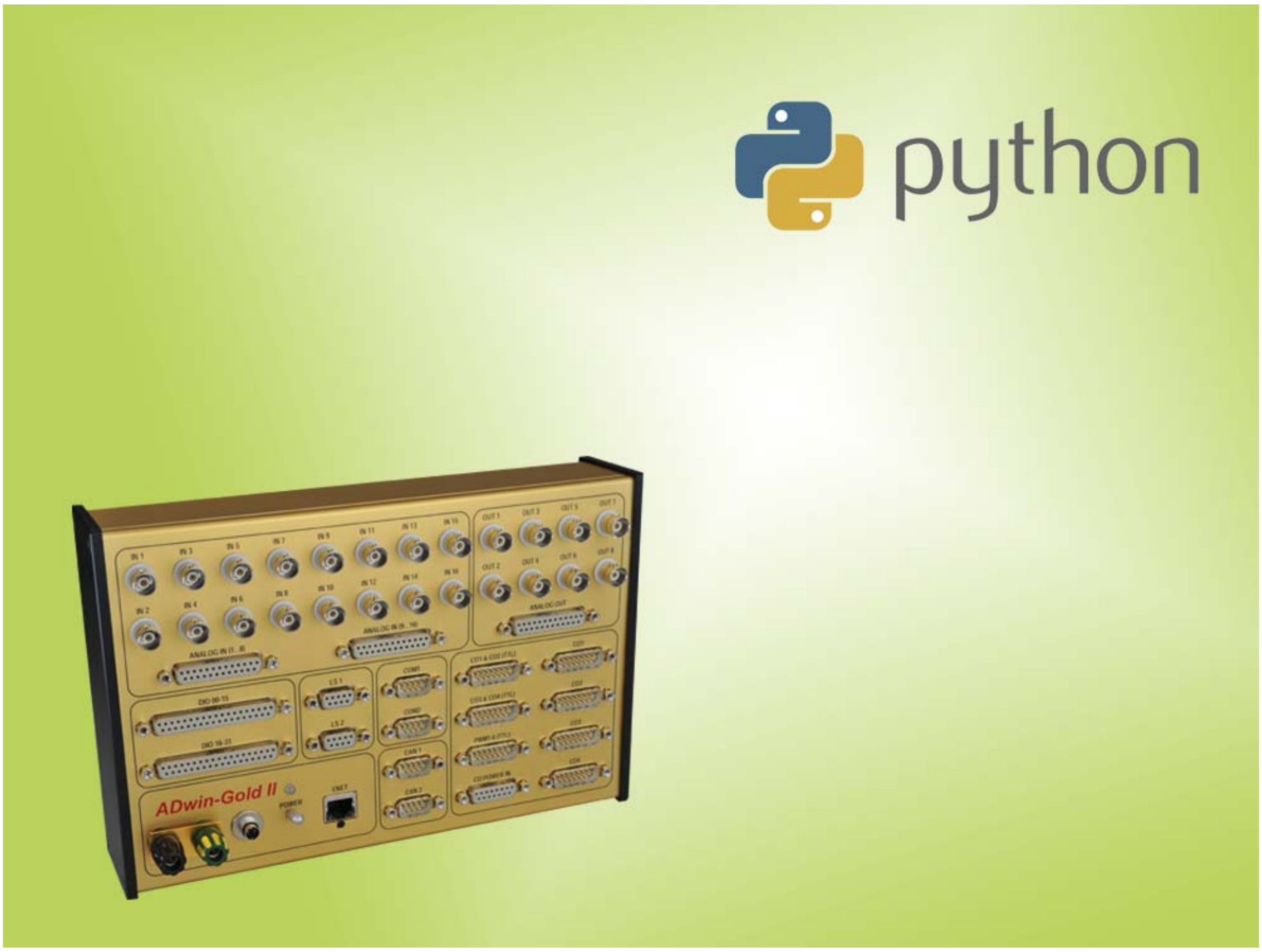
ADwin Driver

Driver for Python



ADwin Driver Python, Manual November 2017

For any questions, please don't hesitate to contact us:

Hotline:	+49 6251 96320
Fax:	+49 6251 568 19
E-Mail:	info@ADwin.de
Internet	www.ADwin.de

Jäger Computergesteuerte Messtechnik GmbH

Jäger Com-
putergesteuerte
Messtechnik GmbH
Rheinstraße 2-4
D-64653 Lorsch
Germany

Table of contents

Typographical Conventions

"Warning" stands for information, which indicate damages of hardware or software, test setup or injury to persons caused by incorrect handling.

You find a "note" next to

– information, which absolutely have to be considered in order to guarantee an error free operation.

– advice for efficient operation.

"Information" refers to further information in this documentation or to other sources such as manuals, data sheets, literature, etc.

<C:\ADwin\ …>

File names and paths are placed in <angle brackets> and characterized in the font `Courier New`.

`Program text`

Program commands and user inputs are characterized by the font `Courier New`.

`Var_1`

Source code elements such as commands, variables, comments and other text are characterized by the font `Courier New` and are printed in color.

Bits in data (here: 16 bit) are referred to as follows:

Bit No.	15	14	13	…	01	00
Bit value	2^{15}	2^{14}	2^{13}	…	2^1=2	2^0=1
Synonym	MSB	-	-	-	-	LSB

1 Information about this Manual

This manual contains comprehensive information about the *ADwin* driver for Python.

Additional information is available in:

– the manual "ADwin Installation", which describes all interface installations for the *ADwin* systems.

  Begin your installation with this manual.

– the manual "ADbasic", which contains all instructions for the compiler *ADbasic*. With this comfortable real-time development tool, you are programming your *ADwin* system.

– the hardware manuals for the *ADwin* systems you are using.

It is assumed that you are familiar with your Python environment.

Please note:

For *ADwin* systems to function correctly, adhere strictly to the information provided in this documentation and in other mentioned manuals.

Qualified personnel

Programming, start-up and operation, as well as the modification of program parameters must be performed only by appropriately qualified personnel.

> *Qualified personnel are persons who, due to their education, experience and training as well as their knowledge of applicable technical standards, guidelines, accident prevention regulations and operating conditions, have been authorized by a quality assurance representative at the site to perform the necessary acivities, while recognizing and avoiding any possible dangers.*
> *(Definition of qualified personnel as per VDE 105 and ICE 364).*

Availability of the documents

This product documentation and all documents referred to, have always to be available and to be strictly observed. For damages caused by disregarding the information in this documentation or in all other additional documentations, no liability is assumed by the company *Jäger Computergesteuerte Messtechnik GmbH*, Lorsch, Germany.

Legal information

OEM products are mentioned without referring to possible patent rights, the existence of which may not be excluded.

Subject to change.

Hotline address: see inner side of cover page.

2 *ADwin*-Python driver

The *ADwin* system consists of an independent on-board CPU, which executes measurement and control tasks very fast and reliably, as well as of an interface under Windows, Linux or Mac OS in order to control the *ADwin* system with Python.

Consequently you transfer all time-critical processes to the *ADwin* system, but with Python you still have control of the processes and data processing.

Please note: You need a Python interpreter version 2.4 or higher. Older versions have not been tested.

How to program the *ADwin* system

ADwin systems are fast, reliable and flexible. You apply the easy-to-learn programming language *ADbasic* in order to use all these advantages.

Before you can apply the here described Python instructions, we recommend to familiarize yourself with *ADbasic*. Please use the *ADbasic* manual and the programming instructions as help. The descriptions will help you to understand the *ADwin* system more easily.

Controlling the *ADwin* systems with Python

Now it’s time to start working with this manual.

The sections 2.1 and 2.2 explain how Python and *ADwin* communicate with each other and deepen your knowledge for the *ADwin* concept.

In chapter 3, the installation and the integration of the new commands are described.

The general use of the Python driver is explained in chapter 4, its functions in chapter 5, which can also be used as a kind of reference documentation.

2.1 Interface for the development environment

The *ADwin*-Python driver is the interface for the Python development environment for the communication with *ADwin* systems.

The combination of Python with an *ADwin* system offers you totally new possibilities. The intelligence and computing power of an *ADwin* system on the one hand and the various Python functions for managing, analysis and documentation of measuring values on the other hand join into a powerful concept.

Typical applications are:

– Control of test stands
– Generating signals
– Measuring with intelligence, collecting data with complex trigger conditions
– Open-loop and closed-loop control
– Online processing, data reduction
– Hardware-in-the-loop, simulation of sensor signals

2.2 Communication with the *ADwin* system

With the development environment, you can control processes in the *ADwin*-system, as well as getting data from there or sending data. Your are programming processes with the real-time development tool *ADbasic*, create a binary file and transfer it to the *ADwin* system (see *ADbasic* manual or online help).

Data and instructions between Python and the *ADwin* system are processed according to the following illustration.

