TOPSPIN

Python Programs

Contents

Chapter 1

Introduction

1.1 Purpose

This document describes how you can extend the TOPSPIN program with your own functionality by writing Python modules or scripts. In this manual we will use the terms Python *module, program,* and Pythons *script* synonymously.

Python is a modern programming language which was embedded in TOPSPIN to provide the possibility of adding functions in a simple manner.

Any TOPSPIN user can easily write Python scripts to open and process data by following the examples presented in this manual. For simple applications, practically no experience in Python programming is required.

For more complicated scripts the reader should familiarize himself with the basics of the Python programming language. Many books are available. TOPSPIN comes equipped with the so-called *Jython* variant of Python. Python applications to be run from TOPSPIN must follow the Jython specifications. These are decribed in detail e.g. in the book *Jython Essentials* by *S. Pedroni* and *N. Rappin* (O'Reilly). *Jython* is the Java implementation of Python, and is therefore particularly well suited for the use with TOPSPIN. On the WEB, *Jython* is represented by the site *www.jython.org*.

Python programs (modules, scripts) written for TOPSPIN are capable of

- executing TOPSPIN commands, e.g. em, ft, apk, by invoking the functions EM(), FT(), APK()
- opening dialog windows for user input
- opening NMR data sets for further processing, e.g. with RE()
- fetching and setting NMR parameters using GETPAR(), PUTPAR()
- accessing NMR data in form of arrays for further manipulation by the Python program
- displaying arrays of data calculated in a Python program
- using the Java *swing* classes which provide a rich set of functions with virtually no limit in user interface and graphics programming
- and much more.

1.2 Accessing Python Modules From TOPSPIN

A Python module (program, script) can be invoked from TOPSPIN

- by entering the TOPSPIN command ***xpy*** in the command line or from ***File →Execute…***

- by adding a new *command* to the TOPSPIN command table. The command definition would include the name and location of the Python module. Entering the command in TOPSPIN's command line (or invoking it from a menu or a tool button) would execute the module.

1.3 Jython – The Python Interpreter

Python modules can only be executed in TOPSPIN if the *Jython* Python interpreter is running. It is started automatically the very first time a Python module is executed. This may cause a short start up delay. From now on, the interpreter remains until TOPSPIN is terminated. However, should you execute a Python script while another one is running, a new interpreter is started: Executing Python scripts in parallel is supported, each one being run by its own interpreter.

1.4 The commands *xpy* and *edpy*

The command ***xpy /x/y/MyModule*** executes the Python module ***MyModule.py*** stored in the directory ***/x/y/***. The command ***xpy*** without an argument prompts for the module to be executed.
xpy MyModule will fetch the module from the Python sub-directory of the experiment directory, e.g.

```
<TOPSPIN installation dir>/exp/stan/nmr/py
```

You may edit a Python script using any convenient text editor. TOPSPIN provides the command ***edpy.*** It allows one to create or modify Python scripts stored in the sub-directory listed above. Python scripts consider text starting with a # character as a comment. ***edpy*** has an option that lists the contents of the py directory together with the first 5 comment lines of each script.

1.5 Adding a New (User-Defined) Command to TOPSPIN

TOPSPIN looks for user-defined commands in the file `cmdtab_user.prop`. This file is located in the *TOPSPIN properties directory* of a user. On a Windows system, for the user 'guest', this directory is for example

```
C:\Documents and Settings\guest\.topspin-EOS\prop
```

It is the subdirectory `.topspin-<cname>\prop` of the home directory of the current user, where `<cname>` is the computer's network name.

When TOPSPIN is running, you can find out the directory valid for your system by entering the command ***hist***. The TOPSPIN history file will be opened, and the third entry is the respective properties directory.

Now create the command file `cmdtab_user.prop` using a text editor. Each line in this file represents a user-defined command.

Example:

```
guest_cmd1=EM=PYMOD /a1/b1/Example1
guest_cmd2=EM=PYMOD /a1/b1/Example2
```

This command file example contains two commands, `guest_cmd1` and `guest_cmd2`. At start up time, TOPSPIN reads `cmdtab_user.prop`, and therefore knows about the commands stored in the file. Now you can enter e.g `guest_cmd1` into TOPSPIN's command line. This would trigger the execution of the Python module `Example1.py`. Note that the '.py' extension is appended automatically. The module file `Example1.py` is assumed to be available in the directory `/a1/b1`.

The command file `cmdtab_user.prop` is completely under your own administration. TOPSPIN will never delete or modify it, but just read it during its start up procedure.

When you add your own commands to TOPSPIN, it is highly recommeded to supply them with a common prefix such as `guest_cmd`. Should you choose a command name which is an existing TOPSPIN command, the latter would be overridden by yours. Command names must be defined in lower case characters and may contain digits and underscores. An underscore and a dot at the beginning is illegal.

The string `EM=PYMOD` appearing in a command definition indicates to TOPSPIN that is has to execute a Python module using the interpreter launched during TOPSPIN startup.

1.6 Simple Python Examples

A detailed discussion of the functions used by the examples is presented in the next chapters.

Example 1:

```
"""
  Process the data set visible in the currently selected TOPSPIN window
  with the command ef, then apk.
"""
from TopCmds import *

EF()
APK()
```

Example 2:

```
"""
Open a simple input dialog and print the user input
"""
import de.bruker.nmr.prsc.toplib as top
input = top.Dia.simpleInputDialog("An input dialog",
   "Please enter something:", "XYZ =", "initialValue", 30)
top.Dia.info(input,"Result of input", "")
```

Chapter 2

Executing TOPSPIN commands

2.1.1 The XCMD() function

This chapter explains how you can execute TOPSPIN commands in a Python program. The most general way of doing this is using the function XCMD described here. For the normal work, however, it is much easier to use the predefined Python functions. Most TOPSPIN commands have a Python equivalent with the same name, but with capital letters. For example, the command 'em' has associated a Python function EM(), 'ft' corresponds to EM(), etc. These function are discussed in the next chapter.

In general, you may invoke any TOPSPIN command as outlined in the following example.

```
from TopCmds import *

"""
      Implements the TOPSPIN command 'pycmd4'.
      Executes ft on the current data set.
Thereafter, executes apk.
"""

XCMD("ft")
XCMD("apk")
```

A further example of using `XCMD` is:

```
XCMD(".ph")
```

When a spectrum is being displayed, it would enter interactive phase correction mode (.ph). This is equivalent to pressing the respective tool button, or to typing .ph.

2.1.2 The XCPR() function

This function should only be used by expert who know what they are doing.

Chapter 3

Predefined Functions Related To TOPSPIN commands

3.1 Introduction

Most TOPSPIN commands have a Python equivalent with the same name, but with capital letters. For example, the command ‘em’ has associated a Python function EM(), ‘em’ corresponds to EM(), etc. Sometimes, arguments are accepted to be specified within the parentheses. All currently available functions are defined in the file `TopCmds.py`, located in the directory

```
<TOPSPIN installation dir>/classes/lib/topspin_py/py/pycmd
```

Here you may check which TOPSPIN commands are available as Python functions. The most important ones are discussed in the following sections. In order to get access to all functions defined in the `TopCmds` module, you must import it by using the statement `from TopCmds import *`. This line must be specified at the beginning of each Python script.

3.2 Opening And Processing Data Sets

This section discusses how to open and process data sets from a Python script.

