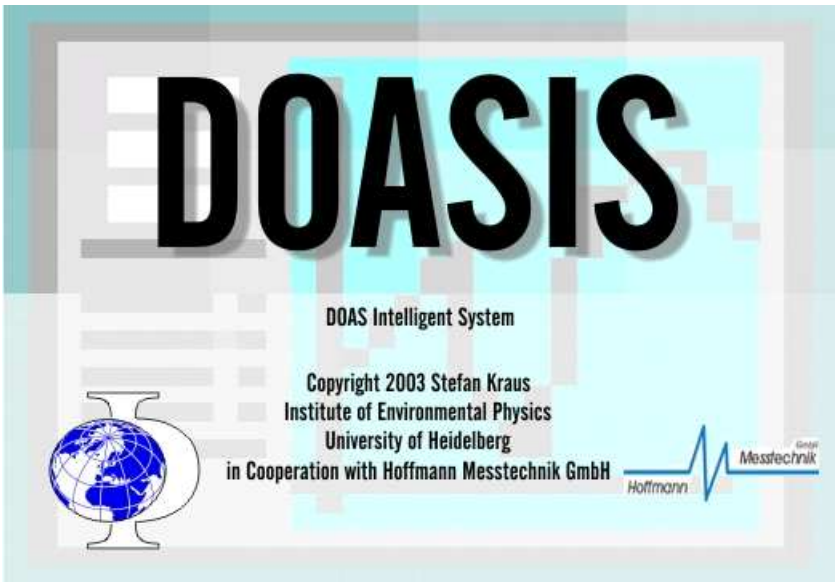
JScript Tutorial

Wolfgang Unger and Tobias Sommer
(tsommer@iup.uni-heidelberg.de)
24.06.2008



Contents

I Introduction **4**

1 What is JScript? **5**

2 For whom this Tutorial is written **7**

3 What is special about this Tutorial? **8**

4 Tips and Tricks **9**
4.1 The help menu of DOASIS 9
4.2 Structuring a JScript program by creating a JScript Project File 9
4.3 Running several JScripts after each other 9
4.3.1 Creating a JScript project file 9
4.3.2 Creating a batch file 9
4.4 Starting a JScript automatically after a hardware reboot 10
4.5 Avoiding inexplicable program crashes 10

II The basics of JScript **11**

5 First steps in JScript **12**
5.1 A first example 12
5.2 A word on "object-orientation" 12
5.3 A second example 13
5.4 The syntax 14

6 Programming in JScript **15**
6.1 Data Types and Type Conversion 15
6.2 Operators 17
6.2.1 Logical Operators 18
6.3 Program Flow 18
6.3.1 if..else-statements 18
6.3.2 for-Loops 20
6.3.3 for..in-Loops 21
6.3.4 while-Loops 21
6.4 Functions 21

7 Important DOASIS-namespaces and their usage **23**
7.1 Overview 23
7.2 DoasCore.Math 25
7.2.1 DoasCore.Math.SpecMath 25
7.2.2 DoasCore.Math.ScanGeometry, DoasCore.Math.JulianDateTime . . . 25
7.2.3 DoasCore.Math.DoasFit 26
7.3 DoasCore.Device 27
7.4 DoasCore.IO 28

7.5 DoasCore.Spectra . . . 28
7.5.1 DoasCore.Spectra.ISpectrum . . . 29
7.5.2 DoasCore.Spectra.Specbar . . . 29
7.6 DoasCore.Script . . . 30
7.7 DoasCore.HMI . . . 31
7.7.1 Progress Bar . . . 31
7.8 System . . . 31
7.9 System.Console . . . 31
7.10 System.Windows.Forms . . . 32
7.11 System.IO . . . 32
7.12 System.Threading . . . 32

III Collection of JScript applications 34

8 Simple JScripts 35
8.1 Hello World . . . 35
8.2 Diplaying variables in the Output window. . . . 35
8.3 Wait functions . . . 36
8.4 Display message boxes . . . 37
8.5 Creating a directory . . . 37
8.6 Creating a text file . . . 38
8.7 Get the current date and time . . . 39
8.8 Control number of digits . . . 40
8.9 Open and Save a spectrum . . . 41
8.10 Saving spectra with the “AutoFileName”-class . . . 42
8.10.1 A word on AutoFileName() . . . 43
8.11 Opening spectra with the “AutoFileName”-class . . . 43
8.12 Modifying the properties of a spectrum . . . 45
8.13 How to compare the time of different spectra . . . 45
8.14 Calculate the solar zenith angle (SZA) . . . 47
8.14.1 A word on the class ScanGeometry . . . 48
8.15 Wavelength Calibration . . . 48

9 JScript Project File 49

10 JScripts communicating with hardware 51
10.1 Reading in data from a serial port (e.g. GPS data) . . . 51
10.2 Measure a spectrum with an Ocean Optics spectrograph . . . 53
10.3 Operate two Ocean Optics spectrographs . . . 54
10.4 Mini-MAX-DOAS (MiniDOAS) control via JScript . . . 54

11 Evaluating spectra 55
11.1 Offset and Dark Current Correction . . . 55
11.2 Offset and Dark Current Correction with a “JScript project file” . . . 56
11.2.1 Preparation . . . 56
11.2.2 The “project file“ . . . 56
11.2.3 The file *variables.js* . . . 56
11.2.4 The file *functions.js* . . . 57
11.2.5 The file *maincorrect.js* . . . 57
11.3 Fitting with JScript . . . 58
11.4 Modifying a fit scenario with JScript . . . 59
11.5 Save the residual of the fit result . . . 62

12 Advanced JScripts **65**
12.1 Write spectrum data to an Excel-Sheet 65
12.1.1 A word on ActiveXObjects 66
12.2 Exception handling 67
12.3 Evaluate data with a fit scenario file 69
12.4 Automatized scanning 70
12.5 Threading 71
12.6 Controlling Mini-MAX-DOAS (MiniDOAS) instruments with JScript 73
12.6.1 Structure of the program 73
12.6.2 Description of the tasks 74
12.6.3 Prerequisites 74
12.6.4 Remarks: 84

13 Literature **85**

Index **86**

Part I

Introduction

Chapter 1

What is JScript?

JScript is a script language which can be used within an application. In DOASIS, this makes possible to automatize measuring and evaluating data. For example, let's say you want to record a huge number of spectra, and you want to adjust the parameters (e.g. the angle) after each measurement. Since you don't want to start a new measurement every 5 minutes, you can use a script which tells DOASIS what measurements to make.

A script language is a programming language, but you are not able to write executable programs with them. Instead, the scripts are started from the program for which the script was written. Scripts are not compiled, but interpreted. That means, the script is not at once translated to machine language and then executed, but read and executed line by line. Resources that your script makes use of (namespaces, objects) are precompiled.

In DOASIS, everything that you need concerning scripts can be found in the Console-Window. More precisely, there are four tabs: the Output-, the Script-, the Mini Scrips- and the Quick Script-Tab.
A script can be started from the Script-Tab in the Console-Window (by specifying the file in which the script is stored) or by entering a script in the Mini-Script-Tab. The output that is generated by the Script can be viewed in the Output-Tab.

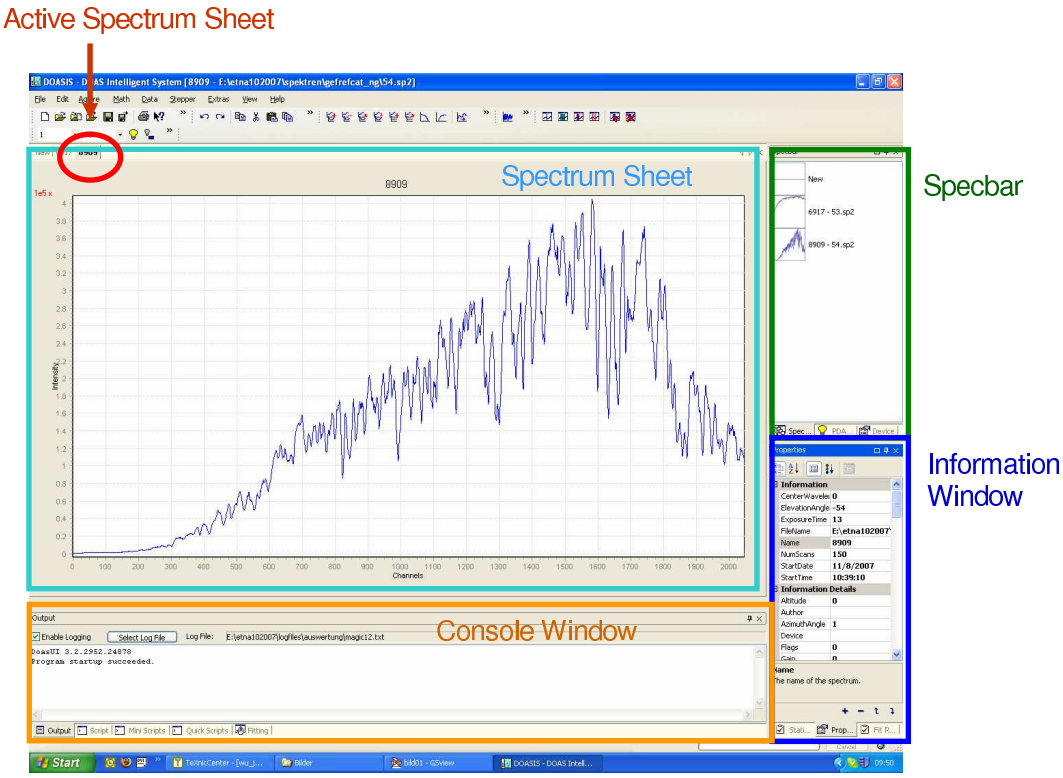


Figure 1.1: The main application window of the graphical user interface part *DoasUI*.

Chapter 2

For whom this Tutorial is written

If you are fairly good at programming and you know Java or JavaScript, this is not for you. In fact, JScript could also be called the "Mircosoft JavaScript". There quite a lot similarities, although Microsoft emphasizes that the technology behind JScript is very different from the JavaScript by Netscape.
But if you are a poor programmer, or even never learned programming, but still want to use *all the great possibilities of DOASIS-JScript* without bothering too much on aspects of programming in general, then this is written for you! Here you will learn just what you need to write JScripts in DOASIS.

Maybe you have experience with other script languages like VBA ("Visual Basic for Applications"). But this will be little help, since JScript has a different syntax, orientated at C/C++. I would recommend to read this anyhow, since there is a lot DOASIS-specific to learn about.

Chapter 3

What is special about this Tutorial?

The Tutorial is split into two parts. In part II, the basic features of JScript and its use in DOASIS are explained. This is more a general overview and interesting for people, who have non or only very little programming experience. If you already have a general idea about programming it is recommended to skip Part II and continue directly with Part III. Part III consists of a selection of practical source code examples, which have been or are still in use by different working groups which use DOASIS for measurement and evaluation purposes. The code examples are explained in a way which can be easily understood and the important commands are commented out.
The code examples can be "**copy-pasted**" directly from your pdf-Reader program (e.g. Adobe Acrobat Reader) to a text editor, then saved with the suffix ".js" and opened within DOASIS in the sheet "Script" of the "Console window". If the programs are very short, than it might be more convenient to "copy-paste" them directly into the "Mini Scripts" sheet of the console window and press "Start". (See as well 5.3)

To get fast access to the commands you are looking for, take advantage of the **Index**, that will guide you to the requested sections.

Chapter 4

Tips and Tricks

4.1 The help menu of DOASIS

The Programming Documentation in the Help menu of DOASIS provides information on all the JScript commands that access the DOASIS source code. Each button or property that can be activated or set in DOASIS can be accessed through JScript as well. This is due to the object orientated programming of DOASIS (with C#, similar to C++). Just press "Help" in the main menu of DOASIS, then "Programming Documentation" and then e.g. under "Search" the command, about which you want to get more information.

4.2 Structuring a JScript program by creating a JScript Project File

A lot of programming work can be saved if, from the beginning, the program is structured in a way, that methods and variables are defined "generally" in a separate file, so that new programs can refer to them as well. It is strongly recommended to establish so called "JScript-projects". In section 9, more information is provided.

4.3 Running several JScripts after each other

If several JScripts should be executed automatically directly after each other two possibilities are presented in the following.

4.3.1 Creating a JScript project file

By creating a JScript project file, the individual JScripts can listed after each other and will be executed successively. See section 9.

4.3.2 Creating a batch file

The advantage of creating a batch file is that the script runs much faster compared to when the script is started within DOASIS. This is because the grafical user interface (GUI) of DOASIS is not used and therefore does not consume calculation time.
Open the text editor and write:

C:\Programme\DOASIS\DoasConsole.exe C:\scripttest\jscript1.js >>C:\logfile1.txt

Save this file for example under *C:\startscript.bat*. By double clicking on the batch file

the DoasConsole of DOASIS is started by *C:\Programme\DOASIS\DoasConsole.exe* and *jscript1.js* (stored in *C:\scripttest\jscript1.js*) is executed. The command *>>C:\logfile1.txt* is optional. It stores the output of the program, which is usually displayed in the Output-Tab of DOASIS in the file *C:\logfile1.txt*. In case the output is stored in a file, it is not shown while the program is executed. The suffix ".bat" is the identifier for the "batch"-file. A batch file contains DOS commands which are executed successively.

Warning:
Make sure that the path name does not contain blanks since DOS does not support them.

Several JScript can be successively run simply by adding more lines to the batch file:

C:\Programme\DOASIS\DoasConsole.exe C:\scripttest\jscript1.js >>C:\logfile1.txt
C:\Programme\DOASIS\DoasConsole.exe C:\scripttest\jscript2.js >>C:\logfile2.txt

4.4 Starting a JScript automatically after a hardware reboot

Measurements controlled by JScripts in remote areas often don't have a reliable power supply. After power is cut, the hardware is often designed in the way to cause a reboot when power is back. In that case the JScript should start by itself after the reboot. This can be done by creating a batch file as described in section 4.3.2 which is executed automatically after the reboot. In the case of Windows XP, a shortcut of the batch file can be placed in the Startup-folder (Start - Programs - Startup). For other operating systems, it should work similarly.

4.5 Avoiding inexplicable program crashes

In the past, it happened quite often that a measurement JScript worked well for hours but then suddenly crashed without any apparent reason. It can be that such crashes are due to "bad" communication between the computer and the measurement device. Especially if the electronics of the measurement instrument are fairly old it can happen that the instrument gets response from the computer, even though it hasn't yet finished its own sending procedure. This can lead to a system crash.
In order to avoid this sort of error, "wait" functions should be implemented after each data transfer. This makes the program more stable. How to implement wait functions is described in section 8.3. By implementing wait functions in the script of section 12.6 the script continued measuring for several months without interruption.

Part II

The basics of JScript

Chapter 5

First steps in JScript

5.1 A first example

Now let's just start to write a JScript! For beginners, it is common to write a so-called "Hello World"-program, which simply writes these words in the output. So that is what we will do first. You can enter the code into the Mini-Script-Window, and then click on start. Here is the code:

```
import System;

System.Console.WriteLine("Hello World!");
```

If you switch to the Output-Window, you'll see the result. Congratulation to your first JScript! But how did it work? There are some remarks I want to make here:

- First of all, we "imported" the "System". This just means that we will make use of certain functions made available by a "namespace" (a kind of library) by that name. Later on, we will import other namespaces, which will be then specific to DOASIS. How else should the interpreter know how to deal with DOASIS-specific commands?!?
- Second, we produced on output by using `System.Console.WriteLine(...);`. The object that we used as a parameter was a static string, which is characterized by the quotation marks. We could also use other parameters, as our next example will show.
- In both commands, we had to finish the line with a semicolon. This is important, because this tells the interpreter when a command ends. You could put the whole code into one line, and the interpreter would still get it right because of the semicolons. But never forget the semicolons, because this most certainly will produce an error!

5.2 A word on "object-orientation"

You might have noticed the dots between the words of the second line of our first script. This is typical for so-called object-oriented programming languages. If you don't know anything about object-orientation, don't panic, you won't need it anyway. The only thing you need to know is that there is a kind of hierarchy of objects. The dot could be read as "belongs to". In our example, the Console-object belongs to the System, and the WriteLine-Method (a method is kind of function or sub-routine) belongs to the System.Console-object. You

need to mention all the names because there could be a WriteLine-method as part of another object with different name. The interpreter wouldn't find the method otherwise. This is all about object-orientation so far. Wasn't that difficult, right?

