OPUS Services

OPUS Interfaces

Description of different interfaces to control OPUS

1849155, Version 3

3rd edition 2019, publication date January 2019

These programming instructions are the original documentation of OPUS Interfaces.

Table of Contents

Table of Contents

1 Introduction

1.1 About this document

This document is a description about different types of interfaces which can be used to control OPUS. The document uses the following format for important information:

i The *i* provides important information on the OPUS Interfaces in particular application fields.

1.2 Motivation

With regard to the practical use of the OPUS spectroscopy software, for many applications it is desirable that OPUS cannot only be operated interactively, but that single steps of frequently recurring routine tasks can be automated.

A separate program must be able to initiate the steps in OPUS, which are normally performed by the user. This mainly applies to the following application fields:

- Measuring spectrum
- Editing spectrum
- Evaluating spectrum
- Displaying evaluation result

1.3 Purpose of this document

The document describes the advantages and disadvantages of the different technical possibilities available to connect with OPUS. The document intends to support the operator to find the technical solution most suitable for his purposes.

If the connection to OPUS has been established, the syntax and content of the commands, which can be used to control OPUS, have to be defined. In chapter 4 et seq., some program sequences exemplify the most important commands. Chapter 5.1 describes the regulations which the commands are based on.

1.4 Term

The term *OPUS Interfaces* is used when the document is described, and the term *OPUS* is used when the spectroscopy software is described.

1.5 Target group

The document is intended to be used by programmers working with spectroscopy software.

1.6 Gender-neutral form

The document uses the male form in a neutral sense and does not differentiate between male and female users. We would kindly ask all female users to have understanding for this simplified form.

1.7 Questions and concerns

If you have questions or concerns about operating the software, contact Bruker at the numbers listed below:

- Service hotline: +49 (0) 72 43 504-2020
- Fax: +49 (0) 72 43 504-2100
- E-mail: service.bopt.de@bruker.com
  service.bopt.us@bruker.com
- Internet: www.bruker.com/optic
- International service: www.bruker.com/about-us/offices/offices/bruker-optics

Introduction 1

2 Interfaces for OPUS Communication

Table 2.1 contains all interfaces which can be used to communicate with OPUS.

Name	Description	Program runs within OPUS
Graphical macro (*.obs)	• Can be used with recurrent measurements and evaluations • Only in connection with the most important OPUS commands[a]	Yes
OPUS macro (*.mtx)	In the simple programming language of the macro editor it is possible to generate commands, which execute OPUS commands, in an automated way. All macro commands are arranged by using the mouse. The programmer does not need any knowledge of the macro language syntax.	Yes

Table 2.1: Interfaces for OPUS communication

Name	Description	Program runs within OPUS
VisualBasic Script (*.obs)	A complete and open-ended programming language is combined with a simple text interface. The interface is used to send commands to OPUS and receive responses from OPUS. The commands to execute an OPUS function are largely created automatically, similar to the OPUS macro.	Yes
Http communication	OPUS provides an interface for the communication via http commands. For the writing process, an URL is opened which contains the OPUS command as part of the URL. The answer is the content of the page accessed by http.	No
Pipe communication	Similar to a file, OPUS can be opened to write and read (named pipe[b]). Via a *named pipe* it is possible to communicate with OPUS.	No

Table 2.1: Interfaces for OPUS communication

a. Typically, these are all commands of the *Measure*, *Manipulate* and *Evaluate* OPUS menu, which allow to generate or edit data.

b. A named pipe is a portion of memory that can be used by one process to pass information to another process. The output of one process is the input of the other process.

3 Description of Interfaces

3.1 Graphical macro

When working with a graphical macro, the following commands are available in the OPUS *Macro* menu:

Command	Definition
New Procedure	Opens the macro wizard to create sequence structures which allow to automate simple procedures.
Edit Procedure	The sequence structure created can be edited.
Execute Procedure	The sequence structure created will be executed.

Table 3.1: OPUS commands for graphical macro

3.1.1 Principle

Using the macro wizard allows to create a sequence structure (flow chart) which shows OPUS commands in the form of nodes. The flow chart allows additional control elements, e.g. if/else branching, countable loopings (for-next), text output by a dialog, as well as variable management.

Figure 3.1 exemplifies a flow chart with OPUS commands.

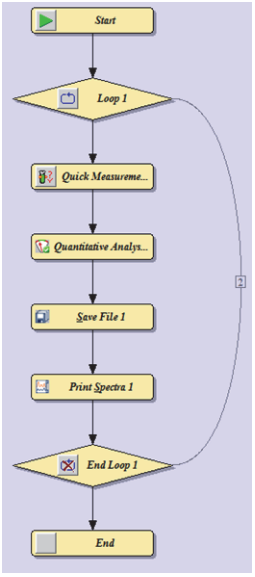

Figure 3.1: Flow chart with OPUS commands

The macro wizard is, however, no programming tool, but more an easy possibility to automate daily routine tasks. Instead of calling the same OPUS commands again and again, they can be described in a flow chart. A flow chart ensures that the commands being part of it are always executed in the same manner.

OPUS converts a flow chart into a VisualBasic Script (*.obs). When executing a sequence, the VisualBasic Script starts. It is not recommended to edit or extend a graphical macro by using the VisualBasic Script editor, as a modification or extension does not allow to restore the original sequence of the flow chart.

Compared to the OPUS macro language (chapter 3.2), the graphical macro is rather limited with regard to the definition of user interfaces, or to the use of programming language elements.

3.2 OPUS macro language (*.mtx)

To work with the OPUS macro language, the following commands are available in the OPUS *Macro* menu:

Command	Definition
Edit Macro	Opens the Macro editor to create and edit macros.
Debug Macro	Macros are executed step-by step, or executed until a specified break point has been reached. Debugging allows to localize and analyze macro errors more quickly.
Run Macro	Starts and executes a macro.