3.2.1 The RE() function

The RE function is used to open and optionally display a data set. RE makes the respective data set the *current* data set of the script on which subsequent processing functions will operate. RE is provided in two variants: RE() and RE(name, expno, procno, dir, user, show). RE() opens a dialog where the user may enter the desired data set. All functions following RE() will then use this data set until another function is encountered (e.g. another RE(), RSER(), ...) which changes the current data set. The second form of RE allows one to specify a data set explicitly in form of a Python list. When omitting the optional argument `show`, the data sets will be displayed. If `show = "n"` is specified, RE will open a data set, but the specified data set will not be displayed. The following example show the usage of the various forms of RE.

```
from TopCmds import *

"""
      Implements the TOPSPIN command 'pyre1'
      Shows various ways how to open data sets and process them
"""

# Executes em;ft;apk on the data set in the selected window
# because no RE() was specified so far
EF()
APK()
```

```
NEWWIN() # open new window
ARRANGE() # arrange windows on screen
RE() # open data entered by user in new window
EF() # process data
APK()

# define test data set
testdata = ["exam1d_13C", "1", "1", "/bruker/topspin", "guest"]

NEWWIN() # open new window
ARRANGE() # arrange windows on screen
RE(testdata) # open testdata in new window
EF() # process test data
APK()

# Open and process data in "background":
# Data do not get displayed
RE(show = "n")
EF()
APK()

# Note that the keyword 'show' is not required here
# (although it may be used)
RE(testdata, "n") # open testdata
EF()
APK()
```

3.2.2 The <u>WR()</u> function

The WR function (“write”) is used to copy the current data set to a new destination. Invoke with: WR(destination) or WR(destination, “n”). The latter command does not ask for confirmation to override destination if existing.

```
"""
from TopCmds import *

"""
      Implements the TOPSPIN command 'pywr1'
      Demonstrates the data set copy command WR.
"""


# define test data set
testdata = ["phspot-10", "10", "1", "c:/nmrdata", "blood"]

RE(testdata) # open testdata in new window

# defined destination
destination = ["phspot", "57", "1", "c:/nmrdata", "bg"]

# copy testdata to destination. "n" means do not display confirm
dialog
# if data exist.
WR(destination, "n")

NEWWIN()
RE(destination) # open destination data in new window
ARRANGE() # make both windows visible
```

3.2.3 The RSER() function

The RSER function extracts one-dimensional fids from multidimensional raw data (*ser* file) and stores them as 1D raw data sets. RSER is provided in two variants: RSER() and RE(fidnum, expno, show). RSER() opens a dialog where the user may enter the fid number to extract, and the destination EXPNO. All functions following RSER() will then use this data set until a function is encountered (e.g. another RSER(), RE(), ...) which changes the data set. The second form of RSER allows one to specify the fid number and the destination explicitly. For Python scripts to operate in background, RSER provides the special argument show=”n”. If specified, RSER will extract a data set, which will not be made visible on screen. The following example show the usage of the various forms of RSER.

```
from TopCmds import *

"""
      Implements the TOPSPIN command 'pyre2'
      Shows various examples how to extract fids from a
      multidimensional raw data set and to process them
"""

# define test data set
testdata = ["exam2d_HC", "1", "1", "c:/Bruker/TOPSPIN", "guest"]

RE(testdata) # open test data set
RSER("25", "999")       # read fid 25 and store it in EXPNO=999
EF()                    # em;ft
APK()                   # apk. Result is in EXPNO=999, PROCNO=1

# Now the 1D result is the current data set
# For further RSERs, we must make the 2D data set again the current
one:
RE(testdata)
RSER()# read fid x and store it in EXPNO=y. x, y from dialog!
EF()  # em;ft
APK() # apk. Result is in EXPNO=y, PROCNO=1 (as original data)


# Read and process some more fids. Store them in subsequent EXPNOs
#
fidnum = 36 # start with this fid number in the raw data file
nfids = 20  # extract this many fids
expno = 800 # store them in EXPNOs starting with this one

for i in range(nfids):  # perform nfids loops

  # Make the 2D data set from which the fids are to be read the
current one
  # but don't display it
  RE(testdata, "n")

  # Read next fid, store in next expno
  # Numbers must be converted to strings
  # Don't display result. If "n" is omitted, each fid is displayed in
its
  #  own window!! With CLOSEWIN(CURDATA()) the current window can be
closed
  #  and only 1 window would be visible while the loop executes
  RSER(str(fidnum + i), str(expno + i), "n")

  EF() # em;ft
  APK()# apk. Results are in EXPNO=800, 801, ...
```

3.2.4 The RSR(), RSC() functions

The RSR/RSC functions extract a row/column from a 2D processed data set and stores it as 1D processed data set. RSR/RSC are provided in two variants: RSR/RSC() and RSR/RSC(num, procno, show). RSR/RSC () open a dialog where the user may enter the row/column number to extract, and the destination PROCNO. All functions following RSR/RSC () will then use this data set until a function is encountered (e.g. RSR(), RSC(), RSER(), RE(), ...) which changes the data set. The second form of RSR/RSC allows one to specify the slice number and the destination explicitly. For Python scripts to operate in background, RSR/RSC provide the special argument show="n". If specified, RSR/RSC will extract a slice, which will not be made visible on screen. The following example show the usage RSR/RSC.

```
from TopCmds import *

"""
      Implements the TOPSPIN command 'pyre3'
      Shows how to extract columns from a
      2D data set and to process them
"""
# define test data set
testdata = ["exam2d_HC", "1", "1", "c:/Bruker/TOPSPIN", "guest"]


# Read and process some columns. Store them in subsequent PROCNOs
#
col = 36 # start with this col number in the 2D data file
ncols = 4   # extract this many columns
procno = 800      # store them in PROCNOs starting with this one

for i in range(ncols):  # perform ncols loops
      CLOSEWIN(CURDATA())     # only 1 col window open at any time

      # Make the 2D data set from which the cols are to be read the
current one
      # and don't display it
      RE(testdata, "n")

      # Read next col, store in next procno
      # Numbers must be converted to strings
      # Display result, because the "n" argument is omitted.
      RSC(str(col + i), str(procno + i))
      ABS() # abs. Results are in PROCNO=800, 801, ...)
```

The CURDATA() function

This function returns the current data set in a Python script. Consider the following code:

```
      testdata = ["exam2d_HC", "1", "1", "c:/Bruker/TOPSPIN",
"guest"]
      RE(testdata)
      RSER("25", "997")
      curdata = CURDATA()
      for items in curdata:
            print item
      FT()
```

The first line opens defines a 2D data set, the second line opens it. The current data set is now the 2D data set. The third line extracts fid=25 and stores it in expno=997. Now the current data set is this 1D data set. Line 4 returns the current data set in a Python list, which is now:

```
      ["exam2d_HC", "997", "1", "c:/Bruker/TOPSPIN", "guest"]
```

Line 5 prints each item in the list:

```
      exam2d_HC
      997
      1
      c:/Bruker/TOPSPIN
      guest
```

Finally, line 5 performs a Fourier Transform of the extracted fid=25.

3.3 Reading Data Points

This section discusses how to read in the points of a data set into a Python list for further manipulation.

3.3.1 The GETPROCDATA() function

GETPROCDATA() reads processed data into a Python list. The result is a list containing values of Python type *float*. Processed data in TOPSPIN are those contained in the processed data files 1r, 1i, 2rr, 2ri, ..., 3rrr, .. , depending on the dimension of the data set. There are two forms of GETPROCDATA:

`GETPROCDATA(from, to)` and `GETPROCDATA(from, to, dataType)`

`from, to` are the ppm limits. The first form returns all data points of the *real processed data* in the limits. The second form allows one to specify which type of imaginary data to return:

dataType = dataconst.PROCDATA_IMAG would return data points from 1i or 2ii or 3iii, depending on the dimension of the data set.

If no data exist, the Python null object `None` is returned.

The following example illustrates the usage of GETPROCDATA():

```
from TopCmds import *
import math

"""
      Implements the TOPSPIN command 'pydat3'
      Changes the phase of a spectrum region by a given angle
      and outputs the result in a dialog box
"""

region = [-0.5, 0.5] # define region in ppm

# read real and imaginary points of the region
reals = GETPROCDATA(region[0], region[1])
imags = GETPROCDATA(region[0], region[1],
      dataconst.PROCDATA_IMAG)
if reals == None or imags == None: EXIT()

# change the phase of the region and output result
N = len(reals)
phi = math.pi/2.0 # define phase angle
result = "Number of points = " + str(N) + "\n"
result += "N  oldReals  newReals\n"
for i in range(N):
      newreal = reals[i] * math.cos(phi) - imags[i] * math.sin(phi)
      result += str(i) + " " + str(newreal) + " " + str(reals[i]) +
"\n"

top.Dia.textViewer("pydat3", "Phased region [ppm]: " +
      str(region), result, 1, 1, 20, 40)
```

3.4 Reading And Modifying Data Set Parameter

A NMR data sets includes a set of parameters used to describe the acquisition and processing state of the data (the *status* parameters), and to prepare the settings for subsequent acquisition and processing functions (the *setup* parameters). The functions GETPAR() and PUTPAR() retrieve and store setup parameters, while GETPARSTAT() and PUTPARSTAT() are provided for accessing status parmeters.