5.3 A second example

Now we want to get more professional, and instead of writing a mini script, we will edit the code in an external file.
So please open your favorite text editor (e.g. PSPad, freeware, very good!) but not Word etc., since they don't save in a plain text format) and enter the following script:

```
import System;
import DoasCore.Spectra;

var Filename;
Filename = Specbar.CurrentSpectrum.Name;
System.Console.WriteLine(Filename);
```

Then save it as a "filename.js". Notice that ".js" is the file extension for JScript. Now you need to specify the file (with the whole path) at the Script-Tab. Alternatively you can browse the file. Before you start the script, open your favorite spectrum in DOASIS. Now please start the script. As you can see in the Output-Tab, the script returns the name of your spectrum. So let us learn some new things from the script:

- This time, we not only imported the System, but also the DoasCore.Spectra namespace. In this namespace all the objects and methods are defined that deal with spectra.
- Then we defined a variable. In other program languages, you need to worry a lot about different types of variables, not so in JScript!!! In the declaration of a variable, no difference is made between integers or floats or strings. All you need to do is to tell that you want to declare a variable. This is done by the `var` command. In our script, we declared the variable "Filename", which is an arbitrary name and you could have used (almost) anything else. Unfortunately, sometimes you need to worry about types nevertheless. Once a variable contains data of a certain type (we assigned a string to "Filename"), you can only use it as a different type later on if a so-called type-conversion is possible. We will come back to this point later on.
- Look what we did next: we assigned a value to our new defined variable, more precisely the name of the current spectrum in the Specbar, i.e. the spectrum that you just opened before starting the script. Assigning a value to a variable is pretty simple: just use a `=`. The right side is then assigned to the left side. The left side therefore always has to be a variable, the right side can also be a constant, an explicit expression (like a number) or a more complicated algebraic expression, a string, or an array. We will see other examples later on.
- Last, we just wrote the value of our variable in the output. And indeed, it is the name of your spectrum, as it should.

Hint: Scripts can also be started from the sheet "Mini Scripts" of the "console window". Advantage: this is faster.

5.4 The syntax

So far, we have already learned that every command has to be completed with a semicolon. There are other syntax rules, i.e. rules that specify how a script should be written in order to be understood by the interpreter. We will discuss them by looking at the next example:

```
//This is an example that illustrates the JScript Syntax

import System;
import DoasCore.Spectra;

var ABCDEFGH;
var abcdefgh;
var aBcDeFgH;
var A123;
/* the text between the stars
will be ignored */
Filename = Specbar.CurrentSpectrum.Name;
System.Console.WriteLine(Filename)
```

- The script starts with a so-called comment, something that the interpreter will ignore since it appears after the comment-symbol "//". Comments that shall extend over several lines, are put between "/*" and "*/".
- It is important that all import-commands are put at the beginning of the script.
- JScript is a case-sensitive language. That means, it does matter if variables, objects or commands are written in uppercase or lowercase. If two names differ in their cases, the interpreter will distinguish between them. So only refer to variables in the way you declared them!

Chapter 6

Programming in JScript

This part of the introduction into the more general aspects of programming in JScript, concerning data types, operators, loops, for-blocks, if-blocks, and more, aims to give you a basic idea of the structure of typical JScripts. Most of the commands are identical to the Java or C/C++ language. If you never came in touch with one of those programming languages, you might also want to consult other learning materials as given in the literature below to learn about details which can not be covered here. Here, you will find the essentials that enable you to write powerful JScripts.

6.1 Data Types and Type Conversion

So far, we only have used so-called untyped variables, i.e. those that are declared by using the `var`-command. In most cases you need not to worry about data types. But sometimes it might be better to declare explicitly the type of a variable. In the table below you can see some examples of the declaration of typed variables. We will now discuss more detailed what data types are, and when it is useful to use them.

In short, data types are specifications of how to handle variables. The interpreter needs to know how much memory he shall reserve for the data, and what can be done with these. For example, the data type integer is of size 32 bit, and you can add, subtract, multiply and divide (with rest) integers. By doing so, another integer is produced. Different data types differ in memory, in how they are interpreted, and in what can be done with them. As another example, strings cannot be added like numbers, but linked, which is also done by the +. Here is a list of data types available in JScript:

To understand the difference between typed and untyped variables, let's compare the following scripts:

```
//A well working script

import System;

var s = 3;
s=s+5;
Console.WriteLine(s);
s = "I'm not an integer!";
Console.WriteLine(s);
```

Data Type	Description	Example
char	a single character, enclosed in single quotation marks	`var c: char = 'a';`
String	a sequence of characters, enclosed in double quotation marks	`var s: String = "Hello";`
int	a 32-bit integer value, ranging from -2.147.483.648 to 2.147.483.647	`var i: int = 1;`
uint	a unsigned 32-bit integer, ranging from 0 to 4.294.967.295	`var i: uint = 1;`
boolean	a boolean value that is either true or false	`var b: boolean = false;`
double	floating point number of double precision (about 15 digits)	`var d: double = 0.123;`
Date	object representing a date and time definition	`var date: Date = "10/08/2003 15:45";`

```
//Example of a type mismatch

import System;

var s:  int = 3;
s = s + 5;
Console.WriteLine(s);
s = "I'm not an integer!";
Console.WriteLine(s);
```

The first script will work perfectly, whereas the second script will produce the error "Type mismatch".

Why then should it be useful to use typed variables if they just produce more errors? The answer is: typed variables are less confusing and protect you against misuse. Look at these examples:

```
import System;

var s = 5;
s = "I'm a string!";
s = s + 3;
Console.WriteLine(s);
```

Output: "I'm a string!3"

```
import System;

var s:  int = 5;
s= "I'm a string!";
s = s + 3;
Console.WriteLine(s);
```

You will get the error message: "type mismatch"

Warning: *If you do not initialize a variable, i.e. if you declare a variable without setting it to a certain value, requesting the value will return NaN (Not a Number) or "undefined", depending on the context in which the variable was interpreted. For example,*

```
import System;
var i;
Console.WriteLine( "Value of i:  " + i);
Console.WriteLine( "Value of 2*i:  " + 2*i);
```

will produce the output "Value of i: undefined" and "Value of i: NaN", because in the first case we have a type conversion to string, in the second to double and then to string.

6.2 Operators

You already know mathematical operators such as +,-,*,/. In a programming language, they can indeed be used to add/multiply numbers (such as integer, double), but there are also other usages. As we have already seen, you can also connect two string variables to one string with the +. But there are some more operators that we have to learn about.

Here is a list of the available operators:

Operator	Description
. [] ()	member access, array access, function call
++ – - !	unary operators
typeof new void	specify return type, object instantiation, undefined value
* / %	multiplication, division, modulo-division
+ - +	addition, subtraction, string concatenation
<< >> >>>	bit-shift
< <= > >=	smaller as, smaller or equal, greater as, greater or equal
== !=	equality, inequality
&&	logical AND
||	logical OR
&	bit-wise AND
^	bit-wise XOR
|	bit-wise OR
?:	conditions
= += -= *= /=	assignment (also with operators)
,	multiple evaluation

6.2.1 Logical Operators

Logical operators enable you to evaluate boolean variables and expressions.

```
import System;

var a:  boolean = true;
var b:  boolean = false;
Console.WriteLine("a AND b = " + (a && b));
Console.WriteLine("a OR b = " + (a || b));
```

Output:
”a And b = false”
”a OR b = true”

6.3 Program Flow

There are many ways to structure the program flow. For example, you might want to execute a certain command only if a variable has a certain value, or you might want to repeat a code block for a certain number of times. Now you will learn how to do that.

6.3.1 if..else-statements

The if..else statement is used to split the flow of your program in two or more branches. Usually, the value of a variable is compared to a certain constant value, and depending on if the comparison holds or not, different code is executed. Let’s start with a simple example:

```
import System;

var i=Math.random();
if (i<0.5) Console.WriteLine("This number " + i + " is smaller than 0.5");
else Console.WriteLine("This number " + i + " is larger than or equals 0.5");
```

Here, the code following the if-condition is executed if the condition is true, and the code following the else-statement is executed if the statement is false. This is what you should notice:

- the boolean expression has to be set in parenthesis
- the if-clause here only consists of one command. It has to be closed by a semicolon
- the else-clause also only consists of one command and has to be closed by a semicolon
- the else-command has to be the next command after the if-clause.

What if we have more than two cases? Here another example that illustrates this point:

```
import System;

 var i=Math.random();
if (i<0.5) Console.WriteLine("This number " + i + " is smaller than 0.5");
else if (i>0.5) Console.WriteLine("This number " + i + " is larger than 0.5");
else Console.WriteLine("This number " + i + " equals 0.5");
```

What if you have more than one command that you want to execute in an if- or else-block? Simply put them between braces as the next example illustrates. This can also be done in other contexts, for example in loops, as we will see later.

```
import System;

var i=Math.random();
if (i==0.5) {
    Console.WriteLine("This number " + i + " equals 0.5");
    Console.WriteLine("Isn't that a nice result?");
}
else {
    Console.WriteLine("This number " + i + " is not equal to 0.5");
    Console.WriteLine("Sorry, can't tell you if < 0.5 or > 0.5.");
}
```

Warning: *Always remember that equality-comparisons are done by "==", not "=". If you use "=", there will not be an error as long as the right-hand side can be assigned to the left-hand side. So you have to be careful.*

Hint: There is a way to shorten the if..else-statements. The if can be replaced by "?", the else by ":". This short notation allows to use if..else-statements even within arguments. For example, all of the following lines are equivalent:

```
if (i==5) Console.WriteLine("five"); else
Console.WriteLine("another number");
(i==5) ?  Console.WriteLine("five"):  Console.WriteLine("another
number");
Console.WriteLine((i==5) ?  "five" :  "another number");
```

6.3.2 for-Loops

The for-loops allow to repeat a command or block of code for a certain time. Usually, a counter is counted up until a certain condition does not hold any more. Here is an example:

```
import System;

var i;
for (i=1;i<10;i++) Console.WriteLine("This is the " + i + ".  line.");
```

Our example produces 9 lines. A for-loop has the following syntax:

- After the for-command, which opens the for-loop, there is a loop-header consisting of three statements, which are separated by semicolons.
- The first statement in the parenthesis is the initializing command. It is executed only once, just before the loop is entered. Usually, the loop variable (here: i) is set to a certain value.
- The second statement is a boolean expression. The loop is (re)entered only if the boolean expression is true. Usually, the loop variable is compared to a certain value which it shall not exceed.
- The third statement is a command that is executed every time the loop was swept. Usually, the loop variable is incremented here.
- The loop itself, the command that is executed several times, follows the loop-header. If you use more than one command, make a block with braces.

6.3.3 for..in-Loops

```
import System;

var weekday = ["monday", "tuesday", "wednesday", "thursday"];
var j;
for (j in weekday) Console.WriteLine(weekday[j]);
```

6.3.4 while-Loops

The while-loops are similar to the for-loops, but the header of the while loop has no section for initializing and no section for incrementation, but just the section for a comparison, the while-condition. This of course only makes sense if the while-condition changes during sweeping the while-loop. Here is an example:

```
import System;

var steps = 10;
while (steps >= 1) {
    if (steps > 1) Console.WriteLine(" There are" + steps + " to do.");
    else Console.WriteLine("Now there is only one more step");
    steps--;
}
```

6.4 Functions

In JScript it is possible to declare classes functions. A function is a piece of code that is encapsulated and can be executed by calling the command under which the function is defined. This has the advantage that code which is used in several places of the script has only to be written once. For the declaration of classes see section 12.5.
Here is an example that calculates the factorial n! of the number n.

```
import System;

function factorial(n)
{
    var i;
    var f=1;
    for (i=1;i<=n;i++) f=f*i;
    return f;
}

Console.WriteLine(factorial(8));
```

The types of the parameters and the return parameter are *not* specified in the declaration of the function (contrary to C/C++/C#). If your function shall not return anything, just omit the return line. If your function takes no parameters, leave the brackets empty, but do not forget the empty brackets ().

Chapter 7

Important DOASIS-namespaces and their usage

7.1 Overview

All the DOASIS functionality is entailed in the DoasCore. There are six namespaces of interest, which need to be imported in order do use its classes (i.e. objects and methods). These are DoasCore.Math, DoasCore.Device, DoasCore.IO, DoasCore.Spectra, DoasCore.Script and DoasCore.HMI. (compare Fig. 7.2).
We will also discuss some namespaces and classes that are offered by the .NET framework, such as the System.Console that we already know from making outputs, the System.Windows.Forms which is useful to design forms like message boxes or dialogs (e.g. to enter paramters), and the System.IO namespace that is used for file handling (compare Fig. 7.1). A complete list of all namespaces can be found here:
`http://msdn2.microsoft.com/de-de/library/ms306608(vs.80).aspx`

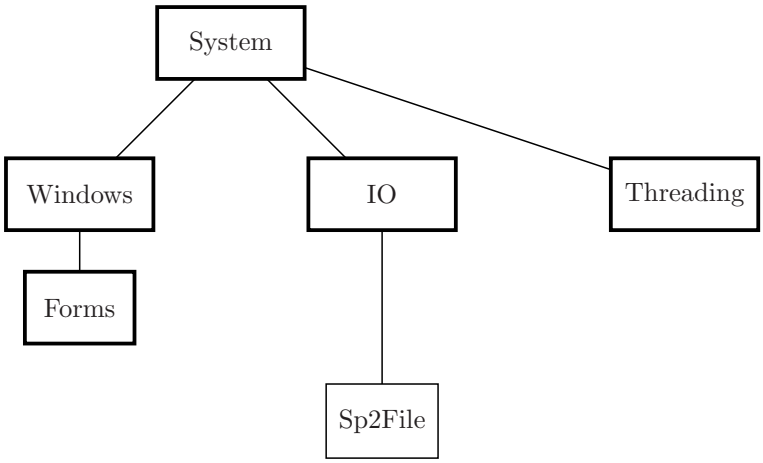


Figure 7.1: The System namespaces and its classes (incomplete - only most common classes listed)

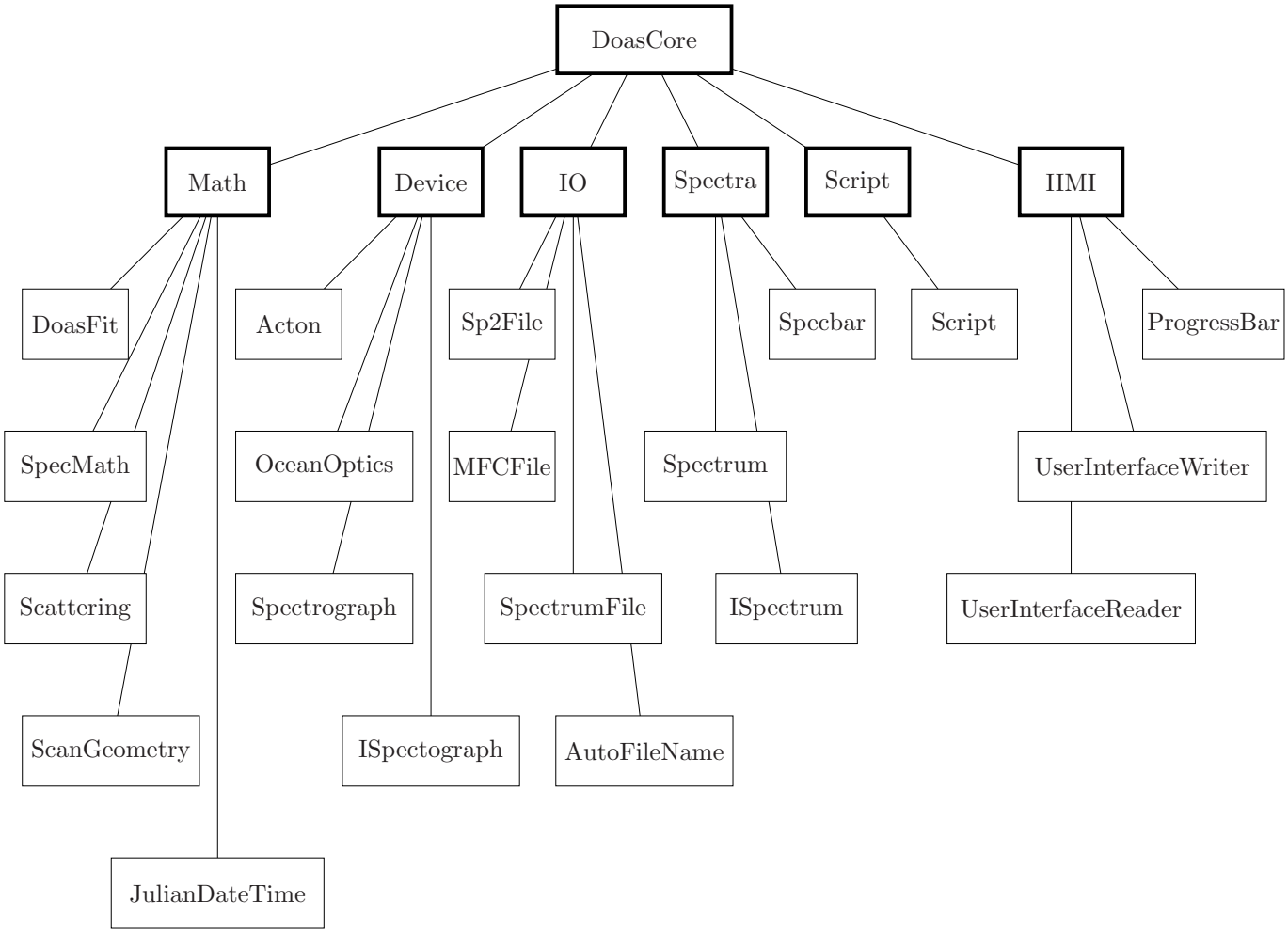


Figure 7.2: The Doasis namespaces and its classes (incomplete - only most common classes listed)

7.2 DoasCore.Math

In this namespace you will find all kinds of mathematical operations provided by Doasis. The classes that you can make use of are:

- DoasCore.Math.SpecMath: basic mathematical methods
- DoasCore.Math.DoasFit: fitting methods
- DoasCore.Math.Scattering: Ring and Raman spectrum calculation
- DoasCore.Math.ScanGeometry: SZA (solar zenith angle) support
- DoasCore.Math.JulianDateTime: Represents a date/time using the Julian date time.