Table 3.2: OPUS commands for macro language

3.2.1 Preconditions

The OPUS macro is only available if the IR software package has been registered in OPUS. To check the OPUS registration proceed as follows:

1. In the *Setup* menu, select the *Register OPUS* command.
2. Check whether the *IR* checkbox is activated under *Available software packages and libraries*.
   ➢ *If the software package is not registered in OPUS, contact Bruker Service (chapter 1.7).*

3.2.2 Principle

The macro editor can integrate each OPUS command into the program code in an automated way. With the editor being open, the particular OPUS command must be called in the same way as being done during the interactive procedure, i.e. the particular parameters must be set up in the OPUS function dialog, and the files must be selected.

If the OPUS function dialog is closed, the command will not be executed but the syntactically correct command will be integrated into the macro code. When executing the macro, the command line will be executed and interpreted as OPUS command with the particular data.

3.2.3 Writing macros

Writing macros using the OPUS macro editor is very easy as all programming commands (user interfaces, control elements, variable assignment, file input and output) and variables are provided in list boxes, so that the program code can almost be generated by using the mouse only. A special knowledge about the syntax of the macro language is not required.

The macro editor is very helpful for programs which had been developed from out of OPUS (chapter 3.4 to 3.5). As the command syntax is identical to OPUS for all interface types, it is advantageous to use the macro editor as command generator for these programs. Instead of struggling to understand the correct command syntax, the OPUS function command can be integrated into the user-specific program code by *Copy & Paste*.

If the macro editor of an OPUS dialog is converted into a command line, the names and meanings of the parameters used in the particular OPUS command (e.g. measurement parameters) are displayed.

The variable management of the editor contains a list of all possible data block types available in OPUS, which allows to write the desired data block reference of the particular OPUS file into the command line.

3.2.4 Disadvantage of macro language

A disadvantage of the macro language is the non-indented code within loopings, which makes it difficult to keep the overview with regard to long macros. The experienced programmer will probably miss very quickly the variety of commands which modern programming languages provide.

The macro editor is, however, a very fast and effective tool to work out most of the programming tasks.

3.3 VisualBasic Script (*.obs)

The VisualBasic Script is most suited for the effective and fast writing of extensive programs, and offers a complete and comfortable programming language as well as a graphical user interface.

In case of VisualBasic Script you can generate the command line of an OPUS command in an automated way. In the script editor, you can individually set up the user interface by means of different control elements, e.g. buttons, option buttons, drop-down lists, checkboxes and entry fields.

3.3.1 Starting script editor

1. On the *File* menu, select the *New* command.
2. In the dialog that opens, select the *VBScript* option and click *OK* to confirm.
   ➢ *The script editor opens.*
3. Use the icons on the toolbar displayed to write the script, and to change between running the script, editing the code or user interface.
   ➢ *The icons are described in the OPUS/Programming user manual.*

3.4 Communicating with OPUS via http

The communication between an external program[1] and an OPUS instance can be performed via http. Each type of programming language provides easy commands which can be used to open and read a http page.

The communication with OPUS is established such that the URL contains the OPUS command, and that the command result is returned as page content. In general, the return value is *OK*, if the command has been executed correctly. In some cases, the return value can also contain the data requested by the command. In case of error, a message can be returned indicating the reason of the error.

i Technically speaking, the communication between the external program and the OPUS instance is done by the http protocol. The answer sent by OPUS, however, is not written in valid html format, but as unformatted string which the recipient has to interpret.

Some programs which allow the communication via an http interface expect an answer in html format, and report an error if no html format is available.

1. A program not running within OPUS

3.4.1 Preconditions

The following conditions must be met in case of the communication via http:

- *OPUS.exe* must be started using the */HTTPSERVER=ON* command line option
- **With OPUS >8.5:** the communication port can be explicitly defined:
  - ☞ In the OPUS command line, enter the */HTTPPORT=<n>* option in addition to the */HTTPSERVER=ON* option.[1]
  - ☞ Then, change the URL for the command call into *http://127.0.0.1:<n>/OpusCommand.htm?<Command>*, with *<n>* being the port number and *<Command>* being the OPUS command used, e.g. GET_VERSION.
- **With OPUS 7.5 and <7.5**: the *Userdatabase* folder with the *Userdatabase.dat* file in the **\OPUS_<version>* path must be shared. To share proceed as follows:
  - ☞ Right click the folder and select the *Properties* command from the pop-up menu.
  - ☞ Click the *Sharing* tab.
  - ☞ Click the *Share* button (figure 3.2).

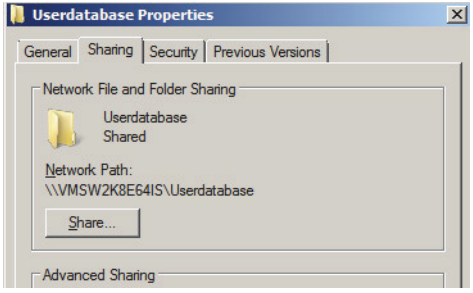

Figure 3.2: Settings to share folder

- **With OPUS 7.2:** the registration must contain the SERVER software package. Bruker extends the prevailing registration free of charge.

1. If the */HTTPPORT* option is not defined, the default value of 80 is used.

3.4.2 Testing communication link

1. Start the Web browser.
2. Enter http://localhost/OpusCommand.htm?GET_VERSION into the address line.

   ➢ *As soon as the page is open, the answer is displayed as page content (figure 3.3).*

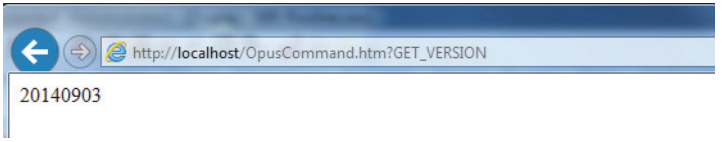

Figure 3.3: Answer as page content

➢ *Up to the question mark the address is always the same. After the question mark follows the actual command (figure 3.3 exemplifies: GET_VERSION). Other commands may call OPUS functions which use the same syntax as used in the macro editor. Example:*

http://localhost/OpusCommand.htm?COMMAND_LINE MeasureSample (, {EXP= experiment.xpm',XPP='C:\Users\Public\Documents\Bruker \OPUS_7.5.18.429\ XPM', NSS=1});

3.5 Communication via named pipe

A named pipe[1] can be used for the communication between OPUS and an external program.

