

Test Automation

Python Modules Guide

Release 2021-A – May 2021

How to Contact dSPACE

Mail:	dSPACE GmbH Rathenaustraße 26 33102 Paderborn Germany
Tel.:	+49 5251 1638-0
Fax:	+49 5251 16198-0
E-mail:	info@dspace.de
Web:	http://www.dspace.com

How to Contact dSPACE Support

If you encounter a problem when using dSPACE products, contact your local dSPACE representative:

- Local dSPACE companies and distributors: <http://www.dspace.com/go/locations>
- For countries not listed, contact dSPACE GmbH in Paderborn, Germany.
Tel.: +49 5251 1638-941 or e-mail: support@dspace.de

You can also use the support request form: <http://www.dspace.com/go/supportrequest>. If you are logged on to mydSPACE, you are automatically identified and do not need to add your contact details manually.

If possible, always provide the relevant dSPACE License ID or the serial number of the CmContainer in your support request.

Software Updates and Patches

dSPACE strongly recommends that you download and install the most recent patches for your current dSPACE installation. Visit <http://www.dspace.com/go/patches> for software updates and patches.

Important Notice

This publication contains proprietary information that is protected by copyright. All rights are reserved. The publication may be printed for personal or internal use provided all the proprietary markings are retained on all printed copies. In all other cases, the publication must not be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form, in whole or in part, without the prior written consent of dSPACE GmbH.

© 1999 - 2021 by:
dSPACE GmbH
Rathenaustraße 26
33102 Paderborn
Germany

This publication and the contents hereof are subject to change without notice.

AUTERA, ConfigurationDesk, ControlDesk, MicroAutoBox, MicroLabBox, SCALEXIO, SIMPHERA, SYNECT, SystemDesk, TargetLink and VEOS are registered trademarks of dSPACE GmbH in the United States or other countries, or both. Other brand names or product names are trademarks or registered trademarks of their respective companies or organizations.

Contents

About This Guide	5
Safety Precautions	7
General Warning.....	7
Introduction to the Python Modules for Test Automation	9
Applying Test Automation Features.....	11
Specific Python Modules.....	11
Structure of the Test Automation Python Modules.....	12
Working with MATLAB	15
How to Execute a matlablib2 Demo Script in PythonWin.....	16
Interfacing MATLAB.....	16
Example of Accessing MATLAB with matlablib2.....	17
Example of Executing a matlablib2 Demo Script Using an External Python Interpreter.....	19
Example of Exchanging Data Between MATLAB and Python.....	20
Reading and Writing MAT Files.....	22
Acquiring Data via the Serial Interface	25
Acquiring Data from External Devices.....	25
Example of Accessing the Serial Interface.....	26
Analyzing and Evaluating the Test	29
ITAE criterion.....	29
Limitations	31
Limitations when Working with MATLAB.....	31
Index	33

About This Guide






Content

The purpose of this guide is to introduce you to the Test Automation Python Modules. With the help of examples, the test phases are considered. As a result, you will be able to automate even complex test scenarios.

It is assumed that you know how to implement control models in Simulink, on dSPACE real-time hardware and VEOS. Furthermore, knowledge in handling the host PC and the Microsoft Windows operating system is a prerequisite.

Symbols

dSPACE user documentation uses the following symbols:

Symbol	Description
 DANGER	Indicates a hazardous situation that, if not avoided, will result in death or serious injury.
 WARNING	Indicates a hazardous situation that, if not avoided, could result in death or serious injury.
 CAUTION	Indicates a hazardous situation that, if not avoided, could result in minor or moderate injury.
NOTICE	Indicates a hazard that, if not avoided, could result in property damage.
Note	Indicates important information that you should take into account to avoid malfunctions.
Tip	Indicates tips that can make your work easier.
	Indicates a link that refers to a definition in the glossary, which you can find at the end of the document unless stated otherwise.
	Precedes the document title in a link that refers to another document.

Naming conventions

dSPACE user documentation uses the following naming conventions:

%name% Names enclosed in percent signs refer to environment variables for file and path names.

< > Angle brackets contain wildcard characters or placeholders for variable file and path names, etc.

Special folders

Some software products use the following special folders:

Common Program Data folder A standard folder for application-specific configuration data that is used by all users.

`%PROGRAMDATA%\dSPACE\<InstallationGUID>\<ProductName>`

or

`%PROGRAMDATA%\dSPACE\<ProductName>\<VersionNumber>`

Documents folder A standard folder for user-specific documents.

`%USERPROFILE%\Documents\dSPACE\<ProductName>\<VersionNumber>`

Local Program Data folder A standard folder for application-specific configuration data that is used by the current, non-roaming user.

`%USERPROFILE%\AppData\Local\dSPACE\<InstallationGUID>\<ProductName>`

Accessing dSPACE Help and PDF Files


After you install and decrypt dSPACE software, the documentation for the installed products is available in dSPACE Help and as PDF files.

dSPACE Help (local) You can open your local installation of dSPACE Help:

- On its home page via Windows Start Menu
- On specific content using context-sensitive help via **F1**

dSPACE Help (Web) You can access the Web version of dSPACE Help at www.dspace.com/go/help.