3.4.1 The GETPAR() functions

The following example displays the usage of GETPAR() for a 1D data set. It opens the data set, fetches both, its setup and status parameters SI, and multiplies the setup parameter SI by 2. The result is shown in a dialog box.

Note: GETPAR() delivers a parameter value in String format. In order to perform a calculation, the String must be converted to a respective number by using the Python `int()` or `float()` functions, depending on the parameter type.

```
from TopCmds import *

"""
      Implements the TOPSPIN command 'pypar1'
      Shows how to read the parameters of a data set
"""
# define test data set
testdata = ["exam1d_13C", "1", "1", "c:/Bruker/TOPSPIN", "guest"]
RE(testdata) # open testdata
si = GETPAR("SI") # get SI as a String
isi = 2 * int(si) # convert to int, mult. by 2
sistat = GETPARSTAT("SI") # get Status param. SI

# print result
top.Dia.info("SI = " + si +
      "\n2*SI = " + str(isi) +
      "\nSI (Status par.) = " + str(sistat),
      "GETPAR Test", "ExamPar1.py")
```

3.4.2 The PUTPAR() functions

The following example displays the usage of PUTPAR() for a 1D data set. It opens the data set, fetches its setup parameter LB, multiplies LB by 2, and stores it back into the data set. The result is shown in a dialog box.

Note: PUTPAR() requires a parameter value in String format. PUTPAR() requires a parameter value in String format. After performing a calculation, the number must be converted to a respective String by using the `str()` functions.

```
from TopCmds import *

"""
      Implements the TOPSPIN command 'pypar2'
      Shows how to change the parameters of a data set
"""
# define test data set
testdata = ["exam1d_13C", "1", "1", "c:/Bruker/TOPSPIN", "guest"]
RE(testdata) # open testdata
oldlb = float(GETPAR("LB")) # get current LB
newlb = 2 * oldlb # mult. by 2
```

```
PUTPAR("LB", str(newlb)) # save new LB (as String!)


# verify and print result
lb = GETPAR("LB") # get current LB

top.Dia.info("Old LB = " + str(oldlb) +
      "\nNew LB = " + lb,
      "PUTPAR Test", "ExamPar2.py")
```

3.4.3 GETPAR(), PUTPAR(), and Multi-Dimensional Data Sets

When dealing with NMR data sets having a dimension bigger than 1, GETPAR() and PUTPAR() can be invoked with an additional argument defining the axis F1, F2, F3, ... of the data set whose parameter is to be fetched or stored in form of an integer 1, 2, 3. The following example displays the usage of GETPAR() and PUTPAR() for a 2D data set. It opens the data set, fetches its setup parameter TD of the axis F1, multiplies it by 2, and stores is back as the new value for TD(F1). The result is shown in a dialog box.

Note: If you omit the axis argument when using GETPAR() and PUTPAR() with multi-dimesional data sets, the program automatically inserts “1”.

```
from TopCmds import *

"""
      Implements the TOPSPIN command 'pypar3'
      Shows how to read/change the parameters of a 2D data set
"""
# define test data set
testdata = ["exam2d_HC", "1", "1", "c:/Bruker/TOPSPIN", "guest"]
RE(testdata) # open testdata
oldtd1 = int(GETPAR("TD", 1)) # get current TD(F1)
print oldtd1
newtd1 = 2 * oldtd1 # mult. by 2
print newtd1
PUTPAR("TD", str(newtd1), 1) # save new TD(F1)


# verify and print result
td1 = GETPAR("TD", 1) # get current TD(F1)
print td1
top.Dia.info("Old TD(F1) = " + str(oldtd1) +
      "\nNew TD(F1) = " + td1,
      "GETPAR/PUTPAR 2D Test", "ExamPar3.py")
```

3.4.4 GETPEAKSFILE()

The function of GETPEAKSFILE() returns the contents of the file *peak.txt* of the current data as a String. This file is created by the TOPSPIN peak picking command, and by the PP() function. The function returns “None” if peak.txt doesn not exist.

```
from TopCmds import *

"""
      Implements the TOPSPIN command 'pypeak1'
      Reads peaks.txt and displays it.
"""
peaksfile = GETPEAKSFILE()
top.Dia.textViewer("Peak List", "Demonstrates usage of
      GETPEAKSFILE()", peaksfile, 1, 1, 30, 80)
```

3.4.5 GETPEAKSARRAY()

The function of GETPEAKSARRAY() returns the contents of the file *peak.txt* of the current data as an array of peak object. Peak objects contain information such as peak position and peak intensity as float numbers. Note that the peak position is an array of floats, whose size is the dimension of the data set. This functions is better suited to access peak information than GETPEAKSFILE(). The function returns "None" if peak.txt does not exist.

```
from TopCmds import *

"""
      Implements the TOPSPIN command 'pypeak2'
      Reads peaks.txt into and array and displays the array contents.
"""
peaklist = GETPEAKSARRAY()
if peaklist == None:
      ERRMSG("Peak list is empty.\nPlease perform peak picking
first.")
else:
      output = "#    position[ppm]    intensity\n\n"
      peaknum = 0
      for peak in peaklist:
            peaknum += 1
            position = peak.position
            intensity = peak.intensity

            output += str(peaknum) + "   "
            for pos in position:
                  output += str(pos) + "           "
            output += str(intensity) + "\n"


      top.Dia.textViewer("Peak List", "Demonstrates usage of
            GETPEAKSARRAY()", output, 1, 1, 30, 80)
```

3.4.6 GETPROCDIM(), GETACQUDIM()

These functions return the the dimensions of the processed data and acquisition (raw) data of the current data set, respectively. The returned values are Integers 1, 2, 3, …, or -1 if an error occurs.

3.5 Dialogs

The purpose of the functions in this section is to display dialog windows of various types suitable for user input.

3.5.1 INPUT_DIALOG()

This is a multi-line input dialog, consisting of a number of input fields with identifier labels displayed in front of the input fields.

Syntax:

```
Result = INPUT_DIALOG(title=<title>, header=<header>, items=<items>,
                  values=<values>, comments =<comments>, types=<types>)
```

where:

title = the dialog window title, e.g. “My Special Title”

header = an additional multiline header, lines to be separated by “\n”, no header if None

items = the list of identifier label names (no labels if None)

values = the list of initial values for the input fields (empty if None)

types = the list of input field sizes (1=single line, >1 multiple line field)

comments = the list of comments appended to the input fields (empty if None), may have several lines to be separated by “\n”

Result = the list of values in the text fields, if the user pressed OK, or None if the user pressed ESC, or Cancel.

Examples:

```
Result = INPUT_DIALOG()
Result = INPUT_DIALOG(items=["EXPERIMENT =", "SOLVENT ="],
    values=["cosy\ncosydqf", "Aceton"])
Result = INPUT_DIALOG("MyTitle" , "MyHeaderLine1\nLine2",
    ["EXPERIMENT =", "SOLVENT ="],
    ["cosy\ncosydqf", "Aceton"],
    ["2", "1"]),
    ["Cosy comment", "Cosydqf comment "],
```

3.5.2 DATASET_DIALOG()

This is a dialog where a user can define an NMR data sets, or to browse for one, or to search for one. The user defined data set is returned.

Syntax:

```
Result = DATASET_DIALOG(title=<title>, values=<values>)
```

where:

title = the dialog window title, e.g. “My Special Title”

values = initial data set specifcations displayed in the dialog

Result = the data set specification of the entered data, if the user pressed OK, or None if the user pressed ESC, or Cancel, e.g. `result = ["exam1d_13C", "1", "1", "c:/bruker/topspin", "guest"]`

Examples:

```
Result = DATASET_DIALOG()
Result = DATASET_DIALOG("MyTitle", CURDATA())
Result = DATASET_DIALOG("", ["exam1d_13C", "1", "1",
           "c:/bruker/topspin", "guest"])

Result = INPUT
```

3.5.3 FIND_DIALOG()

This is a dialog which allows on to search for NMR data sets according to certain criteria, display a list of the found data sets, and perform actions when clicking on the buttons of the list. The action buttons are the same as for the TopSpin “find” command if no options are specified, and the return value is null. If options are specified, only an OK and a Cancel button are present, and the list of selected data sets are returned in form of a list of topspin data paths.