7.2.1 DoasCore.Math.SpecMath

The first step in the evaluation of a recorded spectrum is to correct it: Offset and Dark current have to be subtracted. This can be done at once to a great number of spectra by using a JScript. Here, we are just interested in the method that is used to do the correction, so for simplicity we suppose that the spectrum that shall be corrected and the offset- and dark current-spectrum are loaded into the specbar.

```
import DoasCore.Spectra;
import DoasCore.Math;

var Offset:  ISpectrum = Specbar.GetSpectrum("Offset");
 var Dunkel :  ISpectrum = Specbar.GetSpectrum("Dunkelstrom");
var Meas :  ISpectrum = Specbar.CurrentSpectrum;

SpecMath.CorrectOffset(Meas,Offset);
SpecMath.CorrectDarkCurrent(Meas,Dunkel);
```

In this example, "Offset" and "Dunkelstrom" are the object keys of the spectra by which they can be identified. The methods CorrectOffset and CorrrectDarkCurrent are member of the SpecMath class. To see more examples, read the glossary of the Doasis Tutorial.

7.2.2 DoasCore.Math.ScanGeometry, DoasCore.Math.JulianDateTime

Here is an example that calculates the SZA from the time (given in Julian date time) the longitude, and the latitude:

```
function CalcSZA(Spectr)
 {
    var Julian = new JulianDateTime();
    Julian = new JulianDateTime(Spectr.StartDateAndTime);

    var S = new ScanGeometry();
    S.Latitude = latitude;
    S.Longitude = longitude;
    S.JulianDate = Julian;

    if (S.CalculateSZA()) return(S.SZA);
}
```

7.2.3 DoasCore.Math.DoasFit

With the class DoasCore.Math.DoasFit one is able to construct fit-scenarios and start fit procedures. Here is an example of how to do fit-scenarios (how to evaluate data with fit scenarios in JScript is shown in section 12.3).

```
import DoasCore;
import DoasCore.Spectra;
import DoasCore.Math;
import DoasCore.IO;

var Path = "C:\\Fitszenarios";
var FitFile = "Fitszenario.fs";
var FitLow = 320;
var FitHigh = 420;
var Frauenhofer :  ISpectrum = Specbar.GetSpectrum("Frauenhofer");
var Ring :  ISpectrum = Specbar.GetSpectrum("Ring");
var O3223 :  ISpectrum = Specbar.GetSpectrum("O3(223K)");
var O3246 :  ISpectrum = Specbar.GetSpectrum("O3(246K)");

Frauenhofer.Open(Path + FrauenhoferFile);
Ring.Open(Path + RingFile);
O3223.Open(Path + "Ozon\\Ozon223Gefaltet.sp2");
O3246.Open(Path + "Ozon\\Ozon246Gefaltet.sp2");

var fit :  DoasFit = new DoasFit();
    fit = DoasFit.Open(Path + FitFile);
    fit.FitRanges[0].LimitLow = FitLow;
    fit.FitRanges[0].LimitHigh = FitHigh;
    fit.OffsetPolynomialOrder = 2;
    fit.ReferencesInfo[0].ReferenceSpectrum = Frauenhofer;
    fit.ReferencesInfo[1].ReferenceSpectrum = Ring;
    fit.ReferencesInfo[2].ReferenceSpectrum = O3223;
    fit.ReferencesInfo[3].ReferenceSpectrum = O3246;

DoasFit.Save(fit, Path + FitFile);
```

Warning: *If you want to describe the path of a file in JScipt you have to use two "backslashes" instead of one.*
e.g. "C:\\Fitszenarios\\Ozon\\Ozon223Gefaltet.sp2"

7.3 DoasCore.Device

The DoasCore.Device namespace contains classes for each of the supported devices, and general interfaces. The most common are:

- DoasCore.Device.Spectrograph: this is a common device access class, it supports scanning using the currently selected device.
- DoasCore.Device.ISpectrograph interface that is implemented by all specific spectrographs
- DoasCore.Device.OceanOptics: this spectrograph is implemented in MiniDoas apparatuses

- DoasCore.Device.Acton: another spectrograph often used

Here is an example how to perform one scan with the Ocean Optics Spectrograph:

```
import DoasCore;
import DoasCore.Spectra;
import DoasCore.Device;

var NScan = 1000; var ExpTime = 1; var MiniDoas :  OceanOptics = new OceanOptics();
var Spec1 :  ISpectrum = Specbar.GetSpectrum("Offset1");

MiniDoas.Scan(Spec1,NScan,ExpTime);
```

Note that the Scan method is not only a member of the OceanOptics class in the DoasCore.Device namepace, but also of the Acton class and the interface class ISpectograph, from which all specific spectographs inherit members. The parameters of the Scan method are the name of the spectrum in which the measuremen shall be stored, the number of scans that shall be performed, and the exposure time for each scan. The Properties NumScans and ExposureTime are also members of all devices.

Hint: Using the generic class Spectrograph allows to scan without the need to know the exact hardware used.

7.4 DoasCore.IO

The DoasCore.IO namespace is used for handling the input and output of Doas-specific data, most importantly spectra.

- DoasCore.IO.SpectrumFile: allows to open and save spectra
- DoasCore.IO.AutoFileName: allows to open and save a set of spectra, automatically numbered

The usage of the AutoFileName class is illustrated in section 12.4.

7.5 DoasCore.Spectra

This namespace is used to manage a set of spectra. The important classes are:

- DoasCore.Spectra.Spectrum: Basic object that represents a spectrum within Doasis
- DoasCore.Spectra.ISpectrum: An interface for the spectrum class.
- DoasCore.Spectra.Specbar: A collection of spectra that will be visible in the Specbar of the graphical user interface

7.5.1 DoasCore.Spectra.ISpectrum

The most important object here is the ISpectrum object. For example, one can modify the current spectrum with it. Here is an example of a function that searches the peak of a spectrum:

```
import System;
import DoasCore.Spectra;

var Spec :  ISpectrum = Specbar.CurrentSpectrum;

function SetMarkerToPeak()
{
    Spec.Marker = Spec.MinChannel;
    while(Spec.MarkerValue < Spec.Max) Spec.Marker++;
    System.Console.WriteLine("Peak bei Kanal :  " + Spec.Marker);
}

SetMarkerToPeak();
```

There are quite a lot members in the ISpectrum object: Marker, MinChannel, MaxChannel, MarkerValue, and many more (ObjectKey, FileName, StartTime, StopTime, AzimuthAngle,...) To each property in the property tab of a spectrum there is a member in the ISpectrum class that can be used in a JScript. A complete list can be found in the Programming Documentation in the "help"-menu of DOASIS. Just search for "ISpectrum"!

7.5.2 DoasCore.Spectra.Specbar

The specbar contains the object Specbar.CurrentSpectrum. This allows to access data of the current spectrum (the one that is displayed in the main window). Here is an example that sets the coefficients of the calibration polynomial (which assigns wavelength info to the channels):

```
import Doas.Core;
import DoasCore.Spectra;

var Spec1 :  ISpectrum = Specbar.CurrentSpectrum;
var Spec2 :  ISpectrum = Specbar.GetSpectrum("Halogen1");

Spec1.CalibPolynomialOrder = 2;
Spec1.CalibPolynomial[0]= 297.51431;
Spec1.CalibPolynomial[1]= 0.12086;
Spec1.CalibPolynomial[2]= -9.60429e-6;
Spec1.SaveAs("modified");

 Spec2.ExposureTime= 10000;
Spec2.SaveAs("modified2");

Specbar.CloseAllSpectra();
```

The Spectrum-class also contains methods to open and save (used above) the spectrum directly without the need to use the DoasCore.IO.SpectrumFile class.

7.6 DoasCore.Script

The DoasCore.Script-namespaces helps to handle scripts. Especially important is the property StopScript and StopAllScripts, which signalize when a script was stopped by the user, and the methods SuspendScript() and ResumeScript(). They are illustrated in the script below, which optimizes the exposure time according to the counts given by the range between MinCount and MaxCount.

```
import System;
import DoasCore;
import DoasCore.Spectra;
import DoasCore.Math;
import DoasCore.Device;
import DoasCore.IO;
import DoasCore.Script;

var MiniDoas :  OceanOptics = new OceanOptics();
var OptSpec :  ISpectrum = Specbar.GetSpectrum("Optimize");
    OptSpec.MinChannel = 1400;
    OptSpec.MaxChannel = 2047;
var ETime =10;
var MaxCounts = 4000;
var MinCounts = 3500;

while(!Script.StopAllScripts) {
    MiniDoas.Scan(OptSpec,1,ETime);
    var Faktor :  Double = OptSpec.Max/MaxCounts;
    //******************** too few counts ************************
    while( (OptSpec.Max< MinCounts || OptSpec.Max > MaxCounts)
            && !Script.StopAllScripts) {
        ETime = parseInt(ETime/Faktor);
        if (ETime > 60000) {
            ETime = 60000;
        }
        Console.WriteLine("Exposure Time = " +ETime );
        Console.WriteLine("Maximimum = " + OptSpec.Max);
        MiniDoas.Scan(OptSpec,1,ETime);
        Faktor = OptSpec.Max/MaxCounts;
    }
    Console.WriteLine("Optimization finished.  Exposure Time = " + ETime);
    Console.WriteLine("Maximum :  " + OptSpec.Max);
}
```

Warning: *This script will be repeated until it is stopped by the user by pressing the "Stop" button in the "Script" sheet of the "console" window.*

7.7 DoasCore.HMI

The DoasCore.HMI namespaces (Human Machine Interface) can be used to simplify the interaction between your JScript and the user. The most common used classes are:

- DoasCore.HMI.UserInterfaceWriter: Basic text output to the user
- DoasCore.HMI.UserInterfaceReader: basic text input from the user
- DoasCore.HMI.ProgressBar: Display the progress of your algorithm in the statusbar.

7.7.1 Progress Bar

The progress bar can be used to display the progress of the calculation. Here is an example:

```
import System;
import DoasCore;
import DoasCore.HMI;

var Bar :  DoasCore.HMI.ProgressBar = new DoasCore.HMI.ProgressBar();
    Bar.Value = 0;
    Bar.Minimum = 0;
    Bar.Maximum = 100;
    Bar.Caption = "Bar";
    Bar.Text = "Advance";

var i;
for (i=0;i<100;i++)
{
    doCalculation();
    Bar.Value++;
}
```

The function doCalculation() is not defined here, but can be considered a calculation that has to be done a couple of times.

7.8 System

The System namespace and all its subnamespaces are provided by the .NET Framework. All its classes are also available in programing C++ or C# in the Visual .NET Framework. Since they are not part of Doasis, I will just give a few examples and refer you to the very extensive guide to the .NET framework on the web (see last chapter for addresses).

7.9 System.Console

We already used the WriteLine() method a lot. But there is more functionality of the System.Console namespace.
You can also input data from the console:

```
 import System;

Console.WriteLine("Please enter your Name");
var a=System.Console.ReadLine();
Console.WriteLine("Hello " + a + "!  You have a nice name.");
```

Due to the concept of untyped variables, you can use the same command to read integers or real numbers.

7.10 System.Windows.Forms

The most important and easiest form is the message box:

```
 import System.Windows.Forms;

MessageBox.Show("This is a message box");
```

But there are plenty of other opportunities to design very personal forms including buttons and check boxes and much more.

7.11 System.IO

The System.IO namespace contains classes that provide all kinds of input and output routines, such as serialization (which helps storing data of instants of classes). As an example we will consider the TextWriter class, which can be helpful to save evaluation data of spectra in plain text to files:

```
using System;
using System.IO;

var filename = "MyFile.txt";

StreamWriter sr = File.CreateText(filename);
sr.WriteLine ("I am the text in this file.");
sr.Close();
```

7.12 System.Threading

In the System.Threading namespace, the class thread allows to start more than one threads to execute a portion of your program. A thread is a sequence of instructions. Multiple threads allow parallel computing, comparable to multitasking of programs. For example

you could create two threads, one to measure data, another to do simple math like offset corrections, etc. on all recorded spectra. A sample script about threading is explained in section 12.5

Part III

Collection of JScript applications

Chapter 8

Simple JScripts

The following programs are a collection of fairly simple JScripts, that build the basis for any bigger program.

8.1 Hello World

The first program displays ”Hello World” in the Output window.

```
// The namespace "System" provides the most basic functions and should
// always be imported
 import System;

// Shows "Hello World" in the Output window
Console.WriteLine("Hello World!");
```

8.2 Diplaying variables in the Output window.

If the content of variable should be displayed in the Output window, it can be done in the following way.

```
import System;

var height = 90;
var unit = "cm";
// variables should be separated from the text by "+"
Console.WriteLine("Fresh snow in the alps of "+height+unit+".  Let's go skiing!");
// if the output ends with a variable, the "+" is not necessary
Console.WriteLine("Again:" +height+unit);
Console.WriteLine("It's unbelievable!")
```

The output will then be:

Fresh snow in the alps of 90cm. Let's go skiing!
Again:90cm
It's unbelievable!

The type of the variable (integer, string,...) doesn't necessarily have to be declared in JScript. See section 8.11 Remarks.

8.3 Wait functions

In section 4.5 it was recommended to include "wait" functions after data transfer between the computer and the measurement instrument to prevent system crashes. The example shown here counts from zero to five and after each count it waits for half a second.

```
// The namespace "System.Threading" provides the "Thread"-class which
// includes the "Sleep"-method
 import System.Threading;

var i;

for (i=0; i<=5; i++)
{
    // Wait for 500 ms
    Thread.Sleep(500);
    // Display the variable i in the Output window
    System.Console.WriteLine("Number:  "+i);
}
```

Remarks

Alternatively, a "wait" function can be implemented in the following way:

```
import System;
import DoasCore.HMI;

var window = new DoasCore.HMI.OldUI();
var i;

for (i=0; i<=5; i++)
{
  window.Sleep(500);
  Console.WriteLine("Number: "+i);
}
```

It is slightly more complicated since an instance of the class `DoasCore.HMI.OldUI()` has to be created first. Usually the method mentioned above is given priority.

8.4 Display message boxes

When conducting measurements it can be useful not only to display parameters in the Output window but also to let appear message boxes, if something extraordinary happened. The following example counts from zero to ten and displays the numbers in the Output window. After each count there is a break of 300ms. After number 5 and number 10 a message box appears.

```
import System;
// This namespace provides the "OldUI"-class which includes the message boxes
import DoasCore.HMI;

// Create an instance of the "OldUI"-class
var window = new DoasCore.HMI.OldUI();

var i;

for(i=0; i<=10; i++)
{
    // Display the counter i
    Console.WriteLine("Number:  "+i);
    // Wait for 300ms
    window.Sleep(300);
    // When the counter equals 5, a message box with the title "Message 1"
    // appears displaying the message "Half time".
    // After pressing the "OK" button the script continues.
    if(i==5) window.MessageBox("Half time", "Message 1");
    // Second message box when the counter equals 10
    if(i==10) window.MessageBox("Time over", "Message 2");
}
```

Remarks

- In that case, the "wait"-function of the "OldUI()"-class was chosen (see section 8.3) since this class is used to display the message boxes as well.
- The "OldUI()"-class provides many more message boxes. Information about message boxes can be found in the Programming Documentation of DOASIS (Help menu - Programming Documentation) by searching for "OldUI". See as well the script in section 12.2.

