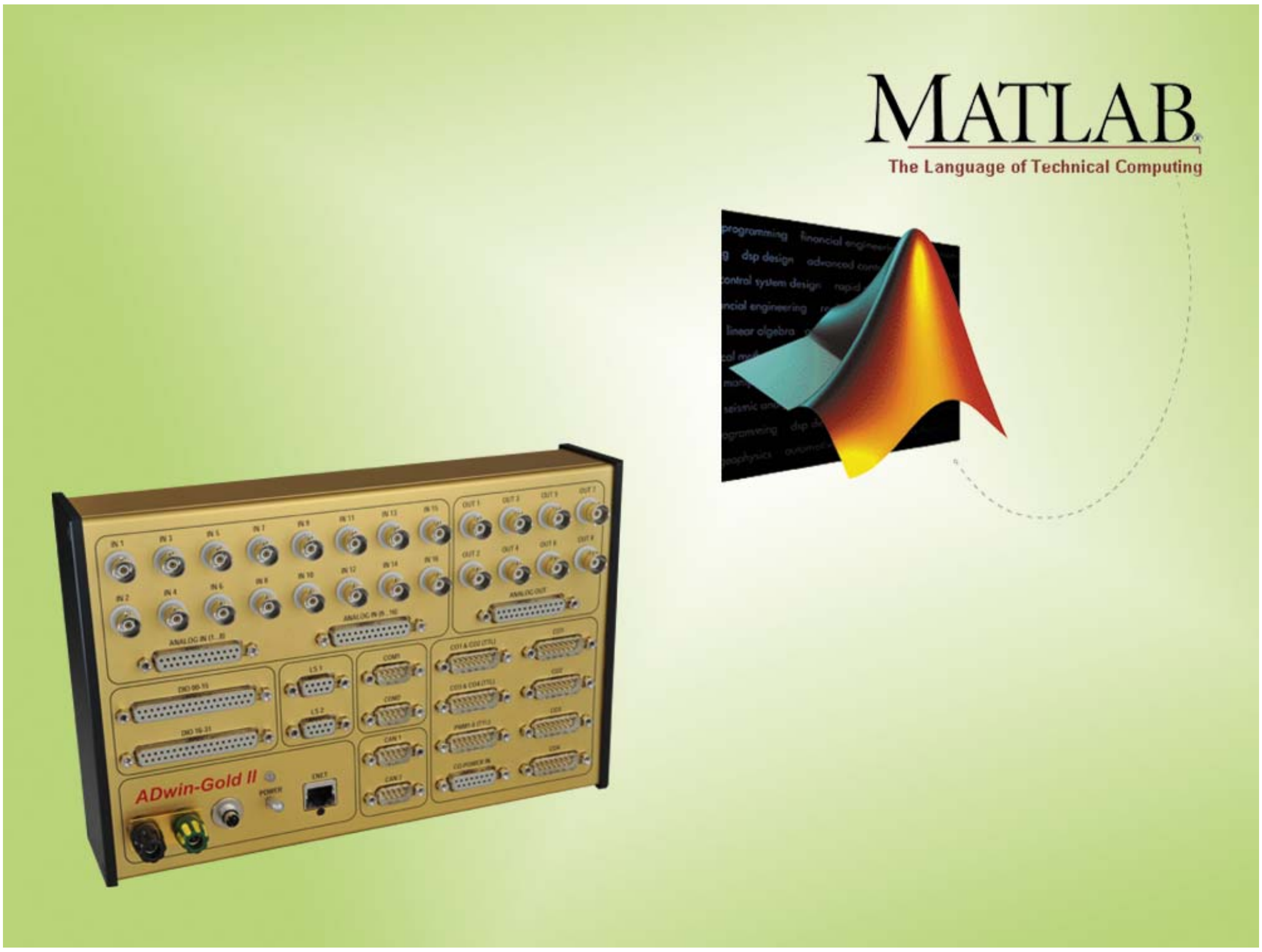
JÄGER
Computergesteuerte
Messtechnik GmbH

ADlab

Driver for MATLAB version 5…9



ADlab driver for `MATLAB`, Manual Sep. 2019

License Key:

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

ADwin

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

The manual gives detailed information about the *ADwin* driver for MATLAB® up to version 9.x.