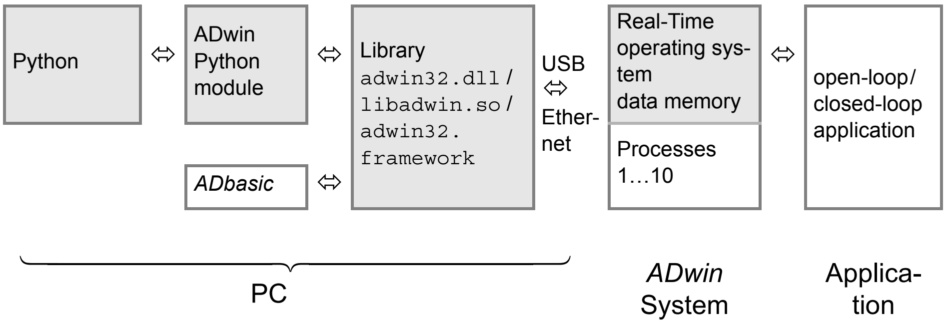


Fig. 1 – *ADwin*-Python Interface

adwin32.dll

The library (Windows: `adwin32.dll`, Linux: `libadwin.so`, Mac OS: `adwin32.framework`) is the main interface to the *ADwin* system for all applications and is therefore used by the *ADwin*-Python driver, too. With this interface, several programs can communicate with the *ADwin* system at the same time: Thus, various development environments, under Windows for example Python, *ADbasic* and *ADtools* can work with the *ADwin* system simultaneously.

Real-time operating system

The library functions communicate with the real-time operating system of the *ADwin* system. Therefore, you must load the operating system (e.g. the file `<adwin9.btl>`) after each power-up. After a successful loading the system will be able to receive and execute processes, receive instructions from the PC and exchange data with it. The processes programmed in *ADbasic*, include the program code for measurement, open-loop or closed-loop control of your application.

The real-time operating system performs the following tasks:

10 processes

– Management of up to 10 real-time processes with low or high priority (selectable). Processes with low priority can be interrupted by processes with high priority, the latter cannot be interrupted by other processes.

Data memory

– Providing global variables:
  - 80 integer variables (`Par_1` … `Par_80`), predefined.
  - 80 float variables (`FPar_1` … `FPar_80`), predefined.
  - 200 data arrays (`DATA_1` … `DATA_200`), length and data type can be set individually.

  You can read and change the values of these variables or data arrays at any time.

Communication

– Communication between *ADwin* system and PC (via program library).

  The communication process is running with medium priority on the *ADwin* system and can interrupt low-priority processes for a short time. It interprets and processes all instructions, which you send from the PC to the *ADwin* system: Control instructions and instructions for data exchange.

The following table shows examples for each group.

Control instructions, for example:	
`Load_Process`	transfers a process to the system.
`Start_Process`	starts a process.
Instructions for data exchange, for example:	
`Get_Par`	provides the current value of a parameter.
`Set_Par`	changes the value of a parameter.
`GetData_Long`	provides the value from a `DATA` array.

☞ The communication process never sends data to the PC without being asked to do so. This assures that data are transferred to the PC only if you have requested these data.

3 Installing the *ADwin* Driver for Python

3.1 Installing *ADwin*

For the installation you need an up-to-date *ADwin* CDROM.

3.1.1 Installation under Linux or Mac OS

Please follow the installation guide in the manual "ADwin Linux / Mac".

After successful installation you will find the files in the folders below `</opt/adwin/share>` (standard installation):

Driver and examples for Python	`./python`
Documentation for the *ADwin* module	`./doc/python`
Examples for *ADbasic*	`./examples/samples_ADwin`

Continue with chapter 3.2 "Installing the ADwin module".

3.1.2 Installation unter Windows

If *ADwin* is installed

If you have already installed an *ADwin* system and software skip this section and continue with chapter 3.2.

Else: New installation

Else, if an *ADwin* system is to be newly installed, please start the installation with the manual "*ADwin* installation", which is delivered with the *ADwin* hardware. It describes how to

– install the software from the *ADwin* CDROM.

– install the communciation driver under Windows.

– install the hardware in the PC (if necessary) and
set up the hardware connections between PC and *ADwin* system.

After successful installation you will find the files in folders below `<C:\ADwin>` (standard installation):

Driver and examples for Python	`.\Developer\Python\…`
Examples for *ADbasic*	`.\ADbasic\samples_ADwin`
Test program for *ADwin-Gold*, *ADwin-light-16* and plug-in boards	`.\Tools\Test\ADtest`
Test program for *ADwin-Pro*	`.\Tools\Test\ADpro`

Continue with the next section "Installing the ADwin module".

3.2 Installing the *ADwin* module

If you want to work with Python and the *ADwin* system you have to install the *ADwin* module.

Follow these steps:

– Enter the following in the command line:
  - Windows: `$> python setup.py install`
  - Linux / Mac: `$> python ./setup.py install`

  Now the *ADwin* module is copied into the folder `site-packages` of the Python installation.

– The *ADwin* module can be used now.

  With `import ADwin`, you include the *ADwin* module and with `adw = ADwin.ADwin(DeviceNo=1, raiseExceptions=1)` you create a new instance of the *ADwin* class with the name `adw`.

The functions of the driver are described in chapter 5. A list of functions in alphabetical order can be found in section A.3.

3.3 Accessing the *ADwin* system

During installation of hardware and software you have successfully checked the access to the *ADwin* system. Please use an example program from chapter A.1 in the annex to check the communication from Python to the *ADwin* system.

If the example program runs correctly all driver functions will operate properly with the *ADwin* system

3.4 Accessing an *ADwin* system via other PCs

If an *ADwin* system is connected to a host PC, but is not accessible within an Ethernet network directly, you can nevertheless get a connection using the program `ADwinTcpipServer`.

Detailed information about the use of `ADwinTcpipServer` is given on the program's online help.

4 General information about *ADwin* functions

4.1 Locating errors

There are 3 possibilities to locate errors upon execution of an *ADwin* function:

– Raising an exception upon run-time errors (exception handling)

  Each error provides an exception, which will be described in a separate program part.

  We recommend processing errors with this method.

– Explicitly querying error codes with `Get_Last_Error` (see next page)

  To handle each error get the error number after each access to the *ADwin* system.

  This method is recommended when you do not use any exceptions.

– Using the return value of functions (see next page)

  When using certain commands the return value contains an error code. It helps to make case differentiation in the program sequence.

  Since not all instructions return an error code, a complete error handling is not possible.

4.1.1 Raising an exception

You can configure the *ADwin*-Python module so that an exception is raised with the name `ADwinError` when run-time errors occur. Structure the program as follows:

```
# Provide ADwin functions and error routines
from ADwin import ADwin, ADwinError

# make an instance of the ADwin class, raise exception
adw = ADwin(DeviceNo=1, raiseExceptions=1)

# error handling with try / except
try:
    …                   # the Python program

except ADwinError as e:
    print('An AdwinError occurred: ', e)
```

The class attribute `raiseExceptions` determines the action of the module during a run-time error. With the value 1, the module generates an exception when a run-time error occurs. If you prevent exceptions using the value 0, you have to to query explicitly the error code after each access to the *ADwin* system (see below).

In the program section, `try` you enter the Python program that also contains the accesses to the *ADwin* system. If an error occurs this part of the program will be executed with the exception attribute `ADwinError`.

4.1.2 Explicitly querying error codes

If you configure the class attribute `raiseExceptions` in such a way that it does not raise any exceptions during run-time errors, you have to query the error number with the function `Get_Last_Error` after each access to the *ADwin* system.

For each error number you will get the text with the functions `Get_Error_Text`. A list of all error messages is in section A.2 in the annex.

The functions `Get_Last_Error` and `Get_Error_Text` are described from page 38 onward.

In the following example, the undefined array `DATA_1` is accessed; the occurring error does not raise an exception (due to the setting of `raiseExceptions` = 0). Instead, the error is queried with `Get_Last_Error`:

```
>>> import ADwin
>>> adw=ADwin.ADwin(DeviceNo=1, raiseExceptions=0)
>>> a = adw.GetData_Long(1,1,10)
>>> adw.Get_Error_Text(adw.Get_Last_Error())
'The Data is too small.'
```

4.1.3 Using the return value of functions

The return value of some functions contains an error code, which you can use to differentiate in the program sequence.

Important:

- If run-time errors raise an exception (see chapter 4.1.1), the functions do not return an error code (type `noneType`).
- The functions use different values to indicate an error.
- The returned error code has nothing to do with the list of error messages in the annex.
- The return value is not always unambigous. If for instance `Get_Processdelay` returns the value 255, it is not quite clear if an error has ocurred or if the parameter `Processdelay` contains the value 255.

  The return value of the following functions is not unambigous, that means it can be understood as error or as value:
  - `Fifo_Empty`
  - `Fifo_Full`
  - `Get_Par`
  - `Get_FPar`
  - `Get_Processdelay`
  - `Free_Mem`

  For an explicit error handling you have to raise an exception or to query error codes (see above).