To access the Web version, you must have a *mydSPACE* account.

PDF files You can access PDF files via the  icon in dSPACE Help. The PDF opens on the first page.

Safety Precautions

Introduction

To avoid risk of injury and/or property damage, read and ensure compliance with the safety precautions given.

General Warning

Danger potential

Using dSPACE software can be dangerous. You must observe the following safety instructions and the relevant instructions in the user documentation.

Improper or negligent use can result in serious personal injury and/or property damage

Using the dSPACE software can have a direct effect on technical systems (electrical, hydraulic, mechanical) connected to it.

The risk of property damage or personal injury also exists when the dSPACE software is controlled via an automation interface. The dSPACE software is then part of an overall system and may not be visible to the end user. It nevertheless produces a direct effect on the technical system via the controlling application that uses the automation interface.

- Only persons who are qualified to use dSPACE software, and who have been informed of the above dangers and possible consequences, are permitted to use this software.
- All applications where malfunctions or operating errors involve the danger of injury or death must be examined for potential hazards by the user, who must if necessary take additional measures for protection (for example, an emergency off switch).

Liability

It is your responsibility to adhere to instructions and warnings. Any unskilled operation or other improper use of this product in violation of the respective safety instructions, warnings, or other instructions contained in the user documentation constitutes contributory negligence, which may lead to a limitation of liability by dSPACE GmbH, its representatives, agents and regional

dSPACE companies, to the point of total exclusion, as the case may be. Any exclusion or limitation of liability according to other applicable regulations, individual agreements, and applicable general terms and conditions remain unaffected.

Data loss during operating system shutdown

The shutdown procedure of Microsoft Windows operating systems causes some required processes to be aborted although they are still being used by dSPACE software. To avoid data loss, the dSPACE software must be terminated manually before a PC shutdown is performed.

Introduction to the Python Modules for Test Automation

Introduction

The increasing complexity of control software means that testing has to be automated. The Test Automation Python Modules are a tried and tested tool for quality assurance in the development of advanced ECUs.

Specific Python modules

The Test Automation Python Modules include specific Python modules that allow you to:


- Access the serial interface (refer to [Acquiring Data from External Devices](#) on page 25)
- Exchange data with MATLAB or invoke MATLAB commands (refer to [Working with MATLAB](#) on page 15)

As a result, the test operation can be automatically adjusted and analyzed while the application is running.

Packaging and licences

Which library you can use depends on the dSPACE product you bought and the corresponding package. The following table shows an overview on the different dSPACE products and the libraries you get with.

Python Module	Contained in Product Set	Required Package
rs232lib2	AutomationDesk	—
	Test Automation APIs	Platform API Package
matlablib2	AutomationDesk	—
	Test Automation APIs	Platform API Package

For further information on packages and licences, refer to [Managing dSPACE Software Installations](#) .

Tip

All dSPACE products with Python interpreter access all Python test automation modules when you installed the Test Automation APIs product set.

Note

As of dSPACE Release 2021-A, the Test Automation Python modules are migrated to Python 3.9. As a consequence, you can use them only with a 64-bit Python 3.9 interpreter.

Test Automation in AutomationDesk

You can use the Test Automation Python Modules in AutomationDesk. You can run the scripts by using the ExecFile block.

ControlDesk Demo Pool

With ControlDesk Demo Pool, your dSPACE installation provides a directory with Python scripts that are useful for automation tasks. It is highly recommended to study these.

The demo scripts specific to ControlDesk Test Automation are located at:

- <ControlDeskInstallationFolder>\Demos\Toolautomation\
- <PythonExtensionsInstallationFolder>\Demos\Python Test Automation\

Python references

For Python references, refer to the documentation of the Python installation.

Where to go from here

Information in this section

Applying Test Automation Features.....	11
Specific Python Modules.....	11
Structure of the Test Automation Python Modules.....	12

Applying Test Automation Features

Introduction To apply test automation features to a control model, which has been implemented using the dSPACE environment, you have to consider the following aspects.

Script-based automation Automating a test means programming a Python script, which is executed by a Python interpreter.

- For information on how to use ControlDesk’s built-in Python Interpreter, refer to [Using the Internal Interpreter \(ControlDesk Automation !\[\]\(4729e517bc6a7cd81c8025b9646574fb_img.jpg\)](#)).
- The Test Automation Python Modules provide specific Python modules, refer to [Specific Python Modules](#) on page 11.


To get an overview of the Python modules for automating ControlDesk, refer to [Programming ControlDesk Automation \(ControlDesk Automation !\[\]\(90a2fb2f2c617b26262139ae4159c0a0_img.jpg\)](#)).