8.5 Creating a directory

During the evaluation of spectra, it is often of advantage to store the results of the evaluation in a file located in a certain directory, named typically for the type of the evaluation. This directory has to be created first. The command is very simple.

```
import System;
 // The namespace "System.IO" must be imported because it includes the class
// "Directory" which is needed to create a directory.
import System.IO;

// Create the directory "C:\SpecialEvaluation".
Directory.CreateDirectory("C:\\SpecialEvaluation");
```

Remarks

The class “Directory” of the“System.IO” namespace includes many more methods, e.g.
-“Directory.Delete()” to delete the path given in the brackets or
-“Directory.Exists()” which returns “true” if the path specified in brackets exists and “false” in case it doesn’t.

8.6 Creating a text file

During evaluation of spectra, it can be useful the create a text file into which important parameters are written. The following example creates a text file `"text.dat"` on drive `"C:"` and puts the following content:

First Column	*Second Column*
20	*30*

```
var textfile
// create an object which enables to create a textfile
var dispFileSystem = new ActiveXObject("Scripting.FileSystemObject");

// provide access to the text file "C:\\text.dat"
textfile = dispFileSystem.OpenTextFile("C:\\text.dat",2,true);
// first line of the text file
textfile.Write("First Column\t Second Column\n");
// second line of the text file
textfile.Write("20\t30 \n");
// close the text file
textfile.Close();
```

Remarks

- In this case no namespace has to be imported.
- For a quick description on ActiveXObjects, see section 12.1.1.

- The method `dispFileSystem.OpenTextFile` takes three arguments.
  The first one is the path and the name of the text file, here `"C:\text.dat"`
  The second one is a number, here “2”.
  “2” means, that the file is for writing.
  “1”: file is for reading only.
  “8”: data is appended to file.
  The third argument, here “true” means, that a new file is created, if the specified file name does not exist. If “false” is used, the file is not created.

- In case you want to open the created file with the program “Origin”, the file should be of the type “.dat”. With “\t”, a new column is started, with “\n”, a new line begins. In “Origin”, it is very important that each line has the same number of columns.

8.7 Get the current date and time

When evaluating spectra, it can be useful to store the results in file which file name includes the current time or date of the computer. This guarantees that if the evaluation is executed again, the results of the old evaluation will not be overwritten but a new file with the new date and time is created. In the following example at first the entire current date and time is displayed in the Output window. Afterwards, month, day, year (date) and hours, minutes, seconds (time) are accessed separately and displayed as well.

```
// the System namespace provides access to date and time
import System;

var month, day, year;
var hours, minutes, seconds;
// store date and time in the variable CurrentDateTime
var CurrentDateTime = new Date();

// display CurrentDateTime
Console.WriteLine("Date and Time:  "+CurrentDateTime);

// retrieve the different parts of the date
// January corresponds to number "0", therefore add "1"
month = CurrentDateTime.getMonth()+1;
day = CurrentDateTime.getDay();
year = CurrentDateTime.getYear();
// display the date
Console.WriteLine("Date:  "+month+"."+day+"."+year);

// retrieve the different parts of the time
hours = CurrentDateTime.getHours();
// make sure "hours" constitutes of two numbers:  e.g.  9 is transformed to 09
if (hours < 10) hours = "0"+hours;
minutes = CurrentDateTime.getMinutes();
if (minutes < 10) minutes = "0"+minutes;
seconds = CurrentDateTime.getSeconds();
if (seconds < 10) minutes = "0"+seconds;
// display the time
Console.WriteLine("Time:  "+hours+":"+minutes+":"+seconds);
```

The output looks like this:

Date and Time: Sat Jun 21 15:33:50 UTC+1 2008
Date: 6.6.2008
Time: 15:33:50

Remarks

The program "Origin" only recognizes time data in the format hh:mm:ss. So make use of this example if you want to store the time in a result file, which you later want to edit with "Origin".

8.8 Control number of digits

When fitting with DOASIS the fit coefficient and other fit results are calculated with very high precision. However many decimal places of numbers are confusing if they are just meant for control purposes e.g. in the Output window. In this example, the decimal number 123456789.123456789 is displayed in different modes and everybody can decide by himself, which on suits best.

```
import System;

var x = 123456789.123456789;

Console.WriteLine("Original number:  "+x);
// includes a blank line
Console.WriteLine("");
Console.WriteLine("Exponential with 5 digits:  "+x.toExponential(5));
 Console.WriteLine("Exponential with 5 digits:  "+x.ToString("e5"));
Console.WriteLine("Number with 5 digits:  "+x.ToString("f5"));
Console.WriteLine("Automatic mode (see Remarks):  "+x.ToString("g5"));
```

The output looks like this:

Original number: 123456789.12345679

Exponential with 5 digits: 1.23457e+8
Exponential with 5 digits: 1.23457e+008
Number with 5 digits: 123456789.12346
Automatic mode (see Remarks): 1.2346e+08

Remarks

The recommended option is:
`x.ToString("g5")`
It generally choses the best way of representing the number. The "5" does represent the number of digits, but the "numbers" altogether.
For example: 4.3 has two "numbers", 13.45 has four, 14.567 has seven, 1.23e+05 has 3 etc.

8.9 Open and Save a spectrum

Make sure you have the spectrum called *oldspec.sp2* in the folder *c:\scripttest*, before running the script.

```
import System;
// The class "Spectra" of the namespace "DoasCore" is necessary to handle spectra
import DoasCore.Spectra;

// open an empty spectrum sheet with the name "Spectrum" and store it
// in the variable "Spec"
var Spec:  ISpectrum = Specbar.GetSpectrum("Spectrum");
// load the spectrum "oldspec.sp2" in the empty sheet
Spec.Open("c:\\scripttest\\oldspec.sp2");
// save it as a new name
Spec.SaveAs("C:\\scripttest\\newspec.sp2");
```

Remarks

- If you handle spectra with JScript, they will always be of the type "ISpectrum". It will not be correct in the sense of informatics, but imagine it this way:
  A whole number is of the type "integer", a spectrum is of the type "ISpectrum".

- The Programming Documentation in the Help menu of DOASIS shows all the members, that are connected with the type "ISpectrum". Many following examples will make this clearer!

- In most cases we will deal with automatized opening and saving of large numbers of spectra. Therefore we will make use of the class "AutoFileName()".

8.10 Saving spectra with the "AutoFileName"-class

When measurements are automatized, it is useful to store the measured spectra in a certain structure. During automatized evaluation, we can then use this structure again. In the following example, the current spectrum will be stored in such a structure. For the following script, the spectrum of the active spectrum sheet will be stored in a folder with the name *S00000000* which will be created under the path *c:\scripttest*. The first file stored will have the name *S0000000.sp2* the next one will be *S0000001.sp2* and so on.

```
import System;
import DoasCore.Spectra;
// IO stands for input/output operations
// AutoFileName is a class of the DoasCore.IO namespace
import DoasCore.IO;

// create the AutoFileName object and store it in the varible afn
var afn :  AutoFileName = new AutoFileName();
    // specify the path where the spectra should be stored in, the path should
    // exist already, it is not created by the script
    afn.BasePath = "c:\\scripttest";
    // Prefix defines the symbols before the digits
    afn.Prefix = "S";
    // Suffix defines the symbols after the digits, usually related to the file type
    afn.Suffix = ".sp2";
    // Number of Digits to be used
    afn.NumberOfDigits = 7;
    // Number of files to be stored in one folder
    afn.FilesPerFolder = 100;
// store current spectrum in the variable Spec
var Spec:  ISpectrum = Specbar.CurrentSpectrum;
// see comment below
afn.FindLastIndex();
// save spectrum with the new index number
afn.Save(Spec);
```

8.10.1 A word on AutoFileName()

- In this class many things are done intrinsically. Here are some examples:
  - The method “FindLastIndex” looks for the last already existing index number and then sets the property “CurrentFileNumber” one higher.
  - The property “CurrentFileNumber” of the “AutoFileName”-class was not used directly in the example above, but the meaning is quite simple. See example 8.11
  - The method “Save” saves a Spectrum and then increases the “CurrentFileNumber” counter by one.
- More Information concerning the “AutoFileName”-class can be found in the Programming Documentation in the Help menu of DOASIS

8.11 Opening spectra with the “AutoFileName”-class

In the following example the property “CurrentFileNumber” of the “AutoFileName”-class will be used to open a specific spectrum. When data has been collected for a very long period of time, several thousand spectra can easily accumulate. Opening specific spectra out of this huge amount can be quite annoying and tiring. Therefore a script is presented here,

that opens a single spectrum. The spectrum number is asked for in the Output window. The demanded spectrum will be opened and some details of the spectrum like "CurrentFileNumber", "ExposureTime" and "AzimuthAngle" are displayed in the Output window. The spectra have to be stored in the folder *C:\scripttest\S0000000*. The spectra themselves must have names like *S0000000.sp2*, *S0000001.sp2*, *S0000002*,.... This is the same structure as how they are stored, when automatic measurements are taken.

```
import System;
import DoasCore.Spectra;
import DoasCore.IO;

var SpectrumNumber;
var Spec:  ISpectrum;
// see section 8.10.1 for "AutoFileName"-class
var afn :  AutoFileName = new AutoFileName();
    afn.BasePath = "C:\\scripttest\\";
    afn.Prefix = "S";
    afn.Suffix = ".sp2";
    afn.NumberOfDigits = 7;
    afn.FilesPerFolder = 100;
// ask for the spectrum number in the Output window
Console.WriteLine("Open spectrum with the number:");
// store the number that is typed in the variable "SpectrumNumber"
SpectrumNumber = Console.ReadLine();
// add a new empty spectrum sheet to the "Specbar"
// the name of the spectrum is the spectrum number, that was specified before
Spec = Specbar.GetSpectrum(SpectrumNumber);
// the property "CurrentFileNumber" of the "AutoFileName" class
// is set to "SpectrumNumber"
afn.CurrentFileNumber = SpectrumNumber;
 // open the spectrum with the demanded spectrum number into the empty
// Spectrum "Spec"
afn.Open(Spec);
// display some properties of the opened spectrum in the Output window
Console.WriteLine(
    " Spectrum:  "+SpectrumNumber+
    " ExposureTime:  "+Spec.ExposureTime+
    " AzimuthAngle:  "+Spec.AzimuthAngle);
```

Remarks

- Referring to the line
  `var Spec:  ISpectrum;`:
  It would be enough as well to write
  `var Spec;`
  only. In JScript, the type of the variable doesn't have to be declared most of the times. But sometimes, it is useful to declare the type of the variable explicitly. See 8.14, where the variable "time" has to be of the type "System.DateTime".

- If the script doesn't open any spectrum when calling the line
  `afn.Open();`
  the problem often is a mistake in the declaration of the "AutoFileName" properties (e.g. "BasePath"," Prefix",...) and it is worth checking those very carefully.
  In order to proof, if the problem is actually caused by the "afn.Open()" method, display the return value of this method in the Output window. The following command can be placed in the script:
  `Console.WriteLine(afn.Open(Spec));`
  "True" is displayed if the file could be opened and "false" if not.

8.12 Modifying the properties of a spectrum

The following script displays the "ObjectKey" (see Fig. 8.1) of the active spectrum sheet in the Output window, changes it to "Nice spectrum" and displays the new ObjectKey in the Output window. Other properties can be changed similarly.

```
import System;
import DoasCore.Spectra;

// The spectrum in the active spectrum sheet is stored in the variable "MeasSpec"
var MeasSpec = Specbar.CurrentSpectrum;

// Display the old object key
Console.WriteLine("Old object key:  "+MeasSpec.ObjectKey);
// Change the object key
MeasSpec.ObjectKey = "Nice spectrum";
// Display the new object key
Console.WriteLine("New object key:  "+MeasSpec.ObjectKey);
```

Remarks

Other spectrum properties like StartDate, StartTime, ExposureTime, NumScans etc. can be accessed in the same way. For a list of all spectrum properties search for "ISpectrum" in the Programming Documentation of DOASIS (Help - Programming Documentation).

8.13 How to compare the time of different spectra

It is possible to compare the date and time of spectra. This can be helpful e.g. when selecting a reference spectrum as close to the measurement spectrum as possible. To compare the spectrum start times the function "TimeSpan.Compare" is used. To compare the "StartTimeAndDate" the function "DateTime.Compare" is used. Both are provided by the "System" namespace. In the example script, the "AutoFileName" structure of section 8.11 is used.

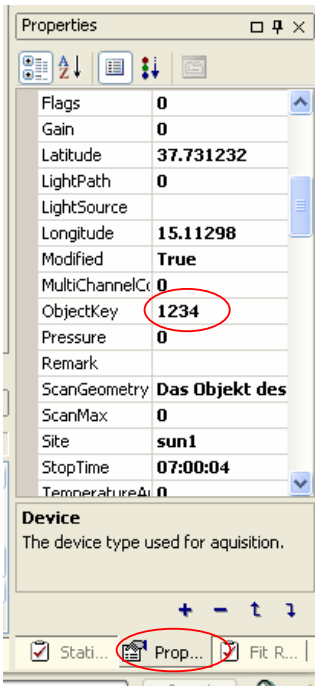


Figure 8.1: The property window of the spectrum.

```
import System;
import DoasCore.Spectra;
import DoasCore.IO;

// add two new empty spectrum sheets to the "Specbar"
var Spectrum1 = Specbar.GetSpectrum("Spectrum1");
var Spectrum2 = Specbar.GetSpectrum("Spectrum2");

// use the same "AutoFileName" structure as in example 8.11
var afn :  AutoFileName = new AutoFileName();
    afn.BasePath = "C:\\scripttest\\";
    afn.Prefix = "S";
    afn.Suffix = ".sp2";
    afn.NumberOfDigits = 7;
    afn.FilesPerFolder = 100;

// the method "FindFirstIndex()" returns the number of the first existing file
afn.CurrentFileNumber = afn.FindFirstIndex();
// open the first spectrum and increase the "CurrentFileNumber" by one
afn.Open(Spectrum1);
// open the next spectrum
afn.Open(Spectrum2);
// display the date and time, when the spectra were recorded
Console.WriteLine("Spectrum1:  "+Spectrum1.StartDateAndTime);
Console.WriteLine("Spectrum2:  "+Spectrum2.StartDateAndTime);
// check which spectrum has been recorded first
if (DateTime.Compare(Spectrum1.StartDateAndTime, Spectrum2.StartDateAndTime) > 0)
    Console.WriteLine("Spectrum1 was recorded later than Spectrum2.")
else if (DateTime.Compare(Spectrum1.StartDateAndTime, Spectrum2.StartDateAndTime) == 0)
    Console.WriteLine("Spectrum1 was recorded at the same time as Spectrum2.")
else
    Console.WriteLine("Spectrum1 was recorded before Spectrum2.")
```

The output can look like this:

Spectrum1: Sun Jul 1 11:31:10 UTC+1 2007
Spectrum2: Sun Jul 1 11:31:12 UTC+1 2007
Spectrum1 was recorded before Spectrum2.

8.14 Calculate the solar zenith angle (SZA)

In order to calculate the solar zenith angle, we need latitude, longitude and time. In this example, latitude and longitude are defined in the code and time is read out of the spectrum, that is active when the script is executed.

```
import System;
import DoasCore.Spectra;
import DoasCore.Math;

// store the active spectrum in the variable Spec
var Spec:  ISpectrum = Specbar.CurrentSpectrum;
// StartDateAndTime provides the time the spectrum was recorded
var time = Spec.StartDateAndTime;
var SZA;

// calculate julian date and store it in the var Julian
var Julian = new JulianDateTime(System.DateTime(time));

// create ScanGeometry class for SZA calculation
var S = new ScanGeometry();
 // feed it with lat, long, and time
S.Latitude = 49.4167;
S.Longitude = 8.7;
S.JulianDate = Julian;
// calculate SZA
S.CalculateSZA();
// store it in the var SZA
SZA = S.SZA;
// displays the SZA on the screen
System.Console.WriteLine("SZA="+SZA);
```

Remarks

- When “JulianDateTime” is called, it has to be made clear, that the argument time is of the type “System.DateTime”, because another type would call a different routine. In this case, “StartDateAndTime” of the spectrum is stored in the variable “time”, then “time” is specified to be of the type “System.DateTime” and then delivered to the class “JulianDateTime”, that afterwards returns the julian date.