4.2 The "DeviceNo."

A "Device No." is the number of a specified *ADwin* system connected to a PC. An *ADwin* system is always accessed via the "Device No."

The "Device No." for the *ADwin* system is generated with the program *ADconfig*. You will find more information about the program's usage in the online help of *ADconfig*. Under Windows, an online help is available, under Linux you call the help with:

`adconfig --help` or
`man /opt/adwin/share/man/man8/adconfig.8`

You indicate the `DeviceNo` when you generate an instance of an *ADwin* class; the default value is 1. That is, you generate an instance for each *ADwin* system.

4.3 Data types

The functions and parameters of the *ADwin*-Python driver use the Python standard data types `int`, `float`, and `str` as well as some C compatible types of the library `ctypes`.

In contrast to Python, *ADbasic* usually uses the following data types:

Data type	Definition
`String`	unsigned integer 32 Bit
`Long`	signed integer 32 Bit
`Float` (until T11) `Float32`	float 32 Bit
`Float64`	float 64 Bit

With 32-Bit floating-point values until processor T11, bit patterns of invalid values in the *ADwin* hardware are converted during transfer to the PC into different values, see following table. Numbers inside the valid value range (normalized numbers) stay unchanged.
With processor T12, the IEEE denominations as `#INF` are displayed.

IEEE denomination	Bit pattern area	Value or display on the PC
+0	00000000h	0
Positive denormalized numbers	00000001h 007FFFFFh	0
Positive normalized numbers	00800000h 7F7FFFFFh	+1,175494 · 10-38 +3,402823 · 10+38
+∞ (Infinity, #INF)	7F800000h	3.402823E+38
Signaling Not a number (SNaN)	7F800001h 7FBFFFFFh	3.402823E+38
Quite Not a number (QNaN)	7FC00000h 7FFFFFFFh	3.402823E+38
-0	80000000h	0

IEEE denomination	Bit pattern area	Value or display on the PC
Negative denormalized numbers	`80000001h` `807FFFFFh`	0
Negative normalized numbers	`80800000h` `FF7FFFFFh`	-1,175494 · 10-38 -3,402823 · 10+38
-∞ (Infinity, #INF)	`FF800000h`	3.402823E+38
Signaling Not a number (SNAN)	`FF800001h` `FFBFFFFFh`	3.402823E+38
Indeterminate	`FFC00000h`	3.402823E+38
Quite Not a number (QNAN)	`FFC00001h` `FFFFFFFFh`	3.402823E+38

4.4 2-dimensional arrays

In *ADbasic*, global `DATA` arrays can be declared 2-dimensional (2D). But the functions of the *ADwin*-Python driver here use only one-dimensional arrays.

In general, there the following rule applies for the relation of an element in a 2D-array from *ADbasic* to an element in a 1D-array from Python:

ADbasic	Python
`DATA_n[i][j]`	`array(s·(i-1)+j-1)`

Here `s` is the second dimension of `DATA_n` when declared in *ADbasic*.

As an example a 2D-array may be declared in *ADbasic* as follows

```
DIM DATA_8[7][3] AS FLOAT 'i.e. s=3
```

The 7×3 elements of the array are read in Python with `GetData_Float`:

```
# transfer elements 1…21 from DATA_8 into array
array = adw.GetData_Float(8,1,21)
```

The data are transferred in the following order: Please note, that in Python array indexes begin at 0, but in *ADbasic* at 1:

Index of `DATA_8`	[1][1]	[1][2]	[1][3]	[2][1]	…	[7][1]	[7][2]	[7][3]
Index of `array`	[0]	[1]	[2]	[3]	…	[18]	[19]	[20]

Thus, the function `GetData_Float` returns element `DATA_8[7][2]` in `array[19]`.

With `s=3`, the general rule results to:

ADbasic	Python
`DATA_n[`**1**`][`**1**`]`	`array[3·(`**1**`-1]+`**1**`-1]  = array[0]`
`DATA_n[`**1**`][`**2**`]`	`array[3·(`**1**`-1]+`**2**`-1]  = array[1]`
…	… …
`DATA_n[`**7**`][`**2**`]`	`array[3·(`**7**`-1]+`**2**`-1]  = array[19]`
`DATA_n[`**7**`][`**3**`]`	`array[3·(`**7**`-1]+`**3**`-1]  = array[20]`

5 Description of the *ADwin* functions

The description of the functions is divided into the following sections:

– System control and system information, page 12
– Process control, page 15
– Transfer of global variables, page 19
– Transfer of data arrays, page 24
– Querying the error code, page 38

In annex A.3, you find an overview of all functions.

Please pay attention to chapter 4, where general aspects for the use of *ADwin* functions are described.

Instructions for accessing analog and digital inputs and outputs are not included in the *ADwin*-Python driver. These applications can be programmed in *ADbasic*.

Program structure

The following program structure belongs to each Python program:

```
# Provide ADwin functions and error routines
from ADwin import ADwin, ADwinError

# Generate an instance of the ADwin class, raise exceptions
adw = ADwin(DeviceNo=1, raiseExceptions=1)

# Error handling with try / except (see chapter 4.1)
try:
    …                      # program section to be monitored

except ADwinError as e:
   print('An ADwinError occurred: ', e)
```

It is assumed that with `adw` an instance of the *ADwin* class has already been generated when using the examples of the *ADwin* functions.

5.1 System control and system information

Initialization of the *ADwin* system and information about the operating status.

Boot

`Boot` initializes the *ADwin* system and loads the operating system.

```
Boot(str Filename)
```

Parameters

`Filename` Path and file name of the operating system file (see table below)

Notes

The initialization deletes all processes on the system and sets all global variables to 0.

The operating system file to be loaded depends on the processor type of the system you want to communicate with. The following table shows the file names for the different processors.

The files are located in the directory `<C:\ADwin\>` or `/opt/adwin/share/btl/`, which can be called with the class attribute `adw.ADwindir`.

Processor	Operating system file
T9	`ADwin9.btl`
	`ADwin9s.btl`[1]
T10	`ADwin10.btl`
T11	`ADwin11.btl`
T12	`ADwin12.btl`
T12.1	`ADwin121.btl`

You can also use the processors T2...T8; in this case call our support (address on the back of the cover page).

The computer will only be able to communicate with the *ADwin* system after the operating system has been loaded. Load the operating system again after each power up of the *ADwin* system.

Loading the operating system with `Boot` takes about one second, with T12.1 about six seconds. As an alternative you can also load the operating system via *ADbasic* development environment. (Icon B).

Example

Please note the advice about the program structurte on page 11.

```
# Load operating system for T10 processor
adw.Boot(adw.ADwindir + '\\ADwin10.btl')
```

1. Optimized operating system with a slightly smaller memory requirement.

Test_Version

`Test_Version` tests, if the right operating system is loaded for the processor and if the processor can be accessed.

```
int Test_Version()
```

Parameters

Return value 0: OK
≠0:error.

Example

```
print('Test_Version:', adw.Test_Version())
```

Processor_Type

`Processor_Type` returns the processor type of the system.

```
int Processor_Type()
```

Parameters

Return value Parameter for the processor of the system.

0: Error	9: T9
2: T2	1010: T10
4: T4	1011: T11
5: T5	1012: T12
8: T8	10121: T12.1

Example

```
print('Processor_Type:', adw.Processor_Type())
```

Workload

`Workload` returns the average processor workload since the last call of `Workload`.

```
int Workload()
```

Parameters

Return value 0…100: Processor workload (in percent)
255: Error

Notes

The processor workload is evaluated for the period between the last and the current call of `Workload`. If you need the current processor workload, you must call the function twice and in a short time interval (approx. 1 ms).

Example

```
print('Workload:', adw.Workload())
```

Free_Mem

`Free_Mem` returns the free memory of the system for the different memory types.

```
int Free_Mem(int Mem_Spec)
```

Parameters

`Mem_Spec`	Memory type: 0 : all memory types (T2, T4, T5, T8 only) 1 : internal program memory (PM_LOCAL); T9…T11 2: internal data memory (EM_LOCAL); T11 only 3 : internal data memory (DM_LOCAL); T9…T11 4 : external DRAM memory (DRAM_EXTERN); T9…T11 5 : Memory, which can provide data to the cache (CACHE-ABLE); T12/T12.1 only. 6 : Memory, which cannot provide data to the cache (UNCACHEABLE); T12/T12.1 only.
Return value	≠255: Usable free memory in Byte, with `Mem_Spec`=5/6 in kByte. 255: Possible error.

Example

```
# Query the free memory (Cacheable)
print('Free_Mem:', adw.Free_Mem(5), 'k7Bytes')
```

5.2 Process control

Instructions for the control of single processes on the *ADwin* system.

There are the processes 1…10 and 15. The processes have the following functions:

– 1…10: You write the process in *ADbasic* yourself.

– 15: Controls flashing of the LED on *ADwin-Gold* or *ADwin-Pro*.

Process 15 is part of the operating system and is started automatically after booting. For detailed information see manual *ADbasic*, chapter "Process Management".