Related topics

Basics

Introduction to the Python Modules for Test Automation.....	9
Specific Python Modules.....	11

References

Overview of the Test Automation Python Modules (Test Automation Python Modules Reference )
--

Specific Python Modules

Introduction The Test Automation Python Modules include several specific Python modules used for automating tests.

- For an overview, see [Structure of the Test Automation Python Modules](#) on page 12.
- For detailed information on the functions, classes, and variables, refer to the [Test Automation Python Modules Reference !\[\]\(e474458956c9a37fbf9586ddb60a7fa1_img.jpg\)](#).

To automate tests, you can also use all functions, classes and variables provided by ControlDesk Tool Automation. Refer to [Tool Automation Demos \(ControlDesk Introduction and Overview !\[\]\(4d1d3f2547aeece54bb6babd23f4121b_img.jpg\)](#)).

Terms and definitions

Classes dSPACE's Python modules provide the classes, functions, and variables you need for automation and test automation. For each class, attributes and methods are defined. Attributes can be simple properties or objects of other classes.

Objects To apply a class definition, you have to declare an instance of the class – an object. The object assumes the class's attributes and methods.

Related topics**Basics**

[Introduction to the Python Modules for Test Automation.....9](#)

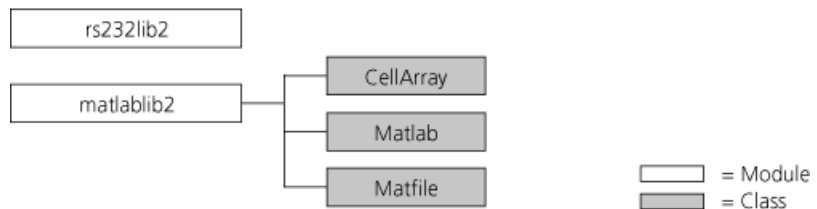
References

[Overview of the Test Automation Python Modules \(Test Automation Python Modules Reference !\[\]\(05be7c7a8995decd503647c99211f7c2_img.jpg\)\)](#)

Structure of the Test Automation Python Modules

Overview

The illustration gives an overview of the modules and classes and an excerpt from the function blocks provided by the Test Automation Python Modules.

**rs232lib2**

The rs232lib2 module provides the functions for communication via the serial interface. The major functions are:

- Opening and closing the connection to the serial interface
- Configuring baudrate, parity, data and stop bits
- Setting input and output buffers
- Receiving data from and sending data to the serial interface

For example, you can use the rs232lib2 functions to access external diagnosis devices or to control laboratory devices remotely.


For further information on rs232lib2, refer to [Acquiring Data from External Devices](#) on page 25.

matlablib2

The matlablib2 module provides access to MATLAB. Using the *Matlab* class, you can exchange data between MATLAB and ControlDesk's or an external Python interpreter or invoke MATLAB functions (for example reuse existing m-scripts). matlablib2 also provides access to the MATLAB file format (MAT-files) via the *Matfile* class.

For information on how to apply matlablib2, refer to [Working with MATLAB](#) on page 15.

Working with MATLAB

Introduction	The matlablib2 module provides an interface to MATLAB and access to MAT files.
Matlab class	Using instances of the class <i>Matlab</i> , you can access MATLAB and exchange data between MATLAB and a Python interpreter. Refer to Interfacing MATLAB on page 16.
Matfile class	Using instances of the class <i>Matfile</i> , you can access MAT files. Refer to Reading and Writing MAT Files on page 22.
Limitations	For restrictions concerning matlablib, refer to Limitations when Working with MATLAB on page 31.
Python Extensions Demo Pool	dSPACE Python Extensions Demo Pool provides several demo scripts for working with MATLAB. It is highly recommended that you study these. Refer to How to Execute a matlablib2 Demo Script in PythonWin on page 16.
Python reference	For detailed information on the functions, classes and methods of matlablib, refer to Interfacing MATLAB (matlablib2) (Test Automation Python Modules Reference ).

Where to go from here

Information in this section

How to Execute a matlablib2 Demo Script in PythonWin.....	16
Interfacing MATLAB.....	16
Example of Accessing MATLAB with matlablib2.....	17

Example of Executing a matlablib2 Demo Script Using an External Python Interpreter.....	19
Example of Exchanging Data Between MATLAB and Python.....	20
Reading and Writing MAT Files.....	22

How to Execute a matlablib2 Demo Script in PythonWin

Objective dSPACE Python Extensions Demo Pool provides several demo scripts that demonstrate how to apply matlablib. You will find these demo scripts at <PythonExtensionsInstallationFolder>\Demos\Python Test Automation\Interfacing Matlab (matlablib2).

Method

To execute matlablib2 demo script

- 1 Open PythonWin (Start - Programs - Python 3.9 - PythonWin).
- 2 From the menu bar, choose File – Open to open a demo script from <PythonExtensionsInstallationFolder>\Demos\Python Test Automation\Interfacing Matlab (matlablib2).
- 3 From the menu bar, choose File – Run to execute the demo script.
- 4 In the MATLAB Command Window, enter **whos** to observe the results (not relevant for the Matfile demo at <PythonExtensionsInstallationFolder>\Demos\Python Test Automation\Interfacing Matlab (matlablib2)).