Syntax:

```
Result = FIND_DIALOG(options)
```

where:

options = not specified or “get_selected” or “get_selected mult_selection”

Result = the list of selected data sets in form of a list of path names. If “mult_selection” is specified, multiple data sets may be selected in the list of found data sets.

Examples:

```
Result = FIND _DIALOG()
Result = FIND _DIALOG("get_selected")
Result = FIND _DIALOG("get_selected mult_selection")
```

3.5.4 MSG(), ERRMSG()

`MSG(message)` and `ERRMSG(message)` both display the specified message in a dialog box. The difference is that the `MSG` dialog includes an information icon, while the `ERRMSG` dialog includes an erro icon. Please note that more sophisticated printing is provided by the toplib package described later in this manual.

3.5.5 CONFIRM()

Displays a confirmation dialog with an OK and a Cancel button.

Syntax:

```
Result = CONFIRM(title=<title>, message=<message>)
```

where:

title = the dialog window title

message = the message printed in the dialog (\n separates lines)

Result = 1 if the user pressed the OK button, otherwise 0

3.5.6 SHOW_STATUS()

`SHOW_STATUS(message)` displays the specified message in the bottom status line of the TOPSPIN main window.

3.6 Window Handling Functions

The purpose of the functions in this section is to manipulate internal TOPSPIN window containing data sets or other NMR objects.

3.6.1 NEWWIN()

`NEWWIN()` creates a new empty internal window. A subsequent RE function will place the data set into this window.

3.7 Window Handling Functions

The purpose of the functions in this section is to manipulate internal TOPSPIN window containing data sets or other NMR objects.

3.7.1 NEWWIN()

`NEWWIN()` creates a new empty internal window. A subsequent RE function will place the data set into this window.

3.7.2 CLOSEWIN()

`CLOSEWIN(dataset)` closes the internal window containing the specified data set. It has no effect if no such window is open. The data set must be specified by a Python *list*. The list contents must have the order [name, expno, procno, dir, user]. All list entries must be Strings. The function `CLOSEWIN(CURDAT())` would close the window containing the current data set of the currently active script.
`CLOSEWIN()`closes all windows on the TOPSPIN desktop, regardless of their contents

3.7.3 ARRANGE()

`ARRANGE(mode)` arranges the currently open internal TOPSPIN windows according to `mode`. If `mode` is omitted (or `mode="$h"`), TOPSPIN uses a default layout. If `mode="$v"`, the windows are arranged vertically. If `mode="$h"`, the windows are arranged horizontally. If `mode` does not start with a Dollar sign it is interpreted as a file name containing layouts of windows of TOPSPIN's desktop, e.g. created from the menu item *Window -> Save Layout...* . If the specified file is not an absolute path name, the file is fetched from the user's properties directory where is store with *Windows -> Save Layout...*. The function `ARRANGE(mode)`, where "mode" is a filename may e.g. be used to write a script which arranges real time fid window, lock window, and gs window according to a previously defined layout stored with *Window -> Save Layout...*.

3.7.4 POSITION_WIN()

`POSITION_WIN(file)` sets the position and size of a window according to the specified file (which must be stored with the extension .prop, although the .prop may be omitted in the function call). In order to generate the file, proceed as follows: Open a window (e.g. and empty window, or a data set) in TOPSPIN, set its position and size as desired, and save it using the *Window→Save Layout...* command. Make sure that only one window is open in TOPSPIN while saving!

The file argument of this function may be an absolute path name, or just a file name. In the latter case the file is looked up in the subdirectory MDILayouts of the user's properties directory.

Example:

```
from TopCmds import *
"""
      Implements the TOPSPIN command 'pypos1'
      1. Sets the position and size of the
        currently open data window according to
        the file 'my_win_1'.
      2. Opens a new window and
        sets its position and size according to
        the file 'my_win_2'.
      3. Re-load the same data set into the new
        window and selects its processing parameter
        tab.
      Result: 2 open windows with the same data set,
        one showing the spectrum, the other one
        the processing parameters.
"""
POSITION_WIN("my_win_1")
NEWWIN()
POSITION_WIN("my_win_2")
RE(CURDATA())
SELECT_TAB(mfw.DatasetPane.PROCPARS)
```

3.7.5 POSITION_WINTYPE()

`POSITION_WINTYPE(windowtype, file)` sets the position and size of a window, whose contents is of the specified type, according to the specified file (which must be stored with the extension .prop, although the .prop may be omitted in the function call). In order to generate the file, proceed as follows: Open a window (e.g. and empty window, or a data set) in TOPSPIN, set its position and size as desired, and save it using the *Window→Save Layout....* command. Make sure that only one window is open in TOPSPIN while saving!

The file argument of this function may be an absolute path name, or just a file name. In the latter case the file is looked up in the subdirectory MDILayouts of the user's properties directory.

Legal window types are:

```
      "LOCK_SIGNAL_WINDOW"
      "REALTIME_FID_WINDOW"
      "GS_CONTROL_WINDOW"
      "VTU_WINDOW"
      "BSMS_WINDOW"
      "CORTAB_WINDOW"
      "DATA_WINDOW"
```

If no window of the specified type is on the TOPSPIN desktop, the function returns without any action. If the specified window type is `"DATA_WINDOW"`, the first data window (or empty window) found is positioned, other data windows are not touched.

Example:

```
from TopCmds import *
POSITION_WIN("LOCK_SIGNAL_WINDOW", "lock1")
POSITION_WIN("REALTIME_FID_WINDOW", "rfid1")
```

3.7.6 SELECT_TAB()

`SELECT_TAB(tab_specifier)` switches the index card of the currently displayed data set so as to display the specified tab, e.g. the Processing Parameters. `tab_specifier` may take on the following values (see the example of the previous section):

```
Mfw.DatasetPane.SPECTRUM
Mfw.DatasetPane.FID
Mfw.DatasetPane.PROCPARS
Mfw.DatasetPane.ACQUPARS
Mfw.DatasetPane.TITLE
Mfw.DatasetPane.PULSPROG
Mfw.DatasetPane.PEAKLIST
Mfw.DatasetPane.INTEGRALS
Mfw.DatasetPane.MOLECULE
```

3.8 Flow Control Functions

3.8.1 The EXIT() function

Normally a Python script ends after having executed all statements defined by the program logic. There is no need to place an `EXIT()` at the end of a script. If the script should terminate e.g. due to an error condition detected by the program itself, the `EXIT()` function my be used, although this is usually not necessary since the same effect can be reached by employing suitable `if...else` clauses (cf. the sophisticated example in the section *Processing Functions*).

3.9 Processing Functions EM(), FT(), ….

Most of the TOPSPIN processing functions and a number of other functions accept arguments. You may find out about the arguments available for a particular function by inspecting `TopCmds.py`. We will discuss here the most important arguments.