8.14.1 A word on the class ScanGeometry

- This class offers calculations of solar and lunar zenith angle (sza, lza) and solar and lunar azimuth angle (saz, laz).
- It is provided by the DoasCore.Math namespace
- You can find the whole functionality of the class, if you search for ScanGeometry in the Help menu of the Programming Documentation of DOASIS

8.15 Wavelength Calibration

Just open a spectrum and run the script. You will see that wavelength information will be added or modified in your spectrum. Therefore a polynom of order 2 is used and the coefficients are defined in the script. Mathematically it looks like this:

$$\lambda = c_0 + c_1 \cdot x + c_2 \cdot x^2$$

λ is the wavelength, c_i are the coefficients and x is the channel.

```
import DoasCore.Spectra;

var Spec :  ISpectrum = Specbar.CurrentSpectrum;
 Spec.CalibPolynomialOrder = 2;
Spec.CalibPolynomial[0]= 297.51431;
Spec.CalibPolynomial[1]= 0.12086;
Spec.CalibPolynomial[2]= -9.60429e-6;
```

Chapter 9

JScript Project File

Using JScript Project Files (jsp-files) to structure JScripts is very recommendable as it saves a lot of work. Variables and functions are declared once in a separate JScript file and can be used later on by simply calling these files in the JScript Project before the "main" JScript, in which the actual task is defined.
A typical project file would look like this:

```
// for path descriptions use always two backslashs!!
C:\\scripttest\\generally\\variables.js
C:\\scripttest\\generally\\functions.js
C:\\scripttest\\evaluate\\mainevaluate.js
```

This project script can be saved as *C:\scripttest\evaluate\evaluate.jsp*. Be aware, that the suffix of a project file is ***.jsp*** and not ***.js***.
The idea:
In *C:\scripttest\generally\variables.js*, all variables are declared.
In *C:\scripttest\generally\functions.js*, all methods, that are used frequently are defined here.
In *C:\scripttest\evaluate\mainevaluate.js*, the actual main routine is defined.
All three JScript files are executed after each other, as if the content was just written in one script. The advantage is that for the next program, a similar project file can be written. Here an example:

```
C:\\scripttest\\generally\\variables.js
C:\\scripttest\\generally\\functions.js
C:\\scripttest\\correct\\maincorrect.js
```

The project script could be saved as: *C:\scripttest\correct\correct.jsp*.
Only the "main-function" has changed and the definitions of the variables and functions can still be used. Therefore, the main routine will be shorter, it will be less work to write and it provides better overview. Within DOASIS, instead of opening a script file with the suffix *.js*, the project file (suffix *.jsp*) can be opened and executed.

A specific example can be looked up in section 11.2

Remarks

- Old JScripts
  If you use old JScripts for DOASIS that were programmed for an older version of DOASIS, it might happen that the script won't run if they use the dispTools-, dispIO or other disp-Objects. These objects are not supported and replaced by equivalent functions in newer DOASIS-Versions.
  However, if you want to use these old scripts nevertheless, you can create a project file referencing the "system.js" and the old script. Then, also old scripts will run without problems. But there is no guarantee.

Chapter 10

JScripts communicating with hardware

10.1 Reading in data from a serial port (e.g. GPS data)

The following script was developed in order read in the signal of a GPS device. In this example, only the data from the GPS is displayed in the Output window. In order to extract the position coordinates and time, specified parts of the data have to be used. This will not be shown here.

```
import System;
// namespace to access the serial port
import System.IO.Ports;
// namespace to abort the script by pressing the "Stop" button
import DoasCore.Script;

var signal;
// specify the properties of the port and the data transfer
var serial = new SerialPort();
    serial.PortName = "COM5";
    serial.BaudRate = 57600;
    serial.DataBits = 8;
    serial.Handshake = Handshake.None;
    serial.Parity = Parity.None;
    serial.StopBits = StopBits.One;

// opens a new serial port connection
serial.Open();

// as long as the "Stop" button is not pressed, the loop is executed
while(!Script.StopAllScripts)
{
    // read in one line of the serial port output
    signal = serial.ReadLine();
    // display it
    Console.WriteLine(signal);
}

// closes the port connection
serial.Close();
```

Remarks

- The output of the GPS used here consisted of a sequences of 6, which are continuously produced:

  *GPRMC,135257.000,V,4925.0709,N,00840.4109,E,0.00,0.00,120608,,,N*78*
  *GPGGA,135257.000,4925.0709,N,00840.4109,E,0,00,99.9,249.6,M,48.0,M,,0000*62*
  *GPGSA,A,1,,,,,,,,,,,,,99.9,99.9,99.9*09*
  *GPGSV,2,1,08,10,39,197,,27,15,070,,07,06,069,,18,09,324,*7B*
  *GPGSV,2,2,08,15,53,298,,09,03,253,,19,07,027,,08,42,063,*7F*

  The first line starting with *GPRMC* provides information about time (13h 52min 57s UTC), latitude 49° 25.0709′ and longitude 8° 40.4109′. For more information search the internet for the NMEA (National Marine Electronics Association) standard protocol.

- To extract specific characters out the lines, the command `signal.Substring(x,y)` can be used. It provides part of the string "signal" which starts at a specified character position x and has a specified length y.

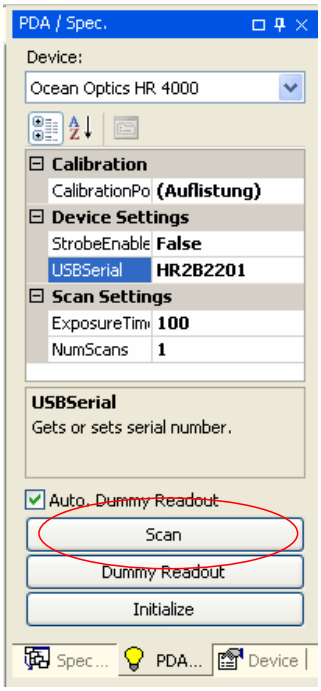


Figure 10.1: Scan button (marked red) to record a spectrum manually.

10.2 Measure a spectrum with an Ocean Optics spectrograph

Before running the script, make sure that you get connection to the spectrograph manually via DOASIS by pressing the Scan button (see Fig. 10.1).
This script was tested with an Ocean Optics HR2000 spectrograph but should work with any other spectrograph of Ocean Optics as well.

```
import System;
import DoasCore.Spectra;
// the namespace "Device" enables the usage of external devices
import DoasCore.Device;

// open an empty spectrum in the specbar
var MeasSpec = Specbar.GetSpectrum("MeasSpec");
var ScanNumber = 10;
 var Inttime = 100;
// create a new ocean optics spectrograph object
var Spectrograph = new OceanOptics();
// assign the serial number of spectrograph
Spectrograph.USBSerial = "HR2B2201";
// sum up 10 spectra, each with an integration time of 100ms
// and display it in the spectrum "MeasSpec"
Spectrograph.Scan(MeasSpec,ScanNumber,Inttime);
```

Remarks

- A new spectrograph object could be created as well like this:
  `var Spectrograph :  OceanOptics = new OceanOptics();`
  The only difference to
  `var Spectrograph = new OceanOptics();`
  is that the "type" of the object is declared after the colon. However, creating the object this way sometimes caused problems when trying to access the properties of the measured spectrum. E.g. the measurement temperature of an Ocean Optics QE65000 spectrograph could not be retrieved.

- Many more objects to control hardware can be created (HMTUSBMultiStepper, Acton,.....). More detailed information can be found in the Programming Documentation in the Help menu of DOASIS. Search for "DoasCore.Device" and a list of all types of available devices will appear.

10.3 Operate two Ocean Optics spectrographs

It has to be said the solution presented here is only a (not very satisfactory) "work around". So far, two spectrograhs could not be controlled with a single JScript.
First, please follow the steps described in the chapter "How to operate two Ocean Optics spectrographs with DOASIS at the same time" of the DOASIS Tutorial. Now, both spectrographs can be run with two instances of DOASIS.
The remains are quite easy:

1. Each of the two instances of DOASIS is connected to one of the spectrographs. So, in each of these instances, choose the proper JScript for controlling the connected spectrograph.

2. Make sure that these JScripts address the spectrographs by using the "OceanOptics" class (see example of section 10.2). In the JScript, create an object of this class and assign to it the serial number of the connected spectrograph.

3. Now, you should be able to operate both spectrographs with the two JScripts at the same time.

10.4 Mini-MAX-DOAS (MiniDOAS) control via JScript

Since the author never worked with MiniDOAS devices, the provided script could not be tested in detail. The JScript controlling a MiniDOAS device is thus presented in section 12.6.

Chapter 11

Evaluating spectra

11.1 Offset and Dark Current Correction

Open the spectrum, that you want to correct and make sure, that you have an offset and a dark current spectrum in the folder that is specified in the code.

```
import Systme;
import DoasCore.Spectra;
import DoasCore.Math;

// store active spectrum in the variable Meas
var Meas :  ISpectrum = Specbar.CurrentSpectrum;
// Open two empty spectra sheets called Offset and DarkCurrent
var Offset:  ISpectrum = Specbar.GetSpectrum("Offset");
var DC: ISpectrum = Specbar.GetSpectrum("Darkcurrent");
// Load the two spectra in the empty spectra sheets
Offset.Open("C:\\scripttest\\offset.sp2");
DC.Open("C:\\scripttest\\darkcurrent.sp2");

// Do the Offset and DarkCurrent correction
SpecMath.CorrectOffset(Meas,Offset);
 SpecMath.CorrectDarkCurrent(Meas,DC);
```

Remarks

- The offset correction will be done by subtracting the offset spectrum scan weighted. The dark current correction will be done by subtracting the dark current spectrum scan time weighted.
  What is actually done in mathematical sense when calling the correction routines is explained in detail in the *DOASIS Tutorial*, section 8.1.2: *Spectrum Operations*.
- Make sure that the dark current spectrum you use has already been offset corrected!

11.2 Offset and Dark Current Correction with a "JScript project file"

In 9 it was strongly recommended to structure JScript programs by using so called JScript project files. A concrete example will be given here. The functionality will be exactly the same as 11.1.

11.2.1 Preparation

Create the three folders *C:\scripttest\generally*, *C:\scripttest\correct* and *C:\scripttest\darkspectra*. In *C:\scripttest\darkspectra*, save a dark current spectrum as *darkcur.sp2* and an offset spectrum as *offset.sp2*.

11.2.2 The "project file"

The project file looks like this:

```
C:\\scripttest\\generally\\variables.js
C:\\scripttest\\generally\\functions.js
C:\\scripttest\\correct\\maincorrect.js
```

Save the project file as *C:\scripttest\correct\correct.jsp*.
The three JScript files *variables.js*, *functions.js* and *maincorrect.js* have to be defined now.

11.2.3 The file *variables.js*

The file *variables.js* includes all "namespaces", that have to be imported, all variables, which are used and it is recommended to specify the different locations, where spectra can be found, too.

```
import System;
import DoasCore.Spectra;
import DoasCore.Math;

var darkpath = "C:\\scripttest\\darkspectra\\";
var DarkCurrentName = "darkcur.sp2";
var OffsetName = "offset.sp2";

var Meas :  ISpectrum;
var Offset:  ISpectrum;
var DC: ISpectrum;
var Spectr:  ISpectrum;
```

As mentioned already in 8.11, the declaration of the type "ISpectrum" is not absolutely necessary.
Save *variables.js* as *C:\scripttest\generally\variables.js*.

11.2.4 The file *functions.js*

All common functions are defined in the file *functions.js*. Offset and Dark Current Correction are one of the most frequent functions and will therefore be specified here.

```
function CorrectOffset(Spectr)
{
    SpecMath.CorrectOffset(Spectr,Offset);
}

function CorrectDC(Spectr)
{
    SpecMath.CorrectDarkCurrent(Spectr,DC);
}
```

Object orientated programming:
The content of the file *functions.js* is a typical example for object orientated programing. Functions are defined, than can be called later in the main routine. The theory of object orientation is quite complex, but the basic meaning should already become clear by studying the file *maincorrect.js*.
Save *functions.js* as *C:\scripttest\generally\functions.js*.

11.2.5 The file *maincorrect.js*

```
Meas = Specbar.CurrentSpectrum;
Offset = Specbar.GetSpectrum("Offset");
DC = Specbar.GetSpectrum("Darkcurrent");

Offset.Open(darkpath+OffsetName);
DC.Open(darkpath+DarkCurrentName);

CorrectOffset(Meas);
CorrectDC(Meas);
```

In the lines
`CorrectOffset(Meas);`
`CorrectDC(Meas);`
the functions from the file *functions.js* are called. The argument, which is passed, is a spectrum, in this case, the spectrum “Meas”.

Save *maincorrect.js* as *C:\scripttest\correct\maincorrect.js*.
Now, in the “Script”-window of DOASIS, *correct.jsp* can be called from the folder *C:\scripttest\correct* and the script will be executed.

11.3 Fitting with JScript

This example does the same, as if you opened a spectrum and a fit scenario in DOASIS and then executed the fit. All properties of the fit, like fit range and reference spectra are stored in the fit scenario. So all you need to run the script is to open a spectrum manually in DOASIS and make it the active spectrum (see Fig. 1.1). This will be the spectrum against which will be fitted. Make sure that it is offset and dark current corrected (see section 11.1) and that the logarithm has been taken. Apart from that, a fit scenario file has to be stored as `c:\scripttest\fitscenario.fs`.
The program runs the fit, shows the fit result and displays the fit coefficient of the first spectrum in the fit scenario in the Output window. Another example for fitting is presented in section 7.2.3.

```
import System;
import DoasCore.Spectra;
// necessary for the DoasFit class
import DoasCore.Math;

// store active spectrum in variable MeasSpec
var MeasSpec :  ISpectrum = Specbar.CurrentSpectrum;
// open an empty spectrum sheet with the title "FitResult"
// in which the fit result will be shown later
var ResultSpec :  ISpectrum = Specbar.GetSpectrum("FitResult");

var fitfile = "c:\\scripttest\\fitscenario.fs";
// open fit scenario
var fit:DoasFit = new DoasFit;
fit=DoasFit.Open(fitfile);

// execute the fit, DoFit returns "true" if the fit was succesfull
if(fit.DoFit(MeasSpec))
{
    // display message for successful fit
    Console.WriteLine("Fit successful!");
    // display the fit result windows in the spectrum "ResultSpec"
    fit.PrepareFitResultSpectrum(ResultSpec);
    // display the fit results an the active spectrum sheet
    Specbar.CurrentSpectrum = ResultSpec;
    // display the fit coefficient of the first reference spectrum
    Console.WriteLine("Fit Coefficient:  "+fit.ReferencesInfo[0].FitCoefficient)
    // write fit results into file
    fit.AppendResultToFile("c:\\scripttest\\result.dat");
 }
else
{
    System.Console.WriteLine("Error while fitting!");
}
```

Remarks:

- Each time the fit is executed the fit results will be appended to the file *result.dat*. If *result.dat* doesn't exist yet it will be created.
- It is very important to create the fit object exactly in the way:
  ```
  var fit:DoasFit = new DoasFit;
   fit=DoasFit.Open(fitfile);
  ```
  Otherwise, the fit coefficients can not be accessed. The reason is unclear.
- Each fit result (fit coefficient, shift, squeeze, chi square...) can be individually accessed. The approach is similar to displaying the fit coefficient in the example above. For more information about fit results of the different reference spectra search for "DoasFitReferenceInfo" in the Programming Documentation of DOASIS (Help menu - Programming Documentation). For properties of the fit itself (like fit range, polynomial...) search for "DoasFit".
- If you want to display e.g. the shift value of the second spectrum of the fit scenario the command would be:
  ```
  var shift;
   shift = fit.ReferencesInfo[1].Shift;
  Console.WriteLine(shift);
  ```
  The entry "0" of the fit.ReferencesInfo-Array refers to the first spectrum from the top, the entry "1" to the second...
  Examples for other properties:

  - Squeeze:
    fit.ReferencesInfo[1].Squeeze;
  - Fit coefficient error:
    fit.ReferencesInfo[1].FitCoefficientError;
  - Chi Square:
    fit.ChiSquare;
- If you don't want to fit against the active spectrum but instead load a spectrum from a file and fit against that spectrum, make use of the commands described in section 8.9.