Related topics

Basics

Working with MATLAB.....	15
--------------------------	----

Interfacing MATLAB

Introduction

With the *Matlab* class, matlablib2 provides the methods to:

- Access MATLAB.
- Exchange data between MATLAB and ControlDesk or an external Python interpreter.

Accessing MATLAB

You can use instances of the *Matlab* class to:

- Open, test, and close a connection to MATLAB.
- Execute MATLAB commands from within the Python interpreter and get MATLAB's output.

Refer to [Example of Accessing MATLAB with matlablib2](#) on page 17.

Exchanging data

Instances of the *Matlab* class also allows you to exchange data between MATLAB and ControlDesk's or an external Python interpreter.

- Using the `GetArray` method, you can get a Python representation of a MATLAB array from a MATLAB workspace variable.
- Using the `PutArray` method, you can assign a Python array or string to a MATLAB workspace variable.

Refer to [Example of Exchanging Data Between MATLAB and Python](#) on page 20.

Related topics**Basics**

[Working with MATLAB..... 15](#)

References

[Interfacing MATLAB \(matlablib2\) \(Test Automation Python Modules Reference !\[\]\(95b425611cbd2b8716a140cf67c81822_img.jpg\)](#))
[Matlab \(Test Automation Python Modules Reference !\[\]\(98475352b625a273242ad989dd0cabc3_img.jpg\)](#))

Example of Accessing MATLAB with matlablib2

Introduction

The dSPACE Python Extensions Demo Pool contains the script `d_ExecuteMatlabCommands.py` that shows how to execute MATLAB commands from within ControlDesk's or an external Python interpreter. You will find the script at `<PythonExtensionsInstallationFolder>\Demos\Python Test Automation\Interfacing Matlab (matlablib2)\ExecutingCommands\`.

The example script defines the `ExecuteDemo` function that creates arrays in MATLAB and plots their contents using the MATLAB plot utility. It also demonstrates the observation of MATLAB results from within ControlDesk's or an external Python interpreter.

Note

To ensure that all matlablib2 objects are deleted at the end of the script, their creation and usage is enclosed in a `try ... finally` block and all objects are set to "None" in the `finally` block (not shown here). This ensures that the communication resources of ControlDesk are released properly. Otherwise, this may lead to unpredictable behavior of the communication components.

The following code fragment is extracted from the `ExecuteDemo` function:

Initialize

```
import matlablib2
...
# the math module is required for the definition of 'pi', 'sin', 'cos' and 'log'
from math import *
```

Open MATLAB interface

```
# Create a new instance of the class Matlab() and open
# access to MATLAB
MyMatlab = matlablib2.Matlab()
MyMatlab.Open()
```

Generate data

```
# Generate arrays to send them to MATLAB and plot them in MATLAB
t=0.0
X=[]
Y1=[]
Y2=[]
while t <= 4*pi:
    X.append(t)
    Y1.append( sin(t) )
    Y2.append( 0.8 * log(t+0.1) )
    t = t + 0.02
```

Send data to MATLAB

```
MyMatlab.PutArray("X", X)
MyMatlab.PutArray("Y1", Y1)
MyMatlab.PutArray("Y2", Y2)
```

Execute MATLAB commands

```
# Execute plot commands in MATLAB for the two curves
MyMatlab.Execute("plot(X,Y1,'b');grid on;hold on; plot(X,Y2,'r');hold off")
...
```

```
# Execute Commands in MATLAB and observe the results in the Python interpreter
print("Show outputs of MATLAB commands in Python window:\n")
MyMatlab.Execute("pwd")
print("Output of 'pwd' is:")
print(MyMatlab.GetOutputs())
CurDir = os.getcwd()
print("Setting working directory of MATLAB to", CurDir)
MyMatlab.Execute("cd %s" % CurDir)
print(MyMatlab.GetOutputs())
MyMatlab.Execute("pwd")
print("Output of 'pwd' now is:")
print(MyMatlab.GetOutputs())
MyMatlab.Execute("whos")
print("Output of 'whos' is:")
print(MyMatlab.GetOutputs())
```

Close MATLAB interface

```
MyMatlab.Close ()
```

Related topics

Basics

[Interfacing MATLAB.....](#) 16

Examples

[Example of Exchanging Data Between MATLAB and Python.....](#) 20

References

[Interfacing MATLAB \(matlablib2\) \(Test Automation Python Modules Reference !\[\]\(870f5d5e9c0d57485634be3ecf52f3ca_img.jpg\)](#))
[Matlab \(Test Automation Python Modules Reference !\[\]\(66b14d8ba452f6f18b47935355b6120a_img.jpg\)](#))

Example of Executing a matlablib2 Demo Script Using an External Python Interpreter

Demo scripts

You find matlablib2 demo scripts at `<PythonExtensionsInstallationFolder>\Demos\Python Test Automation\Interfacing Matlab (matlablib2)`. You can run each of the demo scripts in using an external Python interpreter (e.g. PythonWin).