Load_Process

`Load_Process` loads the binary file of process to the *ADwin* system.

```
Load_Process(str Filename)
```

Parameters

`Filename`	Path and file name of the binary file to be loaded.

Notes

You generate binary files in *ADbasic* with "`Build` ▸ `Make Bin file`".

If you switch off your *ADwin* system all processes are deleted: Load the necessary processes again after power-up

You may load up to 10 processes to an *ADwin* system. Running processes are not influenced by loading additional processes (with different process numbers).

Before loading the process into the *ADwin* system, you have to ensure that no process using the same process number is already running. If there is such a process yet, you first have to stop the running process using `Stop_Process`.

If you load processes more than once, memory fragmentation can happen. Please note the appropriate hints in the *ADbasic* manual.

Example

```
# Load the Testprg.T91 file: Processor T9, Process no. 1
# The file Testprg.T91 can be found in the current
directory
adw.Load_Process('Testprg.T91')
```

Start_Process

`Start_Process` starts a process.

`Start_Process(int ProcessNo)`

Parameters

`ProcessNo` Process number (1...10, 15).

Notes

`Start_Process` has no effect, if you indicate the number of a process, which
- is already running orr
- has the same number as the calling processor or
- has not yet been loaded to the *ADwin* system.

Example

```
adw.Start_Process(1)# start Process 1
```

Stop_Process

`Stop_Process` stops a process.

`Stop_Process(int ProcessNo)`

Parameters

`ProcessNo` Process number (1...10, 15).

Notes

The function has no effect, if you indicate the number of a process, which
- has already been stopped or
- has not yet been loaded to the *ADwin* system.

Example

```
adw.Stop_Process(2)# stops process 2
```

Clear_Process

`Clear_Process` deletes a process from memory.

`Clear_Process(int ProcessNo)`

Parameters

`ProcessNo` Process number (1...10, 15).

Notes

Loaded processes need memory space in the system. With `Clear_Process`, you can delete processes from the program memory to get more space for other processes.

If you want to delete a process, proceed as follows:
- Stop the running process with `Stop_Process`. A running process cannot be deleted.
- Check with `Process_Status`, if the process has really stopped.
- Delete the process from the memory with `Clear_Process`.

Process 15 in Gold and Pro systems is responsible for flashing the LED; after deleting this process the LED does not flash any more.

Please note, that clearing processes can lead to memory fragmentation. You find more information in the *ADbasic* manual, section "Memory fragmentation".

Example

```
# Delete process 2 from memory.
# Declared DATA and FIFO arrays remain.
adw.Stop_Process(2)
while adw.Process_Status(2) <> 0:
    pass
adw.Clear_Process(2)
```

Process_Status

`Process_Status` returns the status of a process.

`int **Process_Status**(int ProcessNo)`

Parameters

`ProcessNo`	Process number (1...10, 15).
Return value	Status of the process: 1 : Process is running. 0 : Process is not running, that means, it has not been loaded, started or stopped. <0:Process is being stopped, that means, it has received `Stop_Process`, but still waits for the last event.

Example

```
# Return the status of process 2
print('Process_Status 2:', adw.Process_Status(2))
```

Get_Processdelay

`Get_Processdelay` returns the parameter Processdelay for a process.

`int **Get_Processdelay**(int ProcessNo)`

Parameters

`ProcessNo`	Process number (1...10).
Return value	≠255: The currently set value ($1…2^{31}$-1) for the parameter `Processdelay` of a process. 255: Possible error or value of `Processdelay`. Please note chapter 4.1.3.

Notes

The parameter `Processdelay` controls the time interval between two events of a time-controlled process (see `Set_Processdelay` as well as the manual or online help of *ADbasic*).

The parameter `Processdelay` replaces the former parameter `Globaldelay`.

Example

```
# Get Processdelay of the ADbasic process 1
print('Processdelay 1:', adw.Get_Processdelay(1))
```

Set_Processdelay

Set_Processdelay sets the parameter Processdelay for a process.

```
Set_Processdelay(int ProcessNo, int Processdelay)
```

Parameters

ProcessNo	Process number (1...10).
Process-delay	Value (1…2^{31}-1) to be set for the parameter Processdelay of the process (see table below).

Notes

The parameter Processdelay controls the time interval between two events of a time-controlled process (see manual or online help *ADbasic*).

For each process there is a minimum time interval: If you fall below the minimum time interval you will get an overload of the *ADwin* processor and communication will fail.

The time interval is specified in a time unit that depends on processor type and process priority:

Processor type	Process priority	
	high	low
T9	25 ns	100 µs
T10	25 ns	50 µs
T11	3.3 ns	0.003 µs = 3.3 ns
T12	1 ns	1 ns
T12.1	1,5 ns	1,5 ns

Example

```
# Set Processdelay 2000 of process 1.
adw.Set_Processdelay(1,2000)
```

If process 1 is time-controlled, has high priority and runs on a T9 processor, process cycles are called every 50 µs (=2000*25 ns).

5.3 Transfer of global variables

Instructions for data transfer between PC and *ADwin* system with the pre-defined global variables Par_1 ... Par_80 and FPar_1 ... FPar_80.

5.3.1 Global variables Par_1 ... Par_80

The global variables Par_1 ... Par_80 have the following range of values:

Par_1 ... Par_80: -2147483648 … +2147483647
$= -2^{31} \dots +2^{31}-1$

Set_Par

Set_Par transfers an integer value of 32 bit precision to a global variable Par.

```
Set_Par(int Index, int Value)
```

Parameters

Index	Number (1 … 80) of a global variable Par_1 … Par_80.
Value	Value to be set for the variable.

Example

```
# Set values of all LONG variables
for i in range(1, 81): adw.Set_Par(i, i)
```

Get_Par

Get_Par returns the value of a global variable Par as integer value of 32 bit precision.

```
int Get_Par(int Index)
```

Parameters

Index	Number (1 … 80) of a global variable Par_1 … Par_80.
Return value	≠255: Current value of the variable. 255: Possible error or value of the variable. Please note chapter 4.1.3.

Example

```
# Read the values of the variables Par_1..Par_10
for i in range(1,11): print(adw.Get_Par(i))
```

Get_Par_Block

`Get_Par_Block` returns the values of several global variables `Par` as integer values of 32 bit precision in an array.

```
ctypes.c_int32_Array Get_Par_Block(int StartIndex,
 int Count)
```

Parameters

`StartIndex`	Number (1 … 80) of the global variable `Par_1` … `Par_80`, to be transferred first.
`Count`	Number (≥1) of values to be transferred.
Return value	Destination array for the transferred values.

Example

Read the values of the variables `Par_10` … `Par_39` and store in `ArrayLong` starting from element 0:

```
ArrayLong = adw.Get_Par_Block(10,30)
```

Get_Par_All

`Get_Par_All` returns the values of all global variables `Par_1` … `Par_80` as integer values of 32 bit precision in an array.

```
ctypes.c_int32_Array Get_Par_All()
```

Parameters

Return value	Destination array for the transferred values.

Example

Read the values of the variables `Par_1` … `Par_80` and store in `ArrayLong`

```
ArrayLong = adw.Get_Par_All()
```

Note: Since the indexing of Python arrays begins at 0 `Par_9` for instance is found in `ArrayLong[8]`.

5.3.2 Global variables `FPar_1` … `FPar_80`

The global variables `FPar_1` … `FPar_80` have the following range of values, according to the *ADwin* processor:

T9…T11 (32 Bit):	negative: $-3.402823 \cdot 10^{+38} \ldots -1.175494 \cdot 10^{-38}$ positive: $+1.175494 \cdot 10^{-38} \ldots +3.402823 \cdot 10^{+38}$
T12/T12.1 (64 Bit):	negative: $-1.797693134862315 \cdot 10^{+308} \ldots -2.2250738585072014 \cdot 10^{-308}$ positive: $+2.2250738585072014 \cdot 10^{-308} \ldots +1.797693134862315 \cdot 10^{+308}$

Set_FPar

`Set_FPar` transfers a float value of 32 bit precision to a global variable `FPar`.

```
Set_FPar(int Index, float Value)
```

Parameters

`Index` Number (1 … 80) of a global variable `FPar_1` … `FPar_80`.

`Value` Value to be set for the variable.

Example

```
# Set variable FPar_6 to 34.7
adw.Set_FPar(6, 34.7)
```

Set_FPar_Double

`Set_FPar` transfers a float value of 64 bit precision to a global variable `FPar`.

```
Set_FPar_Double(int Index, float Value)
```

Parameters

`Index` Number (1 … 80) of a global variable `FPar_1` … `FPar_80`.

`Value` Value to be set for the variable.

Notes

With processor types until T11, the destination variable of the *ADwin* system has 32 bit precision only.

Example

```
# Set variable FPar_6 to 34.7
adw.Set_FPar(6, 34.7)
```

Get_FPar

`Get_Par` returns the value of a global variable `FPar` as float value of 32 bit precision.

```
float Get_FPar(int Index)
```

Parameters

`Index` Number (1 … 80) of a global variable `FPar_1` … `FPar_80`.