11.4 Modifying a fit scenario with JScript

At first it has to be mentioned that so far it is not possible to create a fit scenario solely with JScript. An existing fit scenario has to be loaded and can then be modified. Another important aspect is, that the number of spectra in the fit scenario cannot be changed via JScript. Hence, if a the loaded fit scenario consists of three spectra, it is neither possible to delete one spectrum nor to add one spectrum. In the Fitting window of DOASIS it is easily possible to activate and deactivate spectra manually from the fit scenario by ticking the box in front of the according reference spectrum (see Fig. 11.1). This should be possible in JScript as well, but in reality unfortunately it isn't (so far).

The example presented here loads an existing fit scenario from `C:\scripttest\fitscenario.fs` and overwrites the first spectrum of the fit scenario with the spectrum from `C:\scripttest\testspec.sp2`. The file name of the this first spectrum of the fit scenario is displayed in the Output window before and after the change to verify that the new spectrum was actually implemented. Afterwards the limit parameters for the shift (in channels) of the first spectrum of the fit scenario are set from 2 (Low) to 4 (High)

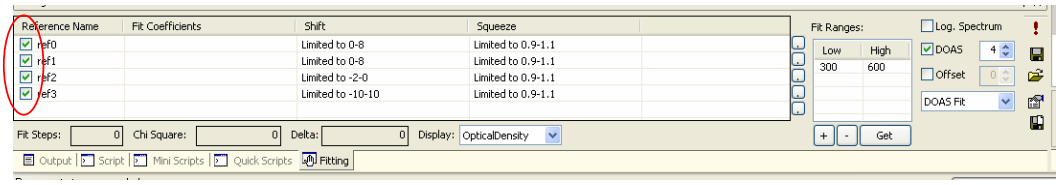


Figure 11.1: Fitting window of DOASIS. By ticking the boxes (marked by the red circle) reference spectra can be activated and deactivated for the fit. This is not possible with JScript.

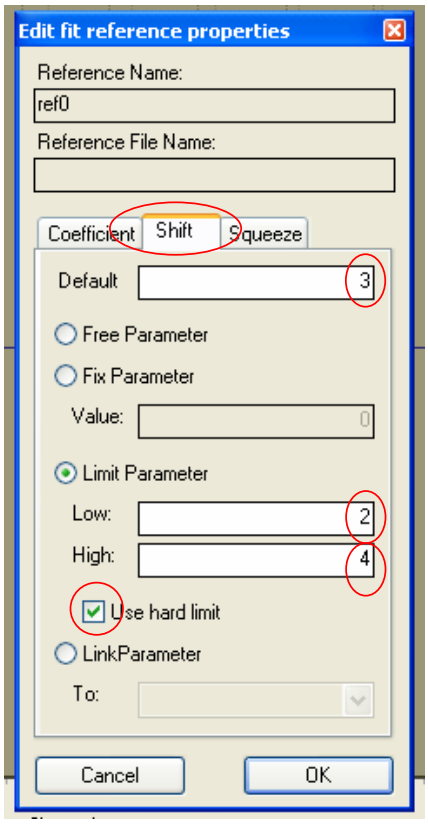


Figure 11.2: Fit reference properties window. It can be activated by double clicking on the reference spectrum in the Fitting window. The marked parameters are set within the JScript. The default value of the shift parameter has to be in between the limit parameters. It constitutes the starting value of the numerical calculations for the shift within the given boundaries.

with 3 being the default value. See Fig. 11.2. The fit range is set from 300 to 600 channels. Finally the fit scenario is saved as `C:\scripttest\newfitscenario.fs`. Before running the script make sure that a fit scenario with at least one reference spectrum exists under `C:\scripttest\fitscenario.fs` and a spectrum is available from `C:\scripttest\testspec.sp2`.

```
import System;
import DoasCore.Spectra;
import DoasCore.Math;

// create an empty spectrum called "Spectrum"
var MeasSpec :  ISpectrum = Specbar.GetSpectrum("Spectrum");
var fitfile = "C:\\scripttest\\fitscenario.fs";
var fit:  DoasFit = new DoasFit();

// open the spectrum
MeasSpec.Open("C:\\scripttest\\testspec.sp2");
// open the fit scenario
fit = DoasFit.Open(fitfile);
// display the file name of the first spectrum of the fit scenario
Console.WriteLine("Old file name:  "+fit.ReferencesInfo[0].ReferenceSpectrum.FileName);
 // replace the first spectrum of the fit scenario by the spectrum "MeasSpec"
fit.ReferencesInfo[0].ReferenceSpectrum = MeasSpec;
// display the new file name of the replaced spectrum
Console.WriteLine("New file name:  "+fit.ReferencesInfo[0].ReferenceSpectrum.FileName);
// set the low limit for the shift
fit.ReferencesInfo[0].SetCoefficientLowLimit(DoasFitReferenceInfo.Coefficients.Shift,2);
 // set the default value for the shift
fit.ReferencesInfo[0].SetCoefficientDefault(DoasFitReferenceInfo.Coefficients.Shift,3);
// set the high limit for the shift
 fit.ReferencesInfo[0].SetCoefficientHighLimit(DoasFitReferenceInfo.Coefficients.Shift,4);
 // activate the hard limits for the shift, be aware that the command covers 3 lines
fit.ReferencesInfo[0].SetCoefficientMode
    (DoasFitReferenceInfo.Coefficients.Shift,
    DoasFitReferenceInfo.CoefficientMode.LimitHard);
 // set the fit range
fit.FitRanges[0].LimitLow = 300;
fit.FitRanges[0].LimitHigh = 600;
// save the modified fit scenario, "DoasFit.Save()" returns "true", if saving
// was successful, if not, "false" is returned
if(DoasFit.Save(fit, "C:\\scripttest\\newfitscenario.fs"))
{
    Console.WriteLine("Fit scenario saved!");
}
else
{
    Console.WriteLine("Error while saving!");
}
```

After the script is executed the modified fit scenario can be opened in DOASIS from `C:\scripttest\newfitscenario.fs` and the newly implemented spectrum will appear to-

gether with the other spectra of the fit scenario in the Specbar and the limits for the shift are visible in the Fitting window.

Remarks

- By loading a new spectrum into the fit scenario, only the spectrum itself, meaning the channel/wavelength-intensity plot and the file name is changed. The "ObjectKey" property however does not change and, if required, must be changed separately by using the command:
  `fit.ReferencesInfo[0].ObjectKey = "..."`

- When limiting the shift it is important that the default value of the shift is in between the low limit and the high limit.

- The "squeeze" of each reference spectrum can be limited in a similar way. Just substitute
  `DoasFitReferenceInfo.Coefficients.Shift`
  by
  `DoasFitReferenceInfo.Coefficients.Squeeze`

- Spectra can only be implemented in a fit scenario with JScript, if the were opened within the same JScript before. Hence it is not possible to include a spectrum which was opened manually beforehand and the using the the command:
  `MeasSpec = Specbar.CurrentSpectrum;`
  In this case
  `fit.ReferencesInfo[0].ReferenceSpectrum = MeasSpec;`
  will not work.

- For setting the low limit of the fit range the command:
  `fit.FitRanges[0].LimitLow = 300;`
  is used. `FitRanges[]` is an array thus several fit ranges can be defined if required. The value "0" here represents the first fit range. It can be useful to define several fit ranges though if e.g. broken pixels or strong absorption lines of interfering absorbers should be excluded.

- For more information to modify the settings of the different reference spectra search for "DoasFitReferenceInfo" in the Programming Documentation of DOASIS (Help menu - Programming Documentation). For properties of the fit itself (like fit range, polynomial...) search for "DoasFit".

- Sometimes it is useful to fix the fit coefficient of certain reference spectra (see Fig. 11.2 "Fix Parameter". For setting the parameters of the fit coefficient the Tab "Coefficient" instead of "Shift" has to be chosen). However it does not work with JScript using the `SetCoefficientFixed()` command. No solution is known by now.

- Unfortunately linking parameters of reference spectra (see Fig. 11.2 "LinkParameter") spectra with JScript does not work either.

11.5 Save the residual of the fit result

In 11.3, we stored the fit results by using the AppendResultToFile function. It stores fit coefficients, errors, etc. in a file. Sometimes it is useful to store as well the whole spectrum of the residual. Unfortunately, there is no easy accessible function implemented in DOASIS so far, so the example given here is quite hard to understand, and it will not be explained

here in detail. Fitting is done as in 11.3, for saving the spectrum see 8.9.

```
import System;
import DoasCore.Spectra;
import DoasCore.Math;

var MeasSpec :  ISpectrum = Specbar.CurrentSpectrum;
var ResidualSpec:  ISpectrum = Specbar.GetSpectrum("Residual");
// the Spectrum "ResidualSpec" will get the same number of channels as
// the spectrum "MeasSpec"
ResidualSpec.NChannel = MeasSpec.NChannel;

var fit = DoasFit.Open("c:\\scripttest\\fitscenario.fs");

if(fit.DoFit(MeasSpec)){
    // get a vector, containing wavelength info from target spectrum
    var wavelength = fit.TargetSpectrum.Wavelength.ToVector();

    // get an empty vector
    var resultSpec = new DoasCore.Math.Fit.Vector(int(wavelength.Size));
    // fill the vector with intensity values of result function
    fit.ResultSpectrum.GetValues(wavelength, resultSpec);

    // create a new vector
    var targetSpec = new DoasCore.Math.Fit.Vector(int(wavelength.Size));
    // fill the vector with the intensity values of the target spectrum
    fit.TargetSpectrum.GetValues(wavelength, targetSpec);

    // subtract resultSpec from targetSpec:  the result is the residual
    targetSpec.Sub(resultSpec);

    // packs the two vectors into the residual spectrum
    ResidualSpec.Wavelength.FromVector(wavelength);
    ResidualSpec.Intensity.FromVector(targetSpec);
}
else
    System.Console.WriteLine("Error while fitting!");
```

Remarks

- ResultSpectrum is a function that you can actually see after fitting manually in the Fit Result window. For a good fit, it should be similar to the “TargetSpectrum”. “TargetSpectrum” is the spectrum in the Active Spectrum Sheet, that we want to fit. By subtracting the ResultSpectrum from the TargetSpectrum we get the residual.
- Don´t create the variable fit by writing `var fit:  DoasFit = new DoasFit()`, because then, TargetSpectrum does not support the function GetValues.
  But: If you want to get access to the properties of fit.ReferencesInfo like FitCoefficient or ObjectKey, you have to create the variable fit in that way! In case you want to get

the Residual and get access to fit.ReferencesInfo use two different variables fit1 and fit2, where you create each in a different way.

- We created three vectors here: targetSpec, resultSpec and wavelength. All vectors have as many entries, as the number of channels of the spectrum that we want to fit (TargetSpectrum). In the vector wavelength, each entry consists of a wavelength corresponding to the specific channel. The other two vectors contain intensity values. Make sure to take the logarithm of the spectrum you want to fit either by pressing "Logarithm" in the "Math" menu of DOASIS, or by activating the "Log. spectrum" control box in the fitting sheet of the console window.
- Is it possible to extract the fit functions of the reference spectra as well? No. At least not very easily.

Chapter 12

Advanced JScripts

The JScripts provided in this chapter are longer and therefore not commented out in great detail anymore. It needs some experience to understand them. Nevertheless, with the gained knowledge of the preceding chapters it shoud be possible to understand and use them.

12.1 Write spectrum data to an Excel-Sheet

Before running the script, open a random spectrum in DOASIS. While the script is executed, Excel will automatically launch and a new worksheet will be created. In the first column of the worksheet, the channels of the spectrum will be written and in the second one the intensity.

```
import System;
import DoasCore.Spectra;

// Start Excel and get Application object.
var oXL = new ActiveXObject("Excel.Application");

// Make the Excel program visible.
oXL.Visible = true;

// Get a new workbook and add a worksheet
var oWB = oXL.Workbooks.Add();
// Make the opened worksheet ready for modifying
var oSheet = oWB.ActiveSheet;

// Add table headers going cell by cell.
oSheet.Cells(1, 1).Value = "Pixel";
oSheet.Cells(1, 2).Value = "Current";

// put the intensity values of the current spectrum into
// the Excel sheet
var i;
for(i = 0; i < Specbar.CurrentSpectrum.NChannel; i++)
{
    oSheet.Cells(2 + i, 1).Value = i;
    oSheet.Cells(2 + i, 2).Value = Specbar.CurrentSpectrum.Intensity[i];
    Console.WriteLine("Put pixel " + i + " to Excel sheet");
}

Console.WriteLine("Ready.");
```

12.1.1 A word on ActiveXObjects

In the example above the command `var oXL = new ActiveXObject("Excel.Application")` is used. It creates an "Application Object" which in this case is of the "type" Excel. This object is put into the variable oXL. We have to imagine, that this variable contains somehow the whole Excel program. By using it's properties like e.g. "ActiveSheet" or "Visible" and methods you can work in Excel "through the channel" of the variable oXL. Under `http://msdn2.microsoft.com/en-us/library/bb149137.aspx` you can find a summary of all members of such an Application Object. In a similar way you could connect e.g. to Outlook.
ActiveXObjects are often used to get an interface to other programs. Here are some more examples for "ActiveXObjects", which can be used instead of "Excel.Application"

- Scripting.FileSystemObject
- WinDoasMotor.DoasMotor
- WinDoasMath.DoasMath

12.2 Exception handling

ßindextry..catch..finally JScript provides the possibility to handle exceptions by putting commands which might cause errors in the “try”-block of a so called try/catch/finally environment. If any command of the code enclosed in the “try”-block causes an error, the **entire** block will be skipped and instead the content of the “catch”-block is executed. If no error occurs in the “try”-block, the “catch” block will not be executed. The content of the “finally” block is executed no matter whether an error occurred in the “try”-block or even in the “catch”-block or not. In the following example, the type of the error is displayed in a pop up window. By pressing “yes” or “no”, the user can either accept the error and continue the script or abort it.

```
import System;
import DoasCore.Spectra;
// This namespace provides the class "OldUI"
import DoasCore.HMI;

// Open an empty spectrum in the "Specbar"
var Spec = Specbar.GetSpectrum("MeasSpectrum");
// Create an instance of "OldUI"-class which contains the method "MessageBoxYesNo"
// to create a checkbox
var checkbox = new OldUI();
var answer;

function main()
{
    // In the "try"-block, a spectrum should be loaded into the empty spectrum
    // "Spec".  If the file "C:\Spectrum.sp2" exists, the spectrum will be
    // opened and the message "Spectrum opened!" is displayed.  If the file
    // does not exist, the entire "try"-block is skipped and the "catch"-block
    // is executed.
    try
    {
       Spec.Open("C:\\Spectrum.sp2");
       Console.WriteLine("Spectrum opened!");
    }
    catch(e)
    {
       // Show a pop up window including a "yes/no" button with the title
       // "An error occurred!" and displaying the reason for
       // the error (e).  If "yes" is pressed the method returns "true", if "no"
       // is pressed it returns "false"
       answer = checkbox.MessageBoxYesNo(e+" Continue?","An error occurred!");
       // If "answer = true", then continue, else exit the "main()"-function
       // and thus the entire script
       if(answer)
       {
          Console.WriteLine("Script is continued!");
       }
       else
       {
          Console.WriteLine("Script is aborted!");
          // Exit the "main()"-function
          return;
       }
    }
    finally
    {
       Console.WriteLine("Always execute this part!");
    }
    Console.WriteLine("End of script!");
}

// Run the main function
main();
```

Remarks

- For the first time in this manual a function is declared here. Making use of functions is a very common and convenient style of programming. The program itself only starts in the last line:
  `main();`
  calling the function "main()" which has been declared in the beginning. In this script the function "main()" was used in order to be able to quit the script by leaving the "main()"-function, if the user clicks on "no" when the checkbox appears.
- The "finally"-block is executed even though "no" is pressed in the checkbox and thus the "return" command is called. However the command
  `Console.WriteLine("End of script!")`
  is not executed in that case.
- It is advisable to be very careful with the usage of the "try/catch" routine. If the error message is not displayed, the program might pretend to work well even though it only jumped over parts of the code.
- More information about checkboxes can be found in the Programming Documentation of DOASIS by searching for "OldUI".