The following documents are also important for the driver description:

– The "*ADwin* Installation Manual" describes the hardware and software installation for all *ADwin* systems

– If you work with Linux or Mac OS: the manual "ADwin for Linux / Mac", which describes the software installation and the *ADbasic* compiler usage from Linux and Mac OS.

– The manual "*ADbasic"* describes the development environment and the instructions of the *ADbasic* compiler. The *ADwin* system is programmed with the easy-to-use real-time development tool *ADbasic*.

– The hardware manuals for your *ADwin* systems.

It is assumed that that the user has a good command of the MATLAB® environment.

Please note:

For *ADwin* systems to function correctly, follow strictly 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

This documentation, including all pictures is protected by copyright. Reproduction, translation as well as electronical and photographical archiving and modification require a written permission by the company *Jäger Computergesteuerte Messtechnik GmbH*, Lorsch, Germany.

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* Driver for MATLAB®

This section introduces into the capabilities of the *ADwin* driver for MATLAB and describes how to communicate with an *ADwin* system from MATLAB under Windows, Linux or Mac OS.

2.1 Interface to the Development Environment

The *ADwin* driver for MATLAB is the interface to communicate with the *ADwin* systems.

The combination of the environment MATLAB with an *ADwin* system provides totally new possibilities. On the one hand you use the intelligence and performance of the *ADwin* system for measurements, open and closed-loop controls. On the other hand you have many MATLAB-functions for administration, analysis, and documentation of the measurement data and a comfortable user interface.

Applications:

– Open-loop control of fast test stands

– Signal generation

– Intelligent measurements, acquiring data under complex trigger conditions

– Open and closed-loop control

– Online processing, data reduction

– Hardware-in-the-Loop, simulation of sensor data

You determine the behaviour of the *ADwin* hardware on your own. There are 2 possible ways to do it:

– *ADbasic*: You program real-time processes using the development environment *ADbasic*, create a binary file and transfer it to the *ADwin* system (see *ADbasic* manual or online help).

– *ADsim* T11: You create a model in Simulink, export it and compile the model with *ADsimDesk* for the *ADwin* hardware (see *ADsim* manual).

2.2 Communication with the *ADwin* System

Using the *ADwin* driver for MATLAB, you can access global variables and arrays of the running *ADwin* hardware and control *ADbasic* processes from MATLAB.

Data and instructions between MATLAB and the *ADwin* system are processed as is shown below.

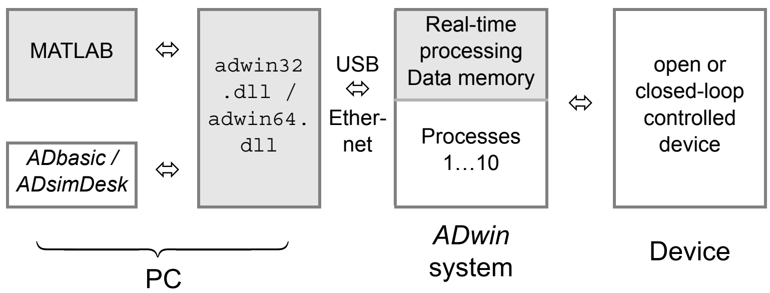


Fig. 1 – *ADwin*-MATLAB interface

`adwin32.dll`

The `adwin32.dll` (with a 64-bit operating system: `adwin64.dll`) is the central interface to the *ADwin* system for Windows applications and is therefore also used by the *ADwin* driver for MATLAB. With this interface, several

Window programs can communicate with the *ADwin* system at the same time: Development environments, *ADbasic*, *ADsimDesk,* and *ADtools* are working simultaneously with the *ADwin* system.