3.9.1 A simple example

```
from TopCmds import *

"""
      Implements the TOPSPIN command 'pycmd1'
      Executes a sequence of TOPSPIN processing functions on
the currently active data set
"""

PUTPAR("SI", "1024")    # set size to 1K
EM()                # perform "em" etc.
FT()
APK()
XCMD("SI")          # request size from dialog
EM()                # perform "em"
```

This example executes the commands `si 1024, em, ft, apk, si, em` in the specified sequence on the currently active data set. Assume you are displaying two different data sets in two internal TOPSPIN windows and you execute the command `pycmd1`. Then this Python program starts processing the data set in the activated window. If you would replace `EM()` by `em` in this program and you would start it, you would get a *Name Error*. The reason is that Python doesn't know anything about `em`. However, it knows the function `EM()` because it is defined in the Python module `TopCmds`. Beware however of omitting the parentheses. If you replace `EM()` by `EM`, *no error is printed*, but the function is not executed either because Python needs the parentheses to trigger execution:

```
EM() # correct
EM   # wrong, no error printed
em   # wrong, Name Error printed
```

3.9.2 The wait argument

By default, processing functions in a Python script do not return before there execution is complete. In a few cases it could be of advantage to carry on with the next function in the script before the currently executing function has terminated. This behaviour can be enforced by the function argument `wait = NO_WAIT_TILL_DONE`. It's usage is illustrate in the following example.

```
import de.bruker.nmr.prsc.toplib as top
from TopCmds import *

"""
        Implements the TOPSPIN command 'pycmd2'
        Executes APK on the currently displayed 1D spectrum
        without waiting for termination.
        Prints a message while APK is in progress.
"""

APK(NO_WAIT_TILL_DONE)

msg = "This dialog is opened as soon as APK was started."
 top.Dia.msg(msg, "Testing NO_WAIT_TILL_DONE", "")
```

3.9.3 A non-trivial example

The following example intensively makes use of the GETPAR(), PUTPAR(), and XCMD(). It implements a 1D automatic phase correction algorithm. The TOPSPIN command `apk0f` is applied to the two spectral regions defines by the variables `reg1_absf1, reg1_absf2, reg2_absf1, reg2_absf2.` From the resulting phases, the overall phase constants `PHC0` and `PHC1` are derived, and a `pk` is executed.

```
from TopCmds import *

"""
      Python script 'apk2r' performing an automatic phase correction
      of a 1D spectrum by applying apk0 to 2 regions.
"""

reg1 = [9.0, 7.0]
reg2 = [0.5, -0.5]

parmod = int(GETPAR("PPARMOD"))

if parmod != 0:
      MSG("Not a 1D Dataset:\n" + CURDATA())
else:
      si = int(GETPARSTAT("SI"))
      sf = float(GETPARSTAT("SF"))
      swp = float(GETPARSTAT("SW_p"))
      offset = float(GETPARSTAT("OFFSET"))

      save_phc0 = float(GETPARSTAT("PHC0"))
      save_absf = [GETPAR("ABSF1"), GETPAR("ABSF2")]

      PUTPAR("ABSF1", str(reg1[0]))
      PUTPAR("ABSF2", str(reg1[1]))

      if XCMD("apk0f") >= 0:
            reg1_phc0 = float(GETPARSTAT("PHC0"))
            PUTPAR("PHC0", str(save_phc0 - reg1_phc0))
            PUTPAR("PHC1", "0")
```

```
PK()

PUTPAR("ABSF1", str(reg2[0]))
PUTPAR("ABSF2", str(reg2[1]))
XCMD("apk0f")
reg2_phc0 = float(GETPARSTAT("PHC0"))

phc = (reg2_phc0 - reg1_phc0) /\
 ((reg1[0] + reg1[1] - reg2[0] - reg2[1])\
                   * sf / (2 * swp))
PUTPAR("PHC1", str(phc))

phc *= (2 * offset - reg1[0] - reg1[1]) * sf /\
 (2 * swp)
PUTPAR("PHC0", str(reg1_phc0 - reg2_phc0 - phc))

XCMD("pk fgphup")

PUTPAR("ABSF1", save_absf[0])
PUTPAR("ABSF2", save_absf[1])
```

3.10 Analysis Functions

3.10.1 Peak Picking (PP)

The `PP()` function can be used for 1D, 2D, and 3D data sets. Here are some examples how to use `PP()`:

Example 1:

```
from TopCmds import *
# Define a 2D data set
testdata = ["exam2d_HC", "1", "1", "c:/Bruker/TOPSPIN", "guest"]

RE(testdata, "n") # Open data set, no display
PP() # Peak pick; uses processing parameters as defined by data set
```

`PP()` can also be invoked as `PP(append = "append")`. In this case the peak list is added to a potentially existing list, rather than replacing it. The peak list is stored in the file `peak.txt` of the processed data set.

Example 2:

```
from TopCmds import *
# Define a 2D data set
testdata = ["exam2d_HC", "1", "1", "c:/Bruker/TOPSPIN", "guest"]

RE(testdata) # Open data set and display it
PP(dialog="y") # Open dialog window to edit parameters
```

In this case the processing parameters used by `PP()` are those specified in the parameter dialog. If the `PP()` was omitted in this example, pick picking would have been performed on the currently selected data set on the screen, if there was one.

3.11 Displaying Data Arrays

The predefined functions described in this section allow you to display arrays of data points in a TOPSPIN internal window using various geometries. Many tool buttons of the TOPSPIN main toolbars and the mouse are active supporting actions such as vertical and horizontal expansion of the data, displaying grids, moving the data arrays up and down within the window. Also, printing the window contents is enabled via CTRL/P.

3.11.1 The GET_DISPLAY_PROPS() Function

This functions returns a list of Strings describing display properties. You must call this function if you want to define your own x axis coordinate range, x axis unit, and more when displaying data. The list generated by this functions may be passed as an argument to the DISPLAY_DATALIST functions to enforce the respective drawing behavior:

```
 GET_DISPLAY_PROPS(xStart = None, xEnd = None, xUnit = None, 
xLegend=None, 
                  yLegend=None, dataTitle=None, drawMode="line")
```

`xStart` – left x axis limit, e.g. 0.0
`xEnd` – right x axis limit, e.g. 10.5
`xUnit` – x axis unit, e.g. `"ppm"`
`xLegend` – extra text displayed below x axis
`yLegend` – extra text displayed left of y axis
`dataTitle` – extra text displayed in the upper left of the data window
`drawMode` – `"dots"` or `"line"`. In the first case the data points are displayed in form of small
circles, without connecting lines. In the second case the data points are drawn in form of a polyline.

3.11.2 The DISPLAY_DATALIST() Function

The purpose of this function is to display several data arrays whose y values are given, while the x axis values are not given explicitly, but are assumed to be equidistant.
Assume, in your Python script you have generated N Python lists each containing a number y floating point values. Then, DISPLAY_DATALIST() will display the lists either in a single (x,y) coordinate system *(single view)*, or in N different coordinate (x,y) systems *(multiple view)*. The general form of DISPLAY_DATALIST() is

```
DISPLAY_DATALIST(dataList, propList = None, winTitle = "", multipleView = 0)
```

where
`dataList` = List of 1, 2, ... or N Python lists with floating points numbers
`propList` = List of 1, 2, ... or N Python lists with Strings (*display property list)*, describing the display properties if defined
`winTitle` = The title of the internal window where the data will be displayed (empty if not specified)
`multipleView` = If specified and == 1, the data list are displayed in multiple views.

The structure of a display property is returned when calling GET_DISPLAY_PROPS().
If no display property list is specified, or arguments of GET_DISPLAY_PROPS() are omitted, default values are applied.

The simplest form is `DISPLAY_DATALIST(dataList)`.
In this case the x axis unit is *index*, which counts the number of data points of the floating number array. This form is used in the following example, which illustrates the usage of DISPLAY_DATALIST() by displaying 3 data arrays.

```
from TopCmds import *
import math

"""
      Implements the TOPSPIN command 'pydsp1'
      Reads in a region of the real part of a spectrum,
      and the corresponding region of the imaginary part.
      Calculates the Magnitude spectrum of the region
      and displays reals, imags, and magnitudes in a single view.
      The simplest form of DISPLAY_DATALIST is used.
"""

region = [80, 72] # define region in ppm

# open testdata, don't display
testdata = ["exam1d_13C", "1", "1", "c:/Bruker/TOPSPIN", "guest"]
RE(testdata, "n")

# read real and imaginary points of the region
reals = GETPROCDATA(region[0], region[1])
imags = GETPROCDATA(region[0], region[1],
      dataconst.PROCDATA_IMAG)
if reals == None or imags == None: EXIT()

# calc. magnitude
magn = []
for i in range(len(reals)):
      magn.append(math.sqrt(reals[i]**2 + imags[i]**2))

# set up list of data to be displayed,
# and respective axis info list
dataList = [reals, imags, magn]

# display the data in the list
DISPLAY_DATALIST(dataList)
```

