined. OPUS opens a Pipe with the name of the program that was launched. This name can be accessed as `argv[0]` and will be added to \.\\PIPE\.

Then a loop tries repeatedly to open the Pipe, using the `fopen` command. If a connection could not be established, the loop will be terminated after a pre-defined amount of time and the program stops.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>

int           main(int argc, char **argv){
float         *data;
FILE          *opuspipe;
char          buffer[255];
char          filename[255];
char          blocktyp[255];
char          pipename[255];
char          *progname;
long          i, numofpoints, entrynum;
double        freqfirstp, freqlastp, scalef;
time_t        starttime;

avail=0;

strcpy(pipename, "\\\\.\\PIPE\\");
progname=argv[0]; /* Remove Path */
if (strchr(progname,':'))
    progname=strchr(progname,':')+1;
while (strchr(progname,'\\'))
    progname=strchr(progname,'\\')+1;
strncat(pipename,progname,255);

starttime=time(NULL);i=0;
while(difftime(time(NULL),starttime)<timeout){
    i++; /* num of tries */
    errno=0;
    if ((opuspipe = fopen(pipename, "rb+")) != NULL)
        break;}
if(difftime(time(NULL),starttime)>=timeout)
    cserror("Timeout - Pipe Open \n");
```

7.2.7 Client/Server Commands

In the next section of the program, a set of commands is processed, following always the same routine. Before a command is processed, the Pipe is reset using the `fseek` command. Then `fprint` writes the command to be transmitted to the Pipe, which is sent immediately by the `fflush` command. `Fwaitgets` transfers the results line by line and also performs an error check.

The program expects a spectrum file indicated on the `Select Program` page of the External Program function, which will be read by the READ_FROM_ENTRY command. DATA_VALUES sets the appropriate mode, and READ_DATA reads the spectral data.

The data will be stored in an array of a size depending on the number of data points.

```
entrynum =1; /* The first file selected in the C/S
box*/
fseek(opuspipe,0,SEEK_SET);
fprintf(opuspipe,"READ_FROM_ENTRY %d\n",entrynum);
fflush(opuspipe);
fwaitgets(buffer, 255, opuspipe);
if (strcmp(buffer,"OK\n"))
    cserror(buffer); /* C/S sent an error code */

fwaitgets(filename, 255, opuspipe);
fwaitgets(buffer, 255, opuspipe);/* contains file num-
ber */
fwaitgets(blocktyp, 255, opuspipe);

fseek(opuspipe,0,SEEK_SET);
fprintf(opuspipe,"DATA_VALUES\n");
fflush(opuspipe);
fwaitgets(buffer, 255, opuspipe);
if (strcmp(buffer,"OK\n"))
    cserror(buffer);

fseek(opuspipe,0,SEEK_SET);
fprintf(opuspipe,"READ_DATA\n");
fflush(opuspipe);
fwaitgets(buffer, 255, opuspipe);
if (strcmp(buffer,"OK\n"))
    cserror(buffer);

fwaitgets(buffer, 255, opuspipe);
sscanf(buffer,"%ld",&numofpoints);
fwaitgets(buffer, 255, opuspipe);
sscanf(buffer,"%lf",&freqfirstp);
fwaitgets(buffer, 255, opuspipe);
sscanf(buffer,"%lf",&freqlastp);
fwaitgets(buffer, 255, opuspipe);
sscanf(buffer,"%lf",&scalef);

if ((data=(float*)malloc(numofpoints*sizeof(float)))
           ==NULL)
   cserror("Out of memory\n");
```

```
for (i = 0; i < numofpoints; i++){/*receive the data */
    fwaitgets(buffer, 255, opuspipe);
    sscanf(buffer,"%f",&data[i]);
    data[i] *= (float)scalef;}
fwaitgets(buffer, 255, opuspipe);
if (strcmp(buffer,"OK\n"))
    cserror(buffer);
```

7.2.8 Data Manipulation

After all data has been read from the OPUS file it is available for processing by the program. In our example the data will only be mutliplied by 2. After the data processing, all data will be written back to the same file. The PRESERVE mode which existed in OS/2, is now obsolete due to the different approach of Windows NT not to manipulate original data.

WRITE_TO_FILE/BLOCK specifies the block type of the target file and WRITE_DATA initiates the write process. Because the spectral data file has been altered by the program, it will be labelled "processed" in the OPUS user interface; if the file was displayed, the display will be refreshed.

```
/* manipulate the data */
for (i = 0; i < numofpoints; i++)
    data[i]*= 2.0;

/* Now Write it Back */
fseek(opuspipe,0,SEEK_SET);
fprintf(opuspipe,"PRESERVE\n"); /* will increment
extension */
fflush(opuspipe);
fwaitgets(buffer, 255, opuspipe);
if (strcmp(buffer,"OK\n"))
    cserror(buffer);

fseek(opuspipe,0,SEEK_SET);
fprintf(opuspipe,"WRITE_TO_FILE %s",filename);/* file-
name contains end of line char !*/
fflush(opuspipe);
fwaitgets(buffer, 255, opuspipe);
if (strcmp(buffer,"OK\n"))
    cserror(buffer);

fwaitgets(buffer, 255, opuspipe);/* contains path +
filename */
fwaitgets(buffer, 255, opuspipe);/* contain fileno */

fseek(opuspipe,0,SEEK_SET);
fprintf(opuspipe,"WRITE_TO_BLOCK %s", blocktyp);/*
blocktyp contains end of line char !*/
fflush(opuspipe);
fwaitgets(buffer, 255, opuspipe);
if (strcmp(buffer,"OK\n"))
    cserror(buffer);
```

```
fseek(opuspipe,0,SEEK_SET);
fprintf(opuspipe,"WRITE_DATA\n");
fflush(opuspipe);
fwaitgets(buffer, 255, opuspipe);
if (strcmp(buffer,"OK\n"))
    cserror(buffer);

fseek(opuspipe,0,SEEK_SET);
fprintf(opuspipe,"%ld\n",numofpoints);
fprintf(opuspipe,"%lf\n",freqfirstp);
fprintf(opuspipe,"%lf\n",freqlastp);
fprintf(opuspipe,"%lf\n",scalef);
fflush(opuspipe);

for (i = 0; i < numofpoints; i++){
    fprintf(opuspipe,"%f\n",data[i]); }

fflush(opuspipe);
fwaitgets(buffer, 255, opuspipe);
if (strcmp(buffer,"OK\n"))
    cserror(buffer);
free(data);
fclose(opuspipe);

return (0); }
```

7.2.9 Reading Data from the Pipe

The program uses `fwaitgets` to read a line of data. However, this can pose a problem, if the complete data set is not yet available or if not all characters have been transferred. Especially a data request happening too early could cause the program to hang, regardless whether the data is being written to the pipe on the server side.

In this aspect the following code is more robust, but requires the use of Windows NT system calls. Because a Pipe is opened similar to a file, a variable of type FILE is used in C to access the Pipe. However, API functions use a system-specific handle instead of this type. First of all, one has to find out the handle of the FILE variable. The API function `PeekNamedPipe` checks if the data is already available. If so, `getc` is used to read the data; otherwise, the routine times out.

```
#include <windows.h>
typedef struct {
        long osfhnd;    /* underlying OS file HANDLE */
        char osfile;    /* attributes of file (e.g.,
open in text mode?) */
        char pipech;    /* one char buffer for handles
opened on pipes */
#if defined (_MT)
        int lockinitflag;
        CRITICAL_SECTION lock;
```

```
#endif  /* defined (_MT) */
    }   ioinfo;
extern _CRTIMP ioinfo * __pioinfo[];
#define IOINFO_L2E          5
#define IOINFO_ARRAY_ELTS   (1 << IOINFO_L2E)
#define _pioinfo(i) ( __pioinfo[i >> IOINFO_L2E] + (i
& (IOINFO_ARRAY_ELTS - \
                              1)) )
#define _osfhnd(i)  ( _pioinfo(i)->osfhnd )
#define _fileno(_stream)  ((_stream)->_file)

#define timeout 200.0

static int avail;

void          cserror(char *errortext)
{
fprintf(stderr, errortext);
fflush(stderr);
exit(3);
}

char *fwaitgets(char *buf, size_t n, FILE *opuspipe){

size_t i, j;
time_t startzeit;

j=0;
startzeit=time(NULL);
for (i=0;i<n;i++){
           if (avail= =0){
           do{
           if
(!PeekNamedPipe((HANDLE)_osfhnd(_fileno(opuspipe)),
0,0,0,&avail,0)){
           LPVOID lpMsgBuf;
           FormatMessage(
           FORMAT_MESSAGE_ALLOCATE_BUFFER |
FORMAT_MESSAGE_FROM_SYSTEM,
           NULL,GetLastError(), MAKELANGID
(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
           (LPTSTR) &lpMsgBuf,0, NULL );
           strcpy(buf,lpMsgBuf);
           LocalFree(lpMsgBuf);
           return (buf);}}
           while ((avail==0)&&(difftime(time(NULL),
           startzeit)<timeout));
           if(difftime(time(NULL),startzeit)>=
           timeout){
           strcpy(buf,"Timeout\n");
           return buf;}
           else{
           buf[i] = fgetc(opuspipe);
           avail--;
```

```
            startzeit=time(NULL);}}
            else{
            buf[i] = fgetc(opuspipe);
            avail--;
            startzeit=time(NULL);}
            if (buf[i]=='\n'){
            i++;
            break;}
}
buf[i]=0;

return (buf);}
```

7.2.10 Changes compared to OPUS-OS/2

The last example already pointed out some changes to programs running on OS/2:

- Changes in the Pipe name
- the use of `fseek`, when switching between read and write
- checking for data with the `PeekNamedPipe` function
- changes in the handling of modified data files
- error messages have changed slightly and are no longer language-specific.

7.2.11 Miscellaneous

A specific Program Pipe will be created every time an external program is launched from the OPUS user interface with the option *Run as OPUS task* set. If this option was not chosen and the program was started using a Pipe, OPUS opens a general Server Pipe named "\\.\PIPE\OPUS" and waits for the external program to connect (this is the reason the OPUS Pipe function has to be started first). After the external program has connected to the Pipe, the connection is of the same quality as a Program Pipe. This type of connection can be used to remote-control OPUS.

To avoid conflicts between several programs running at the same time, OPUS uses the program name as Pipe name.

Note: If you rename the program, the Pipe name will also change!

Identifying the Pipe name during the run time of your program (using a system function) therefore ensures higher stability.

If `Pipe` was selected as communication method, OPUS expects your program to open the Pipe; otherwise an error message will be the result. OPUS closes the Pipe as soon as your program terminates the connection (end of program or connection failure).

It is not recommended to open several Pipes by the same program, for example by starting the same program more than once. The result would be several Pipes with the same name.

In principle, a connection via a network is possible. In this case, the computers network name has to be used instead of the decimal in the Pipe name:

```
\\OPUSPC\PIPE\OPUS
```

would be the Pipe name of OPUS server on the computer OPUSPC.

READ_DATA and WRITE_DATA in binary mode remain the only commands that send binary data and do not use text with an End of Line sequence.

If a command received by OPUS is recognized, *OK* will be returned and the command will be processed (often causing additional data exchange). If the command is not recognized by OPUS (or if arguments are missing), an error message will be returned and OPUS waits for a new command. This is the reason why, after sending a command to the Pipe, the external program should always read a line from the Pipe to ensure that the command has been processed. Certain commands will cause more text to be returned, that also has to be read.

8 Creating Scripts

In this section you will learn more about the OPUS Scripting Editor. All of the examples discussed in this chapter are included on the OPUS CD. The scripts are written in VBScript and show how to call OPUS functions, start a measurement, access spreadsheets and OPUS data files or how to work with timers and DDE communication. Make sure you are granted the right to work with VisualBasic scripts (see *User Settings* dialog).

8.1 VisualBasic Script

A script always consists of a form and program code. The form and the control elements are configurable and are associated to program routines. The first example (first.obs) shows how to process a data file and assign a button to this routine.

8.1.1 Generating Forms and Buttons

Open the Scripting Editor as described in chapter 1.3.3. Click on the button icon of the Toolbox to activate the CommandButton function. Hold down the left mouse key on the form and move the mouse to create a rectangle for the button you want to include. Upon releasing the mouse key, the button will be inserted at this position. You can reposition it by left-clicking on it and moving the mouse; to resize the button, click on the small squares on its border.

If you right-click on the button, you open a pop-up menu through which you can access the *Properties* dialog of the button. Through this dialog, the properties like color and font of the control element can be changed. Enter the text which you would like to be displayed on the button in the *Caption* field. Pressing *Apply* confirms these changes.

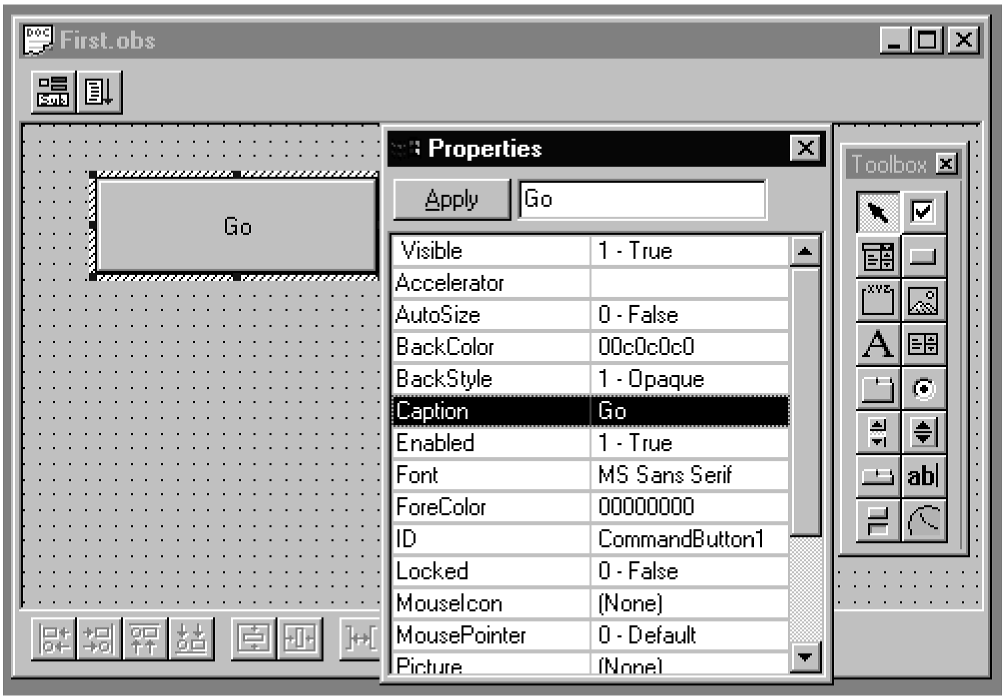


Figure 56: Creating a Button

8.1.2 Objects and Events

In the next step a sub routine will be linked to an event of the button you have just created. Switch to the code window by clicking on the  icon. You will be presented with an empty work space and two drop-down lists. The left list already contains available objects including the entry `CommandButton 1`.

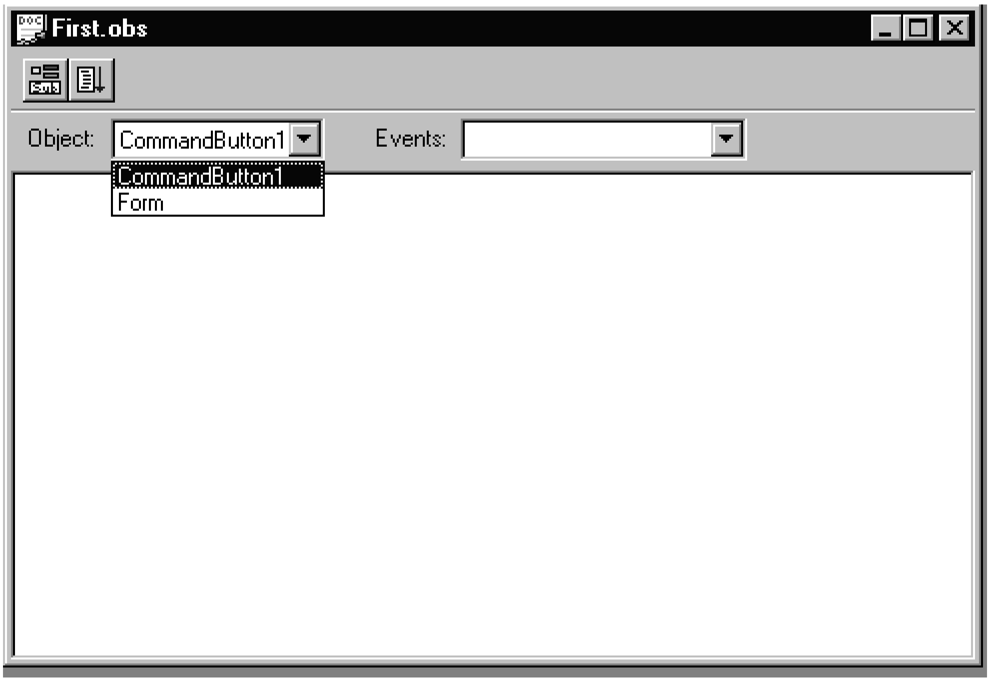


Figure 57: Scripting Editor – Object List

From the *Event* list on the right side you can choose the events associated with the button. Select the event you want to assign to a sub routine An event routine code will be included (here the routine for the event `click`, representing a mouse click). A comment in the code indicates, were you have to include additional statements.

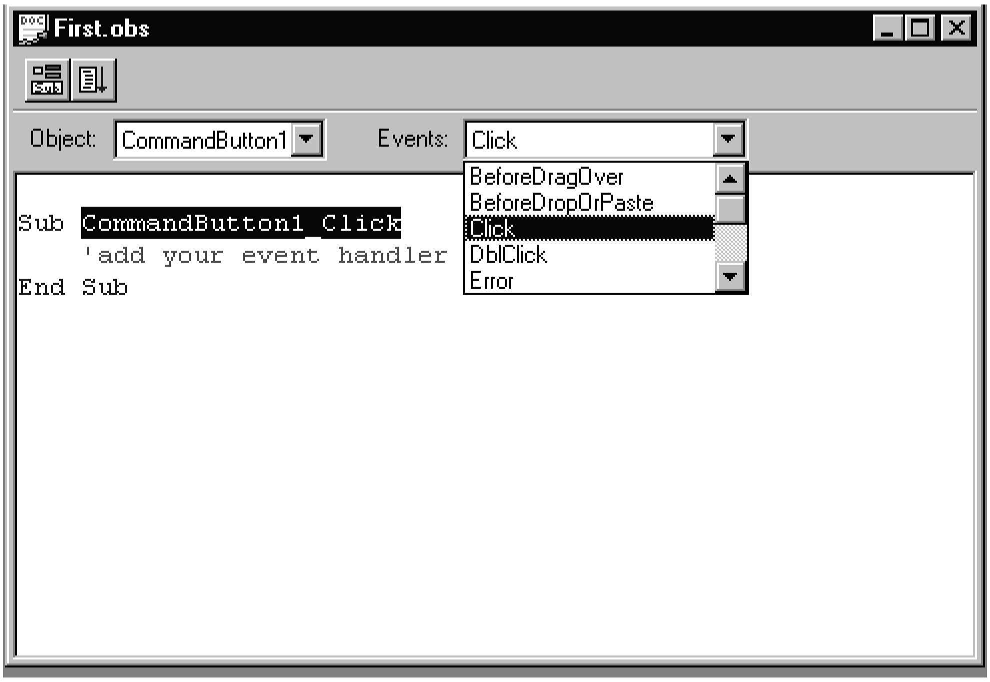


Figure 58: Scripting Editor – Program Routine of the Event *Click*

The routines to be performed upon this mouse click are to be called by the event routine `click.` In case OPUS is to process a file, the `OpusCommand` function is to be used, that forwards the text commands to OPUS:

```
Sub CommandButton1_Click
   Form.OpusCommand("Baseline
           ([""E:\opus\data\abboe05.0""],{})")
End Sub
```

Note the double hyphenation within the command. This is caused by the fact, that the file name has to be enclosed in hyphens, but the script also uses hyphens to indicate text.

Now start the program by clicking on ; the form will be displayed, and the *Go* button is active. If you click on it, OPUS will perform a baseline correction. If no data file has been loaded so far, OPUS loads the file automatically. Switch back to the editing mode by clicking on . Save your work under any name but with the extension .obs (**O**PUS **B**asic **S**cript), using the *Save* command in the *File* menu.

The user right to modify scripts can be assigned in the *User Settings* dialog; if this right was not granted, the user is able to only run the script and is able to perform only the actions defined by the script.

8.1.3 OPUS Functions

Any OPUS function can be included in a text as shown in the example above. Because of the complexity of the functions and function arguments, a shortcut exists to define OPUS functions in a script.

Position the cursor in the section of the code, where you wish to include the function; then simply start the function from the OPUS **tool bar** of the user interface. A dialog differing slightly from the regular one will be displayed. Instead of the usual file selection box, you see an entry field, in which you can enter the file parameter to be used in the script. Clicking on *Correct* will result in the insertion of the OPUS function into the script.

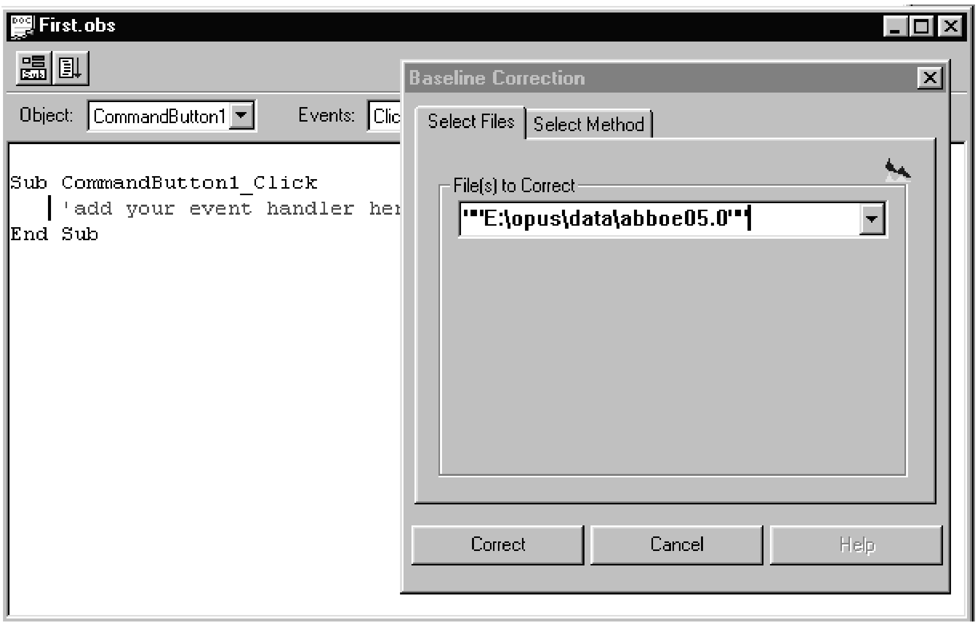


Figure 59: Including an OPUS Function

All parameters of the OPUS function can be defined as usual on the dialog pages. They will be translated to code and appear in the script.

8.1.4 Performing Measurements

The graphical user interface generated by the Basic program in chapter 4.1 can also be generated in a script. Create three buttons and label them *Reference*, *Measure* and *Cancel*, as well as a text box for text output.

Set the parameter `MultiLine` of the text box *Properties* to `true`, to ensure, that long text output will be written line by line. The part of the Basic program, which handles the DDE communication can be omitted in the script. The script is terminated with the function `Close`:

```
Sub CommandButton1_Click
   Form.OpusCommand("TAKE_REFERENCE xxx.xpm")
End Sub
```

```
Sub CommandButton2_Click
   Form.OpusCommand("MEASURE_SAMPLE xxx.xpm")
End Sub

Sub CommandButton3_Click
  Form.Close
End Sub

Sub Form_OnOpusResult( ByVal strResult ,  ByVal
m_strResult2 ,  ByVal m_strResult3 ,  ByVal m_binData )
           TextBox4.Text = strResult + m_strResult2 +
           m_strResult3
End Sub
```

The script makes use of another form function: if a command is processed, the function OnOpusResult will be called, which is an element the form. The result of the OPUS command will be forwarded to this routine and in this example will be written as text to the text box. In case of a terminated measurement this could look like:

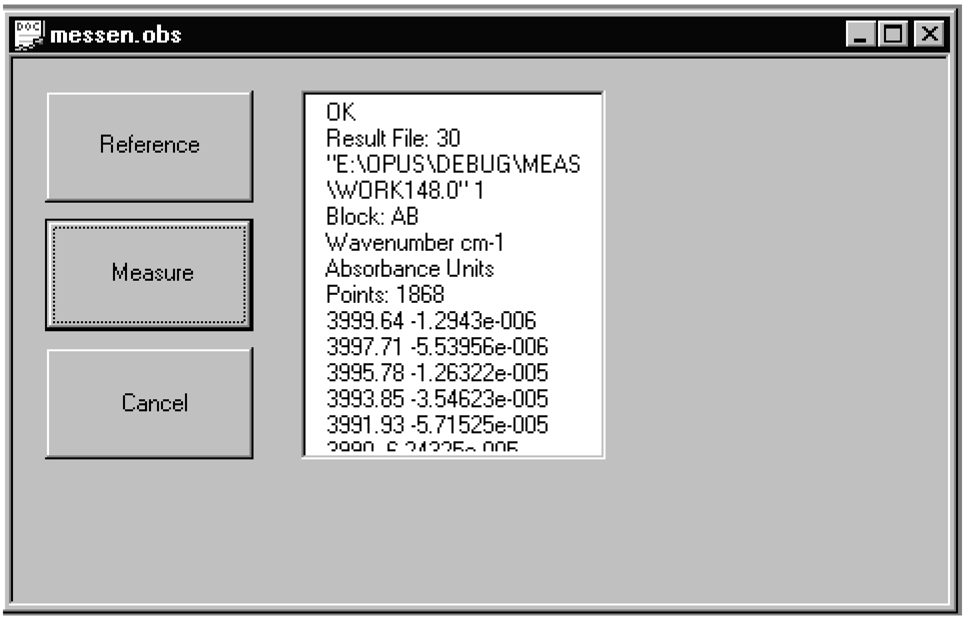


Figure 60: Text Box Messen.obs

The resulting data file consists of a header, comprising the file name, a number, the data block type, and the x and y axis units and the data points.

8.1.5 Accessing Spreadsheets

The following example illustrates how OPUS data can be exported to an Excel spreadsheet. The Excel program must be installed on your computer. The program is started by the script with the `CreateObject("Excel.Sheet")` call; the following command activates the program, which so far is running as a background task, and `Cells` addresses the Excel table cells.

```
Dim ExcelSheet
Sub CommandButton1_Click
  Set ExcelSheet = CreateObject("Excel.Sheet")
  ExcelSheet.Application.Visible = true
  ExcelSheet.ActiveSheet.Cells(1,1).Value = "Hello
  World"
End Sub
```

Further information about which objects and functions (e.g. in Word) are accessible in this way, can be found in the documentation of the Microsoft Office package.

8.1.6 Repeated Calls Using a Timer

A timer object is used to control timed events. While the timer is not visible during run time, it triggers events after a preset amount of time has elapsed. The timer object is placed in the form and the time interval is set in milliseconds.

In the following example, the graphic output of the Infometrix software InStep will be called repeatedly. This also demonstrates how to use the External Program function to establish a DDE connection. Because DDE communication is not supported in scripts, the OPUS function is used in its text command form.

The file ddetest.obs can be found on your OPUS CD; to use the script, you need the InStep software and you have to adjust the path to reflect your environment. The form consists of two buttons to start and stop the repeated addition of data.

```
Dim command
Sub CommandButton1_Click
  command =1
  Form.OpusCommand("ExternalPro-
gram(0,{XPR=F:\instep\instep.exe, XST=2, XCW=0,
DDE=0})")
End Sub

Sub Timer1_Timer
  command=2
  Form.OpusCommand("ExternalProgram(0,{XST=3, DDE=1,
DDS=INSTEP, DDT=DATA, DDI=MacroFile,
DDD=""f:\instep\examples\plat.stp""})")
End Sub

Sub CommandButton2_Click
  Timer1.Interval=0
End Sub

Sub Form_OnOpusResult( ByVal strResult ,  ByVal
m_strResult2 ,  ByVal m_strResult3 ,  ByVal m_binData )
    if command =1 then
        Form.OpusCommand("ExternalProgram(0,{XST=3,
DDE=1, DDS=INSTEP, DDT=DATA, DDI=DataFile,
```

```
DDD=""f:\instep\examples\gasoline.dat""})")
        Timer1.Interval = 10000
        command =3
    end if
    if command =2 then
        Form.OpusCommand("ExternalProgram(0,{XST=3,
DDE=1, DDS=INSTEP, DDT=DATA, DDI=Run, DDD=""  ""})")
        command =4
    end if
End Sub
```

GO (`CommandButton1`) starts the program (XPR = program name, XST = 2 stands for starting up), and the command `DataFile` (DDI) is forwarded to the InStep server, using the `OnOpusResult` routine with the topic (DDT) DATA. This is done by a XTYP_POKE call (DDE = 1), which uses the method name as a parameter (DDD).

The global variable `Command` indicates, which command was terminated when `OnOpusResult is` call. The function `Timer1_Timer` is called every 10 seconds and forwards the name of a macro to be executed (DDI = MacroFile). The InStep macro is started with the DDI = Run command.

Although this example is designed specifically to be used with InStep software, it points out the possibilities of the DDE functionality, and can easily be adapted to other software.

8.1.7 Accessing Spectral Data

Often it is desirable to manipulate OPUS data with the help of self-designed programs. One possibility how to achieve this was shown in chapter 3. The script RWCSTEST.obs basically makes use of the same command routine as the C program in chapter 3.

Here, another function (OpusRequest) of the form is used to call OPUS, which directly returns the result as text. The form consists of a text field for data file name entry, another text field for text output, and a button to start the program.

Initially, the transfer mode is set to allow binary data exchange of type float (BINARY, FLOAT_MODE, FLOATCONV_MODE). Then, the file name and the desired data block is specified using the commands READ_FROM_FILE and READ_FROM_BLOCK. Now, the READ_DATA call to OPUS requires an additional argument to function as a data field. This is the task of the OpusRequestData function.

The additional parameter is a Basic array `Data`, that contains the spectral data upon return. The data field is adjusted automatically to the size of the data block. Therefore, UBound can be applied to find the number of transferred data points.

```
Dim data(10000)
Sub CommandButton1_Click
result= Form.OpusRequest("BINARY")
TextBox2.Text= result
result1= Form.OpusRequest("FLOAT_MODE")
TextBox2.Text= TextBox2.Text+ result1
result2= Form.OpusRequest("FLOATCONV_MODE ON")
TextBox2.Text= TextBox2.Text+ result2
result3= Form.OpusRequest("READ_FROM_FILE
"+TextBox1.Text)
TextBox2.Text= TextBox2.Text+ result3
result4= Form.OpusRequest("READ_FROM_BLOCK AB")
TextBox2.Text= TextBox2.Text +result4
result5 = Form.OpusRequest("DATA_POINTS")
TextBox2.Text= TextBox2.Text+result5
result6 = Form.OpusRequestData("READ_DATA",data)
TextBox2.Text= TextBox2.Text+result6

for i = 1 to UBound(data)
    data(i)= 2*data(i)
    next
```

After data modification, WRITE_DATA writes it back to the file; this shows, that the process of writing data can be split into several commands.