12.3 Evaluate data with a fit scenario file

This is a more elaborate example of how to evaluate your data using a fit scenario file. It was suggested by Stefan Kraus, the programmer of DOASIS.

```
/* EvaluateExample.js\\

   This example script gives an example about how to evaluate existing
   spectra using a already given fit scenario.

   Note: To run the example its necessary to have already some spectra
   saved as
   C:\scanTest
   and that you have an existing fit scenario file
   C:\scanTest\EvaluateExample.fs

   Author: Stefan Kraus
   Version: 1.0 @ 10/28/2003 */

import System;              // use the System libary of the .NET framework
import DoasCore;            // use the DoasCore library
import DoasCore.Device;     // use the Device namespace
import DoasCore.IO;         // use the Input-/Output namespace
import DoasCore.Script;     // use the Script namespace
import DoasCore.Spectra;    // use the Spectra namespace

// create the spectrograph device object
var spectrograph = new Spectrograph();

// create the AutoFileName object to store the files
```

```
var afnFile = new AutoFileName();
afnFile.BasePath = "C:\\scanTest";
afnFile.Prefix = "a";
afnFile.Suffix = ".std";
afnFile.FilesPerFolder = 100,
afnFile.NumberOfDigits = 5;

// look for the first existing file
afnFile.FindFirstIndex();

// create the spectrum object that will receive the scanned data
var specEval = Specbar.GetSpectrum("Eval");

// open the fit scenario
// in this case we need to define the complete namespace hierarchie,
// since another Math namespace exists in the System-library, too!
var fit = DoasCore.Math.DoasFit.Open("c:\\scanTest\\EvaluateExample.fs");

// repeat until the stop button was pressed and a file could be loaded
while(!Script.StopAllScripts && afnFile.Open(specEval))
{
    // set the target spectrum for fitting
    fit.TargetSpectrum = specEval;

    // run the fit
    if(fit.DoFit())
    {
        System.Console.WriteLine("Fit successful for spectrum " + specEval.FileName);
        fit.AppendResultToFile("C:\\scanTest\\EvaluateResult.txt");
    }
    else
        System.Console.WriteLine("Fit failed for spectrum " + specEval.FileName);
}

System.Console.WriteLine("Evaluate stopped.");
```

12.4 Automatized scanning

This example shows how to do an automated scanning for measurement spectra and save them using successive file names. As well suggested by Stefan Kraus.

```
/* ScanExample.js

  This example script gives an example about how to scan using the default spectrograph
  and saves the measured spectra one after another using the AutoFileName scheme.

  Author: Stefan Kraus
  Version: 1.0 @ 10/28/2003 */


import System;                  // use the System library of the .NET framework
import DoasCore;                // use the DoasCore library
import DoasCore.Device;         // use the Device namespace
import DoasCore.IO;             // use the Input-/Output namespace
```

```
import DoasCore.Script;           // use the Script namespace
import DoasCore.Spectra;          // use the Spectra namespace

// create the spectrograph device object
var spectrograph = new Spectrograph();

// create the AutoFileName object to store the files
var afnFile = new AutoFileName();
afnFile.BasePath = "C:\\scanTest";
afnFile.Prefix = "a";
afnFile.Suffix = ".std";
afnFile.FilesPerFolder = 100,
afnFile.NumberOfDigits = 5;

// look for the last existing file and set the file counter to the successor
afnFile.FindLastIndex();

// create the spectrum object that will receive the scanned data
var specScan = Specbar.GetSpectrum("Scan");

// repeat until the stop button was pressed
while(!Script.StopAllScripts)
{
    // show some info in the Output window
    System.Console.WriteLine("Scanning spectrum " +
        afnFile.CurrentFileNumber.ToString() + "...");

    // scan using a fixed number of scans and exposure time
    spectrograph.Scan(specScan, 10, 1000);

    afnFile.Save(specScan);
    System.Console.WriteLine("Saved spectrum " +
            afnFile.CurrentFileNumber.ToString() + "!");
}

System.Console.WriteLine("Scanning stopped.");
```

12.5 Threading

This example shows how to create multiple thread that run at once. Each thread can be supplied with a different information. Also the usage of classes and object is shown.

```
// first include the default namespaces
import System;
import DoasCore;
import DoasCore.Spectra;
import DoasCore.Math;

// use the threading namespace of the .NET framework
import System.Threading;

// Create a class that represent the thread including any additional
// data required to start the thread. In our case the thread has
// one additional data elements: The internal variable 'objectInfoForThread'.
```

```
// The content of 'objectInfoForThread' is set when the object is created.
// The example function that actually runs in a separate thread can access this
// additional information.

class ThreadTest
{
    // an internal variable used to store the additional
    // information for the thread
    private var objectInfoForThread;
    // This variable will store the thread object. Using this object
    // we can control the thread (Start, Stop, Pause, ...)
    private var threadWorker : Thread;

    // this is the constructor of the object.
    // whenever you create an object of this type with the
    // new-operator this function will be called. In our case
    // the contrutor requires one parameter. The data supplied
    // here will be used by the thread.
    function ThreadTest(info)
    {
        // store the additional information in the internal variable
        objectInfoForThread = info;

        // create the thread object. As soon as the thread is started
        // is will run the function 'RunWorker' in a separate thread.
        // See .NET documentation System.Threading.Thread
        // 'RunWorker' is of the type 'ThreadStart'. This has to
        // declared, since the 'Thread' object has several constructors.
        threadWorker = new Thread(ThreadStart(RunWorker));
    }

    // this function actually starts the thread that was created
    // in the constructor
    function StartThread()
    {
        threadWorker.Start();
    }

    // This function waits until the thread function terminates.
    function WaitThreadFinished()
    {
        threadWorker.Join();
    }

    // This is the function that will run in a separarte thread as soon as the
    // 'StartThread' function is called. The thread will run until this function
    // exits or you terminate the thread using the 'threadWorker' object.
    function RunWorker()
    {
        // just do something inside the thread
        var i : int = 0;
        while(i < 100)
        {
          // print the additional information supplied
```

```
            // on creation of the object including the loop counter
            System.Console.WriteLine(objectInfoForThread.ToString() + ": " + i.ToString());

            // reduce the computation requirements by setting the current thread
            // into sleep mode for a few milliseconds
            Thread.Sleep(10);

            i++;
          }
      }
};  // end of class

// create two object of type ThreadTest. Supply each object
// with additional information
var threadOne : ThreadTest = new ThreadTest("Erster Thread");
var threadTwo : ThreadTest = new ThreadTest("Zweiter Thread");

// start both threads
threadOne.StartThread();
threadTwo.StartThread();

// The JScript will run until both threads terminate.
// NOTE: the main thread of the JScript will terminate immediatelly
// after the two worker threads are started. Therefore the script
// dialog will 'think' the JScript already terminated! If you want
// to avoid this, its necessary to wait here in the main thread until
// all worker thread have been terminated.
// uncomment the following lines to see the difference
threadOne.WaitThreadFinished();
threadTwo.WaitThreadFinished();

System.Console.WriteLine("Main thread ends now!");
```

12.6 Controlling Mini-MAX-DOAS (MiniDOAS) instruments with JScript

This is the last and the longest script of the tutorial. It was provided by Denis Pöhler. It hasn't been tested by the author but was applied during several measurement campaigns exactly in the way it is presented here.

12.6.1 Structure of the program

For that program it would have been of advantage to establish a so JScript project (see section 4.2). The script can be divided into three parts:

1. The declaration of variables.

2. The main()-function.

3. The different functions used in the main()-function

Each of these three parts could have been placed in a separate JScript file and then called from a JScript project file. This way the variables and functions would still be available for

further JScripts.
The main()-function only takes about one page out of the ten pages of the entire code. In the main()-function the different functions declared towards the end of script are called. The basic tasks of the Script can already understood by reading the main()-function only, since all the other functions reveal their functionality more or less by their name. It is very important that the main()-function itself is called at the end of the script.

12.6.2 Description of the tasks

The basic task of the script is very simple:
The MiniDOAS device changes the elevation angle of its line of sight in a predefined way and records a spectrum for each elevation angle.
However, the script includes many special functions of different tasks:

- It continuously logs the voltage, current and temperature of the instrument (see `function WriteToLog()`
- It calculates the solar zenith angle (SZA) (see `function aurinko()`)for each measurement for two reasons:
  1. To interrupt the "normal" measurement mode after sunset and from that time onwards only measure offset and dark current in a fixed position.
  2. To save the SZA in the properties of the spectrum for later evaluation purposes.
- The `function TakeSpectrum()` automatically adapts the integration time to the current light conditions by running several test scans and prolonging or shortening the integration time according to the saturation of the spectrum.

12.6.3 Prerequisites

In the script several "ActiveXObjects" are used:

- WinDoasIO.FileIO
- WinDoasMath.DoasMath
- WinDoasTools.DoasTools
- Scripting.FileSystemObject
- HMTUSB.PDZControl

These can be understood a separate "programs" (such as Microsoft Excel, see section 12.1.1) which can be accessed via JScript by ActiveXObjects. Therefore different drivers and programs must be installed. It showed in the past that as well the order of installation is important. Here a list of the required drivers in the right order:

1. Ocean Optics software
2. InstallerHMTUSBDriver
3. InstallerHMTActiveXControll
4. DOASIS
5. If required Mini-Max Software (Udo Friess)
6. VNC