If OPUS is started using the */OPUSPIPE=ON* command line option, OPUS reads and writes on the pipe by using *OPUS* as name (\\.\pipe\OPUS). Sometimes, processing the answers is a problem, as it is not quite clear from the start, how many bytes a command will generate.

1. A named pipe is a portion of memory that can be used by one process to pass information to another process. The output of one process is the input of the other process.

3.6 Using LabView©

LabView© is a graphical programming system, which allows to integrate OPUS to generate data.

Figure 3.4 exemplifies how a virtual instrument performs a measurement based on specific parameters. The data received are indicated as spectrum. This kind of virtual instrument (SAP number 1018930) can be acquired with Bruker, and adapted to user-specific requirements.

The syntax of the commands to control OPUS are identical to the one used for VB-Script, HTTP or Named Pipe.

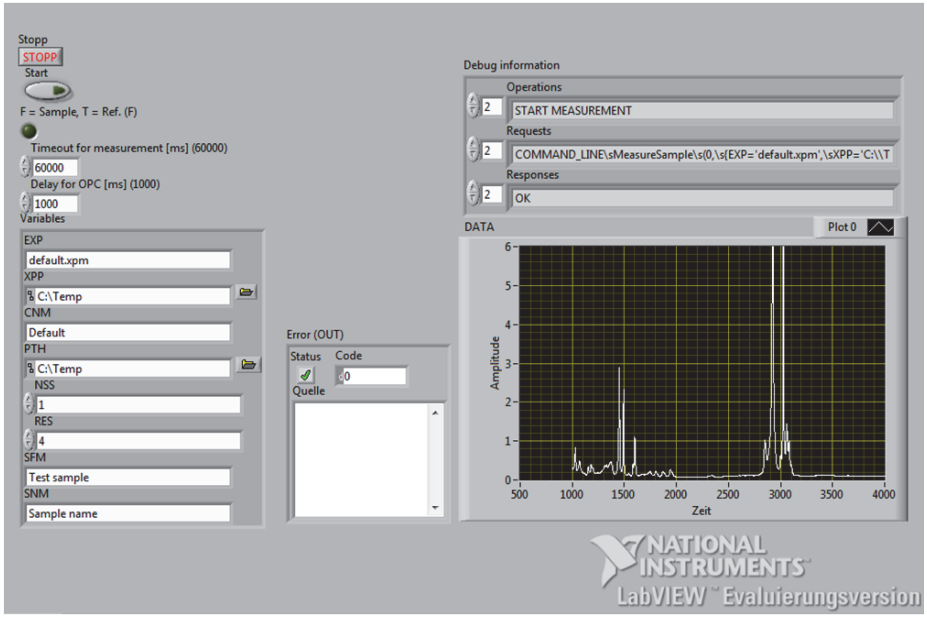

Figure 3.4: LabView©

Description of Interfaces 3

4 Example of a program sequence (in pseudo code)

Frequently-used commands, which can occur in a typical program to control OPUS, are summarized in the following. The commands are green, the literals are red.

i The single code lines are pseudo code and cannot be assigned to a specific programming language.

```
//Previously the program opus.exe /HTTPSERVER=ON /OPUSPIPE=ON has
    been started. (The command line options have to be set only if
    needed).
//We will use the pseudo function CommandToOpus(), to show samples of
    commands which are meaningful.
//In case of VBScript the function Form.OpusRequest(<cmd>), in case
    of http the URL http://localhost/OpusCommand.htm?<cmd> must be
    used.
//In case of named pipe, the <cmd> has to be written directly to pipe
    (hPipe->Write(<cmd>) or similar).

// Test, if OPUS could be connected
Result = CommandToOpus("GET_VERSION_EXTENDED")
Print(Result)  //7.5 Build: 7, 5, 18, 429.B 20140903

//Check if an instrument is connected to OPUS (Check Measure/Setup
    optics and service)
// You can also use the simulator.exe in the OPUS program folder to
    simulate a Tensor instrument.
Result = CommandToOpus("DIAG_STATUS")
Print(Result)  //-1: not connected, 0: OK, 1: warning exists, 2:
    error exists.

// Read the path to the public documents folder of OPUS. Will be the
    source for the XPM folder, where a measurement experiment is
    expected to be.
```

Example of a program sequence (in pseudo code) 4

```
Result = CommandToOpus("GET_OPUSPATH")
Print(Result)  // OK, C:\Users\Public\Documents\Bruker OPUS_
    7.5.18.429
OpusPath = split(Result)[1]
//Run a background measurement. Make sure, that a suitable experiment
    is located in the appropriate folder. Set the experiment interac-
    tively with the OPUS function "Measure/Advanced measurement".
// The command TAKE_REFERENCE is a short cut to the normal OPUS func-
    tion MeasureReference(0,{....}). See MeasureSample for informa-
    tion
Result = CommandToOpus("TAKE_REFERENCE " + OpusPath  + "\XPM\experi-
    ment.xpm")
Print(Result)  // OK, 0

//Run a sample measurement. If you want to overwrite one or more
    experiment parameters, you have to use the normal OPUS function
    syntax as created by macro editor or VB-Script editor. NSS is the
    parameter which determines the number of sample scans. Its value
    (here: 1) will overwrite the setting in the experiment.)
Result = CommandToOpus("COMMAND_LINE MeasureSample (, {EXP= experi-
    ment.xpm',XPP='C:\Users\Public\Documents\Bruker\OPUS_7.5.18.
    429\ XPM', NSS=1});")
Print(Result)
//OK
//1
//"<Path and name of the created file as specified in the xpm>"
    <clone number> 1
//54
//0
//Extract the integer in the 4th line, this is the file id, which
    will be used for further accessing of the measured file
File_id = Split(Result)[3]

//A different way to perform a measurement is to do it asynchronic-
    ally, i.e. the measurement is started and the control is immedi-
    ately given back to this program
Result = CommandToOpus("COMMAND_MODE")  //Switch to mode, where com-
    mands are started in a separate thread
Print(Result)  // OK

Result = CommandToOpus("COMMAND_LINE MeasureSample (0, {EXP= experi-
```