```
result7 = Form.OpusRequest("WRITE_TO_FILE
"+TextBox1.Text)
TextBox2.Text= TextBox2.Text+result7
result8 = Form.OpusRequest("WRITE_TO_BLOCK AB")
TextBox2.Text= TextBox2.Text+result8
result9 = Form.OpusRequest("WRITE_DATA")
TextBox2.Text= TextBox2.Text+result9
result10 = Form.OpusRequest-
Data(CStr(UBound(data))+chr(10)+"1"+chr(10)+
CStr(UBound(data))+chr(10),data)
TextBox2.Text= TextBox2.Text+result10
```

The rest of the script demonstrates, how to access a report block; for this purpose, `PeakPick` is employed to create a peak table in report format. REPORT_INFO, HEADER_INFO and MATRIX_INFO determine the dimensions of the report, header and matrix, respectively. The actual elements of the report are addressed using HEADER_ELEMENT and MATRIX_ELEMENT.

```
Result13 = Form.OpusRequest("COMMAND_LINE PeakPick
(["""+TextBox1.Text+""":AB], {NSP=9, PSM=1, WHR=0,
LXP=400.000000, FXP=4000.000000, QP8='NO',
QP9=0.200000, PTR=20.000000, QP4='NO', QP7=0.800000,
QP6='NO', QP5=80.000000, PPM=1, QP0='NO', QP3=4});")
TextBox2.Text= TextBox2.Text+result13
result14= Form.OpusRequest("READ_FROM_BLOCK AB/Peak")
TextBox2.Text= TextBox2.Text +result14
result15= Form.OpusRequest("REPORT_INFO")
TextBox2.Text= TextBox2.Text +result15
```

```
result16= Form.OpusRequest("HEADER_INFO")
TextBox2.Text= TextBox2.Text +result16
result17= Form.OpusRequest("MATRIX_INFO")
TextBox2.Text= TextBox2.Text +result17
result18= Form.OpusRequest("MATRIX_ELEMENT 1 0 1 1")
TextBox2.Text= TextBox2.Text +result18
result19= Form.OpusRequest("HEADER_ELEMENT 1 0 1")
TextBox2.Text= TextBox2.Text +result19
End Sub

Sub Form_OnLoad
    TextBox1.Text = "E:\opus\data\abboe05.0"
End Sub
```

8.2 JavaScript

Although the Scripting Editor was intended for writing VisualBasic scripts, it can be used as well to generate Java scripts. Therefore, the parameter *ActiveEngine* of the forms *Properties* dialog has to be set to JScript.

Writing Java scripts works the same way as writing VisualBasic scripts, you just have to take into account the specific Java commands. The procedures for calling OPUS functions from within a form are identical. The declaration of functions is slightly different, as you can see from a comparison of the following code (JSCRIP1.OBS).

Java:

```
function CommandButton1::Click( )
{
Form.OpusCommand("NEW_WINDOW 0")
}
```

VisualBasic:

```
Sub CommandButton1_Click
Form.OpusCommand("NEW_WINDOW 0")
End Sub
```

9 Macro Command Reference

This chapter describes all special macro commands. The commands are classified by functionality. You will find an alphabetically sorted list of all commands in section 9.4.

An OPUS Macro consists of three sections, each of which must be present in a macro, even if the sections are empty. Every section begins with its own header:

VARIABLES SECTION
PROGRAM SECTION
PARAMETER SECTION

9.1 VARIABLES Section

This section is reserved for the variable declaration in a macro. Only one declaration per line is allowed. Each line must be terminated by a semicolon. Variables can be of different type. The syntax of the declaration depends on the variable type and is explained in the following sections. STRING, NUMERIC and BOOL and FILE variables can hold a single value as well as an array of values.

General Syntax:

Type <Name> = Value;

*Type*keyword for the variable type.

*<Name>*variable name.

*Value*initial value of the variable.

Variable values can be changed while running a macro in different ways:

- by entering a new value in a user dialog box.
- by calculating a new value using an expression.
- by reading a value from a parameter of a spectrum.
- by reading a value from an info block.
- by reading a value from a report.
- by assigning a value using a macro command.

For array type variables the following additional possibilities exist:

- read a text file (each line is one array element).
- scan a directory (each file name is one array element).
- read the parameter values for some of the measurement parameters.

It is possible to assign a value for an array element without initializing the array elements prior to the selected element. All array elements with lower indices will then be initialized with default values (0 for NUMERIC, BOOL and empty strings for STRING type variables).

Example:

```
<Var>[3] = 123;
```

if this is the first assignment for this variable the elements 0, 1 and 2 will be automatically set to 0.

Usually, the initial value of a variable is declared in the VARIABLES section and will be used whenever a macro is started. Some applications require to revoke any changes made to a variable during the last macro run. This can be achieved by marking the variable.

9.1.1 Variable Types

Five different variable types exist:

STRING	for any string.
NUMERIC	for numerical values, double-precision.
BOOL	can have the values TRUE or FALSE.
FILE	represents a file for processing functions.
BUTTON	command button used for flow control in user dialog boxes*.

* This variables cannot be used as array variable.

9.1.2 Variable Declaration for STRING, NUMERIC and BOOL

The variables must be declared according to the following syntax:

Type <Name>;	declares and initializes a variable with the default values (Numeric = 0, String = empty, Bool = FALSE).
Type <Name> = Value;	declares and initializes a variable with a value.
Type <Name> = ’Value’;	

Note: string values must be enclosed by single quotes.

Type	is the keyword to specify the variable type.
Name	is the unique variable name.
Value	is the start value for the variable.

Examples:

NUMERIC <Index>;	The numeric variable <Index> will be initialized with a value of zero.
NUMERIC <Loop Count> = 10;	Blanks are allowed in variable names; LoopCount is set to “10”.
STRING <Title>;	The string variable <Title> is initialized as empty string.
STRING <Path> = ’d:\data’;	Do not forget to use single quotes for string variable values.
BOOL <Plot?> = TRUE;	BOOL types are used for making decisions in a macro.

Note: The declaration stays the same for variables used as an array type but the array values must be set within the PROGRAM section.

Usage of variables: Simply use the name of the variable enclosed in brackets < >.

9.1.3 Variable Declaration for FILE

The declaration of file variables is almost identical to the other variables. Required data blocks can be specified by assigning them to the variable name.

FILE <Name> = Block1, Block2, ...;	declares a file variable with different data blocks.
Name	is the name of the file variable.
Block n	is the name of the data block.

Examples:

FILE <Input File>;	declares the file variable “Input File” without specifying a data block.
FILE <Input File> = AB;	declares the file variable “Input File” with a single absorbance-block block.

`FILE <Input File> = AB, AB/Peak;`	declares the file variable “Input File” with at least one absorbance block and a peak table of this block.

A file expression is enclosed in square brackets and consists of the variable name, a colon and the block type.

Example:

`Print ([<Input File>:AB/Peak]);`	this is how the file variable is used in a command.

9.1.4 Variable Declaration for BUTTON

The use of a BUTTON variable is restricted to user dialog boxes. It is used to jump to a predefined label in the macro when the button is clicked. The declaration therefore requires a Goto statement to a label i. e. the line indicator for the jump.

BUTTON <Name> = Goto (Label);	declares a button variable with its jump label.
Name	is the variable name and at the same time the text displayed on the button in the dialog.
Label	is the line where the macro execution continues.

Note: Neither the name of the button nor the label for the jump can be changed while executing the macro.

Example:

```
BUTTON <Plot> = Goto (Plot Spectrum);
```

9.1.5 Marking a Variable for Update

Values assigned in the variables section are the initial values whenever a macro is started.

Some cases require, that the variables changed during a run are stored as new initial values for the next run of the macro. You can do this by marking a variable by a preceding “*”. After executing a macro or by using the macro command “SaveVars”, all marked variables are updated to keep their last value. Only variables of type STRING, NUMERIC and BOOL can be marked.

Example:

`*STRING <Text 1> = ’Initial Text’;`	this variable
`.......`	
`<Text 1> = ’New Text’;`	will be updated to ’New Text’
`*STRING <Text 1> = ’New Text’;`	and after running the macro, the above line will read.

9.1.6 Special Characters

Some characters in the macro system are used as control characters in command lines (e.g. {}[]). With two exceptions these characters can be used as all other characters as long as the text is enclosed in single quotes (e.g. ‘123 [g]’).

Exceptions:

Smaller/greater sign “<” and “>” (are used to mark variables)

Single quotes (are used to enclose strings)

To uses these characters within a text line simply enter them twice

Example:

To print a text like	123 < 345	enter: ‘123 << 345’
To print a text like	Result is '123'	enter: ‘result is ‘’123’’’

Square brackets (are used to access array elements)

If used within a text simply type a single character:

To print a text like 123 [g] enter: ‘123 [g]’

In case a variable within a string is followed by square brackets the variable is interpreted as array index, if the leading bracket immediately follows the variable declaration (<Text>[0]). If a blank is inserted between the end bracket “>” of the variable and the leading square bracket (<Text> [0]), the bracket is interpreted as text!

Example:

Two expressions <Text>[0] = ‘ABC’ and <Text>[1] = ‘XYZ’ result in the following combinations:

‘<Text>[0] and <Text>[1]’	is shown as ABC and XYZ
‘<Text> [0] and <Text> [1]’	is shown as ABC [0] and XYZ [1]

9.2 PROGRAM Section

The PROGRAM section is the part of the macro, where command lines are stated. Command lines can be native OPUS Commands, special macro commands or variable assignments.

9.2.1 General Command Syntax

The PROGRAM section consists of several program lines, terminated by semicolons. Each individual line consists of at least three parts:

CommandName (Argument 1, Argument 2, ...);

CommandName	is either a native OPUS command or a special macro command.
Argument n	command arguments; the argument list is enclosed by brackets and the number of arguments depends on the command. Even if a command requires no arguments the brackets have to be present.
Semicolon	The command line must be terminated by a semicolon.

Some Commands are used to assign values to variables. These commands are preceded by the variable name followed by an equals sign.

<Variable> = CommandName (Argument 1, Argument 2, ...);

<Variable>	the name of the variable to be assigned a value. The variable type depends on the command.

9.2.2 Command Names

Command names can either be native OPUS Commands or special macro commands. The commands are listed and described in detail in this chapter and chapter 10.

9.2.3 Command Arguments

Command arguments are necessary to forward specific command parameters to a command. These arguments can be of different type, depending on the purpose of the command:

Text or Numbers

Any text or numbers are forwarded to the command, either as fixed values or as variables.

Keywords

Keywords are specific instructions required for a command. Keywords are always printed in capital letters, e. g. the time behavior of a message is determined by the keyword NO_TIMEOUT.

File

A file argument forwards the file, which is to be processed to the command. A file argument is always enclosed in square brackets and requires the name of the file variable and (separated by a colon) the name of the data block.

[<variable name>:Blocktype]

Example:

```
[<Input File>:AB]   or  [<Result>:AB/Peak]
```

Parameter List

This argument type is mainly needed for OPUS commands and is required, if parameters must be changed during macro execution. A parameter list is enclosed in braces. For each parameter in the list, the three letter parameter name and the parameter value is separated by an equals sign. The parameter values can consist of constants or variables. The different parameters are separated by commas.

{PA1=Value1, PA2=Value2, PA3=Value3, ...}

Return Value

Some commands (e.g. the “Measurement” or the “Load” function) return a value, which has to be assigned to a variable.

These command are preceded by the variable name and an equals sign:

<Variable Name> = Command (Argument 1, Argument2, ...);
[<Variable Name>:Block ID] = Command (Argument 1, Argument2, ...);

9.3 PARAMETER Section

The section marked PARAMETER SECTION contains a list of all necessary function parameters for the OPUS functions used in a macro, which are not included in a command line If this list is empty or if parameters are missing, they will be taken from a default parameter set. Since these parameters can be changed while working with OPUS, results of macros may be unpredictable.

Macros generated automatically by either using the interactive macro editor or by conversion from OPUS-OS/2 macros, will include all required parameters in the single command line and an empty PARAMETER SECTION. For macros written with a text editor, the author is responsible for manually adding a complete parameter set either in the command line or the PARAMETER SECTION.

Syntax:

Name=Value;

Name	three letter parameter name.
Value	default parameter value.

String values must be enclosed by single quotes.

9.4 Macro Functions Sorted Alphabetically

C

CallMacro	run a sub macro
Copy	copies one or more files

D

Delete	deletes one or more files
DisplaySpectrum	shows a spectrum on the screen

E

Else	indicates the point to continue processing after an If statement was FALSE
Endif	closes an If statement
EndLoop	marks the end of a loop
Expressions	use a mathematical expression to assign a value to a variable

F

FindString	searches text within another text
FromReportHeader	reads a value from a report header
FromReportMatrix	reads a value from a report matrix

G

GetArrayCount	retrieves the number of elements in an array
GetEnumList	reads possible parameter values of optics parameters

GetDisplayLimits	retrieves the current display limits of the macro display window
GetLength	retrieves the length of a STRING variable
GetMacroPath	retrieves the path of the current macro
GetOpusPath	retrieves the current OPUS path and assigns it to a string type variable
GetParameter	reads an OPUS parameter from a spectrum file
GetTime	gets system date and time
GetUserPath	retrieves the current User path and assigns it to a string type variable
GetVersion	returns the current OPUS version
Goto	instruction to go to a specified label

I

If	checks a logical expression and act depending on the result

L

Label	jump address within a macro
LoadFile	loads a data file

M

Message	shows a message on screen

O

OpenDisplayWindow	opens a new window for all result files

P

PrintToFile	writes a line of text into a specified text file

R

ReadTextFile	reads the contents of a text file and writes it into an array variable (type STRING)
Rename	renames one or more files

S

SaveVars	updates all selected variables
ScanPath	searches all selected files within a directory and saves their names in an array variable (type STRING)
SetDisplayLimits	sets the frequency limits in a display window
StartLoop	marks the begin of a loop

StaticMessage	shows a permanent message box during execution of a macro

T

TextToFile	writes a text line to a text file
Timer	instruction to achieve time control within a macro

U

UnDisplaySpectrum	hides a spectrum
UserDialog	shows a user-defined dialog box

9.5 Functions Sorted by Categories

Macro commands are available for the following categories:

System Functions

GetOpusPath	retrieves the current OPUS path and assigns it to a string type variable
GetUserPath	retrieves the current user path and assigns it to a string type variable
GetMacroPath	retrieves the path of the current macro
GetVersion	returns the current OPUS version
GetArrayCount	retrieves the number of elements in an array
GetLength	retrieves the length of a STRING variablen
FindString	searches text within another text
CallMacro	runs a sub macro
SaveVars	updates all selected variables

Flow Control Functions

StartLoop	marks the begin of a loop
EndLoop	marks the end of a loop
Label	jump address within a macro
Goto	instruction to go to a specified label
If	checks a logical expression and act depending on the result
Else	indicates the point to continue processing after an If statement was FALSE
Endif	closes an If statement

User Interface Functions

Message	shows a message on screen

StaticMessage	shows a permanent message box during execution of a macro
UserDialog	shows a user-defined dialog box

Input Functions

Enter Expression	uses a mathematical expression for assigning a variable value
FromReportHeader	reads a value from a report header
FromReportMatrix	reads a value from a report matrix
GetEnumList	reads possible parameter values of optics parameters
GetParameter	reads an OPUS parameter from a spectrum file
ReadTextFile	reads the contents of a text file and writes it into an array variable (see alphatbetical list).

Output Functions

PrintToFile	writes a line of text into a specified text file
TextToFile	writes a text line to a text file

File Functions

Copy	copies one or more files
Delete	deletes one or more files
Rename	renames one or more files
LoadFile	loads a data file
ScanPath	scans the path for the specified files and writes them into an array variable (see alphatbetical list).

Time Control Functions

GetTime	gets system date and time
Timer	instruction to achieve time control within a macro

Display Functions

OpenDisplayWindow	opens a new window for all result files
CloseDisplayWindow	closes a display window which had been opened with OpenDisplayWindow
DisplaySpectrum	shows a spectrum on screen
UnDisplaySpectrum	hides a spectrum
GetDisplayLimits	retrieves the current display limits of the macro display window
SetDisplayLimits	sets the display limits for the current macro display window
SetColor	sets the color of the specified spectrum on the display.

9.6 System Functions

System Functions are used to access system values, like for example path names. The following functions are available:

CallMacro	runs a sub macro
FindString	searches text within another text
GetArrayCount	retrieves the number of elements in an array
GetLength	retrieves the length of a STRING variable
GetMacroPath	retrieves the path of the current macro
GetOpusPath	retrieves the current OPUS path and assigns it to a string type variable
GetUserPath	retrieves the current user path and assigns it to a string type variable
GetVersion	returns the current OPUS version
CallMacro	runs a sub macro
SaveVars	immediately saves the current values of marked variables

9.6.1 GetOpusPath

Retrieves the base path from which OPUS was started. The OPUS version number is of type YYYYMMDD (e.g. 19990924).

To allow the design of macros that are machine independent, the path of the OPUS folder can be retrieved at run time and read into a variable. Instead of using fixed path names in a macro, we recommend using this path variable instead.

Syntax:

<Variable> = GetOpusPath ();

<Variable>	name of the variable to receive the current OPUS path.

The variable must be of the type STRING. The path is returned without backslash at the end. If you want to specify a subdirectory of OPUS, you have to insert the backslash between the variable name and the subdirectory name (e.g. <OPUS Path>\methods).

9.6.2 GetUserPath

Retrieves the path to the user specific files and folders of the user currently logged in.

If no user (user name blank at login) is specified, the function acts like the function GetOpusPath and returns the current OPUS path.

To allow to write portable macros, the user specific path can be read into a variable. Instead of using fixed path names in a macro, we recommend to use this path variable instead.

Syntax:

<Variable> = GetUserPath ();

<Variable> name of the variable to receive the current user path.

The variable must be of the type STRING. The path is returned without backslash at the end. If you want to specify a subdirectory of your user path, you have to insert the backslash between the variable name and the subdirectory name (e.g. <User Path>\data)

9.6.3 GetMacroPath

Retrieves the path to the directory that holds the macro currently running and saves it in a STRING variable.

Syntax:

<Variable> = GetMacroPath ();

<Variable> name of the variable to receive the current macro path.

This command requires no parameters.

9.6.4 GetVersion

Gets OPUS version number and assigns it to the specified variable.

Syntax:

<Variable> = GetVersion ();

<Variable> variable to receive the OPUS version number.

The variable must be of type STRING.

9.6.5 GetArrayCount

Determines the number of elements of an array variable.

Syntax:

<Variable 1> = GetArrayCount (<Variable 2>);

<Variable 1> numerical variable, to receive the number of array elements.

<Variable 2> name of the array variable.

9.6.6 GetLength

Determines the length of a STRING variable and stores it in a variable of type NUMERICAL.

Syntax:

<Variable 1> = GetLength (‘<Variable 2>’);

<Variable 1> numerical variable, to receive the number length of the string.

<Variable 2> name of the STRING variable.

Example:

```
STRING <text> = ’Hello world’;
NUMERIC <length> = 0;
<length> = GetLength (‘<text>’);
```

`<length>` has the value 11.

9.6.7 FindString

Finds a specified text within a STRING variable and returns the position of the first character of the search text, starting with zero for the first character of the STRING variable. The return value can be used directly in a text format command. If the search text is not found “-1” will be returned.

Syntax:

<Variable 1> = FindString (‘<Variable 2>’, ‘Text’, Option);

<Variable 1> numerical variable, to receive the result of the query.

<Variable 2> name of the STRING variable, which is used as target.

Text string, to be searched.

Option condition applied for the search
CASEcase-sensitive search
NOCASEsearch not case-sensitive

Example:

```
STRING <text> = ’This is the content of a STRING
variable’;
<Index 2> = FindString (‘<text>’, ’of’, NOCASE);
<Index 1> = FindString (‘<text>’, ’this’, NOCASE);
<Index 2> = <Index 2> - <Index 1>;
<Result> = <[<Index 1>, <Index 2>] text>;
Message (‘<Result>’, ON_SCREEN, NO_TIMEOUT);
```

`<Result>` has the value “the content”.

9.6.8 CallMacro

Function to call a sub macro.

A sub macro is a stand alone macro, which must include a user dialog box. This user dialog box includes all variables, that will be forwarded from the main macro to the sub macro. In addition, a user dialog box can be included in a sub macro as the last command line. This dialog box is used to specify the parameters that will be returned to the main macro. These dialog boxes will not be displayed when running the macro.

Syntax:

CallMacro ('Submacro', { 'Variable A1', 'Variable A2', ...}, {'Variable B1', 'Variable B2', ...});

*Submacro*path and name of the macro to be run.

*Variable An*variables passed from the main macro to the sub macro.

*Variable Bn*variables returned from the sub macro to the main macro.

The variable lists passed to and returned from a sub macro must be consistent with the variable types in the user dialog boxes. That is, the number and type of the variables, as well as their order in the lists and dialogs, must be identical. Blank lines in the dialog box will be skipped. Only variables of type STRING, NUMERIC, BOOL and FILE are allowed in these dialog boxes.

Using sub macros has the advantage, that these macros can be tested individually and called up several times during a macro.

9.6.9 SaveVars

Saves the current values of all selected variables in the macro.

Normally, all selected variables will be saved, if a macro terminates without an error. However, this is not the case, if a macro stops due to a run time error or a power failure. To prevent the macro from starting again with the variable start values e.g. after a power failure, insert this command at the appropriate prosition; restarting the macro will then cause the macro to continue using the last set of values before the power failure.

Syntax:

SaveVars ();

This function requires no parameters.

9.7 Flow Control Functions

Flow control functions are required, if a macro is not intended to run straight from the first to the last line. Flow control function allow to include loops, conditional or unconditional jumps and jumps controlled by buttons in user dialog boxes.

StartLoop	marks the begin of a loop
EndLoop	marks the end of a loop
Label	jump address within a macro
Goto	instruction to go to a specified label
If	checks a logical expression and act depending on the result
Else	indicates the point to continue processing after an If statement was FALSE
Endif	closes an If statement

9.7.1 StartLoop

Marks the beginning of a loop.

A loop is used to repeat a sequence of macro or OPUS commands. The loop count, i.e. the number of repetitions of the command sequence, can either be a constant or a NUMERIC variable. In case that a FILE variable is used, the counter is automatically set to the number of files selected for this variable. This allows to write macros, that account for any number of files.

Each loop begins with the StartLoop statement and ends with the EndLoop statement. Also, each loop is identified by its loop index number. The loop index number facilitates the correlation of StartLoop and EndLoop statements, if loops are nested. Nesting of loops is allowed, as long as the beginning and the end of a nested loop are both within the start and end of the outer loop(s).

Syntax:

StartLoop (LoopCount, LoopIndex);

LoopCount: the loop count can either be a positive number, a numeric variable, or a file variable. If a FILE variable is chosen, the loop count is determined by the number of selected files.

LoopIndex a running index number, needed to correlate the StartLoop with the EndLoop.

A loop count of zero or negative value is not allowed.

9.7.2 EndLoop

Marks the end of a loop.

For details about loops see “StartLoop”.

Syntax:

EndLoop (LoopIndex);

LoopIndex	a running index number, needed to correlate the StartLoop statement with the EndLoop statement.

9.7.3 Goto

Instruction to jump to a label.

Small macros usually are executed sequentially from the first line to the last line. The Goto statement adds more flexibility to a macro, especially if the Goto statement is combined with an If statement. The Goto statement on its own can be used to implement “endless” loops (at the end of the macro a jump to a label at the beginning of the macro).

Syntax:

Goto (Label);

Label	name of the label to jump to.

9.7.4 Label

A label marks the starting point for a Goto instruction.

The label statement itself does not perform any action. Thus labels can be placed anywhere within a macro.

Syntax:

Label (Name);

Name	is the unique name of the label.

The label name must be unique and may not be used as a variable name at the same time. Please note, that labels within a loop are only allowed, if the Goto statement linked to the label (or the user dialog box with the button) is placed within the same loop.

9.7.5 If ... Else ... Endif

Checks a logical expression and executes the sequence of command lines following the If statement, in case the expression is TRUE. If the expression is FALSE, the command sequence is skipped, until either the Else or Endif statement is encountered. Execution continues at the line following the Else or Endif statement. The Endif statement is mandatory. If instructions can be nested.

Syntax:

If ('Value1' .Condition. 'Value2');

Command Sequence 1

.....

Else ();

Command Sequence 2

.....

Endif ();

or

If ('Value1' .Condition. 'Value2');

Command Sequence 1

.....

Endif ();

Value1	first value to compare, can be a number, text, bool or the keyword TIME.
Value2	second value to compare, can be a number, text, bool or the keyword TIME.
.Condition.	logical comparison, the operator is enclosed by decimal points.
.EQ.	equal; to compare strings in an If statement, can be a number, text or bool.
.GT.	greater than; numeric.
.LT.	lower than; numeric.

.LTEQ.	lower than or equal, numeric.
.GTEQ.	greater than or equal, numeric.
.NE.	not equal, numeric
.NOCASE_PARTOF.	part of string, case insensitive.
.CASE_PARTOF.	Part of string, case sensitive.
Command Sequence 1	Sequence of command lines executed if the expression is TRUE.
Else ()	command marking the end of Command Sequence 1 and the beginning of Command Sequence 2.
Command Sequence 2	Sequence of command lines executed if the expression is FALSE.
Endif ()	command marking the end of the If statement, can either be at the end of the Command Sequence 2 if the keyword Else is used or at the end of Command Sequence 1 if no Else statement is used.

Note that all three commands, like all other command lines, must be terminated by semicolons. Also, the Else and Endif commands both require brackets.

The values to be compared can either be specified directly or as variables. The keyword TIME allows to compare a time (format HH:MM:SS) statement with the current system time. Set one of the values to the time to be compared to the system time and use TIME as the second value.

Example 1:

```
If (<Number 1> .GTEQ. <Number 2>);
Message ('<Number 1> is larger than or equal to <Num-
ber 2>', ON_SCREEN, NO_TIMEOUT);
Else ();
Message, ('<Number 1> is smaller than <Number 2>',
ON_SCREEN, NO_TIMEOUT);
Endif ();
```

Example 2:

```
If (<Baseline?> .EQ. TRUE);
Baseline ([<File>:AB], {});
Endif ();
```

Example 3:

```
If (TIME .GT. 12:00:00);
Message ('It is lunch time', ON_SCREEN, NO_TIMEOUT);
Endif ();
```

Example 4:

```
If (...);
....
        If (...);
        ....
        Endif (...);
....
Else ();
....
        If (...);
        ....
              If (...);
              ....
              Endif ();
        ....
        Else ();
        ....
        Endif ();
....
Endif ();
```

9.8 User Interface Functions

User interface functions are used to allow the communication between the operator and the running macro.

Message	shows a message on screen
StaticMessage	shows a permanent message box during execution of a macro
UserDialog	shows a user-defined dialog box

9.8.1 Message

Shows a message box on screen.

The macro execution stops as long as the message box is shown. A confirmation is required (i.e. a click on the OK button) to continue running the macro. If required, a timeout value can be specified, to prevent the hang up of the macro.

Syntax:

Message (’Text’, Option, Timeout);

Text the text to be displayed in the message box. The message can either be text or variables or a combination of both. Note that the text has to be enclosed in single quotes.

Option keyword for the behavior of the message box.

ON_SCREEN	shows a message on the screen.
ON_PRINTER	prints message.
Timeout	specifies, how long the message box will be displayed.
NO_TIMOUT	message will stay on screen, until the user clicks OK.
x	time in seconds, during which the message box stays on the screen.

The option keywords can be combined if both actions are required ON_SCREEN | ON_PRINTER.

If a timeout value is specified, the user still can terminate the message by clicking on *OK*, before the specified time is over.

9.8.2 StaticMessage

Shows a permanent message dialog during the execution of a macro.

This dialog will not interrupt the macro execution and does not require any user confirmation. Up to 14 lines of text can be displayed in the dialog. Depending on the number of text lines displayed, the box will automatically be resized.

Syntax:

StaticMessage (Option, {'Text1', 'Text2', ..., 'Text14'});

Option — keyword for either displaying or hiding the message window.

- *SHOW* — shows the window and updates all lines.
- *HIDE* — removes the window from the screen; if this option is set, the other options are not required (Example: StaticMessage (HIDE, {});).

Text n — text for line number n, can be either pure text or variables or a combination of both. Note that the text has to be enclosed in single quotes.

9.8.3 UserDialog

Shows a user-defined dialog box.

The dialog box can hold up to 14 lines of text. It is intended to enter or select files and variables, as well as buttons for immediate execution of Goto statements. The default box has two buttons *Continue* and *AbortMacro* at the bottom. These buttons can be hidden.

Syntax:

***UserDialog ('Title', Options, Keyword 1:'<Variable 1>',...,
Keyword 14:'<Variable 14>');***

Title the text shown in the title bar of the dialog box, can be a text or a variable.

Option options specifying the behavior and appearance of the dialog box.

0 standard dialog.

NODEFAULTBUTTON do not show the default buttons *Continue* and *Abort Macro*.

Keyword n specifies the type of control shown in line number n. Types can be:

BLANK empty line.

TEXT a text line showing the text of the specified STRING variable.

EDIT an edit field for a STRING or NUMERIC variable. The edit field is preceded by the variable name.

CHECKBOX a check box for a BOOL variable. The check box gets the variable name.

COMBOBOX a combobox showing the contents of an array type STRING or NUMERIC variable. The combobox is preceded by the variable name.

BUTTON a command button which is always connected to a Goto instruction. If two buttons should be displayed two variable names separated by a + sign have to be used (example: `BUTTON: <Button1> + <Button2>`).

*<Variable n>*the variable to be used in line number n.

9.9 Input Functions

Input functions change variables or read parameters from a spectrum, an information block or a report.

Enter Expression	uses a mathematical expression for assigning a variable value
FromReport Header	reads a value from a report header
FromReport Matrix	reads a value from a report matrix
GetEnumList	reads possible parameter values of optics parameters
GetParameter	reads an OPUS parameter from a spectrum file
ReadTextFile	reads the contents of a text file and writes it into an array variable

9.9.1 Enter Expression

Any mathematical expression can be used to assign values to a variable.

Syntax:

<*Result*> = *Expression;*

<Result> variable receiving the result of the expression.

Expression mathematical expression.

Any mathematical expression can be used here. Values can either be represented by numbers or NUMERIC variables. Use the mathematical operators in the same way as a pocket calculator. Use brackets to ensure the correct sequence when calculating an equation (e.g. '(2 + 2) * (4 - 2)').

Use the following case sensitive syntax to access mathematical functions:

SQRT	square root
PI	the number Pi
LN	natural logarithm
LG	decimal logarithm
EXP	exponent
DXP	decimal exponent
sin	sine
cos	cosine
tan	tangens
asin	arc sine
acos	arc cosine
atan	arc tangens
sinh	hyperbolic sine
cosh	hyperbolic cosine
tanh	hyperbolic tangens

You can also assign text to a string variable. In this case, the expression on the right side of the equals sign must be enclosed in single quotes. Text, variables or a combination of both can be used.

Example:

```
<DataPath> = '<OPUS Path>\Data';
```

9.9.2 GetParameter

Function to read an OPUS parameter from a spectrum file.

Syntax:

<Variable> = GetParameter ([<File>:BlockID], Parameter);

<Variable>	the name of the variable for the parameter value.
<File>	name of the file variable.
BlockID	name of the data block to read from (see below).
Parameter	three letter parameter name of the parameter to be read.

If the data block is of the type spectrum, the parameter is read from the parameter block associated with the specified data block. If the data block is of the type INFO, one of the info text lines can be read.

The parameter names for INFO blocks are:

Txx = text definition of line xx (xx = 00 - 99)

Ixx = contents of line xx (xx = 00 - 99)

9.9.3 FromReportHeader

Reads a value from a report header.

Syntax:

<Variable 1> = FromReportHeader (File, Report, Subreport, Line, Option);

Variable 1	the name of the variable for the report value.
File	file expression of the file variable to read from (report block must be specified).
Report	report number (default = 1) in the report block.
Subreport	subreport number (default = 0, reads from main report).
Line	header line to be read, either a constant or a NUMERICAL variable.

Option	keyword stating which part of the line to read.	
	LEFT	left part of the header line, usually the title
	RIGHT	right part of the header line, usually global values (e.g. number of peaks in a peak table)

9.9.4 FromReportMatrix

Reads a value from a report matrix.

Syntax:

<Variable 1> = FromReportMatrix (File, Report, Subreport, Line, Row);

Variable 1 variable for the return value.

File file expression of the file variable to read from (must be a report block).

Report report number (default = 1) in the report block.

Subreport subreport number (default = 0 reads from main report) of the report.

Line number of the column to be read, either a constant or a NUMERICAL variable.

Row number of the line to be read, either a constant or a NUMERICAL variable.

9.9.5 ReadTextFile

Reads a text file into a variable array. Each line transforms to an array element.

Syntax:

<Variable > = ReadTextFile (‘File’);

Variable variable to hold the text lines as a list.

File file specification, including path, name and extension.

9.9.6 GetEnumList

This function has not been implemented yet.

Gets all enum parameter values and writes them to the array elements of the specified variable. Parameters of enum type are mainly used for optic parameters, which have a predefined set of allowed values. In most cases, these values depend on the optics type. Typically, they are chosen from a Combobox in a user dialog.

Syntax:

<Variable> = GetEnumList (Parameter);

<Variable>	name of the array variable to receive the list of allowed values. Each array element is assigned a value.
Parameter	name of the enum parameter.

The variable must be of type STRING.

9.10 Output Functions

Output functions are used to print results on a printer, into a text file or into the print log file. In the current version, only the functions TextToFile and PrintToFile are available which write lines of text into a text file. Due to compatibility reasons, the Print function of OPUS-OS/2 macros is mapped automatically to the function PrintToFile.

TextToFile	writes a text line to a text file
PrintToFile	writes a line of text into a specified text file

9.10.1 TextToFile

This function is the standard macro function to write a line of text into a text file.

Syntax:

TextToFile (‘Path’, ‘File’, ‘Text’, Option);

Path	the path of the text file.
Name	name of the output file (specify with extension).
Text	text line to write into the file.
Option	controls how the text is written to the file

APPEND_TEXT	the new text will be appended to the existing one.
REPLACE_TEXT	the new text will replace the old text; if the file does not exist, it will be created.

9.10.2 PrintToFile

This function writes a line of text into a text file.

The syntax is equivalent to an OPUS command, allowing to easily map the function to the OPUS print function. If the text file does not exist, it will be created. If a file already exists, the text line is appended at the end.

Syntax:

PrintToFile (0, {POP='Path', POF='Name', PTX='Text'});

0	file list; see comment below.
POP	parameter name for output path.
Path	the path of the output file.
POF	parameter name for output file name.
Name	filename of the output file (specify with extension).
PTX	parameter name for text line.
Text	text line to write into the file.

The values of all three parameter can either be text or variables. The first command argument is normally the file list, specifying which report shall be printed. If only a single line of text is printed, this argument is zero. Because the Print function for reports has not been implemented so far, an argument which does not equal zero will cause an error message.

9.11 File Functions

File functions are used to access files within macros

LoadFile	loads a data file
ScanPath	scans the path for the specified files and write them into an array variable
Copy	copies one or more files
Rename	renames one or more files
Delete	deletes one or more files

9.11.1 LoadFile

Function to load one or more data files into OPUS.

Syntax:

<File> = LoadFile (‘Filename’, Option);

<File>	name of the file variable for assigning the loaded data file. The LoadFile function returns the internal file number of the loaded file.
Filename	full path and file name of the file to be loaded, can either be a text or a STRING variable or a combination of both.
Option	option for behavior if a file cannot be loaded (see remark below).
WARNING	shows a dialog box with an error message; this option can be combined with one of the two following options (e.g. WARNING | ABORT).
ABORT	aborts the macro.
Goto (Label)	jumps to the specified label.

If a file could not be loaded, the error condition is TRUE and the FILE variable is not initialized. Therefore, in general the ABORT option should be used. If the file needs not to be processed immediately or at all, the Goto option can be used instead. This gives you the opportunity to use the LoadFile function to check, whether a file exists or not.

Note: This option was only introduced for reasons of compatibility with OPUS-OS/2 Macro. We highly recommend to use the `IT(MACROERROR, .EQ., TRUE);` statement for error checking instead.

The LoadFile function can be used to load more than one file at the same time. You only need to use wildcard characters (*, ?) in the file name. To process all selected files, a StartLoop statement must follow the LoadFile command line, which uses the name of the FILE variable as loop count.

Example:

```
<File> = LoadFile (’D:\OPUS\DATA\SEARCH*.0’, WARN-
ING  | ABORT);
StartLoop (<File>, 0);
....
EndLoop,0);
```

All files beginning with the name SEARCH are loaded and processed in the loop following the LoadFile instruction.

If wildcards are used, then LoadFile first loads all files into OPUS and processes them in the following loop. Loading the files can become time consuming with an increasing number of files. In this case it is preferable to load the files via the standard OPUS *Load* function, as shown in the following example.

Example:

```
<Name> = ScanPath (’D:\OPUS\DATA\SEARCH*.0’);
<Counter> = GetArrayCount ();
StartLoop (<Counter, 0);
[<File>:AB] = Load (0, {DAP=’D:\OPUS\DATA’,
DAF=<Name>
[<Index>]});
....
<Index> = <Index> + 1
Unload ([<File>:AB]);
EndLoop (0);
```

9.11.2 ScanPath

Scans the path for the specified files and writes each file name into an array element of the variable. Wildcard characters should be used; otherwise only a single file will be found. To process all files in a directory, use *.* as the file name.

Syntax:

<Variable> = ScanPath (‘File’);

<Variable>	variable (array) receiving the files found in the path.
File	drive, path and name of the files to be searched for.

9.11.3 Copy

Copies one or more files.

The Copy command also allows to change the file name while copying. Wildcard characters in the file name may be used.

Syntax:

Copy (‘Source’, ‘Destination’);

Source	drive, path and name of file(s) to be copied.
File	drive, path and name of the destination file(s).

Example

```
Copy (‘C:\DATA\TEST*.0’, ‘D:\DATA\TEST*.1’);
```

9.11.4 Rename

Renames one or more files. Files can also be moved to another directory.

Wildcard characters in the file names are allowed.

Syntax:

Rename (‘Source’, ‘Destination’);

Source drive, path and name of file(s) to be copied.

Destination drive, path and name of the destination file(s).

Example

```
Rename (‘C:\DATA\TEST*.*’, ‘C:\DATA\XYZ*.*’);
```

9.11.5 Delete

Deletes one or more files.

Wildcard characters in the file name are allowed.

Syntax:

Delete (‘File’);

File drive, path and name of file(s) to be deleted.

Example

```
Delete (‘C:\DATA\TEST*.*’);
```

9.12 Time Control Functions

Time control functions can be used to control the timing within a macro. Time intervals as well as computer system time can be used.

GetTime	gets system date and time
Timer	instruction to achieve time control within a macro

9.12.1 GetTime

Gets system time and date.

This function gets the current system time and date of the computer. It returns the (numeric) value for the year, month, day, hour, minute and second. All six variables must be specified in the argument list of the command and all must have been previously declared in the VARIABLES section. Arguments which are not required can be replaced by zeros, e.g. `GetTime (0, 0, 0, <Hour>, <Minute>, <Second>);`

Syntax:

GetTime (<Year>, <Month>, <Day>, <Hour>, <Minute>, <Second>);

<Year>,<Month>....	Variables to receive the specified value (year, month, etc.).

The variable type must be NUMERIC. You must use format instructions to convert the floating-point numbers to integers. Later you can change the format to INTEGER, using format instructions.

9.12.2 Timer

Instruction to control the time behavior within a macro.

Syntax:

Timer (Option, Time);

Option	specifies the behavior of the timer.
WAITTIME	waits for the specified time interval.
WAITUNTIL	waits until the specified time of day is reached (only use HH:MM:SS format for time).
Time	time can be specified as single number which is interpreted as seconds or in HH:MM:SS format (HH = hours, MM = minutes, SS = seconds).

Note: You can also use the IF statement to control the time behaviour within a macro.

9.13 Display Functions

Display functions are used to show or hide spectra and to access the display limits

OpenDisplayWindow	opens a new window for all result files
DisplaySpectrum	shows a spectrum on screen
UnDisplaySpectrum	hides a spectrum
GetDisplayLimits	retrieves the current display limits of the macro display window

SetDisplayLimits	sets the display limits for the current macro display win dow

9.13.1 OpenDisplayWindow

Opens a display window, containing all files newly created by the macro.

Commands that only modify an existing file (like “Peak Pick”) will not be affected by the OpenDisplayWindow command. Changes made by these commands will be displayed in the display window, where the original data was shown.

Syntax:

OpenDisplayWindow ();

This command requires no additional parameters. However, if used in a macro, it should be the first command.

9.13.2 CloseDisplayWindow

Closes a display window, which has been opened with OpenDisplayWindow.

Commands that only modify an existing file (like “Peak Pick”) will not be affected by the OpenDisplayWindow command. Changes made by these commands will be displayed in the display window, where the original data was shown.

Syntax:

CloseDisplayWindow ();

This command requires no additional parameters.

9.13.3 DisplaySpectrum

Display the specified spectrum.

Syntax:

DisplaySpectrum ([<File>:BlockID], Option);

<File>	variable name of file to be displayed.
BlockID	name of the data block to be displayed.
Option	keyword for display scaling.

NOAUTOSCALE	keep the current display limits.

SCALE_SELECTED autoscale to the selected file.

SCALE_ALL autoscale to all spectra in the window.

9.13.4 UnDisplaySpectrum

Removes the specified spectrum from the display. The file remains loaded and can be displayed again using the DisplaySpectrum command.

Syntax:

UnDisplaySpectrum ([<File>:BlockID]);

<File> variable name of file to be hidden.

BlockID name of the data block to be hidden.

9.13.5 GetDisplayLimits

Retrieves the current display limits of the display window created with the OpenDisplayCommand and saves them in the functions' variables. If there was no display window created, the active window will be taken instead.

Syntax:

GetDisplayLimits (<X-Start>, <X-End>, <Y-Min>, <Y-Max>;

<X-Start>.... variables used to save the values determined by the command.

All four variables must be specified and be of the type NUMERIC.

9.13.6 SetDisplayLimits

Sets the limits of the display window created with the OpenDisplayCommand to the values specified. If there was no display window created the active window will be taken instead.

Syntax:

SetDisplayLimits (<X-Start>, <X-End>, <Y-Min>, <Y-Max>);

<X-Start>.... variables used to specify the display values.

All four variables must be specified and be of the type NUMERIC.

9.13.7 SetColor

Sets the display color of the specified spectrum.

Syntax:

SetColor (<File>, <Color>);

<File>	name of file variable.
Color	keyword for display scaling.

BEIGE

BLACK

BLACK

BLUE

CYAN

CORAL

GREEN

GRAY

LIME

MAGENTA

MAROON

MIDNIGHT

OLIVE

PURPLE

RED

SEAGREEN

SKY

TEAL

VIOLET

YELLOW

10 OPUS Command Reference

The OPUS commands accessible from the OPUS pull-down menus call OPUS processing functions, that in turn perform the desired manipulation. These OPUS processing function can be included in macros, scripts and external programs. Alternatively to launching a function via the OPUS pull-down menu command, the function name can be typed in the OPUS command line.

10.1 Command Syntax of OPUS Functions

Syntax:

CommandName (Input List 1, ..., Input List n, {PAR 1=Value 1, ..., PAR n=Value n});

CommandName	name of the OPUS command.
Input List n	list of input files (see below).
PAR n	three letter parameter name n.
Value n	value for parameter n.

Syntax for file list:

([<File 1>:BlockID 1] ... [<File n>:BlockID n]

<File n>	name of input file or the file variable n.
BlockID n	name of the data block of file n.

Note that the files in a list are separated by blanks, while the lists themselves are separated by commas. Most functions require only one file list; a few files however, (like *Make Compatible* or *Subtraction*) need several file lists.

10.2 Including OPUS Commands in Macros

We strongly recommend to only use the Macro Editor, if you want to include OPUS commands into macros. Using the Macro Editor guarantees that all relevant parameters required by the command are inlcuded in the command line and PARAMETER section. Furthermore, it is ensured that these parameters are initialized with valid values.

To append an OPUS command to a macro, simply select the command from the OPUS pull-down menu, while the Macro Editor is running. Choose the appro-propriate parameters, files and settings as usual in the dialog box of the command.

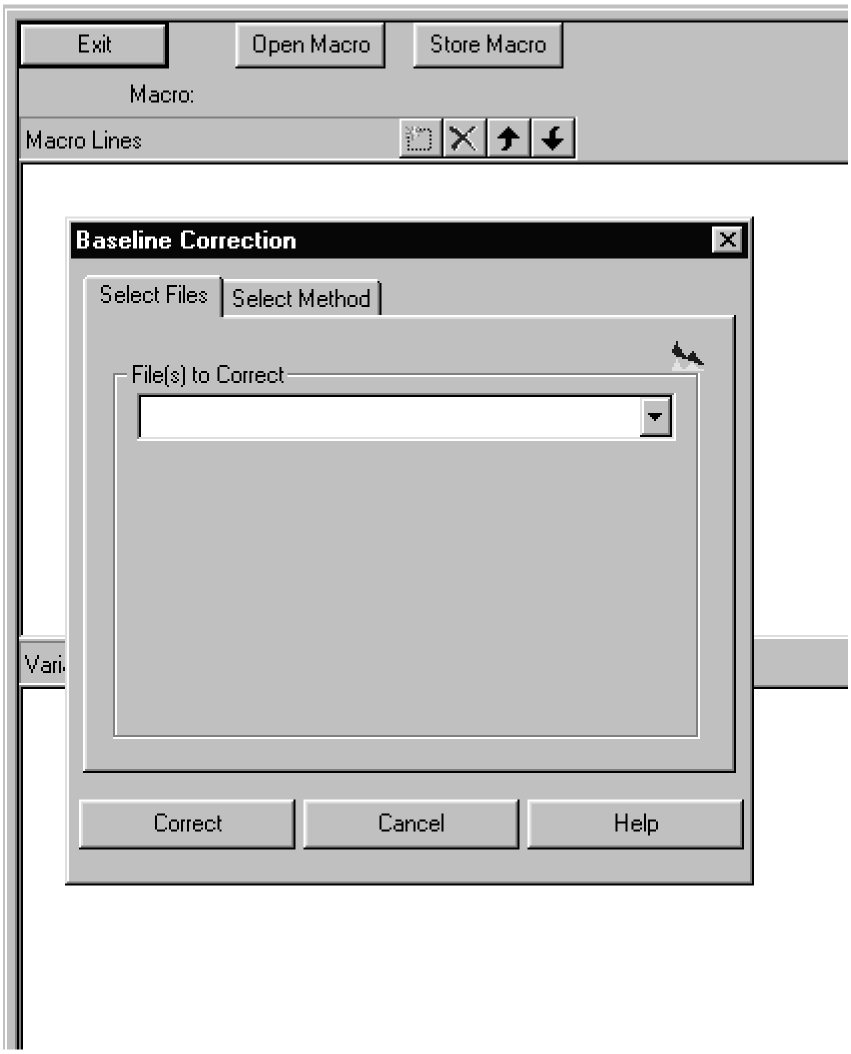


Figure 61: Including an OPUS Command

After clicking on the *Execution* button in the dialog box for executing the command, the OPUS command dialog box is replaced by a parameter dialog, that lists all parameters relevant for the processing function. When you click the *OK* button, the respective OPUS processing function will be appended to the macro.

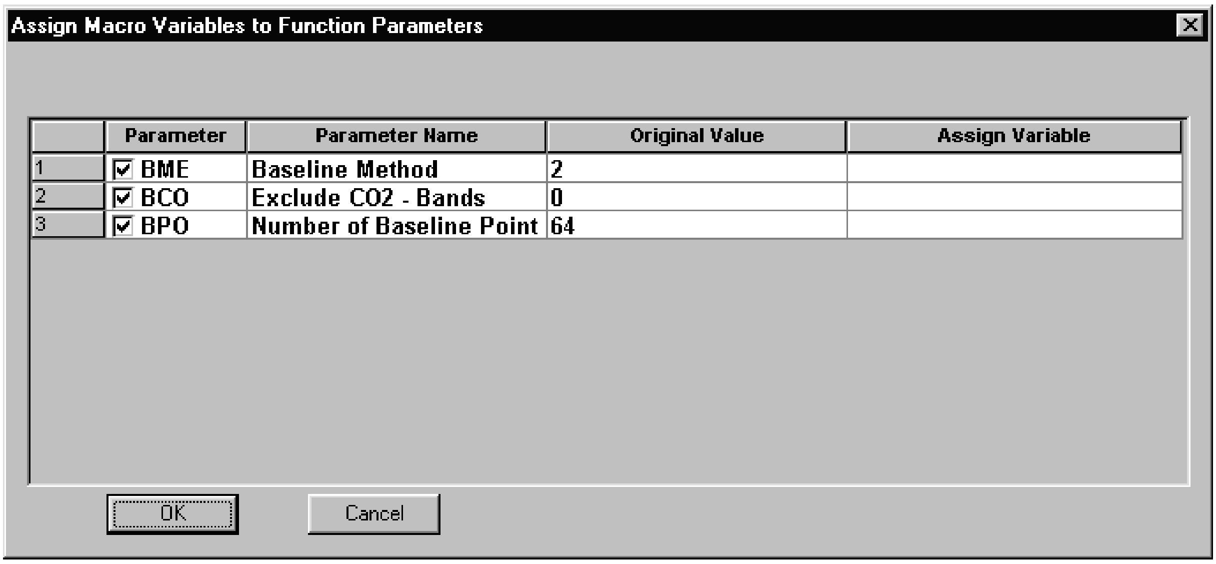


Figure 62: Including an OPUS Command – Parameter Dialgo Box

Column 1: Abbreviation of the parameter and check box

Column 2: Parameter name

Column 3: Parameter value as set in the OPUS command dialog box

Column 4: Assigned macro variable

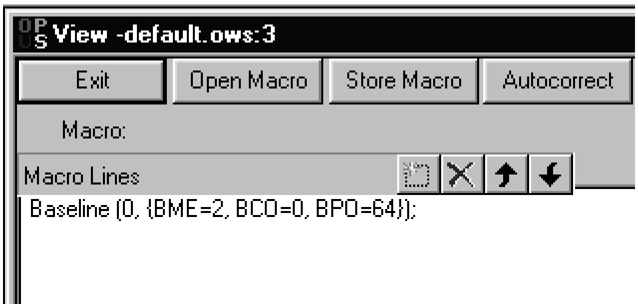


Figure 63: Resulting Command Line

Whether a parameter will be appended to the command line or included in the PARAMETER section is controlled by the check box. If the check box is not selected, a parameter entry will be made in the PARAMETER section. We recommend to always include all parameters in the command line. This ensures, that the commands are using correct parameter values at the time of command execution, in case a command or a goup of commands accessing the same parameter is repetedly used. A parameter may only appear once in the PARAMETER section and therefore, the parameter can only have one value. The only exception to this rule are the measurement commands, explained in detail in the following chapters.

Note: parameters, that have been assigned macro variables must appear in the command line!

A combobox is displayed above the parameter list of OPUS commands, which return a result or a file to the macro. From this box, you have to indicate the variable supposed to hold the returned data. Although the OPUS command will be processed correctly by the macro even if no variable was chosen, the returned data then is not accessible.

10.3 Measurement Commands

As already mentioned, the measurement commands differ from the rest of the OPUS commands. When you include the *Measurement* command in a macro, you will find that only two parameters XPP and EXP are selected by default. XPP represents the directory of the experiment file and EXP the name of the experiment. It is highly recommended to assign macro variables to these parameters. This guarantees, that a measurement started from a macro always uses an existing experiment file (and therefore a defined parameter set). For measurement functions, the remaining parameters won't be included in the PARAMETER section!

Other parameters than XPP and EXP should only be selected, if they are intended to replace values stored in the experiment file or if macro variables should be assigned to these parameters. This will become clear, if one looks at the sequence in which a measurement command is executed.

Measurement without Using an Experiment File (not recommended)

1) The measurement primarily uses the values entered in the PARAMETER section, if anything.
2) Parameter included in the command line override the values declared in the PARAMETER section.

Example:

```
[<File>] = MeasureSample (0, {NSS = 16});
```

Regardless of the original settings the measurement will now run 16 scans.

Measurement Using an Experiment File (XPP and EXP Selected)

The parameters of the PARAMETER section are ignored, and the parameters stored in the experiment file will be used instead. Again, parameters included in the command line override the values stored in the experiment file.

Example:

```
[<File>] = MeasureSample (0, {XPP = '<XMP Path>', EXP =
'default', NSS = 16});
```

Regardless of the settings stored in the experiment file, the measurement will now run 16 scans.

10.4 Reference Section

The following section describes the OPUS commands in detail. The sections are all structured in the same way. You will find:

- the title, which consists of the OPUS command referenced in this section.
- a summary of the command.
- an indication, whether the command modifies files or not.
- an explanation of the syntax.
- a table, listing all command parameters and their function.
- a note, if the command has not been implemented in OPUS to this point in time.

All of the parameters you will find listed in the tables are required, and must be stated as a part of the command. A parameter statement should therefore be included either in the parameter list of the command or in the PARAMETER section of the macro. If no parameter statement was made in a macro, OPUS will use the parameters of the active parameter set, when executing the macro. This usually leads to unpredictable results.

10.5 OPUS Functions Sorted Alphabetically

A

ABTR	absorbance transmittance conversion
Average	averages spectra

B

Baseline	performs a baseline correction of a spectrum
BlackBody	Black Body generation

C

ChangeDataBlockType	changes the type of a data block
Convert	converts spectra
CopyDataBlock	copies a data block from one file to another
Cut	cuts a frequency range out of a spectrum

D

Deconvolution	Fourier self deconvolution
DeleteDataBlock	deletes the specified data block
Derivative	calculates the derivative

E

ExternalProgram	starts an external program
Extrapolation	extrapolates spectra

F

FFT	Fast Fourier transformation
FreqCalibration	frequency calibration

I

InfoInput	adds an information block to a file
Integrate	integrates a spectrum
InverseFT	performs an inverse Fourier transformation

J

JCAMPToOPUS	converts a JCAMP-DX file to OPUS format

K

KramersKronig	performs a Kramers Kronig transformation

M

MakeCompatible	makes spectra compatible
MeasureReference	measures a background spectrum
Merge	merges spectra

N

Normalize	normalizes a spectrum

P

PeakPick	creates a peak table
Plot	plots spectra
PostFTZerofill	Post Zerofilling of a spectrum

R

RamanCorrection	applies Raman correction
Restore	restores an original data file

S

Save	saves a spectrum file
SendFile	sends a file via e-mail
SignalToNoise	calculates the Signal-to-Noise ratio
Smooth	smooths a spectrum
StraightLine	inserts a straight line in a spectrum
Subtract	subtracts one or more spectra from another spectrum

U

Unload	removes a spectrum from the Browser

10.6 OPUS Functions Sorted by Type

Manipulation Functions

ABTR	absorbance transmittance conversion
Average	averages spectra
Baseline	performs a baseline correction of a spectrum
BlackBody	Black Body generation
Convert	converts spectra
Cut	cuts a frequency range out of a spectrum
Deconvolution	Fourier self deconvolution
Derivative	calculates the derivative
Extrapolation	extrapolates spectra
FFT	Fast Fourier transformation
FreqCalibration	frequency calibration
InverseFT	performs an inverse Fourier transformation
KramersKronig	performs a Kramers Kronig transformation
MakeCompatible	makes spectra compatible
Merge	merges spectra
Normalize	normalizes a spectrum
PostFTZerofill	Post Zerofilling of a spectrum
RamanCorrection	applies Raman correction
Smooth	smooths a spectrum
StraightLine	inserts a straight line in a spectrum
Subtract	subtracts one or more spectra from another spectrum

Evaluation Functions

Integrate	integrates a spectrum
PeakPick	creates a peak table
SignalToNoise	calculates the Signal-to-Noise ratio

File Functions

ChangeDataBlockType	changes the type of a data block
CopyDataBlock	copies a data block from one file to another
DeleteDataBlock	deletes the specified data block
Restore	restores an original data file
Save	saves a spectrum file
SendFile	sends a file via e-mail
Unload	removes a spectrum from the Browser

Measurement Functions

MeasureReference	measures a background spectrum
SendCommand	send an optics command to the optics bench
SaveReference	saves a reference spectrum from the AQP to disk
LoadReference	loads a reference spectrum from the disk into the AQP

Library Functions

LibrarySearchInfo	information search in library files
LibrarySearchPeak	peak search in library files
LibrarySearchStructure	structure search in library files
LibrarySearchSpectrum	spectrum search in library files
LibraryInitialize	creates a new, empty library file
LibraryStore	stores a new entry in a library file or replaces an existing one
LibraryEdit	edits an entry, the library description, and the definition of information stored in a library file.
InfoInput	adds an information block to a file or edits an existing one

Miscellaneous Functions

ExternalProgram	starts and external program
Plot	plots spectra

10.7 OPUS Manipulation Functions

10.7.1 ABTR

Absorbance → Transmittance conversion.

This functions modifies the selected spectrum and changes the data block type accordingly.

ABTR ([<File>:BlockID] ...., {...});

Parameter	Value	Description
CCM	1	automatic
	2	AB → TR
	3	TR → AB

10.7.2 Average

Averages spectra.

This command requires three file lists:

File List 1: Spectra to be averaged.

File List 2: (optional) File to store the average result.

File List 3: (optional) File to store the standard deviation result.

If File List 2 and/or 3 are not specified, they have to be set to “0”.

Average ([<File 1>:BlockID 1], [<File 2>:BlockID 2], [<File 3>:BlockID 3], {...});;

Parameter	Value	Description
QA0	0	Do not average with number of scans
	1	Average with number of scans
QA2	0	Don’t create average report
	1	Create average report
QAE	NO	Don’t create standard deviation spectrum
	YES	Create standard deviation spectrum
QAF	NO	Don’t update standard deviation spectrum
	YES	Update standard deviation spectrum
QAL	LIS	Average selected files
	FIL	Average files selected by name and path
QAM	Text	Path of the files to be averaged
QAN	Text	Name of the files to be averaged
QAO	Numerisch	BlockID of the files to be averaged
QFB	Text	Path of the IDENT method
QFC	Text	Name of the IDENT method

10.7.3 Baseline

Performs a baseline correction of a spectrum.

This command modifies the selected spectrum.

Baseline ([<File>:BlockID] ...., {...});

Parameter	Value	Description
BME	1	Rubber Band correction
	2	Scattering correction
BPO	10 ... 200	number of baseline points
BCO	0	include CO_2 bands
	1	exclude CO_2 bands

10.7.4 BlackBody

Calculates a spectrum of a Black Body radiator.

This function adds a single channel sample data block to the selected file(s).

BlackBody ([<File>:BlockID] ...., {...});

Parameter	Value	Description
QTE	pos. number	temperature of the Black Body radiator
QPM	0	energy
	1	photons

10.7.5 Convert

Converts spectra.

This functions modifies the selected spectrum and changes the data block type accordingly.

Convert ([<File>:BlockID] ...., {...});

Parameter	Value	Description
CSD	1	AB, TR, Refl
	2	KM → Refl
	3	AB, TR → ATR
	4	ATR → AB
	5	Refl → lgRefl
	6	lgRefl → Refl
	7	ScSm → Raman
	8	Raman → ScSm

10.7.6 Cut

Cuts out a frequency range of a spectrum file.

This functions modifies the selected spectrum file.

Cut ([<File>:BlockID] ...., {...});

Parameter	Value	Description
CFX	number	X-start frequency
CLX	number	X-end frequency

10.7.7 Deconvolution

Performs a Fourier self deconvolution.

This functions modifies the selected spectrum.

Deconvolution ([<File>:BlockID] ...., {...});

Parameter	Value	Description
DSP		peak form
	LO	Lorentzian
	GA	Gaussian
DEF	pos. number	deconvolution factor
DNR	pos. number	noise reduction factor
DES	number	X-start frequency
DEE	number	X-end frequency
DWR	0	frequency limits
	1	file limits

10.7.8 Derivative

Calculates the derivative of a spectrum.

This functions appends a new data block, containing the derivative of the spectrum, to the original data.

Derivative ([<File>:BlockID] ...., {...});

Parameter	**Value**	**Description**
QSP	5, 9, 13, 17, 21, 25	number of smoothing points
QOD	1...5	order of derivative

10.7.9 Extrapolation

Extrapolates a spectrum.

This functions modifies the selected spectrum.

Extrapolation ([<File>:BlockID] ...., {...});

Parameter	**Value**	**Function**
QX0	number	extrapolate to zero
QX1	number	extrapolate to infinity
QX2	number	lower frequency limit
QX3	number	upper frequency limit
QX4	number	new end frequency

10.7.10 FFT

Performs a Fast Fourier transformation..

This command performs a fast Fourier transformation of an interferogram. The result is a single channel spectrum data block, which will be added to the file.

FFT ([<File>:BlockID] ...., {...});

Parameter	Value	Description
FTS	number	start frequency of the spectrum
FTE	number	end frequency of the spectrum
FZF	pos. number	Zerofilling factor
FTR	pos. number	resolution
FHR	pos. number	phase resolution
FBW		bit code used to indicate forward/backward or even/odd
	1	forward interferogram
	2	backward interferogram
	8	even separation
	16	odd separation
FTA		apodization function
	BX	Boxcar
	TR	Triangular
	4P	Four Point
	HG	Happ-Genzel
	B3	Blackman-Harris 3-term
	B4	Blackman-Harris 4-term
	NBW	Norton-Beer, weak
	NBM	Norton-Beer, medium
	NBS	Norton-Beer, strong
FLR	pos. number	limit resolution
FHZ		phase correction
	ML	Mertz
	SM	Signed Mertz
	PW	Power spectrum
	MS	Mertz stored phase
	NO	No – save complex data
FZF	pos. number	Zerofilling factor
FNL	0	no nonlinearity correction
	1	nonlinearity correction
FNC	pos. number	nonlinearity correction – detector cutoff
FNE	pos. number	nonlinearity correction – mod. efficiency
FSM		ZPD search mode
	AL	largest absolute value

	MN	minimum
	MX	maximum
	MI	mid position between min and max
	NO	use stored value
	MA	manual input
FPP	pos. number	peak position
FSR	pos. number	search range
FSY		symmetry for search range
	0	symmetrical
	1	antisymmetrical
	2	automatic
FTT		to do list — bit list for result data blocks
	1	absorbance
	2	interferogram
	4	single channel
	8	power spectrum
	16	phase spectrum
	64	single channel (real)
	128	single channel (imaginary)

10.7.11 FreqCalibration

Performs a frequency calibration.

This functions modifies the selected spectrum.

FreqCalibration ([<File>:BlockID] ...., {...});

Parameter	Value	Description
QF0	NO	do not restore original values
	YES	restore original values
MWC	number	factor
AWC	number	offset

10.7.12 InverseFT

Performs an inverse Fourier transformation.

This command performs an inverse Fourier transformation of a spectrum. The result is a single channel spectrum data block, which will be added to the file.

InverseFT ([<File>:BlockID] ...., {...});

Parameter	Value	Description
RSY		symmetry
	0	symmetric
	1	Antisymmetric
RXS	number	X-start frequency
RXE	number	X-end frequency
RWR	0	frequency limits used
	1	file limits used

10.7.13 KramersKronig

Performs a Kramers Kronig transformation.

This command performs a Kramers-Kronig transformation of a reflectance spectrum. The real and imaginary part of an absorbance-like spectrum will be calculated. The result is a single channel spectrum data block which will be added to the file.

KramersKronig ([<File>:BlockID] ...., {...});

Parameter	Value	Description
KKR		desired result
	0	refractive index (complex)
	1	absorbance
	2	dielectric function (complex)
	3	phase
KKS	number	X-start frequency
KKE	number	X-end frequency
KWR	0	use specified frequency limits
	1	use file limits

10.7.14 MakeCompatible

Makes spectra compatible.

This functions interpolates the selected spectrum to the frequency limits and point raster of a reference spectrum.

This functions modifies the selected spectrum and changes the data block type accordingly. The reference spectrum remains unchanged.

MakeCompatible ([<File1>:BlockID1], ([<File2>:BlockID2] ...., {...});

<File1> reference file.

<File2> file to be interpolated.

Parameter	Value	Description
CME		interpolation method
	2	interpolation
	3	reduce resolution

10.7.15 Merge

This function has not been implemented yet.

Merges spectra.

Merge ([<File>:BlockID] ...., {...});

10.7.16 Normalize

Normalizes a spectrum.

This functions modifies the selected spectrum.

Normalize ([<File>:BlockID] ...., {...});

Parameter	Value	Description
NME	1	min-max normalization
	2	vector normalization
	3	offset correction
NWR	0	use specified frequency limits
	1	use file limits
NFX	number	X-start frequency
NLX	number	X-end frequency

10.7.17 PostFTZerofill

Performs a post Zerofilling of a spectrum.

This functions modifies the selected spectrum.

PostFTZerofill ([<File>:BlockID] ...., {...});

Parameter	Value	Description
PZF	pos. number	Zerofilling Factor
PZS	number	X-start frequency
PZE	number	X-end frequency
PWR		frequency limits
	0	use specified frequency limits
	1	use file limits

10.7.18 RamanCorrection

Performs a Raman correction.

This functions modifies the selected spectrum.

RamanCorrection ([<File>:BlockID] ...., {...});

Parameter	Value	Description
QC0		background correction
	0	do not perform correction
	1	perform correction
QC1		scatter correction
	0	do not perform correction
	1	perform correction
QC2		restore original data
	0	do not perform correction
	1	perform correction
QC3	Text	path for white light source spectrum
QC4	Text	name of white light source spectrum
QC5	pos. number	reference temperature

10.7.19 Smooth

Smoothes a spectrum.

This functions modifies the selected spectrum.

Smooth ([<File>:BlockID] ...., {...});

Parameter	Value	Description
QSP	5, 9, 13, 17, 21, 25	number of smoothing points

10.7.20 StraightLine

Generates a straight line.

This functions modifies the selected spectrum.

StraightLine ([<File>:BlockID] ...., {...});

Parameter	Value	Description
GFX	number	X-start frequency
GLX	number	X-end frequency

10.7.21 Subtract

Subtracts one or more spectra from another spectrum.

The spectrum from which the others are subtracted is modified. The spectrum/ spectra which are subtracted stay unchanged.

Subtract ([<File A>:BlockIDA], ([<File B>:BlockIDB] ...., {...});

<File A> file to be subtracted from, this file is modified.

<File B> file(s) which are subtracted from <FileA>**.**

Parameter	Value	Description
SUB		subtraction mode
	1	interactive
	3	autosubtraction
	4	use whole range
SUN		number of spectra
SX1		X-start frequency
SX2		X-end frequency

10.8 OPUS Evaluation Functions

10.8.1 Integrate

Integrates a spectrum.

This function adds an integration report to the file.

Integrate ([<File>:BlockID] ...., {...});

Parameter	Value	Description
LPT	text	path for integration method
LFN	text	file name of the integration method
LRM		report mode
	0	overwrite old integration report
	1	merge integration reports
	2	append integration report

10.8.2 PeakPick

Creates a peak table.

This function adds a peak table data block to the file.

PeakPick ([<File>:BlockID] ...., {...});

Parameter	Value	Description
PSM		peak mode
	1	standard peak pick
	2	2. derivative
NSP	5, 9, 13, 17, 21, 25	number of points used for 2. derivative
WHR		frequency limits
	0	use specified frequency limits
	1	use file limits
LXP	number	start frequency
FXP	number	end frequency
PPM		peak definition
	1	autodetect (min or max)
	2	find maximum

	3	find minimum
PTR	pos. number	find peaks > value (absolute)
QP0		decimals
	YES	digits after decimal, user-defined
	NO	digits after decimal, not defined by user
QP3	pos. integer	digits after decimal
QP4		peak limits (%)
	YES	use peak limits
	NO	ignore peak limits
QP5	pos. integer	find peaks < value (%)
QP6		upper absolute peak limit
	YES	use upper absolute peak limit
	NO	ignore upper absolute peak limit
QP7	pos. integer	find peaks < value (absolute)
QP8		lower absolute peak limit
	YES	use lower absolute peak limit
	NO	ignore lower absolute peak limit

10.8.3 SignalToNoise

Calculates the Signal-to-Noise ratio.

This function adds parameters to the data parameter block of the selected spectrum.

SignalToNoise ([<File>:BlockID] ...., {...});

Parameter	**Value**	**Description**
NF1	number	start frequency
NF2	number	end frequency
SN1	number	S/N (RMS)
SN2	number	S/N (peak to peak)
SN3	number	maximum ordinate in S/N region
SN4	number	minimum ordinate in S/N region
SNF		flags

10.9 OPUS File Functions

10.9.1 ChangeDataBlockType

This function has not been implemented yet.

Change the data block type.

This functions does not modify the specified data block, only the block ID is changed.

ChangeDataBlockType ([<File>:BlockID] ...., {...});

10.9.2 CopyDataBlock

This function has not been implemented yet.

Copies a data block from one file to another.

This function adds the specified data block to the selected file in file list B.

CopyDataBlock ([<File A>:BlockID], ([<File B>] ...., {...});

<file A>	source file.
blockID	name of the data block to copy.
<file B>	destination file.

10.9.3 DeleteDataBlock

Deletes the specified data block.

The specified block is removed from the file.

DeleteDataBlock ([<File>:BlockID] ...., {...});

10.9.4 Restore

Restores original File.

This function restores the original file and discards all changes made so far. All changes are lost if the results had not been saved before.

Restore([<File>:BlockID] ...., {});

The function does not require any parameters.

10.9.5 Save, SaveAs

Saves a spectrum file.

This function stores the eventually modified file to disk.

Save ([<File>:BlockID] ...., {...});

Save As ([<File>:BlockID] ...., {...});

Parameter	**Value**	**Description**
OEX		overwrite mode
	0	increment file name
	1	overwrite file
SAN		file name
DAP		target directory
COF		bit combination for save mode
	2	save all data blocks
	4	move file
	16	remove copies
	32	save as JCAMP.dx file
	64	save as x,y table
	128	replace original data
	256	save as Galactics GRAMS file
	512	unload file after saving it
	1024	save as Pirouette file
The following parameters will only be used when saving a file as an x,y table.		
DPA	pos. number	number of decimals, abscissa
DPO	pos. number	number of decimals, ordinate
SEP	character	separator
YON		Y-values
	1	Y-Values only
	0	X and Y-Values
ADP		data points
	1	use all data points
	0	do not use all data points

10.9.6 SendFile

Sends a file via e-mail.

This function does not modify the specified file.

SendFile ([<File>:BlockID] ...., {...});

Parameter	Value	Description
COF		data blocks
	0	send only specified block
	2	send all blocks

10.9.7 Unload

Removes a spectrum from Browser.

This function removes the specified file from the OPUS file list. The file is no longer accessible from the macro.

Unload ([<File>:BlockID] ...., {...});

The function does not require any parameters.

10.10 OPUS Measurement Functions

We strongly recommend to set the measurement parameters for a macro using an experiment file. Most of the parameter are linked and checked for consistency before starting an acquisition. Therefore, an inconsistent or wrong parameter set will most likely not be able to start an acquisition, and can be recognized easily. Only a few of the parameters listed below can be set without any problems either manually or by using variables.

10.10.1 Measurement Commands

The measurement commands always use the same parameters. You should only use the parameters listed here.

1) Measure Reference: ***MeasureReference ({...})***; acquires a background spectrum.
2) Measure Sample: ***<File> = MeasureSample ({...})***; acquires a sample spectrum.
3) Measure Repeated: ***<File> = MeasureRepeated ({...})***; acquires a set of sample spectra.

4) Measure Rapid TRS: ***<File> = MeasureRapidTRS ({...})***; performs a rapid scan acquisition.
5) Measure Step Scan Trans: ***<File> = MeasureStepScanTrans ({...});*** performs a Step Scan acquisition, using a transient recorder.

Parameter	**Value**	**Description**
SNM	text	sample name
SFM	text	sample preparation
CNM	text	operator
XPM	text	experiment file name
XPP	text	path for experiment file
RES	pos. number	resolution
NSS	pos. number	number of scans

10.10.2 SendCommand

Sends an optics command to the optics bench.

This function does not need an input spectrum.

SendFile (0, {...});

Parameter	**Value**	**Description**
UNI	text	text to be sent

10.10.3 SaveReference

Saves a reference spectrum from the AQP to disk.

This function creates a new file.

SaveReference (0, {...});

The function does not require any parameters.

10.10.4 LoadReference

Loads a reference spectrum from disk into the AQP.

This function does not modify the spectrum.

LoadReference ([<File>:ScRf], {...});

The function does not require any parameters.

10.11 OPUS Library Functions

10.11.1 LibrarySearchInfo

Searches for information in a spectrum library.

This function performs a query for information within a spectrum library. The query text must be supplied in a query file (extension .INL); use the OPUS-NT *Information Search* dialog to create and save a query file.

[<File1>:BlockID] = LibrarySearchInfo (0, {...});

Standard search.

<File 1> Contains the search result.

[<File1>:BlockID] = LibrarySearchInfo ([<File2>:BlockID], {...});

Query using an existsing search report.

<File 1> Contains the search result.

<File 2> Contains the search report**.**

Parameter	Value	Description	Remarks
SIH	NUMERIC	maximum number of hits	must be > 0
SIN	STRING	name of the information query file	file name including extension (.INL)
SIP	STRING	path of the query file	path without teminating “\”
LB1	STRING	list of library files to be searched	names of the library files. They must be stated including drive and path but without extension. Separate multiple file names with “@”.

10.11.2 LibrarySearchPeak

Searches for peaks in a spectrum library.

This function performs a query for peaks within a spectrum library. The query R must be supplied in a query file (extension .PKL); use the OPUS-NT *Peak Search* dialog to create and save such a query file.

[<File1>:BlockID] = LibrarySearchPeak (0, {...});

Standard search.

<File 1> Contains the search result.

[<File1>:BlockID] = LibrarySearchPeak ([<File2>:BlockID], {...});

Query using a search report.

<File 1> Contains the search result.

<File 2> Contains the search report.

Parameter	Value	Description	Remarks
SPQ	NUMERIC	minimum Hit quality	Range between 1 and 1000 Will only be used in combination with the *Calculate Hit Quality* algorithm.
SPH	NUMERIC	maximum Hit number	must be > 0
SPA	NUMERIC	search algorithm	
	512	hit if one peak matches	
	1024	hit if all peaks match	
	2048	calculate hit quality	
	4096	count matching peaks	
PNP	STRING	name of the peak query file	file name including extension (.PKL)
PPP	STRING	path of the peak query file	path without teminating “\”
LB1	STRING	list of library files to be searched	names of the library files. They must be stated including drive and path but without extension. Separate multiple file names with “@”.

10.11.3 LibrarySearchStructure

Searches for chemical structures in a spectrum library.

This function performs a query for chemical structures within a library file. The query must be supplied in a structure data block.

LibrarySendStructure([<File1>:BlockID], 0, {...});

Standard search.

<File 1> Contains the query structure.

LibrarySearchStructure ([<File1>:BlockID], [<File1>:BlockID], {...});

Query using an existing search report. The result will be appended to the file containing the structure block.

<File 1> Contains the query structure.

<File 2> Contains the search report**.**

Parameter	**Value**	**Description**	**Remarks**
STH	NUMERIC	maximum number of Hits	must be > 0
LAL	NUMERIC	search algorithm	
	8192	match exact	
	12288	match embedded	
LB1	STRING	list of library files to be searched	names of the library files. They must be stated including drive and path but without extension. Separate multiple file names with “@”.

10.11.4 LibrarySearchSpectrum

Searches for spectra in a spectrum library.

This function performs a query for peaks within a spectrum library. The query spectrum must be absorbance-like.

LibrarySearchSpectrum ([<File1>:BlockID], 0, {...});

Standard search.

<File 1> The query spectrum.

LibrarySearchSpectrum ([<File1>:BlockID], [<File2>:BlockID], {...});

Query using a search report.

<File 1> The query spectrum.

<File 2> Contains the search report.

Parameter	Value	Description	Remarks
LSS	NUMERIC	sensitivity	Range between 1 and 20 Will only be used in combination with the *Standard* algorithm.
SSQ	NUMERIC	minimum Hit quality	Range between 1 and 1000
SSH	NUMERIC	maximum number of Hits	must be > 0
SS1	NUMERIC	search algorithm	When using spectrum correlation algorithms, the value will always be the sum of three options.
	1	standard	
	2	standard, use existing peak table	
	4	spectrum correlation	
	+16	no derivative	one of the three derivatization types must be added to the base value.
	+32	first derivative	
	+64	second derivative	
	+128	vector normalization	one of the two normalization types must be added to the base value.
	+256	min-max normalization	
LB1	STRING	list of library files to be searched	names of the library files. They must be stated including drive and path but without extension. Separate multiple file names with “@”.

10.11.5 LibraryInitialize

Creates a new, empty library.

A method file (extension .MTD) and a text file (extension .TXD) is needed to create a library file.

LibraryInitialize ({...});

The function does not require any parameters.

Parameter	Value	Description	Remarks
LPT	STRING	path of the text definition file	path without teminating “\”
LBT	STRING	name of the text definition file	file name without extension
MTP	STRING	path of the method file	path without teminating “\”
LMT	STRING	name of the method file	file name without extension
LBP	STRING	directory of the new library file	path without teminating “\”
LBN	STRING	name of the library file	file name without extension
LID	STRING	library description	maximum 79 characters
LCP	STRING	copyright	maximum 79 characters

10.11.6 LibraryStore

Stores a new entry, the library description, and the definition of information saved in a library file.

LibraryStore (0, [<File>:BlockID], {...});

The function does not require an input file list.

Parameter	Value	Description	Remarks
LSM	NUMERIC	storage mode	
	1	new entry	
	3	replace entry	
	5	replace info	
	7	insert/replace structure	
LBP	STRING	directory of the library file	path without teminating “\”
LBN	STRING	name of the library file	file name without extension
LBS	NUMERIC	entry number	for all storage modes except “New Entry”.

10.11.7 LibraryEdit

This function loads and deletes entries of a library. Furthermore, the description of the library as well as the description of the stored information can als be edited.

[<File>:BlockID] = LibraryEdit (0, {...});

Syntax to load a spectrum of a library entry.

LibraryEdit (0, {...});

Syntax for any other option

.

Parameter	Value	Description	Remarks
LMO	NUMERIC	edit mode	
	2	load entry	
	5	delete entry	
	13	change information set	
	14	change description	
LBS	NUMERIC	entry number	only required for the “Load Entry” and “Delete Entry” mode.
LBP	STRING	directory of the library file	path without teminating “\”
LBN	STRING	name of the library file	file name without extension
LID	STRING	new information definintion file or library description	only required for the “Chande Info Definition” and “Chande Description” mode. Contains the complete path and name of the new information definition file (extension .TXD) or the description, depending on the mode.

10.11.8 InfoInput

Allows information input.

This function adds an information block to the selected file. Depending on the mode, either the complete info block is replaced, only selected information of an existing info block is replaced, or a new file with an info block will be created.

InfoInput ([<File>:BlockID] ...., {...});

Syntax if a block should be replaced or extended.

[<File>:BlockID] = InfoInput ({...});

Syntax if a new file should be created.

Parameter	Value	Description	
IRM	STRING	information input mode	[a]
	O	the complete info block will be overwritten.	
	R	the complete info block will be replaced.	
	N	generate new info file	
INP	STRING	path of the info definition file	Required for the modes “O” and “N”.
INM	STRING	name of the info definition file	Required for the modes “O” and “N”.
I01	STRING	information of line 1	[b]
I02	STRING	information of line 2	
...			
I99	STRING	information of line 99	
T01	STRING	description of line 1	
T02	STRING	description of line 2	
...			
T99	STRING	description of line 99	

a. If stated, make sure to consider the following points:
   - The parameter IRM is not allowed in the parameter list.
   - Null strings have to be assigned to the parameters INM and INP (e.g. INM = ’’)
   - The paramters Txx have to be specified consecutively, starting with T00. For example, in case of 4 lines, the parameters T00, T01, T02, T03, T04 must be stated.
   - The parameters Ixx responsible for the line content, like all other options, don’t need to be specified consecutively.

b. Specify the text to be entered in the info block using the parameters Ixx. xx represents the line numbers in the info block. You only have to state parameters for the lines in which you wish to enter text. The total number of lines is defined in the info definition file.

10.12 Miscellaneous OPUS Functions

10.12.1 ExternalProgram

Starts an external program.

This function launches an external program, forwards parameters and supplies the means of communication with the external program. DDE connections as well as Named Pipes are supported.

ExternalProgram ([<File>:BlockID] ...., {...});

Parameter	Value	Description
XPF		start as OPUS task
	0	OPUS starts the program, then breaks off all communictation with the external program
	1	program is not detached
XST		type of program start and connection type
	0	start the program; connection via a pipe
	1	don’t start the program; connection via the server pipe
	2	start the program; open a DDE connection
	3	don’t start the program; connection via the server pipe
XPR	Text	name of the program to be launched, includ-ing path
XPA	Text	parameteres to be exchanged
XWI		start 16bit program in its own VDM
	0	use common VDM
	1	extra VDM
XWS		window size at start
	0	normal
	1	maximized
	2	minimized
	3	hidden
XCW		wait for program termination
	0	only start program
	1	wait for result/end

XSB		start in background mode – not supported by Windows NT. Can be replaced by XWS
XEM		OS/2 spezific – no longer supported
XDM		OS/2 spezific – no longer supported
XVP		OS/2 spezific – no longer supported
XPM	<C/S>	OS/2 spezific – no longer supported
DDE		transaction type
	Bit 0 gelöscht	don't send command
	1	poke
	3	execute
	5	request
DDS	Text	DDE server name
DDT	Text	DDE topic
DDI	Text	DDE item
DDD	Text	text-coded binary data

10.12.2 ParameterEditor

Changes the sample parameters.

This function changes the following parameters:

- sample name
- sample form
- user name
- sample number

Note that the statement of all values is required when executing this function. In addition, the axes labels and scaling factors used for the axes can be entered.

ParameterEditor ([<File>:BlockID] ...., {...});

Parameter	Value	Description
CNM	Text	user name
SNM	Text	sample nname
SFM	Text	sample form
RSN	Zahl	sample number
XTX	Text	X-axis label
YTX	Text	Y-axis label
ZTX	Text	Z-axis label
XAF	Number	X-axis scaling factor
YAF	Number	Y-axis scaling factor
ZAF	Number	Z-axis scaling factor

10.12.3 Plot

Plots spectra.

This function does not change the spectrum.

Plot ([<File>:BlockID] ...., {...});;

Parameter	Value	Description
PDV		output device
	Printer	printer
	Clipboard	clipboard
SCP	Text	path of the template used for plotting
SCN	Text	name of the template used for plotting
PUN		devices; currently not evaluated
POP	Text	output path; currently not evaluated
POF	Text	output file; currently not evaluated
PDH		window handle; currently not evaluated
PL2	Number	number of peaks to be labeled
PPA	Text	Codes several parameters in a string that are used for different frames

PPA starts with FRM=n and defines how many frame parameters follow. For each frame the following parameters (separated by commas) are necessary:

Parameter	Value	Description
NPL	Number	number of spectra in the current frame
XSP	Number	X start frequency
XEP	Number	X end frequency
YMN	Number	lowest value of the Y axis
YMX	Number	highest value of the Y axis
ASE	YES/NO	AutoScale the spectrum frame
CWN	YES/NO	use compressed wave numbers
COL	Numbers	colors of each curve, separated by commas

10.12.4 VBScript

Starts a VisualBasic script

This function loads and then runs a VisualBasic script. Parameters and data blocks can be forwarded to the script

VBScript ([<File>:BlockID] ...., {...});

Parameter	Value	Description
VBS	Text	name of the script, including path
VBP	Text	parameters to be forwarded to the script
VBW		wait for termination
	0	immediate return after starting the script
	1	wait for result/end
VBH		start in background mode
	0	start in foreground – script will be displayed
	1	start in background – script will not be displayed

11 OPUS Parameter Reference

The reference section of this chapter presents a complete list of all OPUS parameters. Please keep in mind, that currently some parameters are not yet implemented.

OPUS distinguishes between parameters stored together with a spectrum and such, that are used to control OPUS functions. Not all existing parameters of the former type are necessarily saved in a spectrum file.

Three different types of data exist:

- **numerical** – the parameter is a number
- **text** – the parameter is a text
- **list of values** – the parameter is a text, but only certain values are allowed. A list of possible values is included. The value on the left side of the equal sign is returned to the macro, its explanation is given on the right side.

Sample Parameters

The sample parameters stored with the spectrum:

SNM	(text)	Sample Name
SFM	(text)	Sample Form
CNM	(text)	Operator Name
HIS	(text)	History of Last Operation
PTH	(text)	Measurement Path
EXP	(text)	Experiment
PAT	(text)	Path of File
NAM	(text)	Filename
NA2	(text)	Ch. 2 Filename
ATX	(text)	Annotation text
XPP	(text)	Experiment Path

Data Block-Specific Parameters

These are specific parameters stored with a spectrum. There exists a separate set of paratmeters for each spectral data block stored in a file.

Standard Parameters

DPF	(numerical)	Data Point Format
NPT	(numerical)	Number of Data Points
FXV	(numerical)	Frequency of First Point
LXV	(numerical)	Frequency of Last Point

CSF	(numerical)	Y - Scaling Factor
MXY	(numerical)	Y - Maximum
MNY	(numerical)	Y - Minimum
DXU	(list of values)	X Units
	WN =	Wavenumber cm-1
	MI =	Micron
	NM =	Nanometers
	LGW =	Log Wavenumber
	MIN =	Minutes
	SEC =	Seconds
	PNT =	Points
	EV =	eV
	MM =	Millimeters
	CM =	Centimeters
	MIS =	msec
	MUS =	µsec
DYU	(list of values)	Y Units
	SC =	Single channel
	TR =	Transmittance [%]
	AB =	Absorbance Units
	KM =	Kubelka Munck
	LA =	Log Absorbance
	DR =	Diffuse Reflectance
	ABS =	Absorbance
	REF =	Reflectance
	TRA =	Transmittance
	RRK =	Re (Amplitude Reflectivity Coefficient)
	IRK =	Im (Amplitude Reflectivity Coefficient)
	RTK =	Re (Amplitude Transmission Coefficient)
	ITK =	Im (Amplitude Transmission Coefficient)
	DF1 =	Re (Dielectric Function)
	DF2 =	Im (Dielectric Function)
DAT	(text)	Date of Measurement
TIM	(text)	Time of Measurement

User-Defined Labels

XTX	(text)	X - axis Label
YTX	(text)	Y - axis Label
ZTX	(text)	Z axis Label
XAF	(numerical)	X axis factor
YAF	(numerical)	Y axis factor
ZAF	(numerical)	Z axis factor

Derivatives, Smoothing

DER	(numerical)	Derivative
QS1	(text)	Derivative
SMO	(numerical)	Smoothing points for der.
QS0	(text)	Smoothing points

3D Files

GSQ	(numerical)	GS Base Quality
AOX	(numerical)	Map Origin X
AOY	(numerical)	Map Origin Y
DDX	(numerical)	Map Delta X
DDY	(numerical)	Map Delta Y
NPX	(numerical)	Map Points in X
NPY	(numerical)	Map Points in Y

S/N Ratio

NF1	(numerical)	First S/N Frequency Limit
NF2	(numerical)	Second S/N Frequency Limit
SN1	(numerical)	S/N (RMS)
SN2	(numerical)	S/N (Peak-to-Peak)
SN3	(numerical)	Max. Ordinate in S/N Region
SN4	(numerical)	Min. Ordinate in S/N Region
SNF	(numerical)	S/N Flags

Frequency Calibration

MWC	(numerical)	Mult. for Freq.Calib
AWC	(numerical)	Add for Freq.Calib

Post-Search Specturm Extraction

HQU	(numerical)	Hit Quality
COM	(text)	Compound Name

Instrument Parameters

Parameters used for the spectrometer settings that are stored together with a spectrum:

DPH	(numerical)	Demod. Phase (Degrees)
MOF	(numerical)	Modulation Frequency
NLA	(numerical)	NL Alpha
NLB	(numerical)	NL Beta
HFL	(numerical)	High Folding Limit
LFL	(numerical)	Low Folding Limit
DFR	(numerical)	Digital Filter Reduction
DFC	(numerical)	Number of Filter Coef.
HFF	(numerical)	Digital Filter HFL
LFF	(numerical)	Digital Filter LFL
ASG	(numerical)	Actual Signal Gain
ARG	(numerical)	Actual Ref. Signal Gain
ALF	(numerical)	Actual Low Pass Filter
AHF	(numerical)	Actual High Pass Filter
ASS	(numerical)	Number of Sample Scans
ARS	(numerical)	Number of Background Scans
GFW	(numerical)	Number of Good FW Scans

GBW	(numerical)	Number of Good BW Scans
BFW	(numerical)	Number of Bad FW Scans
BBW	(numerical)	Number of Bad BW Scans
DUR	(numerical)	Scan time (sec)
RSN	(numerical)	Running Sample Number
PKA	(numerical)	Peak Amplitude
PKL	(numerical)	Peak Location
PRA	(numerical)	Backward Peak Amplitude
PRL	(numerical)	Backward Peak Location
SSM	(numerical)	Sample Spacing Multiplicator
SSP	(numerical)	Sample Spacing Divisor
SGP	(numerical)	Switch Gain Position
SGW	(numerical)	Gain Switch Window
INS	(text)	Instrument Type
ITF	(list of values)	Interface Type for Optic
	0 = 25/48	
	1 = 22/28/55/66/88/120/100	
	2 = 85/110/113	
SIM	(numerical)	Simulation Mode
DEB	(numerical)	Debug Printer Mode
LOG	(numerical)	Logfile for Measurement
ADR	(numerical)	AQP Adress
AD2	(numerical)	AQP2 Adress
RMX	(numerical)	Resolution Limit
PLL	(numerical)	Maximum PLL Setting
FFT	(numerical)	Maximum FT Size in K's
MXD	(numerical)	Maximum ADC Sate in Hz
FOC	(numerical)	Focal Length
ABP	(numerical)	Absolute Peak Pos in Laser*2
LWN	(numerical)	Laser Wavenumber
RLW	(numerical)	Raman Laser Wavenumber
RLP	(numerical)	Raman Laser Power in mW
RDY	(list of values)	Ready Check
	0 = OFF	
	1 = ON	
RC0	(list of values)	Raman Background Corrected
	NO = No	
	YES = Yes	
RC1	(list of values)	Raman Scattering Corrected
	NO = No	
	YES = Yes	
SRT	(numerical)	Start time (sec)
ERT	(numerical)	End time (sec)
MAX	(numerical)	X Measurement Position
MAY	(numerical)	Y Measurement Position
AN1	(numerical)	Analog Signal 1
AN2	(numerical)	Analog Signal 2

Data Acquisition Parameters

Parameters used for the acquisition of data that are stored together with a spectrum:

CH2	(list of values) 0 = OFF 1 = ON	Channel 2
SGN	(list of values) -1 = Automatic 0 = 1 1 = 2 2 = 4 3 = 8 4 = 16 5 = 32 6 = 64 7 = 128	Signal Gain, Sample
SG2	(list of values) -1 = Automatic 0 = 1 1 = 2 2 = 4 3 = 8 4 = 16 5 = 32 6 = 64 7 = 128	Signal Gain, Sample
RGN	(list of values) -1 = Automatic 0 = 1 1 = 2 2 = 4 3 = 8 4 = 16 5 = 32 6 = 64 7 = 128	Signal Gain, Background
RG2	(list of values) -1 = Automatic 0 = 1 1 = 2 2 = 4 3 = 8 4 = 16 5 = 32 6 = 64 7 = 128	Signal Gain, Background
GSW	(numerical)	Gain Switch Window
GSG	(list of values) 1 = OFF 8 = ON	Gain Switch Gain
AQM	(list of values) SN = Single Sided DN = Double Sided SF = Single Sided Fast Return SD = Single Sided, Forward-Backward DD = Double Sided, Forward-Backward DF = Double Sided, Fast Return	Acquisition Mode

NSS	(numerical)	Sample Scans
NSR	(numerical)	Background Scans
REP	(numerical)	Repeat Count
DLR	(numerical)	Delay Between Repeats in Sec.
MIN	(numerical)	Sample Meas. Duration in Min.
MIR	(numerical)	Background Meas. Duration in Min.
SOS	(numerical)	Scantime or Scans
SOT	(list of values)	Sample Scans or Time
	0 = Scans	
	1 = Minutes	
STR	(list of values)	BG Scans or Time
	0 = Scans	
	1 = Minutes	
COR	(list of values)	Correlation Test Mode
	NO = No	
	LO = Around Peak,Low	
	HI = Around Peak,High	
	FUL = Full Igram length	
DLY	(numerical)	Stabilization Delay
DEL	(numerical)	Delay Before Measurement
HFW	(numerical)	Wanted High Frequency Limit
LFW	(numerical)	Wanted Low Frequency Limit
RES	(numerical)	Resolution
RE2	(numerical)	Resolution Ch.2
TDL	(numerical)	To do list
PLF	(list of values)	Result Spectrum
	TR = Transmittance	
	AB = Absorbance	
	KM = Kubelka Munk	
	RAM = Raman Spectrum	
	EMI = Emission	
	RFL = Reflectance	
	LRF = Log Reflectance	
	ATR = ATR Spectrum	
	PAS = PAS Spectrum	
SPO	(list of values)	Sample Number
	0 = Background Position	
	1 = 1	
	2 = 2	
	....	
	63 = 63	
RPO	(list of values)	Background Number
	0 = Background Position	
	1 = 1	
	2 = 2	
	.....	
	63 = 63	
WAR	(list of values)	Tr. Rec. Resolution
	4 ns	
	5 ns	
	8 ns	
	10 ns	
	20 ns	
	24 ns	

25 ns
33 ns
40 ns
50 ns
100 ns
200 ns
250 ns
400 ns
500 ns
1 µs
2 µs
2.5 µs
4 µs
5 µs
10 µs
25 µs
40 µs
50 µs
100 µs

WAS	(numerical)	Tr. Rec. Slices
WRC	(numerical)	Tr. Rec. Repeat Count
WTD	(numerical)	Tr. Rec. trigger Delay in points
WPD	(numerical)	Tr. Rec. Stab. Delay after Stepping
WXP	(list of values)	Tr. Rec. Trigger Mode
	1 = Internal	
	2 = External Positive Edge	
	3 = External Negative Edge	
WSS	(list of values)	Tr. Rec. Sampling Source
	0 = External	
	1 = Linear Timescale	
	2 = Compress to Log. Timescale	
W2W	(list of values)	Tr. Rec. Channel 2 Weighting
	0 = Unused	
	1 = Use for Phase Correction	
	2 = Use for Weighting	
	3 = Use for Weighting, discard if < Threshold	
	4 = Discard Experiment if < Threshold	
WXD	(numerical)	Tr. Rec. Experiment Delay
WDV	(list of values)	Transient Recorder
	1 = PAD82A	
	2 = PAD82B	
	3 = PAD82	
	4 = PAD1232a	
	5 = PAD1232b	
	6 = PAD1232c	
WIR	(list of values)	Tr. Rec. Input Range
	1 = ±200mV	
	2 = ±500mV	
	3 = ±1V	
	4 = ±2V	
	5 = ±4V	
	6 = 0..400mV	
	7 = 0..1V	
	8 = 0..2V	

WTH	(numerical)	Tr. Rec. Weighting Threshold
TRR	(numerical)	TRS Resolution in micro sec
TRS	(numerical)	TRS Slices
TRC	(numerical)	TRS Repeat Count
TRD	(numerical)	TRS Exp Delay in msec
TRP	(numerical)	TRS Positionning Delay
TRM	(numerical)	TRS Experiment Trigger Mode
TRX	(numerical)	TRS Sampling Source
ITS	(numerical)	Interleaved Time Slices
ISP	(numerical)	Interleave Time Res. æsec
IDL	(numerical)	Interleave Trigger Delay æsec
ITR	(numerical)	Max. Exp. Trigger Rate Hz
STD	(numerical)	Step Scan Pos. Delay in msec
STC	(numerical)	Step Scan Coadd Count
SMX	(numerical)	Multiplexer positions
SMD	(numerical)	Modulation (0=OFF 1=MOD 2=MOD-DEMOD
4=OLD		
		PHASE 8=AMPL)
SMA	(numerical)	Scanner Modulation Amplitude
SMF	(numerical)	Scanner Modulation Frequency
AMF	(numerical)	Ampl. Modulation Frequency
ADA	(numerical)	Ampl. Demodulation Angle
PDA	(numerical)	Phase Demodulation Angle
CIN	(numerical)	Chrom Integrate Trace
CIM	(text)	Chrom Integration Method
CDT	(numerical)	Chrom Display Trace
CDS	(numerical)	Chrom Display Spectrum
CTM	(numerical)	Chrom Start Trigger Mode
CSV	(numerical)	Chrom Save Mode
CTL	(numerical)	Chrom Trigger Level
GSS	(numerical)	Gram Schmidt Size
GSO	(numerical)	Gram Schmidt Offset
GSP	(numerical)	Gram Schmidt Points
CLD	(numerical)	Limit Run Duration
CMD	(numerical)	Max Run Duration
MLS	(text)	Map XY List
MPO	(numerical)	Map Port (com 1...n)
MSH	(numerical)	Map Shape (1..6)
NDV	(list of values)	Map Device
	0 = Internal (MCLStage)	
	1 = Microscope (MCLStage)	
NOX	(numerical)	Map Origin X
NOY	(numerical)	Map Origin Y
NSX	(numerical)	Map Spacing X
NSY	(numerical)	Map Spacing Y
NGX	(numerical)	Map Gram Schmidt X Base
NGY	(numerical)	Map Gram Schmidt Y Base
MEX	(text)	Map Measurement Experiment
MUN	(list of values)	Map units
	0 = Micron	
	1 = mm	
	2 = cm	
MPX	(numerical)	Map # Pos X
MPY	(numerical)	Map # Pos Y

MSS	(numerical)	Map Save Spectra
MCI	(numerical)	Map Compute Integrals
MIM	(text)	Map Integration Method
MCM	(numerical)	Map Macro
MEM	(text)	Map Evaluation Macro
MSV	(numerical)	Map Save Video
MVM	(text)	Map Video Method
MRL	(numerical)	Map Relative Origin
MGT	(numerical)	Map Gram Schmidt
MGS	(numerical)	Map Gram Schmidt Size
MGO	(numerical)	Map Gram Schmidt Offset
MGP	(numerical)	Map Gram Schmidt Points
MXS	(numerical)	Meas x-Startpoint Display
MXE	(numerical)	Meas x-Endpoint Display
MYS	(numerical)	Meas y-Minimum Display
MYE	(numerical)	Meas y-Maximum Display
MDM	(numerical)	Meas Display Mode
MDP	(numerical)	Meas Display Product
XS2	(numerical)	Meas2 x-Startpoint Display
XE2	(numerical)	Meas2 x-Endpoint Display
YS2	(numerical)	Meas2 y-Minimum Display
YE2	(numerical)	Meas2 y-Maximum Display

FT-Parameters

Parameters used for the Fourier Transformation, that are stored together with a spectrum:

AF2 (list of values) Apodization Function
- BX = Boxcar
- TR = Triangular
- 4P = Four Point
- HG = Happ-Genzel
- B3 = Blackman-Harris 3-term
- B4 = Blackman-Harris 4-term
- NBW = Norton-Beer, Weak
- NBM = Norton-Beer, Medium
- NBS = Norton-Beer, Strong
- US1 = User One
- US2 = User Two

HQ2 (numerical) End Frequency Limit for File

LQ2 (numerical) Start Frequency Limit for File

PH2 (list of values) Phase Correction Mode
- ML = Mertz
- SM = Mertz Signed
- PW = Power Spectrum
- MLP = Mertz / No Peak Search
- SMP = Mertz Signed / No Peak Search
- PWP = Power / No Peak Search

SP2 (list of values) Stored Phase Mode
- NO = No

ZF2 (list of values) Zero Filling Factor
- 1 = 1

	2 = 2	
	4 = 4	
	8 = 8	
	16 = 16	
	32 = 32	
	64 = 64	
	128 = 128	
	256 = 256	
	512 = 512	
APF	(list of values)	Apodization Function
	BX = Boxcar	
	TR = Triangular	
	4P = Four point	
	HG = Happ-Genzel	
	B3 = Blackman-Harris 3-Term	
	B4 = Blackman-Harris 4-Term	
	NBW = Norton-Beer, Weak	
	NBM = Norton-Beer, Medium	
	NBS = Norton-Beer, Strong	
	US1 = User One	
	US2 = User Two	
HFQ	(numerical)	End Frequency Limit for File
LFQ	(numerical)	Start Frequency Limit for File
PHZ	(list of values)	Phase Correction Mode
	ML = Mertz	
	SM = Mertz Signed	
	PW = Power Spectrum	
	MLP = Mertz / No Peak Search	
	SMP = Mertz Signed / No Peak Search	
	PWP = Power / No Peak Search	
	MS = Mertz / Stored Phase	
	NO = No / Save Complex Data	
PHR	(numerical)	Phase Resolution
NLI	(numerical)	Non Linearity Correction
NL2	(numerical)	Non Linearity Correction
DIG	(numerical)	Digital Filter
DI2	(numerical)	Digital Filter
SPZ	(list of values)	Stored Phase Mode
	NO = No	
ZFF	(list of values)	Zero Filling Factor
	1 = 1	
	2 = 2	
	4 = 4	
	8 = 8	
	16 = 16	
	32 = 32	
	64 = 64	
	128 = 128	
	256 = 256	
	512 = 512	

Parameters of the Optics

Parameters used by the optics that are stored together with a spectrum:

IRS	(numerical)	Iris Aperture (micron)
UNI	(text)	Command string for UNI
APT	(list of values)	Aperture Setting, depending on the Optics
AP2	(list of values)	Aperture Setting, depending on the Optics
BMS	(list of values)	Beamsplitter Setting, depending on the Optics
DTC	(list of values)	Detector Setting, depending on the Optics
DT2	(list of values)	Detector Setting
OPF	(list of values)	Optical Filter Setting, depending on the Optics
OF2	(list of values)	Optical Filter Setting, depending on the Optics
PGN	(list of values)	Preamplifier Gain, depending on the Optics
CHN	(list of values)	Measurement Channel, depending on the Optics
DMX	(list of values) 1= ADC 1 2 = ADC 2 3 = ON	Multiplexed Data
ADC	(list of values) 0 = OFF 1 = 1 2 = 2 3 = 1 & 2	Ext. Analog Signals
SON	(list of values) 0 = OFF 1 = ON 2 = Extended	External Synchronisation
SRC	(list of values)	Source Setting, depending on the Optics
VSC	(numerical)	Variable Velocity (Hz)
VEL	(list of values)	Scanner Velocity, depending on the Optics
HPF	(list of values)	High Pass Filter, depending on the Optics
LPF	(list of values)	Low Pass Filter, depending on the Optics
SRL	(numerical)	Raman Laser Power(mW)
RFL	(numerical)	Raman Flags
POL	(list of values) 91 = Out 0 = 0ø 90 = 90ø	Polarizer

Parameters of OPUS Functions

Parameters of the OPUS Functions, sorted by functionality. You will find detailed information about these functions in chapter 11.

General

FIM	(numerical)	File name Incrementing Mode

S/N Ratio

NF1	(numerical)	First S/N Frequency Limit

NF2	(numerical)	Second S/N Frequency Limit
SN1	(numerical)	S/N (RMS)
SN2	(numerical)	S/N (Peak-to-Peak)
SN3	(numerical)	Max. Ordinate in S/N region
SN4	(numerical)	Min. Ordinate in S/N region
SNF	(numerical)	S/N Flags

Subtract

SX1	(numerical)	Start Frequency for Autosub
SX2	(numerical)	End Frequency for Autosub
SUN	(numerical)	Number of Spectra for Subtract
SUB	(numerical)	Subtract Mode

Assemble GC

QA6 (text) Assembled GC Spectrum Path
QA7 (text) Assembled GC Spectrum
QA8 (numerical) Assembled GC Start Frequency
QA9 (numerical) Assembled GC End Frequency
QAA (numerical) Assembled GC Whole x-Range
QAB (list of values) Assembled GC Z Units
- SEC = Seconds
- MIN = Minutes
- NM = Nanometers
- MI = Micrometers
- MM = Millimeters
- CM = Centimeters
- WN = Wavenumber 1/cm
- LGW = Log Wavenumber
- EV = eV
- PNT = Points
- MIS = msec
- MUS = æsec
- NON = None

QAC (numerical) Assembled GC Z-Start
QAD (numerical) Assembled GC Z-End
LST (text) List of Filenames
BLK (text) Blocktype to Assemble
QM0 (list of values) Assembled Map Units
- MI = Micrometers
- MM = Millimeters
- CM = Centimeters
- PNT = Points

QM1 (numerical) Assembled Map x points
QM2 (numerical) Assembled Map Delta x
QM3 (numerical) Assembled Map y points
QM4 (numerical) Assembled Map Delta y

Conformity Test

QCF	(text)	Conform. Test Methd. Path
QCG	(text)	Conform. Test Methd. File
CSM	(text)	CSM Method name

Post-FT ZFF

PZF	(numerical)	Post Zerofill Factor
PZS	(numerical)	Post Zf Start Frequency
PZE	(numerical)	Post Zf End Freqency
PWR	(numerical)	Post Zf Whole Range (0: no; 1: yes)

Fourier Transformation

FHZ (list of values) FT Phase Correction Mode

- ML = Mertz
- SM = Mertz Signed
- PW = Power Spectrum
- MS = Mertz / Stored Phase
- NO = No / Save Complex Data
- FM = Forman
- FS = Forman / Stored Phase
- FP = Forman / Preapodized
- DP = Doubled Phase

FTA (list of values) FT Apodization Function

- BX = Boxcar
- TR = Triangular
- 4P = Four point
- HG = Happ-Genzel
- B3 = Blackman-Harris 3-Term
- B4 = Blackman-Harris 4-Term
- NBW = Norton-Beer, Weak
- NBM = Norton-Beer, Medium
- NBS = Norton-Beer, Strong
- US1 = User One
- US2 =User Two

FZF (list of values) FT Zero Filling Factor

- 1 = 1
- 2 = 2
- 4 = 4
- 8 = 8
- 16 = 16
- 32 = 32
- 64 = 64
- 128 = 128
- 256 = 256
- 512 = 512

FTS	(numerical)	FT Start Frequency
FTE	(numerical)	FT End Freqency
FTR	(numerical)	FT Resolution Limit
FHR	(numerical)	FT Phase Resolution
FLR	(numerical)	FT Limit Resolution
FTT	(numerical)	FT to do list
FBW	(numerical)	Forward/Backward Igram
FNL	(numerical)	Non Linearity Correction
FNC	(numerical)	FT Detector Cutoff Freq.
FNE	(numerical)	FT Modulation Efficiency
FSM	(list of values)	FT ZPD Search Mode

AL = Absolute largest Value
MX = Maximum
MN = Minimum
MI = Mid between Min/Max
NO = No Peak Search
MA = Manually
TW = Mid between largest two
TP = Take from stored Phase

FPP	(numerical)	ZPD Position
FSR	(numerical)	ZPD Search Range
FSY	(numerical)	FT Symmetry

Kramers-Kronig Transformation

KKS	(numerical)	KKT Start Frequency
KKE	(numerical)	KKT End Freqency
KWR	(numerical)	KKT Whole Range
KMT	(numerical)	KKT material (extrapol + cond/not cond)
KKR	(numerical)	KKT result

Deconvolution

DEF	(numerical)	Deconvolution Factor
DNR	(numerical)	Deconv Noise Reduction
DES	(numerical)	Deconv Start Frequency
DEE	(numerical)	Deconv End Frequency
DWR	(numerical)	Deconv Whole Range
DSP	(list of values)	Deconv Line Shape

LO = Lorentz
GA = Gauss

Curve Fitting

FXS	(numerical)	FIT Start Frequency
FXE	(numerical)	FIT End Frequency
FWR	(numerical)	FIT Whole Range

Inverse FT

RXS	(numerical)	Reverse FT Start Frequency
RXE	(numerical)	Reverse FT End Frequency
RWR	(numerical)	Reverse FT Whole Range
RSY	(numerical)	Reverse FT Symmetry

Symmetric FT

FPS	(numerical)	Symmetric FT First Point
LPS	(numerical)	Symmetric FT Last Point
WRS	(numerical)	Symmetric FT Whole Range
SSY	(numerical)	Symmetric FT Symmetry

2D Correlation

2DM	(list of values)	Correlation Mode
	SQ = Squared Correlation	
	RG = Regression	
	CO = Correlation	
	SY = Synchron	
	AS = Asynchron	
2XS	(numerical)	2D X Start Frequency
2XE	(numerical)	2D X End Frequency
2WR	(numerical)	2D Whole X Range
2YS	(numerical)	2D Y Start Frequency
2YE	(numerical)	2D Y End Frequency
2WY	(numerical)	2D Whole Y Range
2DR	(numerical)	2D Digital Resolution
2DF	(numerical)	2D Reduction Factor

DMA Extraction

DMA	(numerical)	DMA extraction mode
DMF	(numerical)	DMA frequency of strain (Hz)
DMP	(numerical)	DMA phase of strain
DMS	(numerical)	DMA additional phase shift

Smoothing, Derivative

QSP	(numerical)	Number of Smoothing Points
QOD	(numerical)	Order of Derivative

Compare Spectra

QFX	(numerical)	Lower Compare Frequency Range
QLX	(numerical)	Upper Compare Frequency Range
QWR	(numerical)	Whole x-Range (0: no; 1: yes)

Identitity Test

QFB	(text)	Identity Test Methods
QFC	(text)	Method name
QFD	(numerical)	Use AQP Flag (0: No; 1: Yes)
QFE	(numerical)	Check Best Hit (0: No; 1: Yes)
QFF	(numerical)	Expected Reference ID
QFH	(numerical)	1st SNM char. for check
QFI	(numerical)	Length of SNM part for check
QFJ	(numerical)	Ident: 0: Analy; 1: Add List; 2: Rem. All
QFK	(text)	Ident: Path of Fl. to be Added
QFL	(text)	Ident: Name of Fl. to be Added
QI0	(numerical)	Ident: BlockID of Fl. to be Added
QI1	(list of values)	Sort for Expected Reference
	SNM = Sample Name	
	ID = ID	
	FIL = File Name	

QI2	(numerical)	1st SNM Char. for Sorting
QQN	(binär)	QIdnt XRanges
QQO	(binär)	QIdnt Deriv

Cluster Analysis

QCA	(text)	Method Path
QCB	(text)	Method Name
QCH	(text)	text-List Path
QCI	(text)	text-List Name (incl. Ext.)
QCJ	(list of values)	Print/Plot
	NO = No	
	DEN = Dendrogram	
	DIA = Diagnosis	
	BOT = Both	
QCK	(list of values)	For Making Dendro
	FIL = File Name	
	SNM = Sample Name	
	NUM = File Number	
	NO = No Name Markers	
QCL	(numerical)	# of Classes (for Diagn.)

Peak Picking

FXP	(numerical)	Peak Pick Start Frequency
LXP	(numerical)	Peak Pick End Frequency
QP0	(list of values)	Precision User-Defined
	NO = No	
	YES = Yes	
QP3	(numerical)	Digits After Decimal Point
QP4	(list of values)	Upper Peak Limit
	NO = No	
	YES = Yes	
QP5	(numerical)	Peaks < [%]
QP6	(list of values)	Upper Peak Limit Abs.
	NO = No	
	YES = Yes	
QP7	(numerical)	Peaks < [abs.]
QP8	(list of values)	Lower Peak Limit Abs.
	NO = No	
	YES = Yes	
QP9	(numerical)	Peaks > [abs.]
PTR	(numerical)	Peaks > [%]
PPM	(numerical)	Peak Pick Mode (Auto,Max,Min)
NSP	(numerical)	Number of Smoothing Points
PSM	(numerical)	Peak Search Method
WHR	(numerical)	Whole x-Range (0: no; 1: yes)

Black Body

QTE	(numerical)	New Entries
QPH	(numerical)	0:Energy; 1:Photons

Raman Correction

QC0	(numerical)	1: Raman Background Correction
QC1	(numerical)	1: Raman Scattering Correction
QC2	(numerical)	1: Undo Correction
QC3	(text)	Calibration Lamp Spectrum Path
QC4	(text)	Calibration Lamp Spectrum
QC5	(numerical)	Temp. of Calibration Lamp

Averaging

QA0	(numerical)	1: Consider Scans
QA1	(numerical)	1: Normalized Spectra
QA2	(numerical)	1: Average Report
QA3	(text)	Av. Spectrum Path
QA4	(text)	Av. Spectrum
QA5	(numerical)	1: Add to List in Ident Method
QAE	(list of values) NO = No YES = Yes	Create Std-Dev Spectrum
QAF	(list of values) NO = No YES = Yes	Update Av. Spectrum
QAG	(text)	Updated Av. Spectrum Path
QAH	(text)	Updated Av. Spectrum
QAL	(list of values) LIS = File List FIL = File Name	Av.: Source of Orig. Spectra
QAM	(text)	Av: Orig. Spectra Path
QAN	(text)	Av: Orig. Spectra
QAO	(numerical)	Av: Orig. Spectra Block ID
QD0	(numerical)	1: New Entries

Quality Test

QQ0	(numerical)	x-Start (x range 0)
QQ1	(numerical)	x-End (x range 0)
QQ2	(numerical)	ymax-ymin(x range 0) must be >
QQ3	(numerical)	ymax-ymin(x range 0) must be <
QQ4	(numerical)	x-Start (x range 1)
QQ5	(numerical)	x-End (x range 1)
QQ6	(numerical)	x-Start (x range 2)
QQ7	(numerical)	x-End (x range 2)
QQ8	(numerical)	x-Start (x range noise)
QQ9	(numerical)	x-End (x range noise)
QQA	(numerical)	y’max (x range noise) must be <
QQB	(numerical)	S/Noise (x range 1) must be >
QQC	(numerical)	S/Noise (x range 2) must be >
QQD	(numerical)	x-Start (x range water)
QQH	(numerical)	x-End (x range water)
QQE	(numerical)	y’max (x water) must be <
QQF	(numerical)	S/Water (x range 1) must be >
QQG	(numerical)	S/Water (x range 2) must be >

QQI	(numerical)	x-Start (x range fringes)
QQJ	(numerical)	x-End (x range fringes)
QQK	(numerical)	y'max (x fringes) must be <
QQL	(text)	Q-Test Method Path
QQM	(text)	Q-Test Method Name

2D Correlation

QC6	(numerical)	2D-Corr. Whole x-Range
QC7	(numerical)	2D-Corr. Start Frequency
QC8	(numerical)	2D-Corr. End Frequency
QC9	(list of values)	2D-Corr. Limit Resolution
	NO = No	
	YES = Yes	
QCC	(numerical)	2D-Corr. Reduction Factor
QCD	(list of values)	2D-Corr. Synchronous
	NO =No	
	YES = Yes	
QCE	(list of values)	2D-Corr. Asynchronous
	NO = No	
	YES = Yes	

Display Limits

XSP	(numerical)	Left X Display Limit
XEP	(numerical)	Right X Display Limit
YMN	(numerical)	Lower Y Display Limit
YMX	(numerical)	Upper X Display Limit
XAU	(list of values)	X - Axis Scaling
	NO = Linear	
	YES = Compressed	

Normalization

NME	(numerical)	Method (1: Min-Max, 2: Vector, 3: Offset)
NFX	(numerical)	First Point
NLX	(numerical)	Last Point
NWR	(numerical)	Whole Range (0: no, 1: yes)

Frequency Calibration

QF0	(list of values)	Restore Orig. Calib.
	NO =No	
	YES = Yes	
MWC	(numerical)	Mult. for Freq.Calib
AWC	(numerical)	Add for Freq.Calib

Baseline Correction

BME	(numerical)	Baseline Method
BCO	(numerical)	Exclude CO2 - Bands
BPO	(numerical)	Number of Baseline Points

Make Compatible

CME	(numerical)	Method (2: Interpolation, 3: Reduce Resolution)

AB $\rightarrow$ TR

CCM	(numerical)	Method

Spectrum Calculator

CDI	(numerical)	Permanent Dialog
FOR	(text)	Formula

Cut

CFX	(numerical)	First Point
CLX	(numerical)	Last Point

Generate Straight Line

GFX	(numerical)	First Point
GLX	(numerical)	Last Point

Convert Spectra

CSD	(numerical)	Method

CarbOx Analysis

FFO	(numerical)	Factor for Oxygen
FFC	(numerical)	Factor for Carbon
FME	(numerical)	Carbox Method
FUN	(numerical)	Units
FRT	(numerical)	Reference Wafer Thickness
FST	(numerical)	Wafer Thickness
FSI	(numerical)	Free Carrier Type
FCN	(numerical)	Charge Carrier Concentration
FOF	(numerical)	Offset
FSL	(numerical)	Slope
FCO	(numerical)	Oxygen Conversion Coefficient
FCC	(numerical)	Carbon Conversion Coefficient

Epi Analysis

EFX	(numerical)	First Point
ELX	(numerical)	Last Point
EWR	(numerical)	Whole Range (0: no, 1: yes)
EMO	(numerical)	Mode
EN1	(numerical)	Refraction Index n1
EN2	(numerical)	Refraction Index n2
EN3	(numerical)	Refraction Index n3
ES1	(numerical)	Sign first echo-peak
ES2	(numerical)	Sign second echo-peak

$cm^{-1} \rightarrow \mu$

LNE	(numerical)	New Entry (cm-1/micron)
LME	(numerical)	Method (cm-1/micron)
LYS	(numerical)	Y scale (cm-1/micron)
LCF	(numerical)	Max. Compression Factor
QL0	(numerical)	Whole x-Range (0: no; 1: yes)
QL1	(numerical)	Lower Compare Frequency Range
QL2	(numerical)	Upper Compare Frequency Range
QL3	(numerical)	Data Points

Integration

LPT	(text)	Integration Method Path
LFN	(text)	Method Filename
LRM	(numerical)	Integration Report Storage Mode

Quant

QPT	(text)	File List Path
QFN	(text)	File List Name
QP1	(text)	Quant 1 Method Path
QF1	(text)	Quant 1 Method
QP2	(text)	Quant 2 Method Path
QF2	(text)	Quant 2 Method
CAP	(text)	Unscrambler Model Path
CAF	(text)	Unscrambler Model Name
GAP	(text)	PLSplus/IQ Calibration File Path
GAF	(text)	PLSplus/IQ Calibration File Name
HEV	(numerical)	Extract Vol [ml]
HWV	(numerical)	Water Vol [l]
HCT	(numerical)	Cell Thickness [cm]
HDF	(numerical)	Dilution Factor
HAR	(numerical)	Aromatics
QSM	(numerical)	No of Smoothing Points
FLC	(numerical)	F Prob Limit Concentration Outliers
FLS	(numerical)	F Prob Limit Spectral Outliers
LLF	(numerical)	Leverage Limit Factor
QPP	(numerical)	Quant 2 Preprocessing Options

Plot Report Parameters

GMS	(numerical)	Marker Size
GMA	(numerical)	Marker Symbol small circle
GMC	(numerical)	Marker Color CLR_NEUTRAL
GXS	(numerical)	Frame X Position 3 cm
GYS	(numerical)	Frame Y Position 2 cm
GDX	(numerical)	Frame dx 12 cm
GDY	(numerical)	Frame dy 12 cm

Info Parameters, JCAMP Setup

INP	(text)	Info text Path
IFP	(text)	Correlation Table Path
INM	(text)	Info Definition Filename
IFN	(text)	Info Definition Filename
IDS	(text)	Info Definition Description

Macro Parameters

MPT	(text)	Macro Path
MFN	(text)	Macro Filename
MDS	(text)	Macro Description

GRAMS Export

GMN	(text)	Macro Filename
GMD	(text)	Macro Description

Data Path Parameters

WOP	(text)	Work File Path
DAP	(text)	Data File Path
FMP	(text)	File Manager Path
DAF	(text)	initial filename for load
QL4	(numerical)	Filter Index

Parameters for Post-Run Extraction

EXS	(numerical)	Extract from start of file
EXE	(numerical)	Extract to end of file
ENT	(numerical)	Entry to extract from
ENE	(numerical)	Entry to extract to
ECO	(numerical)	Coadd all to one
XTP	(text)	Extract Path
XTN	(text)	Extract Filename
XTI	(numerical)	Increment Name (1=Name 0=Ext)
COL	(numerical)	Spectra Color
EAB	(numerical)	Abort extraction if file exists
ELF	(numerical)	Load extracted files

Simulation

SXS	(numerical)	Simul Start Frequency
SXE	(numerical)	Simul End Frequency
SLA	(numerical)	Simul Angle Degrees
SER	(numerical)	Simuls Epsilon Re[]
SEI	(numerical)	Simuls Epsilon Im[]
SFP	(numerical)	Simuls Plasma Frequency
SFS	(numerical)	Simuls Scatter Frequency
SVA	(numerical)	Simuls Valence A
SVB	(numerical)	Simuls Valence B

SVC	(numerical)	Simuls Valence C
SVG	(numerical)	Simuls Energy Gap
SZL	(numerical)	Simuls Lattice Count
SWR	(numerical)	Simul Whole Range
SXP	(numerical)	Simul Points
SLI	(numerical)	Simul Light From
SLP	(numerical)	Simul Polarization
SRI	(numerical)	Simul Extract DF Re/Im
SFV	(numerical)	Simul DF Values Flag
SPC	(numerical)	Simuls Coherence
SDL	(numerical)	Simuls Lattice
SDF	(numerical)	Simuls Free Carriers
SDV	(numerical)	Simuls Valence Electrons
SCQ	(list of values)	Simul Computed Quantity
	ABS = Absorbance	
	REF = Reflectance	
	TRA = Transmittance	
	RRK = Amplitude Reflectivity Coefficient	
	RTK = Amplitude Transmission Coefficient	
SPT	(list of values)	Simuls Type
	HOM = Homogenious	
	IHO = Inhomogenious	
SPD	(text)	Simul Layer Density
SLS	(text)	Simul Layer Stack
SDP	(text)	Simul Layer DF
SDE	(text)	Simul Extract DF
SDD	(text)	Simul Layer DF File
SLF	(text)	Simul Layer Stack File

Extrapolation

QX0	(numerical)	Extrapol R (0)
QX1	(numerical)	Extrapol R (inf.)
QX2	(numerical)	Extrapol i1
QX3	(numerical)	Extrapol i2
QX4	(numerical)	Extrapol ny end

Trace Calculation

QT0	(list of values)	Trace Cal.: Peak Integrals
	NO = No	
	YES = Yes	
QT1	(list of values)	Trace Cal.: Trace Points by Macro
	NO = No	
	YES = Yes	
QT2	(text)	Trace Cal.: Macro Path
QT3	(text)	Trace Cal.: Macro Filename

x Point Adaption

QAI	(numerical)	Adapting: New Entries
QAJ	(text)	Adapting: Method Path
QAK	(text)	Adapting: Method Name

Manipulate GC Blocks

QM5	(text)	Manip. GC: Macro Path
QM6	(text)	Manip. GC: Macro Filename
LB0	(text)	Defaults for Load Box
RS1	(text)	Function Result

Parameter for the Library Search

LBN	(text)	Library Name
LB1	(text)	Library List
LBP	(text)	Library Path
LBT	(text)	Info textfile Name