The following example displays the most general use of DISPLAY_DATALIST().

```
from TopCmds import *
import math

"""
      Implements the TOPSPIN command 'pydsp2'
      Calculates a sine, cosine, and magnitude function.
      Displays the three functions in a multiple view.
"""

sinus = []
cosinus = []
magn = []
for phi in range(360):
      phirad = phi * ((2*math.pi)/360)
      s = math.sin(phirad)
      c = math.cos(phirad)
      m = s**2 + c**2
      sinus.append(s)
      cosinus.append(c)
      magn.append(m)


dataList = [sinus, cosinus, magn]
```

```
propSin = GET_DISPLAY_PROPS(0, end, "degrees", "ANGLE",
      "REL. INTENSITY", "SINE")
propCos = GET_DISPLAY_PROPS(0, end, "degrees", "ANGLE",
      "REL. INTENSITY", "COSINE")
propMagn = GET_DISPLAY_PROPS(0, end, "degrees", "ANGLE",
      "REL. INTENSITY", "SQUARED SUM")
propList = [propSin, propCos, propMagn]

DISPLAY_DATALIST(dataList, propList, "Display Test: pydsp2", 1)
```

3.11.3 The DISPLAY_DATALIST_XY() Function

The purpose of this function is to display several data arrays whose (x,y) values are given. The x axis values are not required to be equidistant, in contrast to DISPLAY_DATALIST().
The general form of DISPLAY_DATALIST_XY() is

```
DISPLAY_DATALIST_XY(ydataList, xdataList, axisList = None, winTitle =
"",
    multipleView = 0)
```

This is quite similar to DISPLAY_DATALIST(), the only difference being the additional argument `xdataList`. The following example shows how to use the function.

```
from TopCmds import *

"""
      Implements the TOPSPIN command 'pydsp3'
      Displays 2 data arrays whose x coordinates are not
      equidistant.
"""

xdata = [1, 10, 100, 1000, 10000, 100000, 1000000, 10000000]
ydata1 = [1,2,3,4,5,6,7,8]
ydata2 = [2,4,9,16,25,36,49,64]

ydataList = [ydata1, ydata2]
xdataList = [xdata, xdata]
data1Prop = GET_DISPLAY_PROPS(xUnit = "MyUnit", xLegend = "Legend 1",
      yLegend = "MyYValues1", dataTitle = "XY TEST 1")
data2Prop = GET_DISPLAY_PROPS(xUnit="MyUnit", dataTitle ="XY TEST 2",
      drawMode = "dots")
propList = [data1Prop, data2Prop]


DISPLAY_DATALIST_XY(ydataList, xdataList, propList, "Display Test:
pydsp3", 1)
```

3.12 Finding Data Sets

This section discusses how to locate data sets on storage media according to given specifications.

3.12.1 FIND_DATA()

The general form of the FIND_DATA() function is

```
      FIND_DATA(dir, name="", expno="", procno="", user="", title="",
            pulprog="", dim="", type="")
```

It searches for NMR data in the list of directories `dir` according to the specifications of the other arguments whose default values are empty Strings. It return the list of found NMR data path names. The `dim` argument, if not empty, may take on the following values: `"Any", "1D+2D+3D", "1D+2D", "1D", "2D", "3D"`. The type argument, if not empty , may take on the following values: `"Any", "spectra", "fids"`. You can force the algorithm to return only NAMEs, or EXPNOs, rather than all found PROCNOs of a data set (which is the default).

The following example shows the usage of the function.

```
from TopCmds import *

"""
	Implements the TOPSPIN command 'pyfind2'
	Find all data set in the specified directory and the
	specified user whose names start with ‘phs’.
	Integrate the same spectral region of this series of
	spectra and display result.
"""

filelist = FIND_DATA(["c:/bruker/topspin"], user="guest", name =
    "exam1d_13C")

region = [3.10, 2.84] # define region in ppm

integrals = []
maxima = []
for file in filelist:
	RE_PATH(file, "n")
	regdata = GETPROCDATA(region[0], region[1])

	sum = 0.0
	max = -3.0e+38

	for point in regdata:
		sum += point
		if point > max: max = point

	integrals.append(sum)
	maxima.append(max)

maxProps = GET_DISPLAY_PROPS(dataTitle="Maximum Intensities 1")
integProps = GET_DISPLAY_PROPS(dataTitle="Integrals")

DISPLAY_DATALIST([integrals, maxima], [maxProps, integProps],
	frameTitle = "Correlation of Integrals & Maximum Intensities",
multipleView = 1)
```

3.13 Data Acquisition Functions

The following functions correspond to their respective TOPSPIN commands, and will not return before they are terminated:

```
ZG(), GO(), GS(), WOBB()
```

The following functions correspond to their respective TOPSPIN commands, and will return immediately after they were started:

```
BSMSDISP(), LOCKDISP(), EDTE()
```

Chapter 4

The `toplib` Package

4.1 Introduction

The `toplib` class package is extremely useful when adding your own applications to TOPSPIN. You can access the package in your Python programs by adding the statement

```
      import de.bruker.nmr.prsc.toplib as top
```

to the beginning of each Python module. This is, however, not required if you already specified `from TopCmds import *`, because then toplib is imported implicitly.

The following sections of this chapter describe the classes of the toplib package in general. The detailed description of all the provided methods are contained in html files and can be opened by a Web browser.

Your extensions of TOPSPIN may, of course, make use of all classes provided by the Java system. The respective import statements must be added to your Python code. For example, the 3 lines

```
      import de.bruker.nmr.prsc.toplib as top
      import java.awt as awt
      import javax.swing as sw
```

would make available Bruker `toplib` under the name `top`, Java `awt` under `awt`, and Java `swing` under `sw`.

The `toplib` class package is a JAVA package. Although this manual provides examples how to use `toplib` from a Python program, a JAVA program may employ it in the same way.

4.2 Examples

The source code of the example programs presented in this chapter can be found in the following directory:

```
      <TOPSPIN installation
directory>/classes/lib/de/bruker/nmr/py/pyexam
```

4.3 Dialog Windows

The class `top.Dia` provides methods to display various kinds of dialog windows. Their use is illustrated by the following example. It can be invoked by the TOPSPIN command ‘pydia1’.

```
import de.bruker.nmr.prsc.toplib as top
```

```
"""
      Implements TOPSPIN command 'pydia1'
      Prints some text in a TOPSPIN info message window
"""
top.Dia.info("This is the message", "This is the title", "THESE ARE
THE DETAILS!")
```

4.4 Internal Windows

The class `top.Win` provides methods to create and manipulate TOPSPIN internal windows (= internal frames), just like the windows containing NMR data sets. Typically you would create such a window in a Python, and add your own graphics elements to the window. The following example can be invoked by the TOPSPIN command ***pywin1***. The source code is available in the file `ExamWin1.py`.

```
from TopCmds import *


"""
      Implements TOPSPIN command 'pywin1'
      - Opens a 2 new empty internal TOPSPIN windows
      - sets their title
      - arranges them horizontally on the TOPSPIN desktop
"""

userPanel1 = top.Win.newUserPanel(userPanel1,
             "My Application Number 1")

userPanel2 = top.Win.newUserPanel(userPanel2,
             "My Application Number 2")
ARRANGE("$h")
```

4.5 Internal Windows and Events

In your own application you want to be able to handle the toolbar events. A toolbar event is generated when a user clicks on a tool button. You can catch such events in an internal window containing your application. You can find out which tool button was clicked and proceed accordingly.
You can also find out if an internal window was selected by a user, or deselected. You may perform clean up work in your application when a window gets closed. And you may provide data object to a caller of your application. The following demonstrates how to catch tool button events, and prints the command assigned to a tool button. The example also shows how you can access TOPSPIN global variables, e.g. the dialog font used, via the `top.Glob` class. The following example can be invoked by the TOPSPIN command ***pywin2***. The source code is available in the file `ExamWin2.py`.

```
import de.bruker.nmr.prsc.toplib as top
import javax.swing as sw


"""
      Implements TOPSPIN command 'pywin2'
      - Opens a new window internal to TOPSPIN,
      - adds a label with red text
      - implements the EventIfc methods to catch internal window
events
      - prints the command assigned to a tool button when clicking on
it
```

```
"""

class ApplicationPanel(sw.JPanel, top.EventIfc):

      def __init__(self):
            self.add(sw.JLabel("Click on one of the tool buttons *2 ,
/2, *8, …!",
                  foreground=(255, 0, 0), font =
top.Glob.dialogFont))


      # the following methods implement 'EventIfc' to catch internal
      # window events
      def performAction(self, actionCode, arg, initiator):
            top.Dia.info(actionCode, "", "")

      def getData(self):
            pass

      def redisplay(self):
            pass

      def notifySelected(self):
            pass

      def notifyDeselected(self):
            pass

      def cleanUp(self):
            pass



userPanel1 = top.Win.newUserPanel("Simple Event Application", 0)
userPanel1.add(ApplicationPanel())
userPanel1.invalidate()
userPanel1.validate()
userPanel1.repaint()
```