Start PythonWin in Start-Programs-Python 3.9-PythonWin. Precondition: install the Test Automation APIs product set, which includes dSPACE Python Extensions.

Example of Exchanging Data Between MATLAB and Python

Introduction

The dSPACE Python Extensions Demo Pool contains two example scripts that show how to exchange data between MATLAB and ControlDesk's or an external Python interpreter.

- `d_SimpleDataTypes.py` concentrates on the more simple data types in MATLAB.
- `d_ComplexDataTypes.py` concentrates on the more complex MATLAB data types `cell array` and `struct array`.

Example script

In the following, the demo script `d_SimpleDataTypes.py` is explained. You will find the script at

`<PythonExtensionsInstallationFolder>\Demos\Python Test Automation\Interfacing Matlab (matlablib2)\ExchangeOfSimpleDataTypes\`.

The example script defines the `ExecuteDemo` function that generates arrays of different types and sizes, inserts them into MATLAB's workspace, and reads them back to ControlDesk's or an external Python interpreter.

Note

To ensure that all `matlablib2` objects are deleted at the end of the script, their creation and usage is enclosed in a `try ... finally` block and all objects are set to "None" in the `finally` block (not shown here). This ensures that the communication resources of ControlDesk are released properly. Otherwise, this may lead to unpredictable behavior of the communication components.

The following code fragment is extracted from the `ExecuteDemo` function:

Initialize

```
import matlablib2
...
```

Open MATLAB interface

```
# Create a new instance of the class Matlab() and open access to MATLAB
MyMatlab = matlablib2.Matlab()
MyMatlab.Open()
```

Put arrays to MATLAB

```
# Send elements of different types to the Matlab workspace
# Put an empty array to MATLAB's workspace
MyMatlab.PutArray("EmptyArray", [])
# Put a string to MATLAB's workspace
MyMatlab.PutArray("PlainString", "dSPACE - solutions for control")
# Put a plain float to MATLAB's workspace
MyMatlab.PutArray("PlainFloat", 42.0)
# 1x1 array
MyMatlab.PutArray("FloatArray1x1", [0.11])
# 1x3 row vector
MyMatlab.PutArray("FloatArray1x3", [0.11, 0.12, 0.13])
# Put a plain int to MATLAB's workspace
MyMatlab.PutArray("PlainInt32", 42)
# 1x1 array
MyMatlab.PutArray("Int32Array1x1", [11])
# 1x3 row vector
MyMatlab.PutArray("Int32Array1x3", [11, 12, 13])
# 3x1 column vector
MyMatlab.PutArray("Int32Array3x1", [[11], [21], [31]])
# 3x2 array
MyMatlab.PutArray("Int32Array3x2", [[11,12], [21,22], [31,32]])
# 3x4 array
MyMatlab.PutArray("Int32Array3x4", [[11, 12, 13, 14],
                                     [21, 22, 23, 24],
                                     [31, 32, 33, 34]])
# 3x4x2 array
MyMatlab.PutArray("Int32Array3x4x2", [[[111, 121, 131, 141],
                                         [211, 221, 231, 241],
                                         [311, 321, 331, 341]],
                                         [[112, 122, 132, 142],
                                         [212, 222, 232, 242],
                                         [312, 322, 332, 342]]])
```

Get arrays from MATLAB

```
# Read some of the just created elements from the MATLAB workspace
elem = MyMatlab.GetArray ("EmptyArray")
print("\n++++")
print("Contents of EmptyArray are: ", elem)
print("++++\n")
elem = MyMatlab.GetArray ("PlainString")
print("++++")
print("Contents of PlainString are: ", elem)
print("++++\n")
elem = MyMatlab.GetArray ("FloatArray1x3")
print("++++")
print("Contents of FloatArray1x3 are: ", elem)
print("++++\n")
elem = MyMatlab.GetArray ("Int32Array3x4x2")
print("++++")
print("Contents of Int32Array3x4x2 are: ", elem)
print("++++\n")
```

Close MATLAB interface

```
MyMatlab.Close()
```

Related topics**Basics**

[Interfacing MATLAB.....](#) 16

Examples

[Example of Accessing MATLAB with matlablib2.....](#) 17

References

[Interfacing MATLAB \(matlablib2\) \(Test Automation Python Modules Reference !\[\]\(10f8862fc183b400327470ea85afe9ae_img.jpg\)\)](#)
[Matlab \(Test Automation Python Modules Reference !\[\]\(4ba8d838a2aa5445d51c9dee78fcb0cc_img.jpg\)\)](#)

Reading and Writing MAT Files

Introduction

For access to MAT files, matlablib2 provides the *Matfile* class.