LTP	(text)	Path for Library text Definitions
LMT	(text)	Method File Name
MTP	(text)	Path for Library Method Definitions
LMN	(numerical)	Method ID
LMO	(numerical)	Library Edit Mode
LSM	(numerical)	Library Store Mode
LBS	(numerical)	Spectrum Number
LSS	(numerical)	Search Sensitivity
LPR	(numerical)	Library Protection
LCP	(text)	Copyright Note
LP1	(numerical)	Password Read
LP2	(numerical)	Password Write
SSQ	(numerical)	Minimum HQ for Spectrum Search
SSH	(numerical)	Maximum Hits for Spectrum Search
SS1	(numerical)	Spectrum Search Algorithm
SIH	(numerical)	Maximum Hits for Info Search
SPQ	(numerical)	Minimum HQ for Peak Search
SPH	(numerical)	Maximum Hits for Peak Search
STH	(numerical)	Maximum Hits for Structure Search
SPA	(numerical)	Peak Search Algorithm
LID	(text)	Library Description
LAL	(numerical)	Structure Search Algorithm
SIN	(text)	Info Query Name
SIP	(text)	Info Query Path
PNP	(text)	Peak Query Name
PPP	(text)	Peak Query Path
MPP	(numerical)	Show Search Report immediately
RNG	(text)	Excluded Ranges File

Temperature Control

TWK	(numerical)	Temperature work for thread
TPO	(numerical)	Temperature Port (com 1...n)
TPD	(numerical)	Use Com Defaults
TMP	(numerical)	Temperature
TCS	(text)	Temperature Command String
TDV	(list of values)	Temperature Control Device

0 = Eurotherm 800 Series
1 = Lake Shore 320
2 = Linkam 93 Series
3 = Eurotherm 2000 Series

Rapid Scan TRS

RAT	(text)	TRS Method name

Communication Parameters

You will find parameters used for data output etc. listed in chapter 10.

JCAMP Parameters

JCC	(list of values) 1 = Yes 0 = No	Generate JCAMP-DX Compound Files
ORP	(list of values) 32 = 32 Bit 16 = 16 Bit	Ordinate Precision
JDT	(list of values) PAC = Packed SQD = Squeezed/Dup DD = Difference/Dup	JCAMP-DX Data Type

Save, Save As, Send File

COF	(numerical)	Copy Flags
OEX	(list of values) 1 = Yes 0 = No	Overwrite Existing Files
REN	(text)	New File Name
SAN	(text)	'Save As' File Name
DAP	(text)	Data File Path

Read Datapoint Table

DFN	(text)	File Name (Data Point Table)
XCO	(numerical)	x Column (Data Point Table)
YCO	(numerical)	y Column (Data Point Table)
OE1	(list of values) 1 = Yes 0 = No	Overwrite (Data Point Table)
SOX	(list of values) 1 = Yes 0 = No	Sort X Values (Data Point Table)
QD1	(numerical)	Max. Number of Data Points
QD2	(numerical)	1st Data Point Line

Write Datapoint Table

DPA	(numerical)	Decimal Places, Abscissa
DPO	(numerical)	Decimal Places, Ordinate
SEP	(text)	Separator
ADP	(list of values) 1 = Yes	All Data Points

	0 = No	
YON	(list of values)	y Values only
	1 = Yes	
	0 = No	
DBT	(numerical)	Data Block Type

External Program

XPF	(numerical)	Run as Opus Task
XST	(numerical)	Start Server
XPR	(text)	Program Name
XPA	(text)	Parameters
XWI	(numerical)	Run in a Window
XWS	(numerical)	Window Size Option
XCW	(numerical)	Close Window on Exit
XSB	(numerical)	Start in Background
XEM	(numerical)	Windows Enhanced Mode
XDM	(text)	VDM Settings Filename
XVP	(numerical)	View Transactions
DDE	(numerical)	DDE Interaction
DDS	(text)	DDE Server Name
DDT	(text)	DDE Topic
DDI	(text)	DDE Item
DDD	(text)	DDE Data
VBS	(text)	VB Script Name
VBP	(text)	Script Parameters
VBW	(numerical)	Wait for Script

Pipe Parameters

PIN	(text)	Pipe Name
PIS	(text)	Pipe String
PIF	(numerical)	Pipe Flags
CO1	(numerical)	COMmunication Flags
CO2	(numerical)	Separator/Terminator Bytes for COMmunication
CO3	(numerical)	Byte Count for COMmunication
CFI	(text)	COMmunication Output File
COT	(text)	COMmunication Output text
TIO	(numerical)	Timeout

Plot Parameters

Parameters used to plot data.

PUN	(list of values)	units (cm / inch)
	CM = cm	
	IN = inch	
PL0	(numerical)	Peak Label Size
LFO	(text)	Peak Label Font
PL1	(numerical)	Option Flags
PL2	(numerical)	’n’ Strongest Peaks to Label
PL3	(numerical)	Decimals for Label Numbers

PL4	(numerical)	Length Stroke 0 - 1
PL5	(numerical)	Length Stroke 1 - 2
PL6	(numerical)	Length Stroke 2 - 3
PL7	(numerical)	Distance Peak <--> Line
PL8	(numerical)	Peak Stroke Length
LGO	(list of values)	BRUKER Logo
	NO = No	
	YES = Yes	
PDV	(text)	Plot Device
SCP	(text)	Script Path
SCN	(text)	Script Name
PRP	(numerical)	Printer Port (LPTn)
LPP	(numerical)	Lines per Page
POP	(text)	Plot & Print Output Path
POF	(text)	Plot & Print Output File Name
PLO	(text)	Print Log File Name
PF1	(numerical)	Plot Option Flags
PFI	(numerical)	First Item Printed
PLI	(numerical)	Last Item Printed
PRF	(numerical)	Print Option Flags
PPA	(text)	Polyline Parameters
PWO	(text)	Plot WYSIWYG Options
PDH	(numerical)	Plot Dialog Window Handle
PTX	(text)	Print text
PLP	(text)	Print Log Position on Screen
PRN	(text)	Print Device
MLW	(numerical)	Minimum Spectrum Line Width in Plot
COS	(text)	Color Settings in Plot
M00	(text)	Plot Message # 0
M01	(text)	Plot Message # 1
M02	(text)	Plot Message # 2
M03	(text)	Plot Message # 3
M04	(text)	Plot Message # 4
M05	(text)	Plot Message # 5
M06	(text)	Plot Message # 6
M07	(text)	Plot Message # 7
M08	(text)	Plot Message # 8
M09	(text)	Plot Message # 9

12 The C/S-Interpreter and its Commands

The Client/Server Interpreter is the module of OPUS responsible for processing commands received through the Pipe-, DDE- or Scripting interface. Therefore, the list of commands is the same for all three interfaces.

The following chapters mainly address users who intend to write their own programs and link them to OPUS or OPUS macros. This is achieved with the OPUS command *External Program*, which was described earlier. In the following we expect the user to be familiar with this command and its options.

A part of these commands was already available under OPUS OS/2 in form of the Client/Server function. Hence, in the following the commands are divided in old and new ones.

12.1 Overview of Available Functions

Currently, you can use a client program to:

- read data from OPUS spectrum files and 3D files; you can either read the whole frequency region or select a part of interest from the data.
- write data to OPUS spectrum files and 3D files; you can either write the whole frequency region or select a part of interest from the data.
- load and unload OPUS files.
- read file information from the Client/Server file list.
- read OPUS parameters from an OPUS file.
- save OPUS parameters to an OPUS file.
- read data from report blocks.
- start OPUS macros.
- exchange parameters with an OPUS macro.

In addition, all functions of the command line, i.e. all OPUS processing functions are supported, according to the syntax described earlier.

12.2 Commands and Command Syntax

In the following you find a list containing all Client/Server commands. The description of all commands is structured in the same manner:

Syntax:

The name of the command and the syntax that has to be applied. Mandatory exchange parameters are indicated with „< >“, optional parameters are enclosed in square brackets „[ ]“.

Description:

A description of the action performed by the command.

Return Value:

A list of the possible return values.

Return Value 2:

Return Value 3:

Some commands return additional text after confirming the execution with *OK*; in this case they must be read.

Errors:

A list of possible error messages.

Comments:

Notes and further comments about the command.

12.3 Old C/S Commands

These commands have been available already in OPUS-OS/2.

12.3.1 Overview

The following commands are still used by OPUS-NT:

TIMEOUT	sets the maximum wait time
CLOSE_PIPE	closes pipe
OVERWRITE	activates overwrite mode
PRESERVE	deactivates overwrite mode
COUNT_ENTRIES	counts entries of the file input list
READ_FROM_ENTRY	sets the entry number and data block
WRITE_TO_ENTRY	sets the entry number and data block for writing
READ_FROM_FILE	selects file for reading
WRITE_TO_FILE	selects file for writing
READ_FROM_BLOCK	specifies the data block for reading
WRITE_TO_BLOCK	specifies the data block for writing

ASCII	sets data point mode to text
BINARY	sets data point mode to binary
DATA_VALUES	sets data point mode to frequencies
DATA_POINTS	sets data point mode to data points
READ_HEADER	reads spectrum header
READ_DATA	reads spectral data
WRITE_HEADER	writes spectrum header
WRITE_DATA	writes spectral data
COPY_DATA	copies spectrum block
LOAD_FILE	loads a file
UNLOAD_FILE	unloads a file
START_MACRO	runs a macro
FILE_PARAMETERS	sets parameter mode to spectrum parameters
OPUS_PARAMETERS	sets parameter mode to OPUS parameters
READ_PARAMETER	reads parameters

12.3.2 CLOSE_PIPE

Syntax:

“CLOSE_PIPE”

Description:

Closes the pipe connection.

Return Value:

“OK”

Comment:

Although it is not strictly required, this command should be send if no further communication with OPUS is necessary. The corresponding program pipe will be closed by OPUS and the resources returned.

12.3.3 COUNT_ENTRIES

Syntax:

“COUNT_ENTRIES”

Description:

Returns the number of data blocks that have been selected in the *Select File* dialog of the *External Program* function.

Return Value:

“OK”

Return Value 2:

<Number of data blocks>

Comment:

This command ensures that all files or data blocks selected in the *Select File* dialog of the *External Program* function can be accessed.

12.3.4 READ_FROM_ENTRY

Syntax:

“READ_FROM_ENTRY <Number>”

Description:

This command specifies the data block accessed by the READ_DATA command.

Return Value:

“OK” or error message.

Error:

“Syntax: READ_FROM_ENTRY <Number>”

“Entry number out of range”

Return Value 2:

<Complete path of the OPUS file>

<File number>

<Data block name>

Comment:

The argument to this command is the number of an entry in the Client/Server file list (between 1 and the number returned from the COUNT_ENTRIES command), from which the client program intends to read. If no error occurs, the complete file name (including drive and path), as well as the data block name of the selected file in text format will be returned as the second return value. The format of the data block name is identical to the one used in the history function.

The file name returned by the command is hyphenated and followed by the number of the copy (clonecount) for further use with the command line.

12.3.5 WRITE_TO_ENTRY

Syntax:

“WRITE_TO_ENTRY <Number>”

Description:

This command specifies the data block accessed by the WRITE_DATA command.

Return Value:

“OK” or error message.

Error:

“Syntax: WRITE_TO_ENTRY <Number>”

“Entry number out of range”

Return Value 2:

<Complete path of the OPUS file>

<File number>

<Data block name>

Comment:

The argument to this command is the number of an entry in the Client/Server file list (between 1 and the number returned from the COUNT_ENTRIES command). If no error occurs, the complete file name (including drive and path), as well as the name of the data block in text format will be returned as the second return value. The name of the data block is returned in the same format used in the history function.

The file name returned by the command is hyphenated and followed by the number of the copy (clonecount) for further use with the command line.

12.3.6 READ_FROM_FILE

Syntax:

“READ_FROM_FILE <Filename> or <File number>”

Description:

Specifies the OPUS file from which the client program intends to read. The argument to this command is the file name which can be specified with or without hyphens. Optionally, the clonecount can be stated. If the file was already loaded in OPUS using this name (including the correct clonecount), this copy will be used. Otherwise, the file will automatically be loaded. For reasons of compatibility to OPUS-OS/2 the file can still be accessed via an internal file number, but this number is no longer limited to the region between 1 to 699.

Return Value:

"OK" or error message.

Error:

"Syntax: READ_FROM_FILE <File name> or <File number>"

"File not Found"

Return Value 2:

<Complete path of the OPUS file>

<File number>

Comment:

Specifies the OPUS file from which the client program intends to read. This command is only able to select a file; the READ_FROM_BLOCK command must subsequently be used to specify the data block in the file, from which to read.

The error message „File not Found" can have multiple causes. In general, it indicates an error while accessing the file.

The file name returned by the command is hyphenated for further use in the command line and is followed by the number of the copy (clonecount).

12.3.7 WRITE_TO_FILE

Syntax:

"WRITE_TO_FILE <File name> or <File number>"

Description:

Specifies the OPUS file to which the client program intends to write. The argument to this command is the file name which can be specified with or without hyphens. Optionally, the clonecount can be stated. If the file was already loaded in OPUS using this name (including the correct clonecount), this copy will be used. Otherwise, the file will automatically be loaded. For reasons of compatibility to OPUS-OS/2 the file can still be accessed via an internal file number, but this number is no longer limited to the region between 1 to 699.

Return Value:

“OK” or error message.

Error:

“Syntax: WRITE_TO_FILE <File name> or <File number>”

“File not Found”

Return Value 2:

<Complete path of the OPUS file>

<File number>

Comment:

Specifies the OPUS file to which the client program intends to write. This command is only able to select a file; the WRITE_TO_BLOCK command must subsequently be used to specify the data block in the file to which to write.

The error message „File not Found“ can have multiple causes. In general, it indicates an error while accessing the file.

The file name returned by the command is hyphenated for further use in the command line and is followed by the number of the copy (Clonecount).

12.3.8 READ_FROM_BLOCK

Syntax:

“READ_FROM_BLOCK <Block name>”

Description:

Specifies the data block from which the client program intends to read. The command always refers to the file that was last specified with the READ_FROM_ENTRY or the READ_FROM_FILE command.

Return Value:

“OK” or error message.

Error:

“Syntax: READ_FROM_BLOCK <Block name>”

“No Filename or Filenumber defined”

“Unknown blocktype”

“Block not found”

Comment:

The argument to the command is the block type which is also used in reports i. e. “AB” for an absorption spectrum, “TR/Multiple” for a transmission block of a 3D file. The command will only be accepted if it was preceded by either the READ_FROM_ENTRY or the READ_FROM_FILE command.

12.3.9 WRITE_TO_BLOCK

Syntax:

“WRITE_TO_BLOCK <Block name>”

Description:

Specifies the data block from which the client program intends to write. The command always refers to the file that was last specified with the WRITE_TO_ENTRY or the WRITE_TO_FILE command.

Return Value:

“OK” or error message.

Error:

“Syntax: WRITE_TO_BLOCK <Block name>”

“No Filename or Filenumber defined”

“Unknown blocktype”

“Block not found”

Comment:

The argument to the command is the block type which is also used in reports, i. e. “AB” for an absorption spectrum, “TR/Multiple” for a transmission block of a 3D file. The command will only be accepted if it was preceded by either the WRITE_TO_ENTRY or the WRITE_TO_FILE command.

12.3.10 ASCII

Syntax:

“ASCII”

Description:

Sets the transfer mode used to transfer data points to ASCII.

Return Value:

“OK”

Comment:

If this mode is chosen (default mode) all data points will be transferred as ASCII text. Each data point is followed by an End of Line sequence.

12.3.11 BINARY

Syntax:

“BINARY”

Description:

Sets the transfer mode used to transfer data points to BINARY.

Return Value:

“OK”

Comment:

If this mode is chosen, all data points will be transferred as 4 byte IEEE floating-point number (REAL*4 in FORTRAN, FLOAT in C). In this mode, the data points will not be terminated. Therefore, the number of bytes transferred is N*4, N being the total number of transferred data points. This mode is faster than the ASCII mode.

12.3.12 DATA_VALUES

Syntax:

“DATA_VALUES”

Description:

The parameters of the READ_HEADER, READ_DATA and COPY_DATA will be interpreted as frequency values.

Return Value:

“OK”

12.3.13 DATA_POINTS

Syntax:

“DATA_POINTS”

Description:

The parameters of the READ_HEADER, READ_DATA and COPY_DATA will be interpreted as data points.

Return Value:

“OK”

Comment:

The data point numbering starts with „1”. Floating-point numbers are always rounded to the next lower integer (e.g. 14.965 will be rounded to 14).

12.3.14 READ_HEADER

Syntax:

“READ_HEADER [<X1>[-<X2>] [<Z1>[-<Z2>]]”

Description:

Reads the header of a spectrum block and returns the frequency range of the spectrum. Several options are available.

Return Value:

“OK” or error message.

Error:

“No Filename or Filenumber defined”

“No Blocktype defined”

“Error Reading File”

“Not implemented”

Return Value 2:

In case of regular spectrum blocks:

<Number of data points (NX = XL - XF + l) in region X>

<Frequency of the first data point in region X>

<Frequency of the last data point in region X>

Return Value 2:

In case of 3D spectrum blocks:

<Number of data points (NX = XL - XF + I) in region X>

<Frequency/number of the first data point in region X>

<Frequency/number of the last data point in region X>

<Number of spectra (NZ = ZL - ZF + 1) in region Z>

<Value (e.g. time) of the first spectrum in region Z>

<Value (e.g. time) of the last spectrum in region Z>

Note:

The output will always be returned as ASCII text, separated by an End of Line sequence, regardless of the selected data transfer mode.

Comment:

Up to four parameters can be forwarded as command arguments.

<X1>, <X2> define the frequency region of the spectrum block. If <X2> is not explicitly stated, only one data point in the vicinity of <X1> will be returned. If no parameters are specified or if <X1> was set to „*", all data stored in the spectrum block will be returned.

<Z1>, <Z2> define the region of the Z axis for which data will be returned (only for 3D files). If <Z2> is not specified, only data in the vicinity of <Z1> will be returned. If no parameters are specified or if <Z1> was set to „*", all data stored in the spectrum block will be returned. In the case of regular spectrum blocks, the parameters <Z1> and <Z2> will be ignored and do not cause an error message in case they have been stated.

All four parameters can either be entered as integer or as floating-point number and will be interpreted either as frequencies or as data points, depending on the settings (see the DATA_VALUES and DATA_POINTS commands).

12.3.15 READ_DATA

Syntax:

“READ_DATA [<X1>[<X2>]] [<Z1>[-Z2]]”

Description:

Reads the header and data points of a spectrum block within the limits indicated. The parameters of the command are similar to the parameters of the READ_HEADER command.

Return Value:

“OK” or error message.

Error:

“No Filename or Filenumber defined”

“No Blocktype defined”

“Error Reading File”

“Not implemented”

Return Value 2:

In case of regular spectrum blocks:

<Number of data points (NX = XL - XF + l) in region X>

<Frequency of the first data point in region X>

<Frequency of the last data point in region X>

<Scaling factor for Y values>

<Y(XF)>, <Y(XF + 1)>, <Y(XF + 2)> ...<Y(XL)>

“OK” or “Error Reading File”

Return Value 2:

In case of 3D spectrum blocks:

<Number of data points (NX = XL - XF + 1) in region X>

<Frequency/number of the first data point (XF) in region X>

<Frequency/number of the last data point (XL) in region X>

<Number of spectra (NZ = ZL - ZF + 1) in region Z>

<Value (e.g. time) of the first spectrum in region Z>

<Value (e.g. time) of the last spectrum in region Z>

<Scaling factor for Y values> for Z = ZF

<Y(XF)>, <Y(XF + 1)>, ... <Y(XL)> for Z = ZF

<Scaling factor for Y values> for Z = ZF+1

<Y(XF)>, <Y(XF + 1)>, ... <Y(XL)> for Z = ZF + 1

<Scaling factor for Y values> for Z = ZF + 2

<Y(XF)>, <Y(XF + 1)>, ... <Y(XL)> for Z = ZF + 2

...

<Scaling factor for Y values> for Z = ZL

<Y(XF)>, <Y(XF + 1)>, ... <Y(XL)> for Z = ZL

“OK1” or “Error Reading File”

Comment:

The header values will always be returned as ASCII text, separated by an End of Line sequence, regardless of the selected data transfer mode. The data points will be returned either as ASCII text, separated by an End of Line sequence, or as floating-point numbers without any separator, depending on the selected data transfer mode. Either „OK” or the error message “Error Reading File” will be appended after the data points.

12.3.16 WRITE_HEADER

Syntax:

“WRITE_HEADER”

Description:

Writes a (new) header for a data block. After the command, the following parameters must be send as ASCII text:

Return Value:

In case of regular spectrum blocks:

<Number of data points (NX = XL - XF + l) in region X>

<Frequency/number of the first data point in region X>

<Frequency/number of the last data point in region X>

In case of 3D spectrum blocks:

<Number of data points (NX = XL - XF + I) in region X>

<Frequency/number of the first data point in region X>

<Frequency/number of the last data point in region X>

<Number of spectra (NZ = ZL - ZF + 1) in region Z>

<Value (e.g. time) of the first spectrum in region Z>

<Value (e.g. time) of the last spectrum in region Z>

“OK” or error message.

Error:

“No Filename or Filenumber defined”

“No Blocktype defined”

“Not implemented”

Comment:

This command serves to edit existing data block headers. Especially, pay attention to the number of data points (especially in Z direction): the number of data points specified must match the actual number of data points stored in the data block. Otherwise, a shift of the data will result.

12.3.17 WRITE_DATA

Syntax:

“WRITE_DATA”

Description:

Writes the header and data points into a data block. After the command, the following parameters must be send:

In case of regular spectrum blocks:

<Number of data points (NX = XL - XF + l) in region X>

<Frequency/number of the first data point (XF) in region X>

<Frequency/number of the last data point (XL) in region X>

<Scaling factor for Y-Values> <Y(XF)>, <Y(XF + 1)>, <Y(XF + 2)> ...<Y(XL)>

In case of 3D spectrum blocks:

<Number of data points (NX = XL - XF + I) in region X>

<Frequency/number of the first data point (XF) in region X>

<Frequency/number of the last data point (XL) in region X>

<Number of spectra (NZ = ZL - ZF + 1) in region Z>

<Value (e.g. time) of the first spectrum in region Z>

<Value (e.g. time) of the last spectrum in region Z>

<Scaling factor for Y values> for Z = ZF <Y(XF)>, <Y(XF + I)>, <Y(XI,)> for Z = ZF

<Scaling factor for Y values> for Z = ZF + 1

<Y(XF)>, <Y(XF + 1)>, ... <Y(XL)> for Z = ZF + 1

<Scaling factor for Y values> for Z = ZF + 2

<Y(XF)>, <Y(XF + 1)>, ... <Y(XL)> for Z = ZF + 2

...

<Scaling factor for Y values> for Z = ZL

<Y(XF)>, <Y(XF + I)>, ... <Y(XL)> for Z = ZL

Return Value:

“OK” or error message.

Error:

“No Filename or Filenumber defined”

“No Blocktype defined”

“Not implemented”

“Error Accessing Data”

Return Value 2:

After the header and all data points have been read by OPUS, either “OK” or an error message will be returned.

Comment:

The header values must always be sent as ASCII text, separated by an End of Line sequence, regardless of the selected data transfer mode. The data points must be returned either as ASCII text, separated by an End of Line sequence, or as floating-point numbers without any separator, depending on the selected data transfer mode.

12.3.18 COPY_DATA

Syntax:

“COPY_DATA [<X1>[-<X2>]] [<Z1>[-<Z2>]]”

Description:

Copies data points from a data block specified by one of the commands READ_FROM_ENTRY or READ_FROM_FILE and READ_FROM_BLOCK to a data block specified by either the WRITE_TO_ENTRY or WRITE_TO_FILE and WRITE_TO_BLOCK command (for parameters see READ_HEADER).

Return Value:

After receiving the command:

“OK” or error message.

Return Value 2:

After processing the command:

“OK” or error message.

Error:

“No Filename or Filenumber defined”

“No Blocktype defined”

“Not implemented”

“Error Reading File”

Comment:

The copy process take place within OPUS. Therefore, no data points are transferred via a pipe.

12.3.19 LOAD_FILE

Syntax:

“LOAD_FILE <File name>”

Description:

Loads the indicated file into OPUS.

Return Value:

“OK” or error message.

Error:

“Syntax: LOAD_FILE <File name>”

“Error reading file”

Return Value 2:

<Path and name of the file>

<File number>

Comment:

OPUS loads the file even if it has already been loaded before. In this case another copy (clone) is generated.

The file name returned by the command is hyphenated for further use in the command line and is followed by the number of the copy (Clonecount).

12.3.20 UNLOAD_FILE

Syntax:

“UNLOAD_FILE <File name> or <File number>”

Description:

Unloads a file from OPUS selection line. The argument to this command is the file name (including clonecount).

Return Value:

"OK" or error message.

Error:

"Syntax: UNLOAD_FILE <File name> or <File number>"

"File not loaded"

Return Value 2:

<Path and name of the file>

<File number>

Comment:

OPUS unloads the selected file. The complete path and file name, as well as the entry number will be returned. If the path of the file is not specified, OPUS searches the "Data Path" directory for the file.

For reasons of compatibility to OPUS-OS/2 the file can still be accessed via an internal file number, but this number is no longer limited to the region between 1 to 699.

The file name returned by the command is hyphenated for further use in the command line and is followed by the number of the copy (clonecount).

12.3.21 START_MACRO

Syntax:

"START_MACRO <Macro file name>[<Number of input parameters>]"

Description:

Starts an OPUS macro. Input parameters can be forwarded to the macro. If parameters are exchanged, the total number of parameters must be defined as the second parameter. If this number is omitted, then it will be set to 0; in this case, no parameters are read. If the number of input parameters is larger than 0, the input parameters <input parameter 1>, <input parameter 2>, ..., <input parameter N> have to be sent by the client program. In addition, the macro can return parameter values to the client program.

Return Value:

Immediately after the command execution (i.e. directly after the starting the macro):

"OK" or error message.

Return Value 2:

After macro execution:

“OK” or error message.

Return Value 3:

only if the macro returned parameters:

<Number of return value parameters>

<Return value parameter 1>

< Return value parameter 2>

...

<Return value parameter N>

Return Value 4:

only if the macro returned parameters:

“OK” or error message.

Error:

“Syntax: START_MACRO <Macro File> <#Parameter>”

“Error in Opus Command Line Execution - ID: %d”

Comment regarding the command:

When executing the command the following order has to be maintained:

- Send command including the macro name and the number of input parameters (optional).
- Read return value: “OK” or error message.
- Send input parameter.
- Read return value: “OK” or error message.
- Read return value parameter.

The individual input parameters must be separated by End of Line sequences.

Macro parameters can also directly follow a command; in this case the second “OK” or error message will not be send.

Comment regarding the macros:

A structure similar to sub macro calls is used to control client programs. Input parameters will be transferred from the client program to the macro using a dialog box, that must be located in the first line of the respective macro. Return values are returned via another dialog box located in the last line of the macro. As in the case of a sub macro call, both dialog boxes will not be displayed. If OPUS cannot find a dialog box in the first macro line, the macro will be started without exchanging parameters, even if they have been sent to OPUS.

In the first dialog box, the input parameters will be assigned from top to bottom; only variables of type FILE, TEXT FOR EDIT, NUMERIC, TEXT FOR OUTPUT or CHECK BOX are allowed. Empty lines and variables of type BUTTON and COMBOBOX will be ignored. If the number of input paramteres exchanged is not equal to the number of variables in the dialog box, OPUS terminates the assignment either after all input parameters have been read or if all macro variables have been assigned. ASCII input paramters will automatically be converted into the format of the macro variable. Accordingly, the return values will be transformed by the last dialog box in the macro from top to bottom into ASCII text, and, delimited by an End of Line character, returned to the client program. Here also, empty lines and variables of type BUTTON and COMBOBOX will be ignored. If no dialog box can be found in the last line of the macro (or if the dialog box is empty), OPUS returns “0” as number of return value parameters immediately after starting the macro. Communication will be resumed without waiting for the macro to terminate.

12.3.22 FILE_PARAMETERS

Syntax:

“FILE_PARAMETERS”

Description:

After this command, the READ_PARAMETER command reads a parameter from the data block of a file specified by the commands READ_FROM_ENTRY, READ_FROM_FILE or READ_FROM_BLOCK.

Return Value:

“OK”

Comment:

This is the default setting for the READ_PARAMETER command.

12.3.23 OPUS_PARAMETERS

Syntax:

“OPUS_PARAMETERS”

Description:

After this command, READ_PARAMETER reads a parameter from the OPUS default parameter set.

Return Value:

“OK”

12.3.24 READ_PARAMETER

Syntax:

“READ_PARAMETER <Parameter name>”

Description:

Reads a parameter either from a specified data block of an OPUS file or from the standard OPUS parameter set.

Return Value:

“OK” or error message.

Error:

“Syntax: READ_PARAMETER <parameter name>”

“No Filename or Filenumber defined”

“No Blocktype defined”

“Parameter not found”

“Invalid Parameter Name”

Return Value 2:

`<Parameter value>`

Comment:

The parameter name forwarded as argument to the command consists of a three-character abbreviation (see chapter 11). After the confirmation by OPUS, the parameter value will be transferred as ASCII text.

12.3.25 WRITE_PARAMETER

Syntax:

“WRITE_PARAMETER <Parameter name> <Parameter value>”

Description:

The WRITE_PARAMETER command writes a parameter or changes an existing one in the OPUS file specified by either READ_FROM_ENTRY or READ_FROM_FILE and READ_FROM_BLOCK.

Return Value:

“OK” or error message.

Error:

“Syntax: WRITE_PARAMETER <Parameter name> <Parameter value>”

“No Filename or Filenumber defined”

“No Blocktype defined”

“Parameter not found”

“Invalid Parameter Name”

Comment:

The parameter name forwarded as argument to the command consists of a three-character abbreviation.

The parameter value will be forwarded as ASCII text file, i.e. numerical values have to be converted to ASCII strings.

12.3.26 RUN_MACRO

Syntax:

Equivalent to START_MACRO

Description:

The RUN_MACRO command starts a macro. Contrary to START_MACRO, the control is returned immediately after the macro was started. The RUN_MACRO command does not wait for the macro to terminate and also doesn’t return any results.

Return Value:

After the command:

“OK” or error message.

Return Value 2:

After transferring the input parameter:

“OK” or error message.

Return Value 3:

<MacroID>: a macro identification number unique for each macro session.

Error:

Similar to START_MACRO

Comment:

See also START_MACRO.

To access the results of the macro started, the MACRO_RESULTS commands is used.

The returned <MacroID> is used as parameter for the MACRO_RESULTS and the KILL_MACRO commands.

12.3.27 MACRO_RESULTS

Syntax:

“MACRO_RESULTS <MacroID>”

Description:

The MACRO_RESULTS command retrieves the result parameters of a macro session that was started with the ID <MacroID>, using the RUN_MACRO command.

Return Value:

“OK” or error message.

Return Value 2:

0 or 1 to indicate whether the macro has already finished or is still running.

Return Value 3:

Containing the results, if the macro was terminated. For a format description see START_MACRO.

Error:

“Syntax: MACRO_RESULTS <MacroID>”

“Invalid Macro ID”

Comment:

In combination with the RUN_MACRO command, this command allows client programs to run different tasks while the macro is still running. Use this command to frequently check, whether the macro has finished and to obtain the return parameters.

12.3.28 KILL_MACRO

Syntax:

“KILL_MACRO <MacroID>”

Description:

KILL_MACRO terminates a macro session started by RUN_MACRO with the specified macro ID.

Return Value:

“OK” or error message.

Error:

“Syntax: KILL_MACRO <MacroID>”

“Invalid Macro ID”

Comment:

In combination with the RUN_MACRO command this command allows client programs to run different tasks while the macro is still running. Under certain conditions a client program can use this command to stop a macro that is still running. This corresponds to the *Abort Task* command of the OPUS task bar.

12.4 Obsolete Commands

The following commands are only supported out of compatibility reasons to OPUS-OS/2. Due to the different concept of OPUS-NT, they are no longer of any practical importance.

12.4.1 OVERWRITE

Syntax:

“OVERWRITE”

Description:

Allows the subsequent commands to overwrite files and data blocks.

Return Value:

“OK”

Comment:

Subsequent to this command, the following commands are allowed to overwrite files and data blocks:

WRITE_TO_ENTRY

WRITE_TO_FILE

WRITE_TO_BLOCK

12.4.2 PRESERVE

Syntax:

“PRESERVE”

Description:

Prevents files and data blocks from being replaced.

Return Value:

“OK” or error message.

Error:

“Set OVERWRITE mode to replace blocks”

Comment:

Subsequent to this command, the following commands cannot replace existing files and data blocks:

WRITE_TO_ENTRY

WRITE_TO_FILE

WRITE_TO BLOCK

If an existing data block was specified in a WRITE_TO_BLOCK command, OPUS returned the message “Set OVERWRITE mode to replace blocks”.

In case of a WRITE_TO_ENTRY or WRITE_TO_FILE command, the file name extension was incremented until the first non-existing file was obtained.

Example:

Assume the files TEST.2 and TEST.3 already exist in the current OPUS\DATA directory. The “WRITE_TO_FILE TEST.1” command is sent twice. The first time the command is executed and generates the file TEST.1. The second time, the file name extension is incremented until the first non-existing file name is obtained (TEST.1), because it is not allowed to replace the now existing file TEST.1.

12.4.3 TIMEOUT

Syntax:

“TIMEOUT <Delay>”

Description:

Sets a delay time (in seconds) for the pipe, which may not be replaced during read and write processes.

Return Value:

“OK” or error message.

Error:

“Invalid time limit”

“Syntax: TIMEOUT<Seconds>”

Comment:

The delay is an integer between l and 1000. Without this command the default value of 10 seconds will be used.

12.5 New Commands

The first view commands of this section serve to further specify the binary transfer mode. They mainly concern the data exchange with scripts. Because scripts allow no direct memory access, the data must be enclosed in a variable field to allow binary data exchange. Hence, the single elements are assigned a certain type: BYTE_MODE, INT_MODE, FLOAT_MODE, and DOUBLE_MODE allow to define, whether the binary OPUS data will be contained in a BYTE, INTEGER, FLOAT, or DOUBLE field in a script.

In case of a pipe, the respective memory region can be transferred directly, which then will be interpreted on the receiving side.

No binary return values are allowed when using DDE connections; these are available in the HEXSTRING_MODE.

12.5.1 BYTE_MODE

Syntax:

“BYTE_MODE”

Description:

Sets the binary transfer mode to single bytes.

Return Value:

“OK”

12.5.2 INT_MODE

Syntax:

“INT_MODE”

Description:

Sets the binary transfer mode to integer.

Return Value:

“OK”

12.5.3 FLOAT_MODE

Syntax:

“FLOAT_MODE”

Description:

Sets the binary transfer mode to floating-point numbers.

Return Value:

“OK”

12.5.4 DOUBLE_MODE

Syntax:

“DOUBLE_MODE”

Description:

Sets the binary transfer mode to double-precision.

Return Value:

“OK”

12.5.5 HEXSTRING_MODE

Syntax:

“HEXSTRING_MODE”

Description:

Sets the binary transfer mode to text.

Return Value:

“OK” or error message.

Comment:

DDE connection default settings for binary mode.

The data is converted to individual strings of numbers, depending on the mode chosen (BYTE_MODE, INT_MODE, FLOAT_MODE and DOUBLE_MODE) and will be transmitted as text.

12.5.6 FLOATCONV_MODE

Syntax:

“FLOATCONV_MODE ON|OFF”

Description:

Switches the conversion of floating-point numbers on and off, when using binary transfer mode.

Return Value:

“OK” or “ON|OFF”

Comment:

When using a pipe for binary data transfer under OS/2, a scaling factor was transferred prior to the actual data. This factor was also transferred binary, but compared to the data transfer at double-precision (8 instead of 4 bytes). In OPUS-NT, this factor is found in the first element of the returned field.

If FLOATCONV_MODE is not selected for the binary data transfer to a script, the first 8 bytes of data (the double-precision scaling factor) will be misinterpreted as two single-precision floating-point numbers.

If neither „ON“ nor „OFF“ is forwarded as parameter the return value text provides the current settings.

12.5.7 GET_DISPLAY

Syntax:

“GET_DISPLAY”

Description:

Provides an identification number of the currently active display window.

Return Value:

“OK”

Return Value 2:

<WindowID>

Comment:

The number returned can be used as parameter for the SET_WINDOW, CLOSE_WINDOW, and POSITION_WINDOW commands.

12.5.8 SET_WINDOW

Syntax:

“SET_WINDOW <WindowID>”

Description:

The window specified by the identification number will be promoted to be the active display window for the current C/S session.

Return Value:

“OK” or error message.

Error:

“Syntax: SET_WINDOW <Window>”

Comment:

If new files are loaded or generated by another OPUS function, they will be displayed in the currently active window. The function is used to define this window.

12.5.9 NEW_WINDOW

Syntax:

“NEW_WINDOW <Window type>”

Description:

Creates a new window of the type specified.

Return Value:

“OK” or error message.

Error:

“Syntax: NEW_WINDOW <Window type>”

“Error creating View”

Comment:

The window type defines, that for example a new report window will be generated.

12.5.10 CLOSE_WINDOW

Syntax:

“CLOSE_WINDOW <WindowID>”

Description:

Closes the window specified by the <WindowID>.

Return Value:

“OK” or error message.

Error:

“Syntax: CLOSE_WINDOW <Window>”

Comment:

The parameter <WindowID> necessary to address the display window can result from either NEW_WINDOW or from GET_DISPLAY.

12.5.11 POSITION_WINDOW

Syntax:

“POSITION_WINDOW <WindowID> <x> <y> <cx> <cy>”

Description:

Positions the display window specified by <WindowID> at the coordinates <x>, <y> and re-sizes it to <cx>, <cy>.

Return Value:

“OK” or error message.

Error:

“Syntax: POSITION_WINDOW <Window> <x> <y> <cx> <cy>\n”

Comment:

The parameter <WindowID>, necessary to address the display window, can result from either NEW_WINDOW or from GET_DISPLAY.

12.5.12 GET_LANGUAGE

Syntax:

“GET_LANGUAGE”

Description:

Retrieves the current language settings of OPUS-NT. The language is set using the command line argument /LANGUAGE when starting OPUS.

Return Value:

“OK”

Return Value 2:

<Language>

Comment:

The name of the language will be returned as text.

12.5.13 GET_OPUSPATH

Syntax:

“GET_OPUSPATH”

Description:

Retrieves the path of the currently running OPUS program.

Return Value:

“OK”

Return Value 2:

<Path>

Comment:

The path can be checked in the *User Settings* dialog box of the *Setup* OPUS pull-down menu.

12.5.14 GET_BASEPATH

Syntax:

“GET_BASEPATH”

Description:

Retrieves the default path of the currently logged in user.

Return Value:

“OK”

Return Value 2:

<Path>

Comment:

The path is set in the *User Settings* dialog box of the *Setup* OPUS pull-down menu.

12.5.15 GET_DATAPATH

Syntax:

“GET_DATAPATH”

Description:

Retrieves the data path of the currently logged in user.

Return Value:

“OK”

Return Value 2:

<Path>

Comment:

The path is set in the *User Settings* dialog box of the *Setup* OPUS pull-down menu.

12.5.16 GET_WORKPATH

Syntax:

“GET_WORKPATH”

Description:

Retrieves the path for work files of the currently logged in user.

Return Value:

“OK”

Return Value 2:

<Path>

Comment:

The path is set in the *User Settings* dialog box of the *Setup* OPUS pull-down menu.

12.5.17 GET_USERNAME

Syntax:

“GET_USERNAME”

Description:

Retrieves the name of the currently logged in user.

Return Value:

“OK”

Return Value 2:

<Name>

Comment:

The user account is set in the *User Settings* dialog box of the *Setup* OPUS pull-down menu.

12.5.18 GET_BENCH

Syntax:

“GET_BENCH”

Description:

Retrieves the configuration file of the currently selected spectrometer.

Return Value:

“OK”

Return Value 2:

<OpticsFile>

12.5.19 UPDATE_BENCH

Syntax:

“UPDATE_BENCH <OpticsFile>”

Description:

Triggers OPUS to initialize the optics configuration using the settings stored in the <OpticsFile>.

Return Value:

“OK” or error message.

Error:

“Syntax: UPDATE_BENCH <inifile>”

“RebuildParmText error”

12.5.20 COMMAND_SAY

Syntax:

“COMMAND_SAY <Text>”

Description:

Returns the transferred commands in text format.

Return Value:

<Text>

Comment:

This command serves to test the communication between OPUS and the client program. It can also be used to forward parameters to scripts. To do this, call the OpusCommand function of a form created with the OpenForm command (or selected with FormByName), and forward parameters using COMMAND_SAY <Parameter>. The form receives the parameter with OnOpusResult <Parameter>.

12.5.21 REPORT_INFO

Syntax:

“REPORT_INFO”

Description:

Retrieves information about the number of main and sub reports of an OPUS report block.

Return Value:

“OK” or error message.

Return Value 2:

<#Main reports>

<#Sub reports 1>

...

<#Sub reports N>

Error:

“No Filename or Filenumber defined”

“No Blocktype defined”

“Error Reading Report”

Comment:

First, the total number of main reports is returned, followed by the number of sub reports contained in each main report. Each line holds only one number.

The information is obtained from the OPUS report block selected by the READ_FROM_FILE, READ_FROM_ENTRY and READ_FROM_BLOCK commands.

12.5.22 HEADER_INFO

Syntax:

“HEADER_INFO <Main report> <Sub report>”

Description:

Returns the number of lines in an OPUS report block header.

Return Value:

"OK" or error message.

Return Value 2:

<Lines>

Error:

"No Filename or Filenumber defined"

"No Blocktype defined"

"Error Reading Report"

Comment:

If no sub report is specified, the number of lines in the header of the main report block is returned instead. If also no main report is specified, the first main report will be taken.

The information is obtained form the OPUS report block selected by the READ_FROM_FILE, READ_FROM_ENTRY and READ_FROM_BLOCK commands.

12.5.23 MATRIX_INFO

Syntax:

"MATRIX_INFO <Main report> <Sub report>"

Description:

Returns the dimension (number of rows and columns) of a matrix stored in an OPUS report block.

Return Value:

"OK" or error message.

Return Value 2:

<Rows>

<Columns>

Error:

"No Filename or Filenumber defined"

“No Blocktype defined”

“Error Reading Report”

Comment:

If no sub report is specified, the number of rows in the main reports’ header will be returned. If also no main report is specified, the first main report will be taken.

The information is obtained form the OPUS report block selected by the READ_FROM_FILE, READ_FROM_ENTRY, and READ_FROM_BLOCK commands.

12.5.24 MATRIX_ELEMENT

Syntax:

“MATRIX_ELEMENT <Main report> <Sub report> <Row> <Column>”

Description:

Reads an element from a data matrix of an OPUS report block. The index of the main/sub report as well as the index of the row and column has to be indicated.

Return Value:

“OK” or error message.

Return Value 2:

<MatrixElement>

Error:

“Syntax: MATRIX_ELEMENT <MainReport> <SubReport> <Row> <Column>”

“No Filename or Filenumber defined”

“No Blocktype defined”

“Error Reading Report”

Comment:

If the main report should be accessed, the sub report number must be set to “0”.

Determine the total number of rows and columns, using the MATRIX_INFO command.

All values are converted to text format prior to the transfer, regardless of the data format of the element.

The information is obtained form the OPUS report block selected by the READ_FROM_FILE/READ_FROM_ENTRY, and READ_FROM_BLOCK commands.

12.5.25 HEADER_ELEMENT

Syntax:

“HEADER_ELEMENT <Main report> <Sub report> <Row>”

Description:

Reads an element from the OPUS report block header. The index of the main/ sub report as well as the number of the row has to be indicated.

Return Value:

“OK” or error message.

Return Value 2:

<ElementName>

<ElementContent>

Error:

“Syntax: HEADER_ELEMENT <MainReport> <SubReport> <Row>”

“No Filename or Filenumber defined”

“No Blocktype defined”

“Error Reading Report”

Comment:

The name of the feature in the selected header row and its value will be returned. If the main report should be accessed, the sub report number must be set to “0”.

Determine the total number of rows and columns using the HEADER_INFO command.

All values are converted to text format prior to the transfer, regardless of the data format of the element.

The information is obtained form the OPUS report block selected by the READ_FROM_FILE/READ_FROM_ENTRY and READ_FROM_BLOCK commands.

12.5.26 COMMAND_MODE

Syntax:

“COMMAND_MODE”

Description:

Sets the mode for processing a command line to COMMAND_MODE. This mode runs commands and programs in the background and returns a message after termination of the program.

Return Value:

“OK”

Comment:

Usually, this mode doesn’t need to be explicitly set, since these modi are predefined for the different transfer types and interfaces or alternatively are set by different calls (like OpusExecute).

12.5.27 EXECUTE_MODE

Syntax:

“EXECUTE_MODE”

Description:

Sets the mode for processing a command line to EXECUTE_MODE. This mode runs commands and programs in the background, but does not wait for the programs to terminate. No message will be returned when a program has finished.

Return Value:

“OK”

Comment:

Usually, this mode doesn’t need to be explicitly set, since these modi are predefined for the different transfer types and interfaces or alternatively are set by different calls (like OpusExecute).

12.5.28 REQUEST_MODE

Syntax:

“REQUEST_MODE”

Description:

Sets the mode for processing a command line to REQUEST_MODE. This mode does not run commands and programs in the background, but waits for the programs to terminate. The result will be returned as soon as the program terminates.

Return Value:

“OK”

Comment:

Usually, this mode doesn’t need to be explicitly set, since these modi are predefined for the different transfer types and interfaces or alternatively are set by different calls (like OpusExecute).

12.5.29 CLOSE_OPUS

Syntax:

“CLOSE_OPUS”

Description:

Terminates OPUS.

Return Value:

No return values.

Comment:

This operation is similar to closing the OPUS user interface window.

12.5.30 TAKE_REFERENCE

Syntax:

“TAKE_REFERENCE <Experiment file>”

Description:

Performs a reference measurement using the specified <Experiment file>.

Return Value:

“OK” or error message.

Error:

“Error in Opus Command Line Execution - ID: %d”

12.5.31 MEASURE_SAMPLE

Syntax:

“MEASURE_SAMPLE <Experiment file>”

Description:

Performs a sample measurement using the specified <Experiment file> and returns the acquired spectral data as text.

Return Value:

“OK” or error message.

Return Value 2:

Result File:<File number>

<File name>

Block: <Block type>

<UnitsX>

<UnitsY>

Points: <Number of points>

<x1> <y1>

...

<xn> <yn>

Error:

“Error in Opus Command Line Execution - ID: %d”

Comment:

All blocks of the new file (specified by the experiment file) are transmitted in succession as data point tables.

12.5.32 COMMAND_LINE

Syntax:

“COMMAND_LINE <Command line>”

Description:

Calls an OPUS function as command lines.

Return Value:

“OK” or error message.

Return Value 2:

Only in combination with COMMAND_MODE

<ThreadID>

Error:

“Error in Opus Command Line Execution - ID: %d”

Comment:

In this exception, the keyword COMMAND_LINE can be omitted, because OPUS tries to interpret all unknown C/S commands in command line notation.

The actual type of command processing depends on the call of the command (in case of scripts for example OpusExecute), or the settings made by COMMAND_MODE, EXECUTE_MODE, and REQUEST_MODE.

If COMMAND_MODE was selected, an identification number is supplied for the background calculation, which can be used to abort the function in case of STOP_THREAD.

12.5.33 STOP_THREAD

Syntax:

“STOP_THREAD <ThreadID>”

Description:

Terminates a OPUS processing function which was started by the COMMAND_LINE function while COMMAND_MODE was selected.

Return Value:

"OK" or error message.

Error:

"Syntax: STOP_THREAD <ThreadID>"

Comment:

In COMMAND_MODE, COMMAND_LINE starts the function in the background and returns an identification number. This number can be used to abort the function. This is similar to the *Abort Task* command of the task manager.

Note: Aborting a program may result in data loss and produce corrupt OPUS files. Therefore, it should only be used in emergencies.

12.5.34 ACTIVATE_DIALOG

Syntax:

"ACTIVATE_DIALOG <Command line>"

Description:

Starts the dialog box of an OPUS function.

Return Value:

"OK" or error message.

Error:

"Syntax: ACTIVATE_DIALOG CommandLine()"

Comment:

Opening an OPUS function dialog box within another program usually is not very practical, since the program cannot control the dialog box once it has been opened. A command line is required as a parameter similar to direct command processing.

12.5.35 LOAD_EXPERIMENT

Syntax:

"LOAD_EXPERIMENT <Experiment file>"

Description:

Loads an experiment file in OPUS and sets the parameters for subsequent data acquisitions.

Return Value:

“OK” or error message.

Error:

“Syntax: LOAD_EXPERIMENT <parameter file>”

“Unable to load Experiment file”

Comment:

This command is similar to the respective function of the OPUS *Measurement* dialog box.

12.5.36 GET_USERRIGHTS

Syntax:

“GET_USERRIGHTS”

Description:

Retrieves the rights of the current user.

Return Value:

“OK”

Return Value2:

A list of user rights separated by semicolons or “No Rights”

Comment:

Allows to adjust programs/scripts to perform different actions, depending on the user rights.

12.5.37 PACKET_AVAILABLE

Syntax:

“PACKET_AVAILABLE <Packet name>”

Description:

Tests if certain OPUS software packages are installed on a computer.

Return Value:

“Yes”, “No” or error message.

Error:

“Syntax: PACKET_AVAILABLE <Packet name>”

Comment:

Allows a script or program to determine, whether a software package or an OPUS function is available at all. This applies to QUANT, SEARCH, 3D etc..

12.5.38 GET_CLIENTAREA

Syntax:

“GET_CLIENTAREA”

Description:

Retrieves the available window size of the OPUS main window. This is dependent on the chosen screen resolution. The result can be used for the positioning of script forms and spectrum windows etc.

Return Value:

“OK”

Return Value2:

<width> <height>

Comment:

The returned values can be used as parameters for POSITION_WINDOW.

12.5.39 ACTIVATE_DISPLAY

Syntax:

“ACTIVATE_DISPLAY” <WindowID>

Description:

A spectrum window can be activated using this command. It will then be displayed in the front. The window specified by the ID number will then be the active window for displaying the spectra.

Return Value:

“OK” or error message

Error:

“Syntax: ACTIVATE_DISPLAY <window>

Comment:

If new files are loaded or created by other OPUS functions, they will then be displayed in the currently active window. The active window can be determined with this function. Whereas SET_WINDOW is only valid for files used in script, here the active window e.g. for manual loading can be set.

12.5.40 GET_LIMITS

Syntax:

“GET_LIMITS <WindowID>”

Description:

Lists the actual display limits of the window.

Return Value:

“OK” or error message

Return Value 2:

<X1> <Y1> <X2> <Y2>

Error:

“Syntax: GET_LIMITS <window>”

Comment:

The <WindowID> can either be a result of NEW_WINDOW or GET_DISPLAY

12.5.41 SET_LIMITS

Syntax:

“SET_LIMITS <WindowID> <X-start> <X-end> <Y-start> <Y-end>”

Description:

Sets the display limits of the window to the given values. this is useful to e.g. enlarge certain areas of the spectrum automatically. The four values determine the coordinates for the new display limits.

Return Value:

“OK” or error message

Error:

“Syntax: SET_LIMITS <window> <xsp> <xep> <ymn> <ymx>”

Comment:

The <WindowID> can either be a result of NEW_WINDOW or GET_DISPLAY

12.5.42 DISPLAY_BLOCK

Syntax:

“DISPLAY_BLOCK <WindowID> <color>”

Description:

Displays a datablock of an OPUS file selected by the commands READ_FROM_ENTRY, READ_FROM_FILE or READ_FROM_BLOCK in a display window determined by <windowID>. <color> determines the color of the curve as RGB value.

Return Value:

“OK” or error message

Error:

“No Filename or Filenumber defined”

“No Blocktype defined”

“Syntax: DISPLAY_BLOCK <window> <color>”

Comment:

The <WindowID> can either be a result of NEW_WINDOW or GET_DISPLAY

12.5.43 UNDISPLAY_BLOCK

Syntax:

“UNDISPLAY_BLOCK <WindowID>”

Description:

Removes a datablock of an OPUS file specified by READ_FROM_ENTRY, READ_FROM_FILE or READ_FROM_BLOCK from the window identified by <WindowID>.

Return Value:

“OK” or error message

Error:

“No Filename or Filenumber defined”

“No Blocktype defined”

“Syntax: UNDISPLAY_BLOCK <window>”

Comment:

The <WindowID> can either be a result of NEW_WINDOW or GET_DISPLAY

12.5.44 ENUM_STRINGS

Syntax:

“ENUM_STRINGS <parametername>

Description:

Possible values for a parameter of type ENUM can be requested at the given conditions, e.g. depending on the spectrometer.

Return Value:

“OK” or error message

Return Value 2:

<number of the following valid strings>

<first parameterstring>

...

<last parameterstring>

Error:

"Syntax: ENUM_STRINGS <parameter name>"

"Invalid Parameter Name"

"No Enum Strings"

Comment:

The parameter name transferred as an argumentis a three letter abbrevation of a parameter.

12.5.45 GET_VERSION

Syntax:

"GET_VERSION>"

Description:

Returns the version of the currently running OPUS NT program.

Return Value:

"OK"

Return Value 2:

<Version>

Error:

Comment:

Enables the reaction on and the controlling of, different current OPUS versions from *one* program.

12.5.46 ASK_THREAD

Syntax:

"ASK_THREAD <ProcessID> <special command>

Description:

Enables the interprocess communication of an external program with a running Opus function.

Return Value:

“OK” or error message

Return Value 2:

depending on the transferred command

Error:

“ASK_THREAD failed”

“Invalid Thread ID”

Comment:

This direct communication with currently running OPUS functions is intended only for very special applications. It is mentioned here only for the sake of completeness. However it is actually reserved to internal programming and is used for the coupling with other instruments. One receives the ProcessID either when starting the function in the COMMAND_MODE or through FIND_FUNCTION

12.5.47 FIND_FUNCTION

Syntax:

“FIND_FUNCTION <function name>”

Description:

Determines whether a certain OPUS function is executed in the background. The returned ID can be used to stop the function or to communicate with it (if supported).

Return Value:

“OK” or error message

Return Value 2:

<ProcessID> for identification

Error:

“Syntax: FIND_FUNCTION <FunctionName> or <ThreadID>

“Function not found”

Comment:

This direct communication with currently running OPUS functions is intended only for very special applications. It is mentioned here only for the sake of completeness. However it is actually reserved to internal programming and is used for the coupling with other instruments.

12.5.48 WORKBOOK_MODE

Syntax:

“WORKBOOK_MODE ON|OFF”

Description:

Turns the tabs for switching between different windows at the bottom of the OPUS window on or off.

Return Value:

“OK” or ON|OFF”

Error:

Comment:

When the buttons are deactivated, switching between different windows is no longer possible. In the case that a simple user interface is required, one prevents thereby deviations from the operational sequence intended.

12.5.49 GET_SELECTED

Syntax:

“GET_SELECTED”

Description:

Supplies the names of the selected (red bordered) files.

Return Value:

“OK” or error message

Return Value 2:

<name of selected files>

Error:

“Error while getting file info”

Comment:

The normal behavior of OPUS, to automatically select marked files for the processing functions, is extended to self-written extension functions.

12.5.50 LIST_BLOCKS

Syntax:

“LIST_BLOCKS”

Description:

Lists all available spectral data blocks of the OPUS file delivered by the command READ_FROM_ENTRY or READ_FROM_FILE.

Return Value:

“OK” or error message

Return Value 2:

<number of block names>

<first block name>

...

<last block name>

Error:

“No Filename or Filenumber defined”

“Error getting blocks”

Comment:

Enables to determine which blocks are containing an unknown file and then work with the correct ones accordingly.

12.5.51 SHOW_TOOLBAR

Syntax:

“SHOW_TOOLBAR <toolbar>”

Description:

Shows a toolbar. Valid parameters are:

MENU, STANDARD, COMMANDLINE, PLE, DISPLAY, MEASURE, MANIPULATE, EVALUATE, MDISPLAY, PLOT_PRINT, MACRO, INFO, USER, SETUP, FILE, BROWSER, STATUSBAR

Return Value:

"OK" or "Already visible"

Error:

"Syntax: SHOW_TOOLBAR <toolbarID>"

"Unknown Toolbar"

Comment:

Warning: The adjustments on the desktop made with this function are stored when leaving OPUS. In order to avoid unwanted effects, all modifications of the original configuration should be cancelled again before the final termination of the self-written program!

12.5.52 HIDE_TOOLBAR

Syntax:

"HIDE_TOOLBAR <toolbar>"

Description:

Hides a toolbar. Valid parameters are:

MENU, STANDARD, COMMANDLINE, PLE, DISPLAY, MEASURE, MANIPULATE, EVALUATE, MDISPLAY, PLOT_PRINT, MACRO, INFO, USER, SETUP, FILE, BROWSER, STATUSBAR

Return Value:

"OK" or "Already visible"

Error:

"Syntax: HIDE_TOOLBAR <toolbarID>"

"Unknown Toolbar"

Comment:

Warning: The adjustments on the desktop made with this function are stored when leaving OPUS. In order to avoid unwanted effects, all modifications of the original configuration should be cancelled again before the final termination of the self-written program!

12.5.53 QUICK_PRINT

Syntax:

“QUICK_PRINT”

Description:

Activates the function “Quickprint”. The currently active window will be printed.

Return Value:

“OK”

Error:

Comment:

To print a certain window with this function, it has to be activated with ACTIVATE_DISPLAY first.

13 Script Commands

In this chapter you find a list of all commands that are available for scripts in OPUS. They are sorted according to the following categories: commands interpreted by OPUS, native VBScript commands and the functions of the objects involved.

13.1 The C/S Interpreter

From within a script all commands of the Client/Server interpreter described in chapter 12 are available. This includes all command line calls and also all commands that can be transferred via a DDE connection or a pipe, as well as VBScript functionalities.

13.2 VBScript Language

In the following you will find a simple tutorial that should make you familiar with the general element of the VBScript language and their use.

13.2.1 VBScript Data Types

In VBScript only one data type exists: variant. Hence, all VBScript functions return this data type. Variant is able to hold different kinds of information, depending on how it is used.

In the simplest case, a variant stores numerical values or strings. A variant behaves like a number if it is used in a numerical context, and like a string if addressed as text. If you work with data that “looks” like numbers, variant will interpret them as such. Of course you can always force numbers to be interpreted as text by enclosing them in hyphens. This is not required if the data is obviously text.

Variant Subtypes

Besides the simple classification of numerical values and strings, the category numerical of a variant can be subdivided. For instance, a date value or a time value can be of the class numerical. In combination with other date and time values, the result will always be expressed in the respective format. Of course there exists a large number of other types of numerical information e.g. boolean values or large floating-point numbers. These classes of information are called subtypes of variant.

Usually it is sufficient to simply assign variant data of a certain type. Variant will automatically behave according to the data type. The next table lists the different sub-types of variant.

Sub-Type	Description
Empty	**variant** is not initialized. Numerical variables are set to 0, string variables are a zero-length string ("").
Null	**variant** intentionally contains no valid data .
Boolean	is either TRUE or FALSE.
Byte	contains an integer ranging from 0 to 255.
Integer	contains an integer ranging from -32,768 bis 32,767.
Currency	contains a number ranging from -922,337,203,685,477.5808 to 922,337,203,685,477.5807.
Long	contains an integer ranging from -2.147.483.648 to 2.147.483.647.
Single	contains a single-precision floating-point number ranging from -3,402823E38 to -1,401298E-45 for negative values and from 1,401298E-45 to 3,402823E38 for positive values.
Double	contains a double-precision floating-point number ranging from -1,79769313486232E308 to -4,94065645841247E-324 for negative values and from 4,94065645841247E-324 to 1.79769313486232E308 for positive values.
Date (Time)	contains a number representing a date between 1. January 100 and 31. December 9999.
String	contains a string of variable length, up to 2 billion characters
Object	contains an object.
Error	contains an error number.

Several conversion functions exist to convert one subtype into another. In addition, the function VarType returns information about how this data ist stored within variant.

13.2.2 VBScript Variables

A variable is a placeholder that refers to a location in the computers memory where programs can store their data. The data may change during run time of the script. For exmple, a variable named “click” can be used to store how many times the user clicks on a certain form. The location of the variable in the computers’ memory is irrelevant. The name of the variable is sufficient to read its value. In VBScript variables always are of the data type variant.

Variable Declaration

Variables are explicitely declared in a script using the Dim, Public and Private statement. For example:

```
Dim DegreesFahrenheit
```

Multiple variables are declared at once by separating them with commas:

```
Dim Top, Bottom, Left, Right
```

A variable can also be declared implicitely by using its name at any position in a script. However, this is looked upon as bad style; you could mistype a variable name at one or more places which in turn leads to unpredictable results when executing the script. Hence, the Option Explicit statement was introduced to force an explicit variable declaration. Therefore, the Option Explicit statement should always be the first statement in a script.

Naming Restrictions

The standard rules for naming language elements in VBScript also apply to variable names:

- they have to start with an alphabet character
- no embedded periods are allowed.
- the maxmum lenght is 255 characters.
- they must be unique within the scope for which they have been declared.

Validity and Life Time of Variables

If a variable was declared within a procedure, only code from within this procedure can access or change the value of that variable. The variable is valid only locally and is therefore called procedure-level variable. In case the variable declaration is not part of a procedure, the variable is recognized by all procedures of the script. The validity scope of this script-level variable is the script level.

The life time of a variable is the time during which a variable exists. The life time extends from the time of the variable declaration until the script is terminated. The life time of a procedure-level variable starts with the variable declaration at the beginning of a procedure and ends with the end of the procedure. Procedure-level variables are ideal as temporary storage while the procedure is running. You can use procedure-level variables of the same name in several procedures, because each variable is only recognized by the procedure in which it was defined.

Assigning Values to Variables

Values are assigned using an expression that contains the variable name on the left side of the equal sign and the value on the right side. For example:

```
B = 200
```

Scalar Data and Arrays

Usually, only a single value is assigned to a variable. These variables are called scalar variables. In some cases it is useful to assign several related values to the same variable. You can create a variable which can contain a series of values, called array variables. They are declared similar to scalar variables, the only difference are patentheses that follow the variable name. A single-dimensional array with 11 elements can be declared as:

```
Dim A(10)
```

Although the number enclosed in patentheses is 10 this array consists of 11 elelment, because the index in VBScripts starts at 0. This type of array is called a fixed size array.

You assign values to the different elements using the index number. Indices running from 0 to 10 are used in the following example to assign values to the array:

```
A(0) = 256
A(1) = 324
A(2) = 100
. . .
A(10) = 55
```

In the same way (using the index of the array) values can be read from the array elements:

```
. . .
AVariable = A(8)
. . .
```

Arrays are not limited to a single dimension. Up to 60 dimensions are allowed, although most people find it difficult to think of more than 3 dimensions. The dimension is declared by introducing more array parameters in the parentheses and separated by a comma. The declaration of a two dimensional array variable Table1 with 6 rows and 11 columns would look like this:

```
Dim Table1(5, 10)
```

The first number in a two-dimensional array always specifies the number of rows and the second the number of columns.

The size of arrays may also vary during run time of a script. This type of array is called a dynamic array. Initially, the array is declared in a procedure using a Dim or a ReDim statement, like any ordinary array. But in this case the number of dimensions is not stated, the brackets are empty:

```
Dim ADataField()
ReDim AndotherDataField()
```

In order to use such an array, the number of dimensions and their size must be defined later using the ReDim command. In the following, ReDim is used to set the initial size of the the dynamic array to 25. The subsequent ReDim statement changes the size to 30, but uses Preserve as keyword, which leaves the content of the array intact during the change of the size:

```
ReDim ADataField(25)
. . .
ReDim Preserve ADataField(30)
```

There is no restriction to how often the size of a dynamic array may be changed. However, decreasing the size of a array will result in loss of data contained in the removed elements.

13.2.3 VBScript Constants

A constant is an expressive name that takes the place of a number or a string and does not change. VBScript defines a number of intrinsic constants.

Creating Constants

User-defined constants are created in VBScripts with the Const statement. It allows to create numerical and string constants and assign them a literal name:

```
Const String1 = "This is my string."
Const Age = 49
```

Note that the string literal is hyphenated (" "). Quotation marks are the most obvious way to distinguish between string values and numeric values. Date/time literals are enclosed in number signs (#):

```
Const Deadline = #6-1-97#
```

There is no difference between constants created in this way and regular variables. Therefore, you may want to adopt a naming scheme to differentiate constants from variables. This will prevent you from accidentially trying to assign a value to a constant while your script is running. For example, you might want to use a “vb” or “con” prefix on your constant names, or you might name your constants in capitals. In any case, you should be able to differentiate between constants and variables to eliminate confusion as you develop more complex scripts.

13.2.4 VBScript Operators

VBScript has a full range of operators, including arithmetic operators, comparison operators, concatenation operators, and logical operators.

Operator Precedence

When several operations occur in an expression, each part is evaluated and resolved in a predetermined order called operator precedence. You can override the order of precedence and force some parts of an expression to be evaluated before others by using parentheses. Operations within parentheses are always performed before those outside. However, within parentheses standard operator precedence is maintained.

If an expression contains operators from different categories, arithmetic operators are evaluated first, comparison operators are evaluated next, and logical operators are evaluated last. Comparison operators all have equal precedence; that is, they are evaluated in the left-to-right order in which they appear. Arithmetic and logical operators are evaluated in the following order of precedence:

Arithmetic		**Comparison**		**Logical**	
Description	**Symbol**	**Description**	**Symbol**	**Description**	**Symbol**
Exponentiation	^	Equality	=	Logical negation	Not
Unary negation	-	Inequality	<>	Logical conjunction	And
Multiplication	*	Less than	<	Logical disjunction	Or
Division	/	Greater than	>	Logical exclusion	Xor
Integer division	\	Less than or equal to	<=	Logical equivalence	Eqv
Modulus arithmetic	Mod	Greater than or equal to	>=	Logical implication	Imp
Addition	+	Object equivalence	Is		
Subtraction	-				
String concatenation	&				

When multiplication and division occur in an expression, each operation is evaluated as it occurs from left to right. Addition and subtraction are handled in the same way, should they occur together in an expression.

The string concatenation (&) operator is not an arithmetic operator, but in precedence it ranks after all arithmetic operators and before all comparison operators. The Is operator is used for object reference comparison. It does neither compare objects nor their values but checks whether two object references refer to the same object.

13.2.5 Using Conditional Statements to Control Program Execution

You can control the flow of your script with conditional statements and looping statements. Using conditional statements, you can write VBScript code that makes decisions and repeats actions. The following conditional statements are available in VBScript:

```
Statement If...Then...Else
Statement Select Case
```

Making Decissions Using If...Then...Else

The If...Then...Else statement is used to evaluate whether a condition is True or False and, depending on the result, to specify one or more statements to run. Usually the condition is an expression that employs a comparison operator to compare one value or variable with another. If...Then...Else statements can be nested to as many levels as you need.

Running Statements if a Condition is True

To run only one statement when a condition is **True**, use the single-line syntax for the **If...Then...Else** statement. The following example shows the single-line syntax. Note that this example omits the **Else** keyword.