Real-Time Processing

The `adwin32.dll` / `adwin64.dll` interface communicates with the real-time processor of the *ADwin* system - the operating system. Therefore, you have to load the operating system first (e.g. the file `<ADwin11.btl>`) before powering up the system.
Only after it has been successfully loaded, you can transfer and run *ADbasic* processes on the *ADwin* hardware, and access global variables and arrays. The processes being programmed in *ADbasic* contain the program code for measurement, open or closed-loop control of your application.

With *ADsim*, the file `*.btl` contains both the operating system and the compiled model, which corresponds to an *ADbasic* process. Therefore, the functions of the *ADwin* driver for process control do not apply here.

The operating system executes the following tasks:

10 processes

– Managing up to 10 real-time processes with low or high priority (individually selectable). Low-priority processes can be interrupted by high-priority processes; high-priority cannot be interrupted by other processes. With *ADsim* T11, there is only a single real-time process of high priority, which is the compiled Simulink model; as an option, an additional low priority process can belong to the model.

Data memory

– Availability of global variables:
  - 80 integer variables (`PAR_1` … `PAR_80`), predefined.
  - 80 floating-point variables (`FPAR_1` … `FPAR_80`), predefined.
  - 200 data arrays (`DATA_1` … `DATA_200`), length and data type can be set individually.

  The values of these variables or data arrays can be read and changed at any time from MATLAB.

Communication

– Communication between *ADwin* system and PC (`adwin32.dll` / `adwin64.dll`).

  The communication process is running at medium priority on the *ADwin* system and can interrupt low-priority processes for a short time. The communication process interprets and processes all instructions you are sending to the *ADwin* system: Control commands and commands for data exchange.

  The following table shows examples of each group.

Process control commands, e.g.	
`Load_Process`	loads an *ADbasic* process to the *ADwin* system
`Start_Process`	starts a process.
Commands for data exchange, e.g.	
`Get_Par`	returns the current value of a global variable
`Set_Par`	changes the value of a parameter.
`GetData_Double`	returns the values from a `DATA` array.

The communication process never sends data to the PC without being told to do so. Thus, it is assured that only then data are transferred to the PC, when they have been explicitly requested before.

2.3 *ADsim* driver for MATLAB

As an addition to the *ADwin* driver for MATLAB Simulink users can also use the *ADsim* driver for MATLAB (see manual *ADsim* Driver for Matlab).

The *ADsim* driver works only under Windows.

While the *ADwin* driver for MATLAB enables you to access *ADwin* variables of the Simulink model, the *ADsim* driver provides access to all other variables and arrays of the Simulink model, i.e. parameters, signals and block states.
Both drivers can be used independently from each other. Please note that the drivers use a very different error handling.

We recommend preferrably using the *ADwin* driver for MATLAB. In comparison, it provides several advantages:

– The driver supports continuous data flow via Fifo arrays.
– Compiled Simulink models can be transferred to *ADwin* hardware using the `Boot` instruction.
– The driver runs under Linux and Mac OS.
– The driver works also with MATLAB versions before R2007b (7.5).

Users of *ADsim* T11 can use all functions of the *ADwin* driver for MATLAB except for the following:

– Free_Mem
– Load_Process, Start_Process, Stop_Process, Clear_Process, Process_Status
– String_Length, SetData_String, GetData_String

2.4 Distribute a Stand-alone Application

Using the MATLAB-Compiler you can create stand-alone applications, which can be run without installing MATLAB. The *ADwin* driver for MATLAB can be integrated into a stand-alone application as well.