Matfile objects

The *Matfile* objects can be used to:

- Create, open, and close MAT files.
- Load MAT files and assign all contained arrays to Python variables.
- Write arrays to MAT files (using the PutArray method).
- Read arrays from MAT files (using the GetArray method).
- Delete arrays from MAT files.

Demo script

The dSPACE Python Extensions Demo Pool contains the script `d_ReadAndWriteMatFiles.py` that shows how to handle MAT files, read from and write to MAT files. You will find the script at `<PythonExtensionsInstallationFolder>\Demos\Python Test Automation\Interfacing Matlab (matlablib2)\ReadingAndWritingMatfiles\`.

The example script defines the `ExecuteDemo` function that creates a MAT file, inserts data, reads the data, and writes it into another MAT file. As a result, two identical MAT files are generated.

Note

To ensure that all matlablib2 objects are deleted at the end of the script, their creation and usage is enclosed in a `try ... finally` block and all objects are set to "None" in the `finally` block (not shown here). This ensures that the communication resources of ControlDesk are released properly. Otherwise, this may lead to unpredictable behavior of the communication components.

Example

The following code fragment is extracted from the `ExecuteDemo` function:

Initialize

```
import matlablib2
...
sMatfilename1 = WorkingDir + "\\DemoTemp\\MatfileExample1.mat"
sMatfilename2 = WorkingDir + "\\DemoTemp\\MatfileExample2.mat"
...
```

Create MAT file

```
# Create a new matfile and write some arrays into it
# Create a new instance of the class Matfile()
MyMatfile = matlablib2.Matfile()
# Open / create the matfile sMatfilename1 for writing
MyMatfile.Open(sMatfilename1, "w")
```

Write to MAT file

```
# Write a few Arrays into the matfile
MyMatfile.PutArray ("PlainString", "dSPACE - solutions for control")
MyMatfile.PutArray ("PlainFloat", 42.0)
MyMatfile.PutArray ("PlainInt32", 12)
# Close the matfile
MyMatfile.Close ()
```

Open MAT files

```
# Create an instance of the class Matfile() for each of the
# files to be opened and open these files
MatfileRead = matlablib2.Matfile ()
MatfileRead.Open (sMatfilename1 , 'r')
MatfileWrite = matlablib2.Matfile ()
MatfileWrite.Open (sMatfilename2 , 'w')
```

Read/write MAT files

```
# Read all the contents of the file MatfileRead and write them
# into the file MatfileWrite
for name in MatfileRead.GetDir():
    array = MatfileRead.GetArray(name)
    MatfileWrite.PutArray(name, array)
```

Close all open files

```
MatfileRead.Close ()
MatfileWrite.Close ()
```

Related topics

Basics

[Working with MATLAB.....](#) 15

References

[Interfacing MATLAB \(matlablib2\) \(Test Automation Python Modules Reference !\[\]\(96cc62f861fdd6e50510c0224a756dff_img.jpg\)\)](#)
[Matfile \(Test Automation Python Modules Reference !\[\]\(e658400d40ca763c7cf4c8c420885c6a_img.jpg\)\)](#)

Acquiring Data via the Serial Interface

Introduction

The Test Automation Python Modules provide several Python modules that allow you to acquire data from different systems.

Where to go from here

Information in this section

[Acquiring Data from External Devices.....](#) 25

The rs232lib2 module allows you to access external devices via the serial interface of your host PC.

[Example of Accessing the Serial Interface.....](#) 26

Acquiring Data from External Devices

Introduction

You can use the functions of the rs232lib2 module to acquire data from external devices via the serial interface of your host PC.

Using the rs232lib2 functions you can:

- Open and close a connection to the serial interface.
- Configure baudrate, data and stop bits.
- Set input and output buffers.
- Send data to the serial interface.
- Receive data from the serial interface.

For an example, refer to [Example of Accessing the Serial Interface](#) on page 26.

Related topics**Basics**

[Acquiring Data via the Serial Interface.....25](#)

References

[Acquiring Data from External Devices \(rs232lib2\) \(Test Automation Python Modules Reference !\[\]\(339a16584d5da0f0a3ca4e9ec17bf6a1_img.jpg\)](#))

Example of Accessing the Serial Interface

Demo scripts

The following demo scripts show how to access the serial interface of your host PC.

Note

To use the demo scripts, the serial interface of your host PC must be connected to the serial interface of the remote device via a null-modem cable. This could be another serial interface of your host PC. On the remote device, you have to start a terminal program for control of its serial interface.

The example script defines the **ExecuteDemo** function that opens a connection to the serial interface, configures the port, sets input and output buffers, transfers data, and finally closes the connection.

Note

To ensure that the connection to the serial interface is closed at the end of the script, the functions accessing the serial interface are enclosed in a **try ... finally** block and the connection is closed in the **finally** block.