Return value ≠255.0: Current value of the variable.
255.0: Possible error or value of the variable.
Please note chapter 4.1.3.

Example

```
# Read the value of the variable FPar_56
print('FPar_56:', adw.Get_FPar(56))
```

Get_FPar_Block

`Get_FPar_Block` returns the values of several global variables `FPar` as float values of 32 bit precision in an array.

```
ctypes.c_float_Array Get_FPar_Block(int StartIndex,
 int Count)
```

Parameters

`StartIndex`	Number (1 … 80) of the first global variable `FPar_1` … `FPar_80` to be transferred.
`Count`	Number (≥1) of values to be transferred.
Return value	Destination array for the transferred values.

Example

Read the values of the variables `Par_10` … `Par_34` and store in `ArrayFloat` starting from element 0:

```
ArrayFloat = adw.Get_FPar_Block(10,25)
```

Get_FPar_All

`Get_FPar_All` returns the values of all global variables `FPar_1` … `FPar_80` as float values of 32 bit precision in an array.

```
ctypes.c_float_Array Get_FPar_All()
```

Parameters

Return value	Destination array for the transferred values.

Example

Read the values of the variables F`Par_1` … `FPar_80` and store in `ArrayFloat` starting from element 1:

```
ArrayFloat = adw.Get_FPar_All()
```

Get_FPar_Double

`Get_Par_Double` returns the value of a global variable `FPar` as float value of 64 bit precision.

```
float Get_FPar_Double(int Index)
```

Parameters

`Index`	Number (1 … 80) of a global variable `FPar_1` … `FPar_80`.
Return value	≠255.0: Current value of the variable. 255.0: Possible error or value of the variable. Please note chapter 4.1.3.

Notes

Until T11, please note: float values in the *ADwin* system have single precision. You should therefore display `FPAR` values only with single precision to avoid misunderstandings.

Example

```
# Read the value of the FLOAT variable FPar_56
print('FPar_56:', adw.Get_FPar(56))
```

Get_FPar_Block_Double

`Get_FPar_Block_Double` returns the values of several global variables `FPar` as float values of 64 bit precision in an array.

```
ctypes.c_float_Array Get_FPar_Block_Double(
 int StartIndex, int Count)
```

Parameters

`StartIndex`	Number (1 … 80) of the first global variable `FPar_1` … `FPar_80` to be transferred.
`Count`	Number (≥1) of values to be transferred.
Return value	Destination array for the transferred values.

Notes

Until T11, please note: float values in the *ADwin* system have single precision. You should therefore display `FPAR` values only with single precision to avoid misunderstandings.

Example

Read the values of the variables `FPar_10` … `FPar_34` and store in `ArrayFloat` starting from element 0:

```
ArrayFloat = adw.Get_FPar_Block(10,25)
```

Get_FPar_All_Double

`Get_FPar_All_Double` returns the values of all global variables `FPar_1` … `FPar_80` as float values of 64 bit precision in an array.

```
ctypes.c_float_Array Get_FPar_All_Double()
```

Parameters

Return value	Destination array for the transferred values.

Notes

Until T11, please note: float values in the *ADwin* system have single precision. You should therefore display `FPAR` values only with single precision to avoid misunderstandings.

Example

Read the values of the variables `Par_1` … `Par_80` and store in `ArrayFloat` starting from element 1:

```
ArrayFloat = adw.Get_FPar_All_Double()
```

5.4 Transfer of data arrays

Instructions for data transfer between PC and *ADwin* system with global `DATA` arrays (`DATA_1`…`DATA_200`):

– Data arrays

– FIFO arrays

– Data arrays with string data

You have to declare each array before using it in *ADbasic* (see *ADbasic* manual).

5.4.1 Data arrays

Declare each array before using it in *ADbasic* with
`DIM` `DATA_x` **`AS LONG/FLOAT/FLOAT32/FLOAT64`**

The value range of an array element depends on the data type:

– **`LONG`**	-2147483648 … +2147483647
– **`FLOAT`** (bis T11), **`FLOAT32`**	negative: $-3.402823 \cdot 10^{+38}$ … $-1.175494 \cdot 10^{-38}$ positive: $+1.175494 \cdot 10^{-38}$ … $+3.402823 \cdot 10^{+38}$
– **`FLOAT64`**	negative: $-1.797693134862315 \cdot 10^{+308}$ … $-2.2250738585072014 \cdot 10^{-308}$ positive: $+2.2250738585072014 \cdot 10^{-308}$ … $+1.797693134862315 \cdot 10^{+308}$

Data_Length

`Data_Length` returns the length of a **`LONG`**, **`FLOAT`**/**`FLOAT64`** or **`STRING`** array, declared in *ADbasic*, that means the number of elements.

```
int Data_Length(int DataNo)
```

Parameters

`DataNo`	Number (1...200) of an array `Data_1`…`Data_200`.
Return value	>0: Declared length of the array (= number of elements) 0: Error, array is not declared -1: Other error

Notes

To determine the length of a string in a **`STRING`** `DATA` array you use the instruction `String_Length`.

Example

In *ADbasic*, `DATA_2` is dimensioned as:

```
DIM DATA_2[2000] AS LONG
```

In Python, the length of the array `DATA_2` is determined as follows:

```
print('length Data_2: ', adw.Data_Length(2))
```

Data_Type

`Data_Type` returns the data type of a `DATA` array declared in *ADbasic*.

`(int, str) Data_Type(int DataNo)`

Parameter

`DataNo`	Array number (1...200)
Return value	Tuple of 2 values (see list below): – Key value (0…8) for the *ADbasic* data type of the array, referring to the processor type. – String for the *ADbasic* data type of the array, referring to the processor type.

Notes

Data type of the array in *ADbasic*	Processor type				
	T9	T10	T11	T12	T12.1
(Byte)	–	–	–	(1)	(1)
(Short)	(2)	(2)	(2)	(2)	(2)
(Integer)	(3)	(3)	(3)	(4)	(4)
Long	4	4	4	4	4
Float (until T11) / Float32 (T12/T12.1 only)	5	5	5	5	5
Float64	–	–	–	6	6
String	3	3	3	8	8

The data types **`BYTE`**, **`SHORT`**, and **`INTEGER`** in *ADbasic* are restricted in use and only listed for the sake of completeness; in the table, these data types are therefore set in parentheses.

Example

In *ADbasic*, `DATA_2` is dimensioned as:

```
DIM DATA_2[2000] AS Float64
```

In C, your receive the data type of array `DATA_2`:

```
ret_val = Data_Type(2,NULL,1,&ErrorNo); // ret_val=6
```

SetData_Long

SetData_Long transfers integer values of 32 bit precision into a DATA array of the *ADwin* hardware.

```
SetData_Long(list|array|ctypes.c_int32_Array
 PC_Array, int DataNo, int Startindex, int Count)
```

Parameters

PC_Array	Source array, from which values are transferred.
DataNo	Number (1...200) of destination array DATA_1 … DATA_200.
StartIndex	Number (≥1) of the first element in the destination array, into which data is transferred.
Count	Number (≥1) of values to be transferred.

Example

Transfer first 100 elements of the source array ArrayLong into the elements 30...129 of the destination array DATA_3:

```
dataType = ctypes.c_int32 * 100
ArrayLong = dataType(0)
for i in range(100): ArrayLong[i] = i+100
adw.SetData_Long(ArrayLong, 3, 1, 100)
```

GetData_Long

GetData_Long transfers values from a DATA array of the *ADwin* hardware as integer values of 32 bit precision into an array.

```
ctypes.c_int32_Array GetData_Long(int DataNo,
 int Startindex, int Count)
```

Parameters

DataNo	Number (1...200) of source array DATA_1 … DATA_200.
StartIndex	Number (≥1) of the first element in the source array to be transferred.
Count	Number (≥1) of values to be transferred.
Return value	Newly created destination array that contains the transferred values.

Example

Transfer elements 1…100 from source array DATA_2 into the destination array ArrayLong starting from index 0:

```
ArrayLong = adw.GetData_Long(2,1,100)
```

SetData_Float

SetData_Float transfers float values of 32 bit precision into a DATA array of the *ADwin* hardware.

```
SetData_Float(list|array|ctypes.c_float_Array PC_Ar-
ray, int DataNo, int Startindex, int Count)
```

Parameters

PC_Array	Source array, from which data are transferred.
DataNo	Number (1...200) of destination array DATA_1 … DATA_200.
StartIndex	Number (≥1) of the first element in the destination array, into which data is transferred.
Count	Number (≥1) of values to be transferred.