4.6 Tool Bars

The class `top.Tbar` provides methods to create and manipulate toolbars. The following example creates a new window and adds a toolbar at the top of the window. The toolbar is left adjusted and contains three icons. When you move the mouse to the icons, a tool tip is displayed near the icon, and on the status line. The following example can be invoked by the TOPSPIN command ***pytbar1***. The source code is available in the file `ExamTbar1.py`.
The toolbar description must be given in the following form:

```
PYTHON_EXAMPLE_TOOLBAR=\
      NM=save.gif, TIP=general_save, CMD=_dotool my_save, END=,\
      NM=saveas_16.gif, TIP=general_saveas, CMD=_dotool my_save_as,
END=,\
      NM=-, END=,\
      NM=undo_16.gif, TIP=setti_undo, CMD=my_undo, END=,\
      NM=-, END=
```

`PYTHON_EXAMPLE_TOOLBAR` is the name of the toolbar which is used in the following Python program. 'NM=' defines the name of an icon. 'TIP=' defines the tool tip text. Note that this is not the text itself, but a 'identifier' of a text (see below). 'CMD=' defines the action to be performed when the icon is clicked. 'NM=-' defines a separator between two icons.

TOPSPIN looks for user-defined toolbar definitions in the file `toolbar_user.prop` , and for tool tip text identifiers in the file `English\txt_toolbar_user.prop`. These files are located in the *TOPSPIN properties directory* of a user. On a Windows system, for the user 'guest', this directory is for example

```
            C:\Documents and Settings\guest\.topspin-EOS\prop
```

It is the subdirectory `.topspin-<cname>\prop` of the home directory of the current user, where `<cname>` is the computer's network name.

The format of `txt_toolbar_user.prop` is shown in the following example:

```
general_save=Save
      general_saveas=Save as…
      setti_undo=Undo
```

When TOPSPIN is running, you can find out the directory valid for your system by entering the command ***hist***. The TOPSPIN history file will be opened, and the third entry is the respective properties directory.

The file `toolbar_user.prop` may contain any number of toolbar definitions which required the syntax of the example above. If the language of TOPSPIN is set to a language other than English (command `set`), the file `txt_toolbar_user.prop` must not be located in the `English` subdirectory, but in one defined by the language.

What happens when you click on one of the icons of the toolbar example shown above?

When you click on the icons `save.gif` or `saveas_16.gif`, the following Python program prints `my_save` or `my_save_as`, because the `performAction` method of the Python program is invoked. However, if you click on the `undo_16.gif` icon, TOPSPIN will print `Command not implemented: my_undo`. The reason is that `_dotool` is missing after 'CMD=' in the tool button definition of `undo_16.gif`. In this case TOPSPIN does not call the `performAction` method. Instead, it tries to execute the String behind 'CMD=' as a command, which e.g. is defined in the user command table `cmdtab_user.prop`. Since the command `my_undo` was not found, the error message was printed. There is a major difference between using the `performAction` method or command execution via the command table. In the latter case the invoked code executes in its own thread! This is often a desired feature. Assume the command must execute a time consuming computation. If not executed in its own thread, the entire TOPSPIN user interface would block until computation is complete. Please check carefully in each case whether you catch a command in the `performAction` method and execute code directly there, or whether you implement a new command in the user command table which gets executed when the user clicks on the respective icon. We generally recommend the implementation as a command because it avoids other problems which can lead to user interface blocking.

And here is the Python program.

```
import de.bruker.nmr.prsc.toplib as top
import javax.swing as sw
import java.awt as awt

"""
      Implements TOPSPIN command 'pytbar1'
      - Opens a new window internal to TOPSPIN,
      - adds a left-adjusted toolbar
      - adds a separator line
      - adds a label "please click on …."
      - implements EventIfc methods to catch internal window events
      - prints the command assigned to a tool button when clicking
        on it (tool buttons 1 + 2 only)
```

```
"""

class ApplicationPanel(sw.JPanel, top.EventIfc):
      def __init__(self):
            self.setLayout(awt.BorderLayout())
            self.setBackground(top.Glob.frameBackground)

            northPanel = sw.JPanel(background=self.background)
            northPanel.setLayout(sw.BoxLayout(northPanel,
           sw.BoxLayout.Y_AXIS))



            tbarPanel =
           sw.JPanel(layout=awt.FlowLayout(awt.FlowLayout.LEFT,0,0),
                  background=self.background)
            toolbar = top.Tbar("PYTHON_EXAMPLE_TOOLBAR")
            tbarPanel.add(toolbar)

            northPanel.add(tbarPanel)
            northPanel.add(sw.JSeparator())
            self.add(northPanel, awt.BorderLayout.NORTH)

            self.add(sw.JLabel("Please click on one of the buttons
*2, /2, *8, …!",
            foreground=(255, 0, 0), font = top.Glob.dialogFont))

      # the following methods implement 'EventIfc' to catch internal
      # window events

      def performAction(self, actionCode, arg, initiator):
            top.Dia.info(actionCode, "", "")

      def getData(self):
            pass

      def redisplay(self):
            pass

      def notifySelected(self):
            pass

      def notifyDeselected(self):
            pass

      def cleanUp(self):
            pass

userPanel1 = top.Win.newUserPanel("Simple Application With Toolbar",
0)
userPanel1.add(ApplicationPanel(), awt.BorderLayout.CENTER)
userPanel1.invalidate()
userPanel1.validate()
userPanel1.repaint()
```

4.7 Menu Bars

The class `top.Mbar` provides methods to create and manipulate menu bars. The following example creates a new window and adds a menu bar at the top of the window. The menu bar is left adjusted and contains two pull down menus. The following example can be invoked by the TOPSPIN command ***pymbar1***. The source code is available in the file `ExamMbar1.py`.
The menu bar description must be given in the following form:

```
MENUTEXTS=txt_menubar_exam.prop
1=NM=py_first, END=, \
      NM=py_new, CMD=_dotool my_new, END=, \
      NM=py_open, CMD=_dotool my_open, END=, \
      NM=-, END=, \
      NM=py_close, CMD=close, END=
2=NM=py_second, END=,\
      NM=py_copy, CMD=my_copy, END=,\
      NM=py_paste, CMD=my_paste, END=,\
      NM=-, END=, \
      NM=py_about, CMD=about, END=
```

The numbers '1=' and '2=' define the two pull down menus. They also indicate the sequence of the menus in the menu bar. 'NM=' defines the name of a pull down menu or of a menu item within a menu. Note that this is not the text itself, but a 'identifier' of a text. 'CMD=' defines the action to be performed when the menu item is clicked. 'NM=-' defines a separator between two menu items.

TOPSPIN looks for menu bar definitions in a file whose name must be passed as an argument to the class constructor which creates the menu bar. In the example below, the menu bar above is stored in the file `menubar_exam.prop`, and the menu text identifiers in the file `English\ txt_menubar_exam.prop`. These files are located in the *TOPSPIN properties directory* of a user. On a Windows system, for the user 'guest', this directory is for example

```
C:\Documents and Settings\guest\.topspin-EOS\prop
```

It is the subdirectory `.topspin-<cname>\prop` of the home directory of the current user, where `<cname>` is the computer's network name.

The format of `txt_menubar_exam.prop` is shown in the following example:

```
py_first=Menu1
py_new=New… `n
py_open=Open… `o
py_close=Close `c
py_second=Menu2
py_copy=Copy `c
py_paste=Paste `p
py_about=Info… `i
```

The characters behind the back apostrophe indicates which character of the text should be underlined in the menu. This is at the same time the short cut to invoke the command from the keyboard when the menu is open.

When TOPSPIN is running, you can find out the directory valid for your system by entering the command ***hist***. The TOPSPIN history file will be opened, and the third entry is the respective properties directory.