Example of a program sequence (in pseudo code) 4

```
    ment.xpm', XPP='C:\Users\Public\Docu-
    ments\Bruker\OPUS_7.5.18.429\XPM'});")

Thread_id = Split(Result)[2]
Result = CommandToOpus("REQUEST_MODE")

Repeat
     //Get the progress of the measurement in %
     Result = CommandToOpus("THREAD_PROGRESS " + Thread_id)
     Print Split(Result)[2]
     Result = CommandToOpus("THREAD_RESULTS " + Thread_id)
Until Split(Result)[2] == 1  //0 : task still running, 1: finished

File_id = Split(Result)[4]

//Manipulate the spectrum (here normalization)
//For detailed explanation of the first argument of the Normalize
    function, see the chapter below
Result = CommandToOpus("COMMAND_LINE Normalize ([ " + File_id +
    ":AB], {NME=1, NWR=1});")
Print(Result)  // OK

//Use this command to specify you want to read a parameter from an
    OPUS file instead of from the OPUS workspace (OPUS_PARAMETERS)
Result = CommandToOpus("FILE_PARAMETERS")
Print(Result)  // OK

//Specify the file id of the OPUS file you want to read the parameter
    from
Result = CommandToOpus("READ_FROM_FILE " + File_id))

Print(Result)
//OK
//"C:\Users\. . . \name.0"
//54

//Read the parameter by specifying the parameter name INS = used
    instrument type
Result = CommandToOpus("READ_PARAMETER INS")
Print(Result)
//OK
```

Example of a program sequence (in pseudo code) 4

```
//Tensor 27 Simulator

//Perform a peak picking on absorbance block of the measured sample.
    Peak picking is an OPUS function from the evaluate menu. Create
    the command line via macro editor to see the meaning of the dif-
    ferent parameters
Result = CommandToOpus("COMMAND_LINE PeakPick ([" + File_id + ":AB],
    {NSP=9, PSM=1, WHR=1, QP8='NO', QP9=0.200000, PTR=20.000000,
    QP4='NO', QP7=0.800000, QP6='NO', QP5=80.000000, PPM=1,
    QPA='OVR', QP0='NO', QP3=4, QPC='NO', QPD=3, QMA=10.000000});")
Print(Result)  // OK

// This command is not necessary, since we already specified it ear-
    lier (same with FILE_PARAMETERS)
Result = CommandToOpus("READ_FROM_FILE " + File_id)
Print(Result)

//Read from the newly created peak block to get the results from peak
    picking
Result = CommandToOpus("READ_FROM_BLOCK AB/Peak")
Print(Result)  // OK

//The Peak pick report has a two column header we want to read from
    the third line (number of peaks found)
Result = CommandToOpus("HEADER_ELEMENT 1 0 3")
Print(Result)  // OK, Number of peaks, 6
//Read all lines from the report matrix, where the peak positions can
    be found.
For I = 1 to split(Result)[3]
Result = CommandToOpus("MATRIX_LINE 1 0 1")
     PeakPosition = split(Result)[2]
     PeakIntensity = split(Result)[4]
Next

//Save the changed file (peak report has been added).
//It's not necessary to specify a block, since the whole file shall
    be saved
Result = CommandToOpus("COMMAND_LINE Save ([" + File_id + "], {});")
Print(Result)  // OK,...

//Now read the spectrum data from the AB block
```

Example of a program sequence (in pseudo code) 4

```
Result = CommandToOpus("READ_FROM_FILE + File_id)
Result = CommandToOpus("READ_FROM_BLOCK AB")
//Read the number of data points and the x-range of the AB-block
Result = CommandToOpus("READ_HEADER")
Print(Result)
// OK
//7466
//3999.35
//399.742
//Specify, that you want the interpolated data at specific x-values.
    Use DATA_POINTS to get the y values as stored in the block.
Result = CommandToOpus("DATA_VALUES")
Print(Result)  // OK, 0
//Read spectrum data in the specified range
Result = CommandToOpus("READ_DATA 3000-2000")
Print(Result)   //Numbers are just samples. See programming.pdf for
    details.
//OK
//2466
//2999.34640113379
//2000.741761097168
//1
//0.0344783
//0.0343538
//0.0342202
// . . .

//Unload the measured file
Result = CommandToOpus("COMMAND_LINE Unload ([" + File_id + "],
    {});")
```

Example of a program sequence (in pseudo code) 4

5 OPUS data blocks and their addressing in the code

The chapter describes how single data blocks of an OPUS file can be addressed in the code.

5.1 Syntax of a command which executes an OPUS function

Each command which corresponds to an OPUS function is structured according to one of the following variants:

```
FunctionName(0, {Parameter1='Value1', Parameter2='Value2', ...});
```

```
FunctionName([FileId1:BlockName1][FileId2:BlockName2]..., [File-
Id3:BlockName3][FileId4:BlockName4]...,{Parameter1='Value1',
Parameter2='Value2', ...});
```

The first variant is used for OPUS functions which have no input data, e.g. the measurement. In the second variant, zero is replaced by a description of the data blocks belonging to one or several OPUS files used in the OPUS function.

Dialogs of an OPUS function may contain list boxes. Each list box, which OPUS data blocks can be moved into by drag & drop, must correspond to the particular area - separated by commas - in the argument list of the command for the OPUS function. Several data blocks within a list box are written in the command without any separator, i.e. directly one after another.