Example

Transfer first 80 elements of the source array ArrayFloat into the elements 20…99 of the destination array DATA_3:

```
dataType = ctypes.c_float * 80
ArrayFloat = dataType(0)
for i in range(80): ArrayFloat[i] = i+100.1234
adw.SetData_Float(ArrayFloat, 3, 20, 80)
```

GetData_Float

GetData_Float transfers values from a DATA array of the *ADwin* hardware as float values of 32 bit precision into an array.

```
ctypes.c_float_Array GetData_Float(int DataNo,
 int Startindex, int Count)
```

Parameters

DataNo	Number (1...200) of source array DATA_1 … DATA_200.
StartIndex	Number (≥1) of the first element in the source array to be transferred.
Count	Number (≥1) of values to be transferred.
Return value	Newly created destination array that contains the transferred values.

Example

Transfer elements 1…100 from source array DATA_2 into the destination array ArrayFloat starting from index 0:

```
ArrayFloat =adw.GetData_Float(2,1,100)
```

SetData_Double

`SetData_Double` transfers float values of 64 bit precision into a `DATA` array of the *ADwin* hardware.

```
SetData_Double(list|array|ctypes.c_double_Array
 PC_Array, int DataNo, int Startindex, int Count)
```

Parameters

`PC_Array`	Source array, from which data are transferred.
`DataNo`	Number (1...200) of destination array `DATA_1` … `DATA_200`.
`StartIndex`	Number (≥1) of the first element in the destination array, into which data is transferred.
`Count`	Number (≥1) of values to be transferred.

Notes

Until T11, please note: float values in the *ADwin* system have single precision. You should therefore display `FPAR` values only with single precision to avoid misunderstandings.

Example

Transfer first 80 elements of the source array `ArrayDouble` into the elements 20…99 of the destination array `DATA_3`:

```
dataType = ctypes.c_double * 80
ArrayDouble = dataType(0)
for i in range(80): ArrayDouble[i] = i+100.1234
adw.SetData_Double(ArrayDouble, 3, 20, 80)
```

GetData_Double

`GetData_Double` transfers values from a `DATA` array of the *ADwin* hardware as float values of 64 bit precision into an array.

```
ctypes.c_double_Array GetData_Double(int DataNo,
 int Startindex, int Count)
```

Parameters

`DataNo`	Number (1...200) of source array `DATA_1` … `DATA_200`.
`StartIndex`	Number (≥1) of the first element in the source array to be transferred.
`Count`	Number (≥1) of values to be transferred.
Return value	Newly created destination array that contains the transferred values.

Notes

Until T11, please note: float values in the *ADwin* system have single precision. You should therefore display `FPAR` values only with single precision to avoid misunderstandings.

Example

Transfer elements 1…100 from source array `DATA_2` into the destination array `ArrayDouble` starting from index 0:

```
ArrayDouble =adw.GetData_Double(2,1,100)
```

Data2File

`Data2File` saves values from a `DATA` array of the *ADwin* system to a file (on the hard disk).

```
Data2File (str Filename, int DataNo, int Startindex,
    int Count, int Mode)
```

Parameters

`Filename`	Pointer to path and file name. If no path is indicated, the file is saved in the project directory.
`DataNo`	Number (1...200) of the source array `DATA_1` … `DATA_200`.
`Startindex`	Number (≥1) of the element in the source array to be transferred first.
`Count`	Number (≥1) of values to be transferred.
`Mode`	Write mode: 0: File will be overwritten. 1: Data is appended to an existing file.

Notes

The `DATA` array must not be defined as **`FIFO`**.

The data are saved as binary file. The data type (Long, Float32, Float64) is adapted to the `DATA` array being read.

If not existing, the file will be created.

Example

Save elements 1…1000 from the *ADbasic* array `DATA_1` into the file `<Test.dat>` in the project direcory:

```
Data2File('Test.dat', 1, 1, 1000, 0);
```

File2Data

`File2Data` copies values from a file (on the hard disk) into a `DATA` array of the *ADwin* system.

```
File2Data (str Filename, int DataType, int DataNo,
    int Startindex)
```

Parameters

`Filename`	Pointer to path and source file name. If no path is indicated, the file is searched for in the project directory.
`DataType`	Data type of values saved in the file. Select one of the following contants: `2`: Values of type integer (32 bit, signed). `5`: Values of type float (32 bit). `6`: Values of type float (64 bit).
`DataNo`	Number (1…200) of the destination array `DATA_1` … `DATA_200`.
`Startindex`	Index (≥1) of the element in the destination array to be written first.

Notes

The file values are expected to be saved as binary in one of the formats Long, Float or Double, and `DataType` must be set appropriately.

The `DATA` array must not be defined as **`FIFO`**. The array must be dimensioned great enough to hold all values of the file.

If the destination array has a different data type than `DataType`, the values of the source file are converted into the destination data type. According to the processor typ, there are the destination data types Long, Float32 and Float64.

Example

In *ADbasic*, `DATA_1` is dimensioned as:

```
DIM DATA_1[1000] AS LONG
```

Copy values of type long from file `<Test.dat>` in the project direcory into the *ADbasic* array `DATA_1`, starting from element `DATA_1[20]`. The file may contain up to 980 values as to not exceed the `DATA_1` array size.

```
File2Data('Test.dat', TYPE_LONG, 1, 20);
```

5.4.2 FIFO arrays

Instructions for data transfer between PC and *ADwin* system with global `DATA` arrays (`DATA_1…DATA_200`), which are declared as FIFO.

You must declare each FIFO array before using it in ADbasic (see "*ADbasic*" manual): **DIM** `DATA_x[n]` **AS** `TYPE` **AS FIFO**

The value range of a FIFO array element depends on the data type:

- **LONG** -2147483648 … +2147483647
- **FLOAT** (bis T11), **FLOAT32** negative: $-3.402823 \cdot 10^{+38} \ldots -1.175494 \cdot 10^{-38}$
  positive: $+1.175494 \cdot 10^{-38} \ldots +3.402823 \cdot 10^{+38}$
- **FLOAT64** negative: $-1.797693134862315 \cdot 10^{+308} \ldots -2.225073858507201 4 \cdot 10^{-308}$
  positive: $+2.2250738585072014 \cdot 10^{-308} \ldots +1.797693134862315 \cdot 10^{+308}$

To ensure that the FIFO is not full, the **FIFO_EMPTY** function should be used before writing into it. Similarly, the **FIFO_FULL** function should be used to check if there are values, which have not yet been read, before reading from the FIFO.

Fifo_Empty

`Fifo_Empty` returns the number of empty elements in a FIFO array.

```
int Fifo_Empty(int FifoNo)
```

Parameters

`FifoNo` Number (1...200) of FIFO array `DATA_1` … `DATA_200`.

Return value ≠255: Number of empty elements in the FIFO array.
255: Possible error or number of empty elements.
Please note chapter 4.1.3.

Example

In *ADbasic*, `DATA_5` is dimensioned as:

```
DIM DATA_5[100] AS LONG AS FIFO
```

In Python, you will get the number of empty elements (≤100) in `DATA_5`:

```
print('Fifo_Empty 5:', adw.Fifo_Empty(5))
```

Fifo_Full

`Fifo_Full` returns the number of used elements in a FIFO array.

```
int Fifo_Full(int FifoNo)
```

Parameters

`FifoNo` Number (1...200) of FIFO array `DATA_1` … `DATA_200`.

Return value ≠255: Number of the used elements in the FIFO array.
255: Possible error or number of used elements.
Please note chapter 4.1.3.

Example

In *ADbasic*, `DATA_12` is dimensioned as:

```
DIM DATA_12[2500] AS FLOAT AS FIFO
```

In Python, you get the number of used elements (≤2500) in `DATA_12`:

```
print('Fifo_Full 12:', adw.Fifo_Full(12))
```

Fifo_Clear

`Fifo_Clear` initializes the write and read pointers of a FIFO array. Afterwards, the data in the FIFO array are no longer available.

```
Fifo_Clear(int FifoNo)
```

Parameters

`FifoNo`	Number (1...200) of FIFO array `DATA_1` … `DATA_200`.

Notes

During start-up of an *ADbasic* program the FIFO pointers of an array are not initialized automatically. We therefore recommend calling `Fifo_Clear` at the beginning of your *ADbasic* program.

Initializing the FIFO pointers during program run is useful, if you want to clear all data of the array (because of a measurement error for instance).

Example

```
# Clear data in the FIFO array DATA_45
adw.Fifo_Clear(45)
```

SetFifo_Long

`SetFifo_Long` transfers integer values of 32 bit precision ifrom the PC into a FIFO array of the *ADwin* hardware.

```
SetFifo_Long(int FifoNo,
 list|array|ctypes.c_int32_Array PC_Array, int
Count)
```

Parameters

`FifoNo`	Number (1...200) of FIFO array `DATA_1` … `DATA_200`.
`PC_Array`	Source array, from which values are transferred.
`Count`	Number (≥1) of transferred values.