If you want to distribute a stand-alone application to an end user, please note:

– The end user requires–apart from the stand-alone application–an *ADwin* software package, which is to be installed.
– The installation of the *ADwin* software package is described in chapter 3.1. There are no further installation steps required for an end user.
– The end user will not require any license key from Jäger Messtechnik., neither for the *ADwin* driver for MATLAB (because it is comprised in the stand-alone application) nor for *ADbasic*.

3 Installing the *ADwin* Driver for MATLAB®

3.1 Do the "*ADwin* driver installation"

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

3.1.1 Installation under Linux or Mac

Please follow the installation guide in the manual "ADwin Linux / Mac". Please pay attention to installing the archive `adwin-labview-x.y.tar.gz` at last.

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

Drivers and examples for MATLAB.	`./Developer/Matlab`
Examples for *ADbasic*	`./samples_ADwin`

Continue with chapter 3.2 "Including the ADwin Driver for MATLAB®".

3.1.2 Installation under 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

- to install the software from the *ADwin* software package.
- to install the communications driver under Windows.
- to install the hardware in the PC (if neccessary) and
  set up the hardware connections between PC and *ADwin* system.

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

Drivers and examples for MATLAB.	`.\Developer\MATLAB\…`
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`

64-bit version requires Visual Studio

Please note for Matlab 64-bit version: According to Mathworks you must install an additional C compiler to use the *ADwin* driver.
On the homepage `www.mathworks.de`, Mathworks provides a list of suitable compilers for `MATLAB` releases under the keyword "Supported and Compatible Compilers". Mostly the list provides download links.

The urge to use an external compiler in Matlab (64 bit) existis for all external program packages (DLLs). Matlab uses the compiler to create an internal file, with which it can use the DLL.

32-bit version: configure compiler

If the C compiler in a 32-bit version does not run as expected the compiler may need to be configured first. To configure, enter the instruction `mex -setup` in Matlab.

3.2 Including the *ADwin* Driver for MATLAB®

Make the driver files available in MATLAB® as follows below, in order to access the functions of your *ADwin* system.

New license key

– Release the *ADwin*-MATLAB® driver by entering your new license key in *ADbasic* (Menu: "`Help` ▸ `About ADbasic`").

  You find the license key on the firsts page of the *ADbasic* manual.

  Under Windows NT, 2000, XP, Vista, 7, 8, 10, you must be member of the user group "Administrators". It is not sufficient to have full access rights on the PC. Ask your system administrator.

– From MATLAB® start the "`Path Browser`". Add the path `<C:\ADwin\Developer\Matlab\ADwin>` (standard path name) to the list of used directories and save the list.

  If there is an older version of the file `<ADlab.DLL>` in the MATLAB root directory, delete this file. Otherwise MATLAB® uses the older file version.

– The used MATLAB version leads to the required *ADwin* driver package. Each package suits both 32-bit and 64-bit versions.

  For Matlab versions since R2007b (*ADwin* driver package P3), all further settings are already done.

Matlab Version	*ADwin* driver package		
	P1	P2	P3
7.0 or before	x		
7.1	x	x	
R2006a (7.2)	x	x	
R2006b (7.3)		x	
R2007a (7.4)		x	
R2007b (7.5)		x	x
R2008a (7.6)		x	x
R2008b (7.7)		x	x
R2009a (7.8)		x	x
R2009b (7.9)		x	x
R2010a (7.10)		x	x
R2010b (7.11)		x	x
R2011a (7.12) or later			x

– If you require one of the previous *ADwin* driver packages P1 or P2:
  - Delete the folder `<.\ADwin>` in the directory `<C:\ADwin\Developer\Matlab>`.
  - According to the selected version, create a copy of the folder `<.\ADwin_P1>` or `<.\ADwin_P2>`.
  - Rename the copied folder into `<.\ADwin>`.

The *ADwin* driver is now available in MATLAB.

3.3 Accessing the *ADwin* System

With the Installation of hardware and software, you have successfully checked the access to the *ADwin* system. The following test shows whether MATLAB has access to the *ADwin* system.