```
Sub FixDate()
    Dim myDate
    myDate = #2/13/95#
    If myDate < Now Then myDate = Now
End Sub
```

To execute more than one line of code, the multiple-line (or block) syntax must be used. This syntax includes the End If statement, as shown in the following example:

```
Sub AlertUser(value)
    If value = 0 Then
        AlertLabel.ForeColor = vbRed
        AlertLabel.Font.Bold = True
        AlertLabel.Font.Italic = True
    End If
End Sub
```

Running Certain Statements if a Contidition is True

You can use an If...Then...Else statement to define two blocks of executable statements: one block will be executed if the condition is True, the other block to run if the condition is False.

```
Sub AlertUser(value)
    If value = 0 Then
        AlertLabel.ForeColor = vbRed
        AlertLabel.Font.Bold = True
        AlertLabel.Font.Italic = True
    Else
        AlertLabel.Forecolor = vbBlack
        AlertLabel.Font.Bold = False
        AlertLabel.Font.Italic = False
    End If
End Sub
```

Differentiating Between Several Alternatives

The If...Then...Else statement allows you to choose from several alternatives. Adding the ElseIf clause expands the functionality of the If...Then...Else statement and allows you to control program flow based on different possibilities:

```
Sub ReportValue(value)
    If value = 0 Then
        MsgBox value
    ElseIf value = 1 Then
        MsgBox value
    ElseIf value = 2 then
        Msgbox value
    Else
        Msgbox "Wert außerhalb des Bereichs!"
    End If
```

Depending on your needs you can add as many ElseIf clauses as you want to provide alternative choices. Extensive use of the ElseIf clauses often becomes cumbersome. A better way to choose between several alternatives can bbe realized with the Select Case statement.

Making Decisions with Select Case

The Select Case structure provides an alternative to If...Then...ElseIf for selectively executing one block of statements from among multiple blocks of statements. A Select Case statement provides capability similar to the If...Then...Else statement, but it makes code more efficient and readable.

A Select Case structure works with a single test expression that is evaluated once, at the top of the structure. The result of the expression is then compared with the values for each Case in the structure. If there is a match, the block of statements associated with that Case is executed:

```
 Select Case CardType
    Case "MasterCard"
        DisplayMCLogo
        ValidateMCAccount
    Case "Visa"
        DisplayVisaLogo
        ValidateVisaAccount
    Case "American Express"
        DisplayAMEXCOLogo
        ValidateAMEXCOAccount
    Case Else
        DisplayUnknownImage
        PromptAgain
End Select
```

Note that the Select Case structure evaluates an expression once at the top of the structure. In contrast, the If...Then...ElseIf structure can evaluate a different expression for each ElseIf statement. You can replace an If...Then...ElseIf structure with a Select Case structure only if each ElseIf statement evaluates the same expression.

13.2.6 Loops

Using loops allows you to repeat a group of statements. Some loops repeat statements until a condition is False; others repeat statements until a condition is True. There are also loops that repeat statements a specific number of times.

The following looping statements are available in VBScript:

- Do...Loop: loops while or until a condition is True.
- While...Wend: loops while a condition is True.
- For...Next: uses a counter to run statements a specified number of times.
- For Each...Next: repeats a group of statements for each item in a collection or each element of an array.

Using Do Loops

You can use Do...Loop statements to repeatedly run a block of statements. The statements are repeated either while a condition is True or until a condition becomes True.

Repeating Statements While a Condition is True

Use the While keyword to check a condition in a Do...Loop statement. You can check the condition before you enter the loop (as shown in the following ChkFirstWhile example), or you can check it after the loop has run at least once (as shown in the ChkLastWhile example). In the ChkFirstWhile procedure, if `myNum` is set to 9 instead of 20, the statements inside the loop will never run. In the ChkLastWhile procedure, the statements inside the loop run only once because the condition is already False.

```
Sub ChkFirstWhile()
    Dim counter, myNum
    counter = 0
    myNum = 20
    Do While myNum > 10
        myNum = myNum - 1
        counter = counter + 1
    Loop
    MsgBox "The Loop was repeated " & counter &
    " times."
End Sub

Sub ChkLastWhile()
    Dim counter, myNum
```

```
        counter = 0
        myNum = 9
        Do
            myNum = myNum - 1
            counter = counter + 1
        Loop While myNum > 10
        MsgBox "The Loop was repeated " & counter &
        " times."
    End Sub
```

Repeating a Statement Until a Condition Becomes True

You can use the Until keyword in two ways to check a condition in a Do...Loop statement. You can check the condition before you enter the loop (as shown in the following ChkFirstUntil example), or you can check it after the loop has run at least once (as shown in the ChkLastUntil example). As long as the condition is False, the looping occurs.

```
    Sub ChkFirstUntil()
        Dim counter, myNum
        counter = 0
        myNum = 20
        Do Until myNum = 10
            myNum = myNum - 1
            counter = counter + 1
        Loop
        MsgBox "The Loop was repeated " & counter &
        " times."
    End Sub

    Sub ChkLastUntil()
        Dim counter, myNum
        counter = 0
        myNum = 1
        Do
            myNum = myNum + 1
            counter = counter + 1
        Loop Until myNum = 10
        MsgBox "The Loop was repeated " & counter &
        " times."
    End Sub
```

Exiting a Do...Loop Statement from Inside the Loop

You can exit a Do...Loop by using the Exit Do statement. Because you usually want to exit only in certain situations, such as to avoid an endless loop, you should use the Exit Do statement in the True statement block of an If...Then...Else statement. If the condition is False, the loop runs as usual.

In the following example, `myNum` is assigned a value that creates an endless loop. The If...Then...Else statement checks for this condition, preventing the endless repetition.

```
Sub ExitExample()
    Dim counter, myNum
    counter = 0
    myNum = 9
    Do Until myNum = 10
        myNum = myNum - 1
        counter = counter + 1
        If myNum < 10 Then Exit Do
    Loop
    MsgBox "The Loop was repeated " & counter &
    " times."
End Sub
```

13.2.6.1 Using While...Wend

The While...Wend statement is provided in VBScript for developers who are familiar with its usage. However, because of the lack of flexibility in While...Wend, it is recommended that you use Do...Loop instead.

Using For...Next

For...Next statements are used to run a block of statements a specific number of times. For loops, use a counter variable whose value is increased or decreased with each repetition of the loop.

For example, the following procedure causes a procedure called `MyProc` to be executed 50 times. The For statement specifies the counter variable x and its start and end values. The Next statement increments the counter variable by 1.

```
Sub DoMyProc50Times()
    Dim x
    For x = 1 To 50
        MyProc
    Next
End Sub
```

Using the Step keyword, you can increase or decrease the counter variable by the value you specify. In the following example, the counter variable j is incremented by 2 each time the loop repeats. When the loop is finished, total is the sum of 2, 4, 6, 8, and 10.

```
Sub TwosTotal()
    Dim j, total
    For j = 2 To 10 Step 2
        total = total + j
    Next
    MsgBox "Die Summe ist " & total
End Sub
```

To decrease the counter variable, you use a negative Step value. You must specify an end value that is less than the start value. In the following example, the counter variable myNum is decreased by 2 each time the loop is repeated. When the loop is finished, total is the sum of 16, 14, 12, 10, 8, 6, 4, and 2.

```
Sub NewTotal()
    Dim myNum, total
    For myNum = 16 To 2 Step -2
        total = total + myNum
    Next
    MsgBox "The sum is " & total
End Sub
```

You can exit any For...Next statement before the counter reaches its end value by using the Exit For statement. Because you usually want to exit only in certain situations, such as when an error occurs, you should use the Exit For statement in the True statement block of an If...Then...Else statement. If the condition is False, the loop runs as usual.

Using For Each...Next

A For Each...Next loop is similar to a For...Next loop. Instead of repeating the statements a specified number of times, a For Each...Next loop repeats a group of statements for each item in a collection of objects or for each element of an array. This is especially helpful if you don’t know how many elements are in a collection.

13.2.7 VBScript Procedures

In VBScript there are two kinds of procedures; the Sub procedure and the Function procedure.

Sub Procedures

A sub procedure is a series of VBScript statements, enclosed by Sub and End Sub statements, that perform actions but don’t return a value. A Sub procedure can take arguments (constants, variables, or expressions that are passed by a calling procedure). If a Sub procedure has no arguments, its Sub statement must include an empty set of parentheses ().

The following Sub procedure uses two intrinsic (or built-in) VBScript functions, MsgBox and InputBox, to prompt a user for some information. It then displays the results of a calculation based on that information. The calculation is performed in a Function procedure created using VBScript. The Function procedure is shown after the following discussion.

```
Sub ConvertTemp()
   temp = InputBox("Enter the Temperature in
   Fahrenheit.", 1)
   MsgBox "The Temperature is " & Celsius(temp) &
   " Degree Celsius."
End Sub
```

Function Procedures

A Function procedure is a series of VBScript statements enclosed by the Function and End Function statements. A Function procedure is similar to a Sub procedure, but can also return a value. A Function procedure can take arguments (constants, variables, or expressions that are passed to it by a calling procedure). If a Function procedure has no arguments, its Function statement must include an empty set of parentheses. A Function returns a value by assigning a value to its name in one or more statements of the procedure. The return type of a Function is always a Variant.

In the following example, the Celsius function calculates degrees Celsius from degrees Fahrenheit. When the function is called from the ConvertTemp Sub procedure, a variable containing the argument value is passed to the function. The result of the calculation is returned to the calling procedure and displayed in a message box.

```
Sub KonvertTemp()
    temp = InputBox("Enter the Temperature in
    Fahrenheit.", 1)
    MsgBox "The Temperature is " & Celsius(temp) &
    " Degree Celsius."
End Sub

Function Celsius(GradF)
    Celsius = (GradF - 32) * 5 / 9
End Function
```

Forwarding Data To or From Procedures

Each piece of data is passed into your procedures using an argument. Arguments serve as placeholders for the data you want to pass into your procedure. When you create a procedure using either the Sub statement or the Function statement, parentheses must be included after the name of the procedure. Any arguments are placed inside these parentheses, separated by commas. For example, in the following example, fDegrees is a placeholder for the value being passed into the Celsius function for conversion:

```
Function Celsius(fDegrees)
   Celsius = (fDegrees - 32) * 5 / 9
End Function
```

To get data out of a procedure, you must use a Function. Remember, a Function procedure can return a value; a Sub procedure can’t.

Using Sub and Function Procedures in Code

A Function in your code must always be used on the right side of a variable assignment or in an expression. For example:

```
Temp = Celsius(fDegrees)
```

or

```
MsgBox "The temperature is " & Celsius(fDegrees) &
" Degree Celsius."
```

To call a Sub procedure from another procedure, you can just type the name of the procedure along with values for any required arguments, each separated by a comma. The Call statement is not required, but if you do use it, you must enclose any arguments in parentheses.

The following example shows two calls to the `MyProc` procedure. One uses the Call statement in the code; the other doesn't. Both do exactly the same thing.

```
Call MyProc(firstArg, secondArg)
MyProc firstArg, secondArg
```

Notice that the parentheses are omitted in the call when the Call statement isn't used.

13.2.8 VBScript Coding Converntions

Conventions for Programming

Coding conventions are suggestions that may help you write code using Microsoft Visual Basic Scripting Edition. Coding conventions can include the following:

- Naming conventions for objects, variables, and procedures
- Commenting conventions
- Text formatting and indenting guidelines

The main reason for using a consistent set of coding conventions is to standardize the structure and coding style of a script or set of scripts so that you and others can easily read and understand the code. Using good coding conventions results in precise, readable, and unambiguous source code that is consistent with other language conventions and as intuitive as possible.

Constant Naming Conventions

Earlier versions of VBScript had no mechanism for creating user-defined constants. Constants, if used, were implemented as variables and distinguished from other variables using all uppercase characters. Multiple words were separated using the underscore (_) character. For example:

```
USER_LIST_MAX
NEW_LINE
```

While this is still an acceptable way to indentify your constants, you may want to use an alternative naming scheme, now that you can create true constants using the Const statement. This convention uses a mixed-case format in which constant names have a “con” prefix. For example:

```
conYourOwnConstant
```

Variable Naming Conventions

For purposes of readability and consistency, use the following prefixes with descriptive names for variables in your VBScript code.

Subtype	Prefix	Example
Boolean	bln	BlnFound
Byte	byt	BytRasterData
Date (Time)	dtm	DtmStart
Double	dbl	DblTolerance
Error	err	ErrOrderNum
Integer	int	IntQuantity
Long	lng	LngDistance
Object	obj	ObjCurrent
Single	sng	SngAverage
String	str	StrFirstName

Variable Scope

Variables should always be defined with the smallest scope possible. VBScript variables can have the following scope.

Scope	Where Variabel Is Declared	Visibility
Procedure Level	Event, Function, or sub procedure	Visible in the procedure in which it is declared.
Script Level	HEAD section of an HTML page, outside any procedure.	Visible in every procedure in the script.

Variable Scope Prefixes

As script size grows, so does the value of being able to quickly differentiate the scope of variables. A one-letter scope prefix preceding the type prefix provides this, without unduly increasing the size of variable names.

Subtype	Prefix	Example
Procedure Level	None	dblVelocity
Script Level	s	sblnCalcInWork

Descriptive Variable and Procedure Names

The body of a variable or procedure name should use mixed case and should be as complete as necessary to describe its purpose. In addition, procedure names should begin with a verb, such as InitNameArray or CloseDialog.

For frequently used or long terms, standard abbreviations are recommended to help keep name length reasonable. In general, variable names greater than 32 characters can be difficult to read. When using abbreviations, make sure they are consistent throughout the entire script. For example, randomly switching between Cnt and Count within a script or set of scripts may lead to confusion.

Object Naming Conventions

The following table lists recommended conventions for objects you may encounter while programming VBScript.

Object type	Prefix	Example
CheckBox	chk	chkReadOnly
ComboBox, drop-down ListBox	cbo	cboDeutsch
CommandButton	cmd	cmdExit
CommonDialog	dlg	dlgFileOpen
Frame	fra	fraLanguage
horizontal ScrollBar	hsb	hsbVolume
Image	img	imgIcon
Label	lbl	lblHelpMessage
Line	lin	linVertical
ListBox	lst	lstPolicyCodes
Spin	spn	spnPages
TextBox	txt	spnLastName
vertical ScrollBar	vsb	vsbRate
Slider	sld	sldScale

Code Commenting Conventions

All procedures should begin with a brief comment describing what they do. This description should not describe the implementation details (how it does it) because these often change over time, resulting in unnecessary comment maintenance work, or worse, erroneous comments. The code itself and any necessary inline comments describe the implementation.

Arguments passed to a procedure should be described when their purpose is not obvious and when the procedure expects the arguments to be in a specific range. Return values for functions and variables that are changed by a procedure, especially through reference arguments, should also be described at the beginning of each procedure.

Procedure header comments should include the following section headings. For examples, see the “Formatting Your Code” section that follows.

Section Heading	Comment Contents
Purpose	What the procedure does (not how).
Assumptions	List of any external variable, control, or other element whose state affects this procedure.
Effects	List of the procedure’s effect on each external variable, control, or other element.
Inputs	Explanation of each argument that isn’t obvious. Each argument should be on a separate line with inline comments.
Return Values	Explanation of the value returned.

The following points should be taken into account:

- Every important variable declaration should include an inline comment describing the use of the variable being declared.
- Variables, controls, and procedures should be named clearly enough that inline comments are only needed for complex implementation details.
- At the beginning of your script, you should include an overview that describes the script, enumerating objects, procedures, algorithms, dialog boxes, and other system dependencies. Sometimes a piece of pseudocode describing the algorithm can be helpful.

Formatting Your Code

Screen space should be conserved as much as possible, while still allowing code formatting to reflect logic structure and nesting. Here are a few pointers:

- Standard nested blocks should be indented four spaces.
- The overview comments of a procedure should be indented one space.
- The highest level statements that follow the overview comments should be indented four spaces, with each nested block indented an additional four spaces.

13.2.9 VBScript Functions

A complete reference of all available VBScript functions is beyond the scope of this manual. The functions listed in the following are only a part of what is available in the full edition of VisualBasic. If a function also exists in VBScript, it is used similar; a description can be taken from the VisualBasic documentation. Hence we restrict the following list to all functions that are also available in VBScript.

Control Flow

Do...Loop
For...Next
For Each...Next
If...Then...Else
Select Case
While...Wend
Array
Dim, Private, Public, ReDim
IsArray
Erase
LBound, UBound

Dates/Times

Date, Time
DateAdd, DateDiff, DatePart
DateSerial, DateValue
Day, Month, Weekday, WeekdayName, Year
Hour, Minute, Second
Now
TimeSerial, TimeValue

Declarations

Const
Dim, Private, Public, ReDim
Function, Sub

Input/Output

InputBox
LoadPicture
MsgBox

Error Handling

On Error
Err

Comments

Comments using ’ or Rem

Constants/Literals

Empty
Nothing
Null
True, False

Conversions

Abs
Asc, AscB, AscW
Chr, ChrB, ChrW
CBool, CByte
CCur, CDate
CDbl, CInt
CLng, CSng, CStr
DateSerial, DateValue
Hex, Oct
Fix, Int
Sgn
TimeSerial, TimeValue

Literals

Empty
False
Nothing
Null
True

Math

Atn, Cos, Sin, Tan
Exp, Log, Sqr
Randomize, Rnd

Objects

CreateObject
Err-Objekt
GetObject

Operators

Addition (+), Subtraction (-)
Exponentiation (^)
Modulo arithmetic (Mod)
Multiplication (*), Division (/), Integer Division (\)
Negation (-)
String Concatenation (&)
Equality (=), Inequality (<>)
Less Than (<), Less Than or Equal To (<=)
Greater Than (>), Greater Than or Equal To(>=)
Is
And, Or, Xor
Eqv, Imp

Options

Option Explicit

Procedures

Call
Function, Sub

Rounding

Abs
Int, Fix, Round
Sgn

Script Engine ID

ScriptEngine
ScriptEngineBuildVersion
ScriptEngineMajorVersion
ScriptEngineMinorVersion

Variants

IsArray
IsDate

IsEmpty
IsNull
IsNumeric
IsObject
TypeName
VarType

Miscellaneous

RGB-Functions

Strings

Asc, AscB, AscW
Chr, ChrB, ChrW
Filter, InStr, InStrB
InStrRev
Join
Len, LenB
LCase, UCase
Left, LeftB
Mid, MidB
Right, RightB
Replace
Space
Split
StrComp
String
StrReverse
LTrim, RTrim, Trim

Formatting Strings

FormatCurrency
FormatDateTime
FormatNumber
FormatPercent

Assignments

Set

13.2.10 File and System Handling

Files are accessed via the objects of the VBScript run time library, which provides the following objects: Dictionary, Drive, File, Folder, FileSystemObjekt, TextStream. These in turn provide the functions listed here.

Dictionary

Add
Exists
Items
Keys
Remove
RemoveAll
Count
Item
Key

Drive, File, Folder

Copy
Delete
Move
OpenAsTextStream
Attributes
Count
DateCreated
DateLastAccessed
DateLastModified
Drive
ParentFolder
Name
Path
ShortName
ShortPath
Size
AvailableSpace
DriveLetter
DriveType
FileSystem
FreeSpace
IsReady
RootFolder
SerialNumber
ShareName
TotalSize
VolumeName

FileSystemObject

BuildPath
CopyFile
CopyFolder

CreateFolder
CreateTextFile
DeleteFile
DeleteFolder
DriveExists
FileExists
FolderExists
GetAbsolutePathName
GetBaseName
GetDrive
GetDriveName
GetExtensionName
GetFile
GetFileName
GetFolder
GetParentFolderName
GetSpecialFolder
GetTempName
MoveFile
MoveFolder
OpenTextFile
Drives

TextStream

Close
Read
ReadAll
ReadLine
Skip
SkipLine
Write
WriteBlankLines
WriteLine
AtEndOfLine
AtEndOfStream
Column
Line

13.3 JavaScript

JavaScript will also be processed by the OPUS Scripting Engine, in this manual however we document and support mainly VBScript in view of a uniform use. If you prefer JavaScript you find a good introduction and reference on the following web pages.

```
http://msdn.microsoft.com/scripting/default.htm?/
scripting/jscript/default.htm
```

Regrettably, most documentation is oriented towards using scripts for the design of HTML pages.

13.4 Functions/Events of Forms

The functions and events of an OPUS script form have already been used to transmit commands to OPUS. In the following you will find an extensive list of all available functions and events:

Visible

Property of the form, indicates whether the form is visible or not, can either have the value true or false.

Show

Method to visualize the form.

Hide

Method to hide the form.

Close

Method to close the form and end the script.

Enable

Enables input to the form by keyboard or mouse, in combination with a parameter true or false.

Minimize

Method to minimize the form.

Maximize

Method to maximize the form.

Restore

Method to reset the size of the form to its initial value.

GetApp

Returns an object of type application. This object represents the OPUS application and in turn provides functions for the handling of forms. This allows to address other forms dynamically, create new scripts from a running script, and exchange data with them.

OpenForm

Opens a new form, the name of the script file (including the path) has to be specified. You can indicate (with the values true or false) if a script should be used which was already opened and if the file should be opened in edit mode.

NewForm

Opens a new, blank form.

FormByName

Returns an interface to a form object already running. The internal form name is transferred as the parameter. The form name is listed as *ID* in the forms *Properties*.

Caption

Caption reads or writes a text as title in the window of the form.

DoEvents

DoEvents hands the process control over to the system until the system has processed all pending events (like paint messages).

Caution! Each time the process control is temporarily transferred to another thread, care has to be taken that the procedure will not be called by any part of the code, before the first call is terminated. Otherwise the results are unpredictable.

OpusCommand

Function to start OPUS commands. The function returns immediately after transmitting the text command. The command is then processed by OPUS and then the result forwarded to the form by means of an OnOpusResult event.

OpusExecute

Executes an OPUS command. The command will run as background task and the result will not be returned.

OpusRequest

Executes an OPUS command and waits until OPUS has finished the command processing. The result will be returned directly as text. While normally only a single event is being processed at a time, this function allows processing of additional events until OPUS returns its result. The execution of the event which calls OpusRequest is postponed until OPUS answers. Events, which are normally called after this procedure may have already been executed. The consequences of the independent time lines have to be taken into account in the script.

OpusRequestData

Sends commands to OPUS, similar to OpusRequest. In addition, a data array parameter is exchanged. This parameter is able to transfer a data field to OPUS or to receive binary data from OPUS. Like in the case of OpusRequest precautions have to be taken to avoid unwanted side effects resulting from parallel data processing.

SetWindowPos

Positions and dimensions a window using four coordinates: x, y, dx, dy.

SetResult

Sets the result of a script as text, to be transmitted to the requesting OPUS function upon closing the script. To make use of this result, the script has to be called using the OPUS *VisualBasic Script* funtion.

GetDocName

Returns the name including path of the active script. Makes it possible to run a script on different machines by referring to relative path statements. This command becomes first available after the form has been loaded (and not upon loading the form).

HideControl

Hides a control element. The name of the object in the form has to be stated (e.g. `CommandButton1`).

ShowControl

Reveals a hidden control element. The name of the element has to be stated in the form (i.e. `CommandButton 1`).

OnLoad

Event which is triggered upon loading a form.

OnUnLoad

Event which is triggered upon unloading a form.

OnOpusResult

Event which is triggered after an OPUS command has been processed. Three text fields and if necessary a binary data array will be returned. The contents of the text fields may vary with the function executed; the first string usually is the "OK" statement or an error statement, that indicates whether the command could be processed successfully. The other fields contain the result. If an OPUS data manipulation function has been started with a text command, the first parameter holds the names (numbers) of the resulting files and the second parameter and return values. If binary data transfer was chosen, the fourth parameter contains the data array.

OnOpusInform

This function is used by OPUS to transmit a parameter to a script. If for example the function VisualBasic Script is called in combination with script parameters, an OnOpusInform event is triggered at the start of the program. The text is forwarded to the event handling routine as a parameter.

This event is also employed for the **A**utomatic **A**ccessory **R**ecognition; if an accessory with AAR support is inserted into the spectrometer the accessory code will be transferred to a special script.

13.5 Microsoft Forms

Microsoft Forms are control elements that can be used to create and modify custom forms and dialog boxes. They are also used to create VisualBasic for Application Forms in Microsoft Word. This chapter gives a brief overview of the existing elements their use.

All functionalities of these controls can be classified to one of the following categories:

Properties – By opening the Properties dialog of an element in the Form Editor you can find out which properties and values are supported by this element. All available properties will be listed. Usually, they can also be set or read from the program.

Events and related handling routines – An event box in the Code/Modules view shows which events can occur for an object. Here, all required functions can be created.

Methods – Elements use methods to independently process certain tasks. Since no selection box exist for the methods, short lists are is presented here. An extensive treatment can be found in Microsofts documentation.

13.5.1 Checkbox

Displays the selection state of an item. Use a CheckBox to give the user a choice between two values such as Yes/No, True/False, or On/Off. When the user selects a CheckBox, it displays a special mark (such as an X) and its current setting is Yes, True, or On; if the user does not select the CheckBox, it is empty and its setting is No, False, or Off. Depending on the value of the TripleState property, a CheckBox can also have a null value.

13.5.2 Combobox Control

Combines the features of a ListBox and a TextBox. The user can enter a new value, as with a TextBox, or the user can select an existing value as with a ListBox.

Supported Methods:

```
AddItem
Clear
DropDown
RemoveItem
Copy
Cut
Paste
```

13.5.3 CommandButton

Starts, ends, or interrupts an action or series of actions. The macro or event procedure assigned to the CommandButton's click event determines what the CommandButton does. For example, you can create a CommandButton that opens another form. You can also display text, a picture, or both on a CommandButton.

13.5.4 Frame Control

Creates a functional and visual control group. All option buttons in a Frame are mutually exclusive, so you can use the Frame to create an option group. You can also use a Frame to group controls with closely related contents. For example, in an application that processes customer orders, you might use a Frame to group the name, address, and account number of customers.

You can also use a Frame to create a group of toggle buttons, but the toggle buttons are not mutually exclusive.

Supported Methods:

```
Copy
Cut
Paste
RedoAction
Repaint
Scroll
SetDefaultTabOrder
UndoAction
```

13.5.5 Image Control

Displays a picture on a form. The Image lets you display a picture as part of the data in a form. For example, you might use an Image to display employee photographs in a personnel form.

The Image lets you crop, size, or zoom a picture, but does not allow you to edit the contents of the picture. For example, you cannot use the Image to change the colors in the picture, to make the picture transparent, or to refine the image of the picture. You must use image editing software for these purposes

13.5.6 Label Control

Displays descriptive text. A Label control on a form displays descriptive text such as titles, captions, pictures, or brief instructions.

13.5.7 ListBox Control

Displays a list of values and lets you select one or more. If the ListBox is bound to a data source, then the ListBox stores the selected value in that data source.

The ListBox can either appear as a list or as a group of OptionButton controls or CheckBox controls.

Supported Methods:

```
AddItem
Clear
RemoveItem
```

13.5.8 Multipage Control

Presents multiple screens of information as a single set. A MultiPage is useful when you work with a large amount of information that can be sorted into several categories. For example, use a MultiPage to display information from an employment application. One page might contain personal information such as name and address; another page might list previous employers; a third page might list references. The MultiPage lets you visually combine related information, while keeping the entire record readily accessible.

New pages are added to the right of the currently selected page rather than adjacent to it.

Note: The MultiPage is a container of a Pages collection, each of which contains one or more Page objects.

13.5.9 OptionButton Control

Shows the selection status of one item in a group of choices. Use an OptionButton to show whether a single item in a group is selected. Note that each OptionButton in a Frame is mutually exclusive.

If an OptionButton is bound to a data source, the OptionButton can show the value of that data source as either Yes/No, True/False, or On/Off. If the user selects the OptionButton, the current setting is Yes, True, or On; if the user does not select the OptionButton, the setting is No, False, or Off.

Depending on the value of the TripleState property, an OptionButton can also have a null value.

You can also use an OptionButton inside a group box to select one or more of a group of related items.

13.5.10 ScrollBar Control

Returns or sets the value of another control based on the position of the scroll box. A ScrollBar is a stand-alone control you can place on a form. It is visually like the scroll bar you see in certain objects such as a ListBox or the drop-down portion of a ComboBox. However, unlike the scroll bars in these examples, the stand-alone ScrollBar is not an integral part of any other control.

To use the ScrollBar to set or read the value of another control, you must write code for the ScrollBar's events and methods. For example, to use the ScrollBar to update the value of a TextBox, you can write code that reads the Value property of the ScrollBar and then sets the Value property of the TextBox.

Note: To create a horizontal or vertical ScrollBar, drag the sizing handles of the ScrollBar horizontally or vertically on the form.

13.5.11 SpinButton Control

Increments and decrements numbers. Clicking a SpinButton changes only the value of the SpinButton. You can write code that uses the SpinButton to update the displayed value of another control. For example, you can use a SpinButton to change the month, the day, or the year shown on a date. You can also use a SpinButton to scroll through a range of values or a list of items, or to change the value displayed in a text box.

To display a value updated by a SpinButton, you must assign the value of the SpinButton to the displayed portion of a control, such as the Caption property of a Label or the Text property of a TextBox. To create a horizontal or vertical SpinButton, drag the sizing handles of the SpinButton horizontally or vertically on the form.

13.5.12 TabStrip Control

Presents a set of related controls as a visual group. You can use a TabStrip to view different sets of information for related controls.

Note: The TabStrip is implemented as a container of a Tabs collection, which in turn contains a group of Tab objects.

13.5.13 TextBox Control

Displays information from a user or from an organized set of data. A TextBox is the control most commonly used to display information entered by a user. Also, it can display a set of data, such as a table, query, worksheet, or a calculation result. If a TextBox is bound to a data source, then changing the contents of the TextBox also changes the value of the bound data source.

Formatting applied to any piece of text in a TextBox will affect all text in the control. For example, if you change the font or point size of any character in the control, the change will affect all characters in the control.

Supported Methods:

```
Copy
Cut
Paste
```

13.5.14 ToggleButton Control

Shows the selection state of an item. Use a ToggleButton to show whether an item is selected. If a ToggleButton is bound to a data source, the ToggleButton shows the current value of that data source as either Yes/No, True/False, On/Off, or some other choice of two settings. If the user selects the ToggleButton, the current setting is Yes, True, or On; if the user does not select the ToggleButton, the setting is No, False, or Off. If the ToggleButton is bound to a data source, changing the setting changes the value of that data source. A disabled ToggleButton shows a value, but is dimmed and does not allow changes from the user interface.

You can also use a ToggleButton inside a Frame to select one or more of a group of related items.

13.5.15 Timer Control

Although the Timer Control is rather a part of the Internet Explorer we list it here, because it is used in a similar manner. It serves to call functions after predefined periods of time.

Supported Methods:

`AboutBox`

13.5.16 Debugging Scripts

To debug OPUS scripts in single-step mode you should preferably use an universal script debugger. On its home page Microsoft offers a freeware version of a script debugger. We also recommend the InterDev packet which is part of the Microsoft Visual Studio. This software, if installed, it will automatically open when an error in an OPUS script occurs. You can view and edit variables, and process lines step-by-step. If the script is intended to run from the start in the debugger, insert a stop command at the beginning of the script.

Index

S

T

U

V

W

X