Example

Check FIFO destination array `DATA_12` for enough free elements and transfer 1000 elements from the source array `ArrayLong`:

```
dataType = ctypes.c_int32 * 1000
ArrayLong = dataType(0)
for i in range(1000): ArrayLong[i] = i
if adw.Fifo_Empty(12) >= 1000:
    adw.SetFifo_Long(12, ArrayLong, 1000)
```

GetFifo_Long

`GetFifo_Long` transfers values from a FIFO array of the *ADwin* hardware as integer values of 32 bit precision into an array.

```
ctypes.c_int32_Array GetFifo_Long(int FifoNo,
 int Count)
```

Parameters

`FifoNo`	Number (1...200) of FIFO array `DATA_1` … `DATA_200`.
`Count`	Number (≥1) of values to be transferred.
Return value	Generated destination array that contains the transferred values.

Example

Check FIFO source array `DATA_2` for enough used elements and transfer 3000 elements of `DATA_2` into `ArrayLong` starting from index 0:

```
if adw.Fifo_Full(2) >= 3000:
    ArrayLong = adw.GefFifo_Long(2,3000)
```

SetFifo_Float

`SetFifo_Float` transfers float values of 32 bit precision from the PC into a FIFO array of the *ADwin* hardware.

```
SetFifo_Float(int FifoNo,
list|array|ctypes.c_float_Array PC_Array, int Count)
```

Parameters

`FifoNo`	Number (1...200) of FIFO array `DATA_1` … `DATA_200`.
`PC_Array`	Pointer to the source array, from which values are transferred.
`Count`	Number (≥1) of values to be transferred.

Example

Check FIFO destination array `DATA_12` for enough free elements and transfer 1000 elements of the source array `ArrayFloat`:

```
dataType = ctypes.c_float * 1000
ArrayFloat = dataType(0)
for i in range(1000): ArrayFloat[i] = i+0.1234
if adw.Fifo_Empty(12) >= 1000:
    adw.SetFifo_Float(12, ArrayFloat, 1000)
```

GetFifo_Float

`GetFifo_Float` transfers values from a FIFO array of the *ADwin* hardware as float values of 32 bit precision into an array.

```
ctypes.c_float_Array GetFifo_Float(int FifoNo, int
Count)
```

Parameters

`FifoNo`	Number (1...200) of FIFO array `DATA_1` … `DATA_200`.
`Count`	Number (≥1) of values to be transferred.
Return value	Generated destination array that contains the transferred values.

Example

Check FIFO source array `DATA_2` for enough used elements and transfer 200 elements of `DATA_2` into `ArrayFloat` starting from index 0:

```
if adw.Fifo_Full(2) >= 200:
    ArrayFloat = adw.GetFifo_Float(2, 200)
```

SetFifo_Double

`SetFifo_Double` transfers float values of 64 bit precision from the PC into a FIFO array of the *ADwin* hardware.

```
SetFifo_Double(int FifoNo,
list|array|ctypes.c_double_Array PC_Array, int
Count)
```

Parameters

`FifoNo`	Number (1...200) of FIFO array `DATA_1` … `DATA_200`.
`PC_Array`	Pointer to the source array, from which values are transferred.
`Count`	Number (≥1) of values to be transferred.

Example

Check FIFO destination array `DATA_12` for enough free elements and transfer 1 000 elements of the source array `ArrayDouble`:

```
dataType = ctypes.c_double * 1000
ArrayDouble = dataType(0)
for i in range(1000): ArrayDouble[i] = i+0.1234
if adw.Fifo_Empty(12) >= 1000:
    adw.SetFifo_Double(12, ArrayDouble, 1000)
```

GetFifo_Double

`GetFifo_Double` transfers values from a FIFO array of the *ADwin* hardware as float values of 64 bit precision into an array.

```
ctypes.c_double_Array GetFifo_Double(int FifoNo, int Count)
```

Parameters

`FifoNo`	Number (1...200) of FIFO array `DATA_1` … `DATA_200`.
`Count`	Number (≥1) of values to be transferred.
Return value	Generated destination array that contains the transferred values.

Example

Check FIFO source array `DATA_2` for enough used elements and transfer 200 elements of `DATA_2` into `ArrayDouble` starting from index 0:

```
if adw.Fifo_Full(2) >= 200:
    ArrayDouble = adw.GetFifo_Double(2, 200)
```

5.4.3 Data arrays with string data

Instructions for data transfer between PC and *ADwin* system with global `DATA` arrays (`DATA_1...DATA_200`) that contain string data.

You must declare each `DATA` array before using it in *ADbasic* (see manual "*ADbasic*"): **`DIM`** `DATA_x[n]` **`AS STRING`**.

An element in the `DATA` array of type `STRING` may contain a character with ASCII value 0 … 127. The ASCII value 0 (termination char or NULL) marks the end of a string in a `DATA` array.

String_Length

`String_Length` returns the length of a data string in a `DATA` array.

```
int String_Length(int DataNo)
```

Parameters

`DataNo`	Number (1...200) of array `DATA_1` … `DATA_200`.
Return value	≠-1: String length = number of characters. -1: Error

Notes

`String_Length` counts the characters in a `DATA` array up to the termination char (NULL). The termination char is not counted as character.

To determine the declared length of a `DATA` array you use the instruction `Data_Length`.

Example

In *ADbasic*, `DATA_2` is dimensioned as:

```
DIM DATA_2[2000] AS STRING
DATA_2 = 'Hello World'
```

In Python, the length of the array `DATA_2` is determined as:

```
adw.String_Length(2)# returns 11
```

SetData_String

`SetData_String` transfers a string into a `DATA` array.

```
SetData_String(int DataNo, str String)
```

Parameters

`DataNo`	Number (1...200) of destination array `DATA_1` … `DATA_200`.
`String`	String to be transferred.

Notes

`SetData_String` appends the termination char (NULL) to each transferred string.

Example

```
adw.SetData_String(2,'Hello World')
```

The string "Hello World" is written into the array `DATA_2` and the termination char is added.

GetData_String

`GetData_String` transfers a string from a `DATA`-array to the PC.

```
ctypes.c_char_Array GetData_String(int DataNo,
 int MaxCount)
```

Parameters

`DataNo`	Number (1...200) of source array `DATA_1` … `DATA_200`.
`MaxCount`	Max. number (≥1) of the transferred characters without termination char.
Return value	Array with the transferred string from the source array.

Notes

If the string in the `DATA` array contains a termination char (NULL), the transfer stops exactly there, that is the termination char will not be transferred. The number of read characters without termination char is the return value.

If *`MaxCount`* is greater than the number of string chars defined in *ADbasic*, you will receive the error "`Data too small`" via `Get_Last_Error`().

If you set *`MaxCount`* to a high value, the function will have an appropriately long execution time, even if the transferred string is short.
For time-critical applications with large strings, it may be faster to proceed as follows:

- You determine the actual number of chars in the string using `String_Length`().
- You read the string with `Getdata_String`() and pass the actual number of chars as *`MaxCount`*.

Example

Get the current string from `DATA_2` and copy it to ArrayString:

```
count = adw.String_Length(2)
ArrayString = adw.GetData_String(2,count)
```

5.5 Querying the error code

The following instructions are only useful in certain circumstances. Therefore, pay attention to chapter 4.1 "Locating errors".

Get_Last_Error

`Get_Last_Error` returns the number of the error that occurred at last in the *ADwin* program library.

```
int Get_Last_Error()
```

Notes

The function is only useful when you do not use any exceptions for error handling (see chapter 4.1 on page 7). In this case, you have to query the error number after each access to the *ADwin* system and act accordingly; after a successful access the error number is automatically set to 0.

Even if several errors occur, `Get_Last_Error` will only return the number of the error that occurred last.

To each error number you will get the text with the function `Get_Error_Text`. You will find a list of all error messages in section A.2 in the annex.

The call of `Get_Last_Error` itself does not influence the error number.

Example

```
# raiseExceptions = 0: In case of a run-time error,
# no exception is raised
adw.raiseExceptions = 0
Status = adw.Process_Status(2)
# read error number after each access
# to the ADwin system
ErrNum = Get_Last_Error()
if ErrNum: print('error: ', \)
    adw.Get_Error_Text(ErrNum)
```

Get_Error_Text

`Get_Error_Text` returns the error text to a given error number.

```
str Get_Error_Text(int ErrorNumber)
```

Parameters

`Last_Error`	Error number = return value of the function `Get_Last_Error`.
Return value	Error text.

Notes

The function can be called in combination with `Get_Last_Error`.

Example

```
# raiseExceptions = 0
adw.raiseExceptions = 0
Status = adw.Process_Status(2)
ErrNum = Get_Last_Error()
print('error: ', adw.Get_Error_Text(ErrNum))
```

Annex

A.1 Example programs

The ***ADwin*** driver for Python contains simple example programs that describe the interaction between Python and the ***ADwin*** system. All examples contain both the executable files and the source codes.

The Python examples use the Qt-Toolkit, version 4, and the Python module PyQt4. Toolkit and module are available in the internet from Riverbank: `http://www.riverbankcomputing.co.uk/software/pyqt/download.`

A complete example consists of a Python and an ***ADbasic*** program. Both programs characteristically perform the following different tasks:

– The Python program starts, monitors and stops a process on the ***ADwin*** system and displays the transferred data.