Type the following lines in the „`Command window`“:

```
>> ADwin_Init(); %only required since Matlab version 7.1
>> Boot('C:\ADwin\ADwin9.btl', 0);
>> Get_Last_Error()
ans =
       0
```

This is what the lines do:

– You initialize MATLAB for communication with *ADwin* systems; the target Device No. is automatically set to 1.

  If installation with *ADconfig* was done with a different device number, set it separately with `Set_DeviceNo`.

– You load the operating system of the processor T9 to the *ADwin* system (= booting).

  The filenames for other processors than T9 are given on page 14.

– You query the error code, which was created after booting. The value 0 confirms the *ADwin* system to be ready.

  An error code > 0 denotes an error during booting. A list of all error messages is given in chapter A.2 in the annex.

You may now use all driver functions to get access to the *ADwin* system.

As an introcduction we recommend working with the program examples in the annex, section A.1.

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.

Former driver versions used the functions `Net_Connect` and `Net_Disconnect` instead of `ADwinTcpipServer`. The functions are obsolete now and should not be used any more.

4 General Information about *ADwin* Functions

4.1 Detecting Errors

There are 2 possibilities to locate errors upon execution of an *ADwin* function:

Return value of the function

1. The return value of a function indicates if an error has occurred.

   Please note:
   - The functions are using different values to indicate an error.
   - Functions that read more than one value at a time will not have an error number as return value:
     `Get_Par_Block`, `Get_Par_All`, `Get_FPar_Block`, `Get_FPar_All`, `Get_FPar_Block_Double`, `Get_FPar_All_Double`, `GetData_Double`, `GetFifo_Double`.
   - In the following functions, the return value is not quite explicit, that means, it can be interpreted either as error or as value. Therefore, you have to use the function `Get_Last_Error` here:
     `Fifo_Empty`, `Fifo_Full`, `Get_Par`, `Get_FPar`, `Get_FPar_Double`, `Get_Processdelay`, `Free_Mem`.

Get_Last_Error

2. The function `Get_Last_Error` (see page 38) returns the number of the error that occurred last.

   To handle each error, call `Get_Last_Error` after each access to the *ADwin* system.

Because the first variant has some disadvantages we recommend always querying errors with `Get_Last_Error`.

For instance an error in the function `Get_Processdelay` can only be detected with `Get_Last_Error`.

First the unsafe variant:

```
gd_2 = Get_Processdelay(2); % Processdelay of process 2
if (gd_2 ~= 255)
  … % no error
end
```

If `Get_Processdelay` returns the value 255, it is not quite clear if an error has occurred or if the parameter contains the value 255.
Therefore, `Get_Last_Error` must be used:

```
gd_2 = Get_Processdelay(2); % Processdelay of process 2
if (Get_Last_Error() == 0)
  … % no error
end
```

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*.

All functions of the *ADwin* driver for MATLAB use an internal variable `DeviceNo` to access an *ADwin* system. The function `Set_DeviceNo` is used to set the internal variable. The default number is 1.

4.3 Data Types

The functions and parameters of the *ADwin* driver for MATLAB use the following data types:

ADwin driver for MATLAB		ADwin hardware
Data type	Definition	Data type
char	unsigned integer 8- bit	String
int32	signed integer 32- bit	Long
single	float 32- bit	Float / Float32
double	float 64- bit	Float64

Variables (1x1 matrix) and row vectors can be used as function parameters.

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
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

4.3.1 Converting integer data types during data transfer

Upon reading integer variables the *ADwin* functions return values of data type `double`. Vice versa, upon writing model variables you will normally use values of data type `double`.

With integer values, counter values and bit patterns require to use data types `int32` or `uint32` in MATLAB. To do so, the data type must be converted. You have to distinguish:

– Counter values: In order to calculate with counter values (typically differences between 2 counter values), values of data type `int32` have to be used.
  This concerns both event counters as well as timers.