Import modules

```
import rs232lib2    # Baselib for the demo script
import win32api     # Used for Sleep command
import win32ui      # Used for MessageBox
import win32con     # Used for styles for the MessageBox
import sys          # Used for stdout command
```

Define ExecuteDemo function

```
def ExecuteDemo():
    # Initialize the variable 'h' to a defined value. This is
    # used for cleaning up later.
    h = None
```

Begin of try ... finally block

```
# Use try and finally to clear the system at the end or if
# an error occurs
try:
```

Open connection

```
# Open a connection to a serial communication port.
h = rs232lib2.Open("COM2")
```

Configure interface

```
# Configuration of the serial port with baudrate: 4800,
# databits: 8, no parity and 1 stopbit.
rs232lib2.SetConfig(h, 4800, 8, "NO", 1)
# Set input and output buffer size to 2048 bytes
rs232lib2.SetBuffers(h, 2048, 2048)
# Set the timeout value for read operations to 8 second
rs232lib2.SetReadTimeout(h, 8000)
```

Check input buffer

```
# Determine the number of bytes in the input buffer
win32ui.MessageBox("The number of available bytes in the input \n" + \
    "buffer is issued in the command window in 100 ms steps \n" + \
    "for 8 seconds. Please transmit bytes to increase the input buffer.", \
    "d_rs232lib2", win32con.MB_SYSTEMMODAL | win32con.MB_ICONINFORMATION)
for i in range(80):
    print("Number of bytes in input buffer(%i):\t\t%i" \
        %(i,rs232lib2.GetNumInBytes(h)))
    # wait 0.1 second
    win32api.Sleep(100)
# Print arrived bytes in the command window
print("Buffer content:\t\t\t\t\t%s" \
    %rs232lib2.Read(h, rs232lib2.GetNumInBytes(h)))
```

Read input buffer

```
# Read and print input buffer
win32ui.MessageBox("Arriving characters at the serial \n" + \
    "communication will be read and printed in the command \n" + \
    "window as long as the arriving character is not a 'q'.\n", \
    "t_rs232lib2", win32con.MB_SYSTEMMODAL | win32con.MB_ICONINFORMATION)
print("Arriving character:\t\t\t\t\t", end=' ')
c = 0
while c != 'q':
    c = rs232lib2.Read(h)
    # same as 'print c,', but without spaces between characters
    sys.stdout.write(c)
    # "update" interpreter output
    win32ui.PumpWaitingMessages()
```

Read three bytes

```
win32ui.MessageBox("Please fill the input buffer with at least \n" + \
    "three characters." , "t_rs232lib2", \
    win32con.MB_SYSTEMMODAL | win32con.MB_ICONINFORMATION)
win32api.Sleep(5000) # wait 5 seconds
num = rs232lib2.GetNumInBytes(h)
if num >= 3:
    print("\nThe first 3 characters in input buffer:\t%s" \
        %rs232lib2.Read(h, 3))
else:
    print("\nNot enough bytes in input buffer, read command not execute...")
```

Write string

```
win32api.Sleep(1000)
win32ui.MessageBox("Write string to output buffer \n", "t_rs232lib2", \
    win32con.MB_SYSTEMMODAL | win32con.MB_ICONINFORMATION)
rs232lib2.WriteString(h, "\n\rHello World\n\r")
```

Write five bytes

```
for i in range(65, 70):
    rs232lib2.Write(h, i)
```

End of demo script

```
win32api.Sleep(1000) # wait 1 second
win32ui.MessageBox("Demo finished...", "t_rs232lib2", \
    win32con.MB_SYSTEMMODAL | win32con.MB_ICONEXCLAMATION)
```

End of try ... finally block

```
finally:
    # It is important to close the connection to the serial
    # port if it is not used furthermore.
    # Has the connection been opened successfully?
    if not None == h:
        rs232lib2.Close(h)
    h = None
```

Main program

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

Related topics**Basics**

[Acquiring Data from External Devices..... 25](#)

References

[Acquiring Data from External Devices \(rs232lib2\) \(Test Automation Python Modules Reference !\[\]\(c1168d6a8b365d11e842ece304635fa7_img.jpg\)](#))

Analyzing and Evaluating the Test

Introduction

To evaluate test results, you can use the basic calculation functions of the standard Python language or MATLAB functions provided by the Python module `matlablib2`.