Each data block is addressed by the numerical ID of the file which the data block belongs to, and by the name of the block type. Example: *[37:AB]* (AB = absorption data block). The file ID can be found in the return value of an OPUS function, which generates a new file in the OPUS browser (e.g. *Load* or *MeasureSample*).

In case of the example for the *Spectrum subtraction* function the dialog shown in figure 5.1 corresponds to the following command line:

```
"COMMAND_LINE Subtract ([57:ScSm], [58:ScSm][59:ScSm], {. . .});"
```

The numbers written in the example must be the actual file numbers.

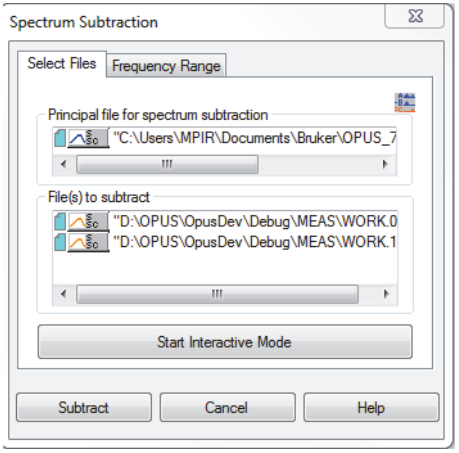

Figure 5.1: Dialog of spectrum subtraction

5.1.1 Data block type

Data blocks can be put up in many ways as:

1. simple data block: e.g. AB, TR, History etc.
2. spectrum block of a 3D file: e.g. AB/Multiple, TR/Multiple etc.[1]
3. spectrum block in connection with a data block; in this case, the spectrum block is either
   - a simple block type: e.g. AB, TR, KM, lgRefl, ScSm etc. or
   - a derivation of a spectrum block: e.g. AB/1.Der., TR/n.Der.[2]

5.1.1.1 Simple data block type

Table 5.1 contains the simple data block types which may occur in connection with a macro.

Spectrum block	Definition
AB	Absorption spectrum
AriAB	Arithmetic result (absorption)
AriTR	Arithmetic result (transmission)
ATR	ATR spectrum
Emis	Emissions spectrum
IgRf	Interferogramm, Reference

Table 5.1: Simple data block type

1. With spectrum blocks of a 3D file the evaluation results are written into the TRACE data block.
2. The evaluation results are attached in the particular data block, e.g. AB/1.Der./Quant.

OPUS data blocks and their addressing in the code 5

Spectrum block	Definition
KM	Kubelka Munk spectrum
lgRefl	Logarithm of reflection
lgSm	Interferogramm, sample
PAS	Photoacoustic spectrum
PhSm	Phase spectrum, sample
PwRf	Power spectrum, reference
PwSm	Power spectrum, sample
Raman	Raman spectrum
Refl	Reflection spectrum
ScRf	Single-channel spectrum, reference
ScSm	Single-channel spectrum, sample
Spec	Ratio spectrum; placeholder for AB, TR, KM and others
TR	Transmission spectrum

Table 5.1: Simple data block type

5.1.1.2 Data blocks of derivation

Data block	Definition
/1.Der.	First derivation
/2.Der.	Second derivation
/n.Der.	n-th derivation

Table 5.2: Data blocks of derivation

5.1.1.3 Data blocks for evaluations and reports

Data block	Definition
/AvRep	Average report
/Class	Classification
/Conf	Conformity test
/Integ	Integration report
/ME	Multi Evaluation
/Multiple	3D file (e.g. map or GC spectrum file)
/Peak	Evaluation result of peak search
/QCompRe-port	Quick compare report

Table 5.3: Data blocks for evaluations and reports

OPUS data blocks and their addressing in the code 5

Data block	Definition
/Quant	Evaluation result of a quantitative analysis
/Report	IDENT report
/Search	Search report
/Subtract	Subtraction report
3DMethod	Method of a 3D spectrum file
Fit	Curve fit
GenRep	General report
Graphics	Video image(s)
History	Audit trail
Info	Information data block
Method	Method
Report	Evaluation result of an IDENT analysis
SampleInfo	Concentration values and names of sample components in case of quantitative analysis
Search	Search result
Strc	Molecular structure
Trace	Spur (Trace)

Table 5.3: Data blocks for evaluations and reports

6 Programming Examples

The following chapters list various programming languages which use different interfaces to OPUS. The chapter title contain the programming language used, and the particular interface for those cases where there is no internal communication with OPUS.