Of course the different file and path names mentioned in the script have to be adapted accordingly.

```
import System;
import DoasCore;
import DoasCore.IO;
import DoasCore.Spectra;
import DoasCore.Math;
import DoasCore.Device;

// ************* Declarations of variables ***********************************


// controls the file handling similar to the AutoFileName() class in DOASIS
var dispIO = new ActiveXObject("WinDoasIO.FileIO");
// tool e.g. to correct offset and to calculate solar zenith angle
var dispMath = new ActiveXObject("WinDoasMath.DoasMath");
// tool for spectrum handling
var dispTools = new ActiveXObject("WinDoasTools.DoasTools");
// provides message boxes
var window = new DoasCore.HMI.OldUI();
// create an object which enables to store data in a text file
var dispFileSystem = new ActiveXObject("Scripting.FileSystemObject");
// provides access to "Hoffmann Messtechnik Elektronik (HMT)" which
// the stepper motors
var dispHMTUSB = new ActiveXObject("HMTUSB.PDZControl");

var serialHMT = "13291";                        // serial of HMT hardware
var serialUSBHMT = "2E7150";                    // serial saved in HMT hardware
var serialUSB2000 = "2E4670";                   // serial of USB2000
var MeasSite = "Joelle_1";
var CampInit = "a1";
var Latitude = 42.0;                            // south is negative
var Longitude = 8.0;                            // west is negative
var ViewingAzimuth = 200;                       // 0 is north
var SZAnight = 96;
var Tini=0;                                     // detector temperature
var CalOrder = 2;
var CalZero = 302.74;
var CalFirst : double = 0.087134;
var CalSecond : double = -9.0073e-6

var straylight = new Array();
var AngleSeries = new Array(90,20,10,5,2,1);   // multiples of 0.9
var AngleTimes  = new Array(1,1,1,1,1,1);      // reps for each angle
var MotorHorizontalPosition = 6000;            // horizontal was 6200
var MotorFrequency = 740;                      // typical MiniDOAS = 740
var MotorOSDCPosition = -10;          // offset dark current position in degree
var MotorStep90 = 10000               // number of steps per 90°
var s;
var smin = 1;                         // at least one scan
var tmax = 10000;                     // maximum time per scan in ms
var smax = 1000;                      // maximum number of scans
var t = 100;                          // initial integration time
```

```
var ttotal = 100000;                    // total integration time
var sOS = 10000;                        // number of offset scans
var tOS = 3;                            // offset integration time in ms
var tDC = 20000;                        // dark current integration time
var sat = 3000;                         // saturation level(max 4095)
var tNap = 600000;                      // break between two offset mesurements(ms)
var waitscan = 150;                     // break after scan
var waitmotor = 500;

var BasicPath = "D:\\MAXDOAS_Amundsen08\\Amundsen\\080312\\";
var MFCPath = BasicPath + "specMFC";
var OSDCPath = BasicPath + "specOSDC";
var specPath;
var OS0 = BasicPath + "TestAnDS\\offset2";

// create an Ocean Optics spectrograph object
var specOOI = new DoasCore.Device.OceanOptics();
specOOI.USBSerial = "USB" + serialUSB2000;
var Tcold, Tdet;                                // current temp, set temp
var currentelevation;
var ADCVal = new Array();
var SolarValues;
var SZALongitude = -Longitude;
// create files to log data
var outfile=dispFileSystem.OpenTextFile(BasicPath + "DOASlogfile.txt",8, true);
var DoasLog=dispFileSystem.GetFile(BasicPath + "DOASlogfile.txt");
outfile.Close();
var outfile=dispFileSystem.OpenTextFile(BasicPath + "ADClogfile.txt",8, true);
var AdcLog=dispFileSystem.GetFile(BasicPath + "ADClogfile.txt");
outfile.Close();
var f, i, j, k, l, m, n, o, time, CurrentDate, CalAnswer, TempAnswer, fullname;
var newspec, measspec, offset0, offset1, offset2, dark1, dark2;


// ************** main() function ***********************************************

function main()
{
    dispIO.Silent = true;
    WriteToLog(AdcLog,"UBatt [V]; USteuer [mV];
        TCold [C]; TWarm [C]; TSet [C]; IPelt [A]; ");
    // initialise the Hoffmann controllers
    InitHMTUSB();
    // check if script was aborted by pressing the stop button in the Script window
    StopCheck();
    // MakeSpec() creates empty spectra in the Specbar
    offset0 = MakeSpec("OOoffset0");
    offset1 = MakeSpec("OOoffset1");
    offset2 = MakeSpec("OOoffset2");
    dark1   = MakeSpec("OOdark1");
    dark2   = MakeSpec("OOdark2");
    dispTools.CurrentSpectrum = offset0;

    for (j=0;j<AngleSeries.length;j++)
```

```
    {
        straylight[j] = MakeSpec(AngleSeries[j].toString());
    }

    while (!window.CheckStop())
    {
        // calculate the solar zenith angle
        SolarValues = aurinko(Latitude,SZALongitude);
        StopCheck();
        // only measure during day time
        if (SolarValues.SZA < SZAnight)
        {
            WriteToLog(DoasLog, "Started MAX routine");
            MotorMoveHome();
            StopCheck();
            for (j=0;j<AngleSeries.length;j++)
            {
                StopCheck();
                for (k=0;k<AngleTimes[j];k++)
                {
                    StopCheck();
                    {
                        StopCheck();
                        SolarValues = aurinko(Latitude,SZALongitude);
                        WriteToLog(DoasLog, "SZA = "
                            +parseInt(SolarValues.SZA*1000)/1000+"°  "
                            +"SAA = "+parseInt(SolarValues.SAZ*1000)/1000+"°");
                        // calculate integration time and measure spectrum
                        TakeSpectrum(AngleSeries[j])
                    }
                }
            }
        }
        // during night time record offset and dark current
        else if (SolarValues.SZA > SZAnight)
        {
            MotorMoveToPos(MotorOSDCPosition);
            WriteToLog(DoasLog, "Started Offset and DC routine");
            TakeOffsetDarkCurrent();
            StopCheck();
        }
    }
    WriteToLog(DoasLog, "Stopped");
}

// ********* End of main() function ********************************************

// ********* Declare the individual functions called in the main() routine ******

// Initialize device
function InitHMTUSB()
{
    window.Status = "Initializing USB device " + serialHMT + "/"
        +serialUSBHMT + " ...";
```

```
    dispHMTUSB.SetSerial(serialHMT + "/" + serialUSBHMT);
    if (!dispHMTUSB.IsDevicePresent())
    {
        WriteToLog(DoasLog, "Device not found - sleep 1 min.");
        window.Sleep(60000);
        StopCheck();
    }
    else window.Status = "Device found!";
    dispHMTUSB.UndervoltageDetection == false;
    if(!(dispHMTUSB.UndervoltageDetection))
        window.Status = "Low battery warning is turned off!";
    else
    {
        WriteToLog(DoasLog, "Low battery warning could not be turned off");
        return;
    }
    dispHMTUSB.SetMotorMode(1, 2);                  // half step mode
    dispHMTUSB.SetMotorFreq(1, MotorFrequency);     // frequency
    MotorMoveHome();

    // check Temp settings
    dispHMTUSB.Temperature = Tini;
    WriteToLog(DoasLog, "Temperature set to " + Tini + " deg C!");
    return;
}

// similar function as InitHMTUSB()
function InitHMT()
{
    window.Status = "Initializing USB device " + serialHMT + "/"
        + serialUSBHMT + " ...";
    dispHMTUSB.SetSerial(serialHMT + "/" + serialUSBHMT);
    if (!dispHMTUSB.IsDevicePresent())
    {
        WriteToLog(DoasLog, "Device not found - sleep 1 min.");
        window.Sleep(60000);
        StopCheck();
    }
    else window.Status = "Device found!";
    dispHMTUSB.UndervoltageDetection == false;
    if(!(dispHMTUSB.UndervoltageDetection))
        window.Status = "Low battery warning is turned off!";
    else
    {
        WriteToLog(DoasLog, "Low battery warning could not be turned off");
        return;
    }
    dispHMTUSB.SetMotorMode(1, 2);                  //half step mode
    dispHMTUSB.SetMotorFreq(1, MotorFrequency);     //frequency
    MotorMoveHome();
}

// calculate solar zenith angle
// computer time must be set to GMT!!!
```

```
function aurinko(lat,lon)
{
   time = new Date();
   var y = time.getFullYear();
   var m = time.getMonth()+1; //Bug!!
   var d = time.getDate();
   var h = time.getHours();
   var min = time.getMinutes();
   var s = time.getSeconds();
   var tz = time.getTimezoneOffset();
   if (tz != 0) tz /= 60;

   var juliantime;
   juliantime = dispMath.MakeJulianDate(y,m,d,h+tz,min,s);
   SolarValues = dispMath.GetSZA(juliantime,lat,lon);
   return(SolarValues);
}

// create empty spectra in the Specbar of DOASIS
function MakeSpec(specname)
{
    newspec = dispTools.GetSpectrum(specname);
    newspec.Name = specname;
    newspec.MinChannel = 2;
    newspec.maxchannel = 2048;
    return(newspec);
}

// calculates integration time and records spectra
function TakeSpectrum(angle)
{
    specPath = MFCPath;
    StopCheck();
    for (l=0;l<AngleSeries.length;l++)
    {
        if (angle == AngleSeries[l])
        {
            measspec = straylight[l];
            if (angle != currentelevation)
            {
                MotorMoveToPos(angle);
                currentelevation = angle;
            }
        }
    }

    dispTools.CurrentSpectrum = offset0;
    if (offset0.Average < 100)
        specOOI.Scan(offset0,100,3);      // measure offset spectrum
        window.Sleep(waitscan);

    dispTools.CurrentSpectrum = measspec;
    specOOI.Scan(measspec,1,t);           // measure test spectrum
    window.Sleep(waitscan);
```

```
dispMath.Correctoffset(measspec,offset0)  // correct measspec for offset
// determine integration time
while (measspec.Max > sat)
{
    StopCheck();
    t = (parseInt(0.8 * t));
    window.Status = "IT= "+t;
if (t < 3)
     {
        t = 3;
        break;
     }
    specOOI.Scan(measspec,1,t);
    window.Sleep(waitscan);
    dispMath.Correctoffset(measspec,offset0)
}
specOOI.Scan(measspec,1,t);
window.Sleep(waitscan);
while (measspec.Max < (0.8*sat))
{
     StopCheck();
     t = (parseInt((1.2 * t)+0.5));
     window.Status = "IT= "+t;
     if (t > tmax)
     {
        t = tmax;
        break;
     }
     // the integration time has been determined with t
     // the number of scans still has to be calculated
     specOOI.Scan(measspec,1,t);
     window.Sleep(waitscan);
     dispMath.Correctoffset(measspec,offset0)
}
// calculate the number of scans s
s = parseInt(ttotal/t);
if (s>smax)
{
s=smax;
}
if (s < 1)
{
s=smin;
}
// now s has been determined
dispIO.PathSplitting(1,specPath,100);
dispIO.AutoFileNumber( 1,CampInit,6,1);
i = dispIO.AutoFileNumberFindLastIndex() + 1;
window.Status = "Taking spectrum #" + i + "  " + s + "x" + t +"ms @ "+angle+"°";
// the actual spectrum, which will be used for evaluation later is recorded now!
specOOI.Scan(measspec,s,t);
window.Sleep(waitscan);
//
StoreADCValues(measspec);
```

```
    measspec.CalibPolynomialOrder = 2;
    measspec.CalibPolynomial[0] = CalZero;
    measspec.CalibPolynomial[1] = CalFirst;
    measspec.CalibPolynomial[2] = CalSecond;
    measspec.Site = MeasSite;
    // the spectrum is now saved!!!!!!!!
    dispIO.Save(measspec);
    // check if the file of the spectrum exists
    if (CheckFileSize(measspec.FileName))
        WriteToLog(DoasLog, measspec.FileName+" looks ok - IT was " + s
            + " scans at" + t + " ms");
    // modify integration for next spectrum if spectrum was over- or undersaturated
    if ((measspec.Max/s) > sat)
    { t = parseInt(0.8 * t)   }
    if ((measspec.Max/s) < (0.8*sat))
    {
        t = parseInt(1.2 * t);
    }
    // checks, if stored spectrum shows signal
    CheckIfSpecOk(measspec);
    i++;
}

function TakeOffsetDarkCurrent()
{
    StopCheck();
    specPath=OSDCPath;
    window.Status = "Offset spectrum 1: " + sOS + " x " + tOS + " ms";
    // measure offset or dark current depending on given parameters (see function below)
    OSDC(offset1,100,sOS,tOS);
    window.Status = "Offset spectrum 2: " + sOS + " x " + tOS + " ms";
    OSDC(offset2,100,sOS,tOS);
    window.Status = "Dark current spectrum 1: 1 x " + tDC + " ms";
    OSDC(dark1,200,1,tDC);
    window.Status = "Dark current spectrum 2: 1 x " + tDC + " ms";
    OSDC(dark2,200,1,tDC);
    window.Status = "Went for a nap - " + tNap/60000 + " min ... See you soon!!!";
    window.Sleep(tNap);
    StopCheck();
}

// measure offset or dark current
function OSDC(spec,wl,ss,tt)
{
    StopCheck();
    dispTools.CurrentSpectrum = spec;
    specOOI.Scan(spec,ss,tt);
    window.Sleep(waitscan);
    StoreADCValues(spec);
    spec.Wavelength1=wl;
    dispIO.PathSplitting(1,specPath,100);
    dispIO.AutoFileNumber( 1,CampInit,6,1);
    o = dispIO.AutoFileNumberFindLastIndex() + 1;
    dispIO.Save(spec);
```

```
    if (CheckFileSize(spec.FileName))
        WriteToLog(DoasLog, spec.FileName+" looks ok");
    o++
    StopCheck();
}

// store the values of the analog digital converter (Temperature, Voltage....)
function StoreADCValues(spec)
{
        CheckIfSpecOk(spec);
        ADCVal[0] = dispHMTUSB.ADC(0)/100;          //UBatt
        ADCVal[1] = dispHMTUSB.ADC(1);              //USteuer
        ADCVal[2] = (dispHMTUSB.ADC(2)-2048)/50;    //Tcold
        ADCVal[3] = dispHMTUSB.ADC(3)/50;           //Twarm
        ADCVal[4] = (dispHMTUSB.ADC(4)-2048)/50;    //Tset
        ADCVal[5] = dispHMTUSB.ADC(5)/1000;         //IPelt
        ADCVal[6] = dispHMTUSB.ADC(6);
        WriteToLog(AdcLog, ADCVal[0] + "; " + ADCVal[1] + "; " + ADCVal[2]
            + "; " + ADCVal[3] + "; " + ADCVal[4] + "; " + ADCVal[5] + "; " + ADCVal[6]);
        var RAWADC = new Array();
        for (m=0;m<6;m++)
            RAWADC[m] = dispHMTUSB.ADC(m);
        spec.Latitude=Latitude;
        spec.Longitude=Longitude;
        Tcold = ADCVal[2]
        spec.Temperature=Tcold;
        if (Math.abs(Tcold-Tini) > 1)
        {dispHMTUSB.Temperature = Tini;
        WriteToLog(DoasLog, "Set Temperature " + Tini + " deg C, not Reached!
            Set Temp new to ADC Controller.");
        }
}


function MotorMoveToPos(angle_L)
{
    var pos = parseInt(angle_L/90*MotorStep90) + MotorHorizontalPosition;
    dispHMTUSB.MoveToPos(1,pos);
    window.Sleep(waitmotor);
    var Position = dispHMTUSB.GetMotorPosition(1);
    window.Sleep(waitmotor);
    if (Position == pos)
        WriteToLog(DoasLog, "Motor position "+pos+" ("+angle_L+"°) reached");
    else
    WriteToLog(DoasLog, "Motor position "+pos+" not reached. Motor responds "+Position);
}

function MotorMoveHome()
{
    dispHMTUSB.MoveHome(1);
    window.Sleep(waitmotor);
    if (dispHMTUSB.GetMotorStatus(1)==3)
        WriteToLog(DoasLog, "Home Position reached");
    else
```

```
    {
        WriteToLog(DoasLog, "Home Position not reached");
        return;
    }
    window.Sleep(waitmotor);
    if (dispHMTUSB.GetMotorPosition(1)==0)
        WriteToLog(DoasLog, "Home Position set to 0");
    else
   {
        WriteToLog(DoasLog, "Home Position not 0");
        return;
   }
   StopCheck();
}

function WriteToLog(file,message)
{
    window.Status = "log> " + message;
    var out=file.OpenAsTextStream(8);
    out.WriteLine(UTCDate() + " ; " + message);
    out.Close();
}

//returns the date e.g. as 05.06.2008 12:05:32
function UTCDate()
{
   var d, s="";
   var day,month,hours,min,sec;
   d = new Date();
   day = d.getUTCDate();
   if (day.toString().length == 1) day = "0" + day;
   s += day + ".";
   month = d.getUTCMonth()+1;
   if (month.toString().length == 1) month = "0" + month;
   s += month + ".";
   s += d.getUTCFullYear()+" ";
   hours = d.getUTCHours();
   if (hours.toString().length == 1) hours = "0" + hours;
   s += hours + ":";
   min = d.getUTCMinutes();
   if (min.toString().length == 1) min = "0" + min;
   s += min + ":";
   sec = d.getUTCSeconds();
   if (sec.toString().length == 1) sec = "0" + sec;
   s += sec;
   return(s);
}

// check if spectrum contains signal
function CheckIfSpecOk(spectrum)
{
    if (spectrum.Average == 0e-20)
    {
    WriteToLog(DoasLog, "Average is ZERO");
```

```
    WriteToLog(DoasLog, "No Spectrum taken, intensity is
        zero due to an error! New initialisation of HMT device");
    window.Sleep(1000);
    InitHMT()
    }
}


// check if file saved spectrum actually exists
function CheckFileSize(filespec)
{
    fullname = specPath + "\\" +filespec.substr(0,6)+"00\\"+filespec;
    System.Console.WriteLine(fullname);
    f = dispFileSystem.GetFile(fullname);
    if (f.size == 0)
        WriteToLog(DoasLog, f.Name + " has " + f.size + " bytes. Restart DOASIS");
    else
        return(f.size);
}

// check, if stop button in Script window has been pressed
function StopCheck()
{   if (window.CheckStop())
    {   WriteToLog(DoasLog, "Stopped");
        return;
    }
}

main();
```

12.6.4 Remarks:

- The motor status checked by the command
  `dispHMTUSB.GetMotorStatus()`
  can have the values:
  - 0: motor stopped
  - 1: motor is moving
  - 2: positive end reached
  - 3: negative end (homing position) reached
- The command `window.sleep()` is used frequently in the script. It helps to guarantee reliable data transfer between computer and measurement instrument (see Tips and Tricks of section 4.5).

Chapter 13

Literature

JScript Reference Microsoft
`http://msdn.microsoft.com/en-us/library/x85xxsf4.aspx`

JScript Textbooks

- Rogers, Justin (2002). Microsoft JScript .NET Programming. Indianapolis: SAMS
- Stone, M. (1996). How to Program Microsoft Jscript, Scripting Interface: Ziff Davis
- Jaworski, J. (1999). Mastering JavaScript and JScript (Mastering). London: Sybex Inc.

Google

The best way to google for JScript commands is to type:
"msdn2 + command"
For example:
"msdn2 ToString"
returns as the first link the page:
`http://msdn.microsoft.com/en-us/library/system.object.tostring.aspx`
which should hopefully be exactly what you want.

Index

ActiveXObject, 38, **66**, 74
AND, 18
AppendResultToFile, 58
AutoFileName, 28, **42**
    errors related to, 45
    Open, 43
    Save, 43
automatic start of JScript, 10
automized measurements, 70

backslash, 27
batch file, 9
baud rate, 52
boolean, 18
break, 36

CalcSZA, 26
calculate
    integration time, 30
calibration
    polynomial, **48**
    wavelength, 48
calibration polynomial, 29
catch, 67
check box, 32, 69
ChiSquare
    display, 59
class, 71
colon, 20
comments, 14
Console, 31
correct
    offset and dark current, 25, **55**
CorrectDarkCurrent, 55
CorrectOffset, 55
crash
    avoid, 10
CurrentFileNumber, 44

data transfer, 10
data types, 15
    conversion, 15
    lost of, 16
date
    get, 39
DateTime, 45

decimal numbers, *see* digits
Device, 27, **54**
digits
    control, 40
directory
    create, 37
DoasCore, 23
    IO, 28
DoasCore.Spectra, 13
DoasFit, 26, 59
DoasFitReferenceInfo, 59
driver, 74

else if, 19
equals, 20
Excel
    ActiveXObject, 66
    create worksheet, 65
    write data into sheet, 65
exception handling, 67

factorial, 21
false, 18
FileSystemObject, 38
find peak of spectrum, 29
FindLastIndex, 43
fit, 26, **58**, 69
    access properties, 63
fit coefficient
    display, 58
fit range, 61
fit result
    show, 58
fit scenario, 26, **58**
    active/deactivate spectra, 59
    modify, 59
    number of spectra, 59
    replace spectrum, 59
    set fit range, 61
FitCoefficientError
    display, 59
fix parameter, 62
for loop, 20
for..in-loop, 21
Forms, 32

functions, 21, 57

g5, 41
GetSpectrum, 25
google, 85
GPS, 51

handshake, 52
hardware
    communication with, 51
Hello World, 12, **35**
HMI, 31, **36**

if..else, 18
import, 14
input, 32
input/output, 28
InstallerHMTActiveXControll, 74
InstallerHMTUSBDriver, 74
internet, 85
IO, **38**
ISpectrum, 29, 42, 45
    members, 29

JScript project file, 9, **49**, 56
jsp, *see* JScript project file
Julian, 47
JulianDateTime, 25, 47

LAZ, *see* lunar azimuth angle
LimitHard, 61
link parameter, 62
lunar azimuth angle, 48
lunar zenith angle, 48
LZA, *see* lunar zenith angle

main function, 74
marker, 29
Math, 25
message box, 32, **37**
MinChannel, 29
Mini Scripts, 13
MiniDOAS
    control, 54
MiniDoas, **73**
minus, 17

namespace, 12
namespaces
    overview, 23
NaN, 17
notation
    scientific, 41

object orientation, 12, **57**
ObjectKey, 25, **45**, 62
Ocean Optics, 53
    control two, 54
OldUI, 36
OpenTextFile, 39
operators, 17
    list of, 17
    logical, 18
OR, 18
Origin, 39

parity, 52
plus, 17
port, 52
PrepareFitResultSpectrum, 58
Progress Bar, 31
Project file, 9
project files, *see* JScript project file

question mark, 20

ReadLine, 32
ReferencesInfo, 61
residual
    save, 62
ResultSpectrum, 63
run several JScripts, 9

SaveAs, 29
SAZ, *see* solar azimuth angle
Scan
    OceanOptics, 28, **53**
ScanGeometry, 25, **47**
scanning
    automized, 70
Script, 30
semicolon, 12
serial port
    communication with, 51
SetCoefficientDefault, 61
SetCoefficientHighLimit, 61
SetCoefficientLowLimit, 61
SetCoefficientLowMode, 61
shift
    display, 59
    limit parameter, 61
sleep, 36, 84
solar azimuth angle, 48
solar zenith angle, 25
    calculate, **47**
Specbar, 13, 29
SpecMath, 25
Spectra, 28
spectrograph
    control, 53

control two, 54
serial number, 53
spectrum
file name, 59
measure, 53
modify, 29
name, 13
open, 41
optimize, 30
properties, 45
save, 41
Squeeze
display, 59
limit, 62
start JScript, 5
Stop button, 31
StopAllScripts, 30
stopbits, 52
String, 12
syntax, 14
System, 12, 31, **35**
IO, 32
SZA, 25, *see* solar zenith angle

TargetSpectrum, 63
text file, 32
create, 38
textbooks, 85
Threading, 32, **36**, 71
time
compare, 45
get, 39
TimeSpan, 45
true, 18
type mismatch, 17

variable
case sensitive, 14
conversion, 15
declare, 13
initialize, 17
type, 44
types, 13
variables
display, 35
vectors, 64

wait, 36, 84
while loop, 21
WriteLine, 12, 35