ITAE criterion

Introduction

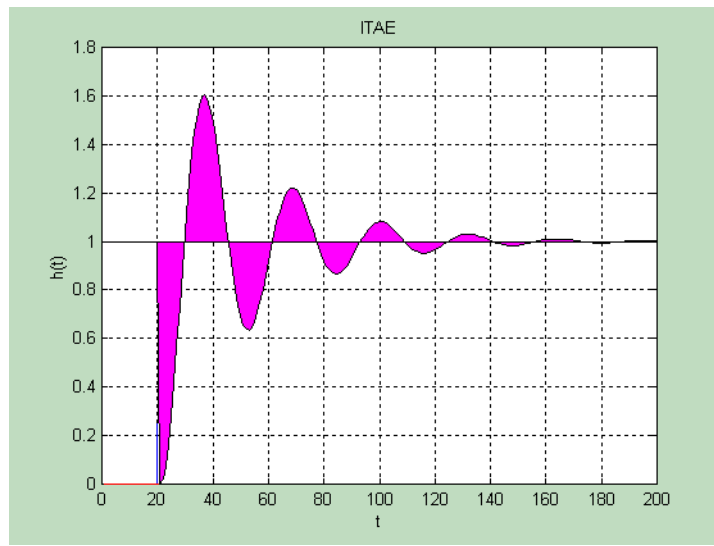
The ITAE criterion (abbreviation for Integral Time weighted Absolute Error) is used in the design of PID controllers to judge the control deviation for a step impulse. It calculates the area between step impulse and step response using the following formula:

$$ITAE = \frac{1}{N} \cdot \sum_{k=0}^N k \cdot |e(k)|$$

With:

- **e**: control deviation
- **N**: interval length

The following illustration shows a typical step response. The selected areas show the absolute error, which is time weighted when using the ITAE criterion.



Using standard Python functions

The following code fragment shows how to calculate the ITAE criterion using standard Python functions. The variables **RefSignal** and **ControlledVariable** store the values describing the reference signal and the output signal.

```
ITAE = 0.0
N = len(RefSignal)
for k in range(N):
    ITAE = ITAE + abs((k + 1) * (RefSignal[k] - ControlledVariable[k]))
ITAE = ITAE / float(N)
```

Using MATLAB interface

The following code fragment shows how to calculate the ITAE criterion using the MATLAB functionality provided by the `matlablib2` module. The variables **RefSignal** and **ControlledVariable** are used to store the values that describe the reference and output signals.


```
import matlablib2
ITAE = 0.0
ITAEScript = "N = length(RefSignal),\n"
            "ITAE = sum(abs([1:N]' .* (RefSignal(1:N) -\n"
            "            ControlledVariable(1:N)) ))/N;"
MLInstance = matlablib2.Matlab()
MLInstance.Open()
MLInstance.PutArray("RefSignal", RefSignal)
MLInstance.PutArray("ControlledVariable", ControlledVariable)
MLInstance.Execute(ITAEScript)
ITAE = MLInstance.GetArray("ITAE")
MLInstance.Close()
MLInstance = None
```

For detailed information on the MATLAB interface, refer to [Interfacing MATLAB](#) on page 16.

Limitations

Introduction For technical reasons, there are a few limitations that apply to the current version of the Test Automation Python Modules. These are listed according to the used elements.

Limitations when Working with MATLAB

MATLAB corresponding version To access MATLAB using matlablib2 you have to install MATLAB on your host PC. matlablib2 supports only the MATLAB version which corresponds to the dSPACE release. For further MATLAB compatibility information, refer to [Managing dSPACE Software Installations](#)  .

A MATLAB installation is not necessary to use the Matfile class.

Related topics	Basics
	Working with MATLAB..... 15

A

- accessing MATLAB functionality 17
- accessing serial interface 25
- acquiring data
 - from external devices 25
 - overview 25
- analyzing test 29
- attributes (Python) 12
- automation
 - overview 9

C

- classes (Python) 12
- Common Program Data folder 6
- ControlDesk Demo Pool
 - test automation 10

D

- Demo Pool
 - see Python Extensions Demo Pool 15
- demos for Test Automation
 - d_ComplexDataTypes.py 20
 - d_ExecuteMatlabCommands.py 17
 - d_ReadAndWriteMatFiles.py 22
 - d_SimpleDataTypes.py 20
- RS232 26
- Documents folder 6

E

- evaluate test 29
- external diagnosis devices 12

I

- ITAE criterion 29
 - using MATLAB 30
 - using Python 30

L

- limitations
 - matlablib2 31
- Local Program Data folder 6

M

- MATLAB
 - exchanging data with Python Interpreter 16
 - executing matlablib2 demos 16
 - interfacing MATLAB 16
 - working with MATLAB 15
- matlablib2
 - limitations 31
 - Matfile class 22
 - Matlab class 16
 - overview 13
 - test evaluation 30
 - working with 15
- method (Python) 12

O

- objects (Python) 12

P

- Python
 - attributes 12
 - classes 12
 - module matlablib2 13
 - module rs232lib2 12
 - objects 12
 - Test Automation modules 12
 - test evaluation 30
- Python Extensions Demo Pool
 - interfacing MATLAB 15
- Python Interpreter
 - accessing MATLAB functionality 17
 - accessing serial interface 25
 - acquiring data from external laboratory devices 25
 - executing MATLAB commands 17
 - interfacing MATLAB 16
 - reading from MAT files 22
 - writing to MAT-files 22

R

- RS232
 - demos for Test Automation 26
- rs232lib2
 - details 25
 - overview 12

T

- test analysis 29
- Test Automation
 - applying features 11
 - overview 9
 - Python modules overview 12
- test evaluation 29