6.1 *Python*: Named Pipe

```
# Program to show the main commands to communicate with OPUS:
Measurement, manipulation, evaluation, reporting.
#

def main():
    print("\nStart of program...")
    global hPipe
    global DefaultPrintout
    global XpmPath
    global XpmName

    # Defining globals
    XpmName = 'python.xpm'
    DefaultPrintout = 2  # set bit 1 to print out the pipe commands,
     set bit 2 to print out the return strings. (3 does both)
    hPipe = open(r'\\.\pipe\OPUS', 'r+b', 0)

    #Check connection by getting the OPUS version
    if (GetVersion() == False) :
        print("Pipe connection failed.")
        return

    # Check if OPUS is connected to an instrument
    if (CheckInstrumentStatus() == False) : return

    #Read OPUS path from workspace
    XpmPath = GetExperimentPath()
```

```
if (XpmPath == '') :
    print ( "Error on trying to evaluate the OPUS path.")
    return
else :
    print ("\nMake sure, you have a suitable experiment file
 defined in: " + XpmPath)
   print ("The experiment file must have the name " + XpmName +
".")


#Begin measurement
if (DoBackgroundMeasurement(1) == False) :
    print ("Background measurement failed.")
    return
print ("\nBackground measurement successfully performed.")

fileid = DoSampleMeasurement(', NSS=1')  # NSS=1 is setting the
 number of sample scans different from the value in the experi-
 ment. All experiment parameters can be overwritten.
if (fileid == 0) :
    print ("Sample measurement failed.")
    return

print ("\nSample measurement successfully performed. Returned
 file id is: " + fileid)

# Manipulate the spectrum: Normalize it from 0 to 2
if (NormalizeSpectrum(fileid) == False) :
    print ("Normalization failed.")
    return

print ("\nNormalization successful. Check spectrum for changed
 axis labels.")

# Reading parameter from OPUS file
instrument = ReadParameter(fileid, 'INS')
if ( instrument == False) :
    print ("Reading parameter for instrument type failed.")
    return

print ("\nInstrument type is: " + instrument)
```

```
    # Evaluate the spectrum: Do Peak picking
    if (PeakPick(fileid) == False) :
        print ("Peak picking failed.")
        return

    # Save file
    if (Save(fileid) == False) :
        print ("Saving failed.")
        return
    print ("\nSaving was successful")


    # Read spectrum data
    SpectralData = ReadData(fileid)
    if ( SpectralData == "") :
        print ("spectrum data reading failed.")
        return
    num, start, end = SpectralData.split(chr(10))[1:4]
    delta = abs((float(end)-float(start))/(float(num)-1))
    print( "\nData read between {0} and {1}. Number of points: {2}.
     Distance between points: {3}.".format(start, end, num, delta))
    print( "\nFirst lines of total return string:\r\n" + Spectral-
     Data[0:300])

    return


#=========================================================================
#=========================================================================

def PipeCommand(cmd, show):
    if (show == -1) : show = DefaultPrintout
    "print out the command or not and shows the result"
    if (show & 1) : print (cmd)
    hPipe.write(bytes(cmd + '\r\n', 'utf-8'))
    data = hPipe.read(1000)
    Mystr = data.decode('utf-8')
    total = Mystr
    if show & 2 : print (total)
    return total
```

```
def GetVersion() :
    return  (PipeCommand('GET_VERSION_EXTENDED', 3) != "")

def CheckInstrumentStatus() :
    result = PipeCommand('DIAG_STATUS', -1)
    arr = result.split(chr(10))
    if (arr[0] == 'OK') :
        number = int(arr[1])
        if (number == -1) :
            print( "\nNo instrument connected. Check 'Measure/Setup
     optics and services'.")
            return False
        if (number == 1) :
            print( "\nInstrument warnings present. Nevertheless
     measurements are possible.")
            return True
        if (number == 2) :
            print( "\nInstrument errors present. Measurement not pos-
     sible.")
            return False
        if (number == 0) :
            print( "\nConnection to instrument is ok")
            return True
    print( "DIAG_STATUS was not successful.")
    return False

def GetExperimentPath () :
    arr = PipeCommand('GET_OPUSPATH', -1).split(chr(10))
    if (arr[0] == 'OK') :
        return arr[1] + '\\XPM'
    return ''

def DoBackgroundMeasurement(do) :
    if (do == 0) : return True
    arr = PipeCommand('TAKE_REFERENCE ' + XpmPath + '\\' + XpmName, -
     1).split(chr(10))
    if (arr[0] == 'OK') :
        return True

def DoSampleMeasurement(addParms) :
```

```
    arr = PipeCommand("COMMAND_LINE MeasureSample (, {EXP='" + Xpm-
     Name + "', XPP='" + XpmPath + "'" + addParms + "});", -
     1).split(chr(10))
    if (arr[0] == 'OK') :
        return arr[3]
    return 0

def NormalizeSpectrum(fileid) :
    arr = PipeCommand("COMMAND_LINE Normalize ([" + fileid + ":AB],
     {NME=1, NWR=1});", -1).split(chr(10))
    return (arr[0] == 'OK')

def ReadParameter(fileid, name) :
    arr = PipeCommand("FILE_PARAMETERS", -1).split(chr(10))
    if (arr[0] == 'OK') :
        arr = PipeCommand("READ_FROM_FILE " + fileid, -
     1).split(chr(10))
        if (arr[0] == 'OK') :
            arr = PipeCommand("READ_PARAMETER " + name, -
    1).split(chr(10))
            if (arr[0] == 'OK') :
                return arr[1]
    return False

def PeakPick(fileid) :
    arr = PipeCommand("COMMAND_LINE PeakPick ([" + fileid + ":AB],
     {NSP=9, PSM=1, WHR=1, QP8='NO', QP9=0.200000, PTR=20.000000,
     QP4='NO', QP7=0.800000, QP6='NO', QP5=80.000000, PPM=1,
     QPA='OVR', QP0='NO', QP3=4, QPC='NO', QPD=3, QMA=10.000000});",
     -1).split(chr(10))
    if (arr[0] == 'OK') :
        print ("\r\nPeak pick on absorbance spectrum successfully
     finished.")
        arr = PipeCommand("READ_FROM_FILE " + fileid, -
    1).split(chr(10))
        if (arr[0] == 'OK') :
            arr = PipeCommand("READ_FROM_BLOCK AB/Peak", -
    1).split(chr(10))
            if (arr[0] == 'OK') :
                arr = PipeCommand("HEADER_ELEMENT 1 0 3", -
    1).split(chr(10))  # Read the number of peaks found
```

```
                if (arr[0] == 'OK') :
                    PeakCount = int(arr[2])
                    print (str(PeakCount) + " peaks found")
                    if (PeakCount > 0) :
                        print ("{pos:20s}{intens:20s}".format(
    pos="Position", intens="Intensity"))
                        for i in range(1, PeakCount) :
                            arr = PipeCommand("MATRIX_LINE 1 0 " +
     str(i), -1).split(chr(10))  # Read the number of peaks found
                            if (arr[0] == 'OK') :
                                print ("{pos:20s}{intens:20s}".for-
     mat(pos=arr[2], intens=arr[4]))
                    return True
    return False;

def Save(fileid) :
    arr = PipeCommand("COMMAND_LINE Save ([" + fileid + "], {});", -
     1).split(chr(10))
    return (arr[0] == 'OK')

def ReadData(fileid) :
    arr = PipeCommand("READ_FROM_FILE " + fileid, -1).split(chr(10))
    if (arr[0] == 'OK') :
        arr = PipeCommand("READ_FROM_BLOCK AB", -1).split(chr(10))
        if (arr[0] == 'OK') :
            arr = PipeCommand("READ_HEADER", -1).split(chr(10))
            if (arr[0] == 'OK') :
                count = int(arr[1])
                begin = float(arr[2])
                end   = float(arr[3])
                arr = PipeCommand("DATA_VALUES", -1).split(chr(10))
                if (arr[0] == 'OK') :
                    hPipe.write(bytes("READ_DATA 5000-0" + "\r\n",
     'utf-8'))
                    total = ""
                    while True :
                        data = hPipe.read(1000)
                        Mystr = data.decode('utf-8')
                        total += Mystr
                        if (Mystr[-3]+Mystr[-2] == "OK") : break
                    return total
```

```
    return ""

if __name__ == "__main__":
    main()
```