  Files can be found in the ***ADwin*** directory, see chapter 3.1 on page 5.

– The ***ADbasic*** program defines the processes running on the ***ADwin*** system, as well as measurement, open and closed loop control, and time-critical evaluation.

The following examples are available:

– BAS_DMO1: Online evaluation of measurement values

– BAS_DMO2: Online setting of control parameters

– BAS_DMO3: The example <BAS_DMO3> acquires a sequence of measurement values and displays them as curve.

– BAS_DMO7: Signal generation

All example programs are written for ***ADwin-Gold*** with the device no. 1. For different settings of the ***ADwin*** systems you have to adapt the source code and recompile.

For Windows, follow these steps:

– Open the ***ADbasic*** source code `<BAS_DMOx.bas>` and adapt the settings under `"Options ▸ Compiler"`.

  For ***ADwin-Pro I*** there are separate example programs in the directory `<C:\ADwin\ADbasic\samples_ADwin_Pro\>`.

– For ***ADwin-light-16*** the source code remains unchanged.

  If you use ***ADwin-Gold II*** or ***ADwin-Pro II*** you have to include the corresponding include files and adapt the instructions `ADC` and `DAC`.

– Generate a new binary file with `"Build ▸ Make Bin File"`.

– In the Python source code,
  - change the constant `DEVICENUMBER` to the number of the specified ***ADwin*** system.
  - adapt the name of the operating system files `<ADwin9.btl>` and the name of the binary file `<BAS_DMOx.T91>` an.

For Linux, these steps be used appropriately. The compiler use under Linux is described in the manual "ADwin for Linux/Mac".

BAS_DMO1

Online evaluation of measurement values

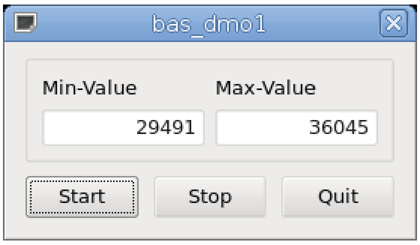
The example <BAS_DMO1> acquires measurement values in cycles, evaluates them online and displays the result.

The example executes the following tasks:

– The Python program loads the ***ADwin*** operating system for the T9 processor: `<ADwin9.btl>`.

– The Python program loads the ***ADbasic*** binary file for the T9 as process 1: `<BAS_DMO1.T91>`.

– With the button `Start`, you start
  - the loaded ***ADbasic*** process 1 and
  - the timer.



  The ***ADbasic*** process acquires 1000 measurement values at analog input 1, evaluates the minimum and maximum value and saves the two values in the global variables *Par_1* and *Par_2*. This measurement cycle is repeated until the process is stopped. After the first measurement cycle the process sets the global *Par_10* to value 1.

  The timer checks five times per second if the global variable *Par_10* has the value 1. If so, the function reads the values of the global variables *Par_1* and *Par_2* and displays them in the windows for minimum and maximum values.
  If you have no signal at the analog input 1 the displayed values fluctuate around zero, that is around the value 32768.

– With the button `Stop`, you stop
  - the ***ADbasic*** process 1 and
  - the timer.

Online setting of control parameters

BAS_DMO2

The example <BAS_DMO2> sets the control parameters of a closed loop control process, a digital P-controller. The control parameters may be changed online.

The example executes the following tasks:

– The Python program loads the ***ADwin*** operating system for the T9 processor: `<ADwin9.btl>`.

– The Python program loads the ***ADbasic*** binary file for T9 as process 1: `<BAS_DMO2.T91>`.

– With the button `Start`, you start the loaded ***ADbasic*** process 1.

– Using the slide controls you adjust the `Setpoint` and the `Gain` of the digital P-controller.

  With each new setting, the Python program writes the values of the slide controls into the global variables `Par_1` and `Par_2`.
  The ***ADbasic*** program continuously reads the global variables and uses them as control parameters.

– With the button `Stop`, you stop the ***ADbasic*** process 1.

You find a more detailed description of the ***ADbasic*** process in the tutorial.

BAS_DMO3

The example <BAS_DMO3> acquires a sequence of measurement values and displays them as curve.

The example executes the following tasks:

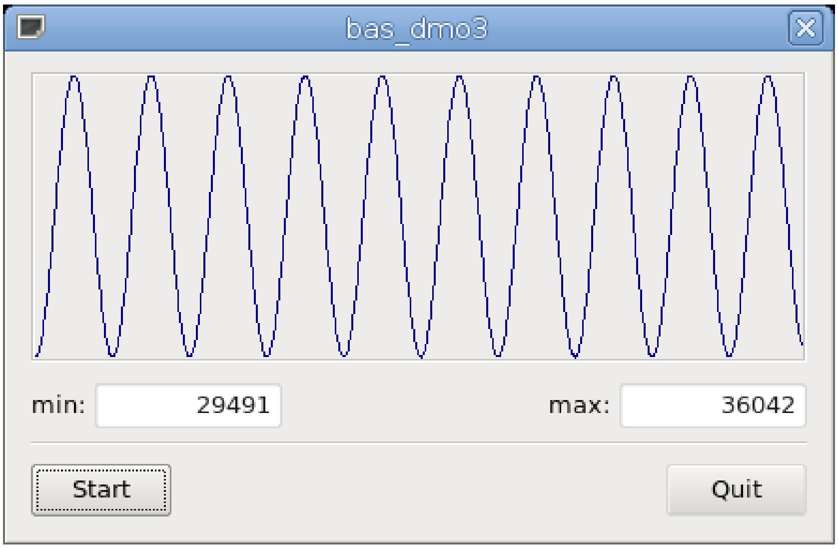
– The Python program loads the ***ADwin*** operating system for the T9 processor: `<ADwin9.btl>`.

– The Python program loads the ***ADbasic*** binary file for T9 as process 1: `<BAS_DMO3.T91>`.

– With the button `Start`, you start
  - the loaded ***ADbasic*** process and
  - the Python function `Timer1`.

  The ***ADbasic*** process `BAS_DMO3` acquires 1000 measurement values at analog input 1 and saves them in the global array *`DATA_1`*. Then the process sets the global variable *`Par_10`* to the value 1 and stops.

  The Python function `Timer1` checks ten times per second if the global variable *`Par_10`* is set to the value 1. If so, the function reads 1000 values from the global array *`DATA_1`* and displays them as curve.



  The displayed curve depends on the signal sequence at analog input 1.

– You can repeat the acquisition of a measurement sequence whenever you want to.

Signal generation

BAS_DMO7

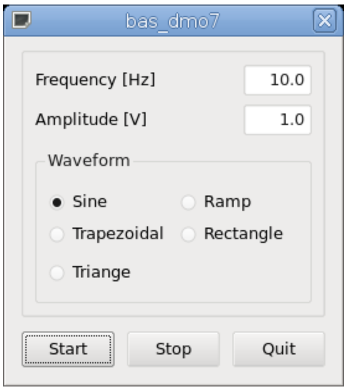
The example <BAS_DMO7> contains an ***ADbasic*** process running as function generator. Via user interface the signal form, the frequency and the amplitude of the output signal are set online.

The example executes the following tasks:

– The Python program loads the ***ADwin*** operating system for the T9 processor: <ADwin9.btl>.

– The Python program loads the ***ADbasic*** binary file for T9 as process 1: <BAS_DMO7.T91>.

– With the button Start, you start the loaded ***ADbasic*** process 1.



– Set the parameters for the output signal:
  - Frequency: 0…1000 Hz
  - Amplitude: 0…10 V
  - Signal form: Sine, ramp, trapecoid, rectangle, triangle.

  As soon as you change one of the parameters, the Python program writes the parameter values into the global variables Par_1, Par_2 and Par_3.

  The ***ADbasic*** program reads the global variables and generates the appropriate output signal.

– With the button Stop, you stop the ***ADbasic*** process 1.

A.2 Error messages

No.	Error message
0	No Error.
1	Timeout error on writing to the ADwin-system.
2	Timeout error on reading from the ADwin-system.
10	The device No. is not allowed.
11	The device No. is not known.
15	Function for this device not allowed.
20	Incompatible versions of ADwin operating system, driver (ADwin32.DLL) and/or ADbasic binary-file.
100	The Data is too small.
101	The Fifo is too small or not enough values.
102	The Fifo has not enough values.
103	The Data array is not declared.
150	Not enough memory or memory access error.
200	File not found.
201	A temporary file could not be created.
202	The file is not an ADBasic binary-file.
203	The file is not valid.[1]
204	The file is not a BTL.
2000	Network error (TcpIp).
2001	Network timeout.
2002	Wrong password.
3000	USB-device is unknown.
3001	Device is unknown.

1. Possibly the file `<ADwin5.btl>` has no memory table, or another file was renamed to `<ADwin5.btl>` or the file is damaged.

A.3 Index of functions