A menu bar definition file (in our example `menubar_exam.prop)` may contain just one menu bar definition which required the syntax of the example above. This means, should your applications required several menu bars, you will have to set up the respective numbers of menu bar definition files. You may, however, use the same text identifier file for all menu bars just by specifying its name at the top (in our example: `MENUTEXTS=txt_menubar_exam.prop`). If the language of TOPSPIN is set to a language other than English (command `set`), the file `txt_menubar_exam.prop` must not be located in the `English` subdirectory, but in one defined by the language.

What happens when you click on one of the menu items of the pull down menu examples shown above?

When you click on the first or second menu items of the first menu (defined by `py_new` and `py_open`), the following Python program prints `my_new` or `my_open`, because the `performAction` method of the Python program is invoked. However, if you click on the py_copy or py_paste menu items of the second menu, TOPSPIN will print Command not implemented: my_copy (or my_paste). The reason is that _dotool is missing after 'CMD=' in the menu item definition of py_copy and py_paste. In this case TOPSPIN does not call the performAction method. Instead, it tries to execute the String behind 'CMD=' as a command, which e.g. is defined in the user command table cmdtab_user.prop. Since the commands my_copy and my_paste were not found, the error message was printed. There is a major difference between using the performAction method or command execution via the command table. In the latter case the invoked code executes in its own thread! This is often a desired feature. Assume the command must execute a time consuming computation. If not executed in its own thread, the entire TOPSPIN user interface would block until computation is complete. Please check carefully in each case whether you catch a command in the performAction method and execute code directly there, or whether you implement a new command in the user command table which gets executed when the user clicks on the respective icon. We generally recommend the implementation as a command because it avoid other problems which can lead to user interface blocking.

What happens if you click on the menu item py_close of the first or on py_about of the second menu? Since the corresponding commands close and about are legal TOPSPIN commands, i.e. they are defined in the main command table of TOPSPIN, they will be executed and the window is closed, or the TOPSPIN version is printed.

And here is the Python program.

```
import de.bruker.nmr.prsc.toplib as top
import javax.swing as sw
import java.awt as awt

"""
      Implements TOPSPIN command 'pymbar1'
      - Opens a new window internal to TOPSPIN,
      - adds a left-adjusted menu bar
      - adds a separator line
      - adds a label 'please click on any menu item'
      - implements EventIfc methods to catch internal window events
      - prints the command assigned to a tool button when clicking on
it
"""

class ApplicationPanelMbar1(sw.JPanel, top.EventIfc):
      def __init__(self):
            self.setLayout(awt.BorderLayout())
            self.setBackground(top.Glob.frameBackground)

            northPanel = sw.JPanel(background=self.background)
            northPanel.setLayout(sw.BoxLayout(northPanel,
                 w.BoxLayout.Y_AXIS))


            mbarPanel =

      w.JPanel(layout=awt.FlowLayout(awt.FlowLayout.LEFT,0,0),
            background=self.background)
            menubar = top.Mbar("menubar_exam.prop")
            mbarPanel.add(menubar)
            northPanel.add(mbarPanel)
            northPanel.add(sw.JSeparator())
            self.add(northPanel, awt.BorderLayout.NORTH)
            self.add(sw.JLabel("Please click on any menu item!",
                  foreground=(255, 0, 0), font =
                  top.Glob.dialogFont))
```

```
	# the following methods implement 'EventIfc' to catch internal
	# window events
	def performAction(self, actionCode, arg, initiator):
		top.Dia.info(actionCode, "", "")

	def getData(self):
		pass

	def redisplay(self):
		pass

	def notifySelected(self):
		pass

	def notifyDeselected(self):
		pass

	def cleanUp(self):
		pass

userPanel1 = top.Win.newUserPanel("Simple Application With Menu Bar",
0)
userPanel1.add(ApplicationPanelMbar1(), awt.BorderLayout.CENTER)
userPanel1.invalidate()
userPanel1.validate()
userPanel1.repaint()
```

4.8 Window With Menu Bar and Tool Bar

A simple combination of the examples of the last 2 sections of this chapter creates an internal TOPSPIN window with a menu bar and a tool bar. The example can be invoked by the TOPSPIN command ***pymtbar***. The source code is available in the file `ExamMbarTbar.py`.

4.9 Window With Menu Bar and Tool Bar, and Scalable Rectangle

An extension of the example of the previous section creates an internal TOPSPIN window with a menu bar and a tool bar. In addition it displays a red rectangle. When clicking *2 or /2 on the lower main tool bar of TOPSPIN the rectangle is resized accordingly. The example can be invoked by the TOPSPIN command ***pyrect1***. The source code is available in the file `ExamRect1.py`.

4.10 NMR Data Sets

This section discusses how you can open NMR data sets, read the fid or spectrum or parameters, and manipulate the data points using your own algorithms. The functionality described here is available in the Java classes `BNMRDataSet` and `BNMRComponent` of the `toplib` package. `BNMRDataSet` contains functions such as `open()`, `getProcData()`, `getRawData()`, `getPar()`, `getPeakList`, etc. `BNMRComponent` provides functions to access components of the objects returned by the functions of `BNMRDataSet`. These are the actual data arrays or parts of them, e.g. `getColumn()`, `getRowFromIndex()`, etc. The following subsections display the usage of these functions.

Most of the functions dealing with data sets throw the exception `MfrException` if an error occurs. You can catch the exception and proceed accordingly in your program. If your script does not catch such exceptions, it is aborted in the case of an error, and a dialog is opened with the stack trace of your script.

4.10.1 Reading a complete 1D spectrum and fid

The following example explains the usage of the functions `open(),getProcData(),getRawData(),getDataDouble()`. The latter function returns the actual floating point array representing the complete processed or raw data. The example also shows how to read parameters (here: `SI` and `TD`).

```
from TopCmds import *

"""
      Implements the TOPSPIN command 'pydat1'
      Shows how to open a data set, and to
      read the spectrum and the fid.
      Calculates the total integrals of spectrum and fid
      and displays the result.
      Note that no exceptions are caught!
"""

# define and open test data set
testdata = ["exam1d_13C", "1", "1", "c:/Bruker/Topspin", "guest"]
dataset = top.BNMRDataSet.open(testdata)

# get real processed data ("1r")
data = dataset.getProcData().getDataDouble()
si = dataset.getPar("SI", 1)
spec_integral = 0 # build sum
for point in data:
      spec_integral += point

# get raw data ("fid")
data = dataset.getRawData().getDataDouble()
td = dataset.getPar("TD", 1)
fid_integral = 0 # build sum
for point in data:
      fid_integral += point

top.Dia.info("spec_integral = " + str(spec_integral) +
      " , SI = " + si + "\n" +
      "fid_integral = " + str(fid_integral) +
      " , TD = " + td,
      "Integrate", "")
```

4.10.2 Reading a section of a 1D spectrum

While the previous example reads a complete spectrum, the following example reads the data points from 76 to 80 ppm using the function `getDataMatrix(4.0,5.0)`. It also shows how to catch an the exception thrown if the data set can't be opened: It displays an error message in a dialog window. Note that a `from ... import` statement must be used to make the exception known to the script.

```
import de.bruker.nmr.prsc.toplib as top
from de.bruker.nmr.jutil.except import MfrException

"""
      Implements the TOPSPIN command 'pydat2'
      Shows how to open a data set, and to
      read a section of the spectrum.
      Calculates the integral of the section
      and displays the result.
      If "open" terminates with an exception, the
      exception is caught, and a message is printed.
"""
```

```
# define and open test data set
testdata = ["exam1d_13C", "1", "11", "c:/Bruker/Topspin", "guest"]
try:
      dataset = top.BNMRDataSet.open(testdata)
except MfrException, e  :
      top.Dia.info("Data set not found:\n" + str(testdata),
 "Exception", str(e))
else:
      # get real processed data defined by section
      data = dataset.getProcData().getDataMatrix(76.0, 80.0)
      spec_integral = 0 # build sum
      for point in data:
            spec_integral += point

      top.Dia.info("spec_integral = " + str(spec_integral) +
            " , #Points = " + str(len(data)), "Integrate section",
"")
```