6.2 *PERL*: http

i In case of the http interface, the simulator cannot be used instead of a spectrometer type, as both devices communicate via port 80. With OPUS 8.5 and higher you can explicitly define the simulator port. When starting *simulator.exe* you have to use the */PORT=<n>* command line option, with *<n>* being the port number. This allows the simultaneous use of the simulator and the http interface.

```
use strict;
use Cwd;

# Get working dir
my $homedir = getcwd;

# How to remote control OPUS from a website
#
# You must have a running instance of OPUS before you can perform
    remote control.
# Start OPUS with the appropriate command line or use the Create pro-
    cess of Win32 as shown below.

use Win32::Process;  # to create an OPUS instance
use LWP::Simple;     # to communicate with OPUS via http protocol

my $opus;  # not needed to get the instance run

if (! OpusAlreadyStarted())
{
    Win32::Process::Create($opus, "d:\\opus\\opus.exe",
/HTTPSERVER",
0, NORMAL_PRIORITY_CLASS, ".") || die ErrorReport();
```

```
# If you use HTTPSERVER as a command line argument you must assure,
# that the path to the userdatabase is a path available from the net-
    work.
# (this is due to the SERVER option of OPUS, where all user informa-
    tion is gathered from network, not locally)

    sleep(20); # Allow OPUS to be started
}

# Easy command: get the version date from OPUS. The answer string is
    directly returned from get
MyCommand('GET_VERSION', 3);

# If connected to an instrument (e.g. a TANGO) you can start a
    measurment using an exisiting experiment.
# Check for correct paths in the example
my $result;
$result = MyCommand('DIAG_STATUS', 3);
#print ("--", (split(/\n/, $result))[1], "--\n");
die "Instrument not connected!\n" if (split(/\n/, $result))[1] eq "-
    1" ;
MyCommand('REQUEST_MODE', 3);  # Waits for the command to return
MyCommand('COMMAND_MODE', 3);  # starts new thread and returns imme-
    diately (thread id will be returned)
$result = MyCommand("COMMAND_LINE MeasureReference (0,
    {EXP='Trans.xpm', XPP='$homedir/XPM'});", 3);
my $id = (split(/\n/, $result))[2];
TrackThreadProgress($id); # Track the progress until the measurement
    is finished

$result = MyCommand("COMMAND_LINE MeasureSample (0,
    {EXP='Trans.xpm', XPP='$homedir/XPM'});", 3);
$id = (split(/\n/, $result))[2];
my $file = TrackThreadProgress($id);
print ( "file created: $file\n");
# Make manipulation, then peak pick
MyCommand('REQUEST_MODE', 3);  # Waits for the command to return
MyCommand("COMMAND_LINE Normalize ([$file:AB], {NME=1, NWR=1});",
    0);
MyCommand("COMMAND_LINE PeakPick ([$file:AB]0, {PSM=1, WHR=1,
    PTR=7.000000, PPM=1, QPA='OVR'});", 0);
```

```
MyCommand("READ_FROM_FILE " . $id, 0);
MyCommand("READ_FROM_BLOCK AB/Peak", 3);
my $number = (split(/\n/, MyCommand("HEADER_ELEMENT 1 0 3", 0)))[3];
printf( "%8s  %8s\n",  " Pos (cm-1)", "Intens.");
for (1..min(5,$number))
{
printf( "%8.1f  %8.2f\n",  (split(/\n/, MyCommand("MATRIX_LINE 1 0
    $_", 0)))[2,4]);

}

exit;

sub  OpusAlreadyStarted()
{
my $list = `tasklist.exe`;
return $list =~ m/OPUS.exe/im;
}

sub TrackThreadProgress
{
my $id = shift;
my $res;
die "'$id' not numerical" if !$id;
MyCommand('REQUEST_MODE', 3);
while (1)
{
sleep(2);
$res = MyCommand("THREAD_RESULTS $id", 0);
last if (split(/\n/, $res))[2];
$res = MyCommand("THREAD_PROGRESS $id", 0);
print ( (split(/\n/, $res))[2], " %\n");
}
return (split(/\n/, $res))[4];  # Return file id
}


sub MyCommand
{
  my $cmd = shift;
  my $flag = shift;
```

```
  $cmd = "http://localhost/OpusCommand.htm?$cmd";
  print "$cmd\n" if $flag & 1;
  my $result = get($cmd);

  print "$result\n" if $flag & 2;
  return $result;
}

sub min {return ($_[0] <= $_[1] ? $_[0] : $_[1]);}
```

6.3 OPUS Macro

More example macros are saved in the *Public\Public Documents \Bruker\OPUS_<version>\Macro* directory.

```
VARIABLES SECTION

FILE <$ResultFile 1> = AB, AB/Peak;
NUMERIC <PeakCount> = 0;
NUMERIC <i> = 0;
STRING <Position> = '';

PROGRAM SECTION

MeasureReference (, {EXP='sim3.xpm', XPP='C:\Users\Public\Docu-
    ments\Bruker\OPUS_7.5.18.429\XPM'});
<$ResultFile 1> = MeasureSample (0, {EXP='sim3.xpm',
    XPP='C:\Users\Public\Documents\Bruker\OPUS_7.5.18.429\XPM'});
Baseline ([<$ResultFile 1>:AB], {BME=2, BCO=0, BPO=64, BIO=10});
PeakPick ([<$ResultFile 1>:AB], {NSP=9, PSM=1, WHR=1,
    LXP=400.000000, FXP=4000.000000, QP8='NO', QP9=0.200000,
    PTR=20.000000, QP4='NO', QP7=0.800000, QP6='NO', QP5=80.000000,
    PPM=1, QPA='OVR', QP0='NO', QP3=4, QPC='NO', QPD=3,
    QMA=10.000000});
<PeakCount> = FromReportHeader ([<$ResultFile 1>:AB/Peak], 1, 0, 3,
    RIGHT);
If (<PeakCount>, .GT., 5);
<PeakCount> = 5;
Endif ();
```

```
StartLoop (<PeakCount>, 0);
<i> = <i> + 1;
<Position> = FromReportMatrix ([<$ResultFile 1>:AB/Peak], 1, 0, <i>,
    1);
Message ('Peak Posittion <i>: <Position>', ON_SCREEN, 2);
EndLoop (0);

PARAMETER SECTION
```

6.4 VisualBasic Script

If the OPUS installation has been finished, more examples for VisualBasic Scripts are available in the *Public\Public Documents\Bruker\OPUS_<version>\VBSample* path.

The script described in this chapter contains only one button (*CommandButton1*) used to start measurement and evaluation, as well as one list box (Listbox1) which shows the evaluation results.

Figure 6.1 shows the start dialog displayed after the VisualBasic Script has been started.

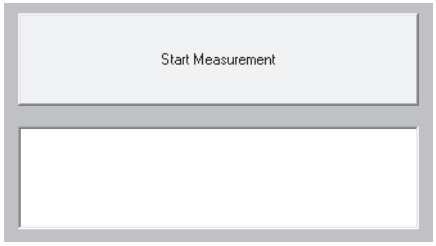

Figure 6.1: Start dialog of VisualBasic Script

After clicking the *Start Measurement* button, the actions contained in the code are executed and the results are displayed, e.g. an integration of the spectrum measured (figure 6.2).

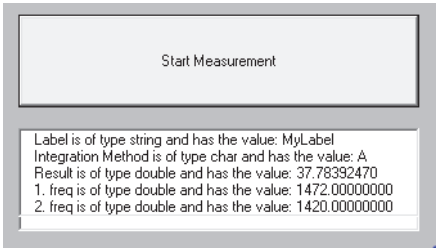

Figure 6.2: VisualBasic Script - Integration result

The code could be as follows:

```
option explicit

' This function is called on creating the user interface at the start
    of the script.
Sub Form_OnLoad
  CommandButton1.Caption = "Start Measurement"
End Sub

' This is the handler of the click on the button of the user inter-
    face
Sub CommandButton1_Click
  'Empty Listbox
  while Listbox1.ListCount >=1
    Listbox1.RemoveItem (0)
  wend

  dim Result, file_id
  ' Do a background measurement
  Result = Form.OpusRequest("COMMAND_LINE MeasureReference (,
    {EXP='sim3.xpm', XPP='C:\Users\Public\Documents\Bruker\
    OPUS_7.5.18.429\XPM');")
  if InStr(Result, "OK") = 1 then
    ' Do a sample measurement
    Result = Form.OpusRequest("COMMAND_LINE MeasureSample (,
      {EXP='sim3.xpm', XPP='C:\Users\Public\Docu-
      ments\Bruker\OPUS_7.5.18.429\XPM');")
    if InStr(Result, "OK") = 1 then
```

```
  ' get the file_id
  file_id = split(Result, chr(10))(3)
  ' Do a baseline correction
  Result = Form.OpusRequest("COMMAND_LINE Baseline ([" & file_id
& ":AB], {BME=2, BCO=0, BPO=64, BIO=10});")
  if InStr(Result, "OK") = 1 then
    ' Do an integration (we use no integration method, but give
the integration parameters directly)
    Result = Form.OpusRequest("COMMAND_LINE Integrate ([" &
file_id & ":AB], {SDI=1472, EDI=1420, MDI='A', ILN='MyLa-
bel'});")
    if InStr(Result, "OK") = 1 then
      'Specify the file to be read from
      Result = Form.OpusRequest("READ_FROM_FILE " & file_id)
      'Specify the block to be read from
      Result = Form.OpusRequest("READ_FROM_BLOCK AB/Integ")
     'Read from the first report, read from the main report (sub-
report=0), read from the 1st line
      Result = Form.OpusRequest("MATRIX_LINE 1 0 1")
      if InStr(Result, "OK") = 1 then
        dim i, typ, value, column
        for i = 0 to 4
          typ = split(Result, chr(10))(1+2*i)
          value = split(Result, chr(10))(2+2*i)
          column = ""
          if typ = "0" then typ = "long"
          if typ = "1" then typ = "double"
          if typ = "2" then typ = "string"
          if typ = "3" then typ = "enum"
          if typ = "10" then typ = "bool"
          if typ = "11" then typ = "char"

          if i = 0 then column = "Label"
          if i = 1 then column = "Integration Method"
          if i = 2 then column = "Result"
          if i = 3 then column = "1. freq"
          if i = 4 then column = "2. freq"
          Listbox1.AddItem(column & " is of type " & typ & " and
has the value: " & value)
        next
       exit sub 'Everything was ok, skip the global error message
```

```
            end if
          end if
        end if
      end if
    end if
    msgbox "Something went wrong. Check the result values for each com-
      mand call."
End Sub
```

6.5 *C*: Named Pipe

The code for an exemplified program is supplied with the OPUS installation. The code by pipetest.c is provided in the *C:\User\Public\Public Documents\BRUKER\OPUS_<version>\VBSample* path.

Index

Index