– Bit patterns: Bit patterns must be processed with data type `uint32`, also to enable the use of MATLAB functions for bit operations.

  Bit patterns are used with the following *ADwin* block signals:
  - Status of 16/32 digital channels
  - Data bytes of CAN messages
  - Signals of an SSI encoder (Gray coded)
  - packed values: 2 values of 16 bit in a 32-bit value.

– Boolean values: The data type `double` displays the boolean values 0 and 1 correctly, a type conversion is not required.

  Boolean values e.g. be found as status of single digital channels.

– Numerical values: The data type `double` displays numerical values correctly, a type conversion is not required.

Data type int32

This is how to convert values into data type `int32`:

– Read variable and convert into data type `int32`:
```
ValueID = Register_Value(ModelFile, VariableNodePath)
value = Get_Int32(ValueID);
val_int32 = cast(value, 'int32');
```

– While writing a value the data type has not to be converted.

  The workflow is shown here nevertheless, i.e. convert to `int32` and write into variable:
```
val_int32 = cast(value, 'int32')
ValueID = Register_Value(ModelFile, VariableNodePath)
Set_Int32(ValueID, val_int32);
```

Data type uint32

This is how to convert values into data type `uint32`:

– Read variable and convert into data type `uint32`:
```
ValueID = Register_Value(ModelFile, VariableNodePath)
value = Get_Int32(ValueID);
val_int32 = cast(value, 'int32');
val_uint32 = typecast(val_int32, 'uint32')
```

– Convert value into data type `int32` and write into variable:
```
val_int32 = typecast(val_uint32, 'int32')
ValueID = Register_Value(ModelFile, VariableNodePath)
Set_Int32(ValueID, val_int32);
```

4.4 Exchange Data of Two-Dimensional Arrays

In *ADbasic*, global `DATA` arrays can be declared as 2-dimensional arrays (2D). But the functions of the *ADwin* driver use only row vectors in MATLAB.
A row vector may be easily changed into a 2-dimensional array using the MATLAB function `reshape`.

In general, the following table shows how an element in a 2D array in *ADbasic* is related to an element in a row vector in MATLAB:

ADbasic	MATLAB
`DATA_n[i][j]`	`Vector[s·(i-1)+j]`

Here `s` is the second dimension of `DATA_n` when you declare the array in *ADbasic*.

Please see the notes on 2-dimensional arrays in the *ADbasic* manual, too.

Example: A 2D array in *ADbasic* is declared as

```
DIM DATA_8[7][3] AS FLOAT 'that is s=3
```

The 7×3 elements of the array are read in MATLAB with `GetData_Double`:

```
>> vector = GetData_Double(8,1,21);
```

The data are transferred in the following order:

Index of `DATA_8`	[1][1]	[1][2]	[1][3]	[2][1]	…	[7][1]	[7][2]	[7][3]
Index of `vector`	[1]	[2]	[3]	[4]	…	[19]	[20]	[21]

Thus, the function `GetData_Double` transfers the element `DATA_8[7][2]` into `vector[20]`.

The general formula `s=3` results in:

ADbasic	MATLAB	
`DATA_n[1][1]`	`Array[3·(1-1)+1]`	`= vector[1]`
`DATA_n[1][2]`	`Array[3·(1-1)+2]`	`= vector[2]`
...	...	...
`DATA_n[7][2]`	`Array[3·(7-1)+2]`	`= vector[20]`
`DATA_n[7][3]`	`Array[3·(7-1)+3]`	`= vector[21]`

5 Description of *ADwin* Driver Functions

The description of functions is divided into the following sections:

– System control and information, page 13
– Process control, page 17
– Transfer of Global Variables, page 22
– Transfer of Data Arrays, page 28
– Error handling, page 38

Call function `ADwin_Init` first, before accessing the *ADwin* system with other functions.

In appendix A-3, you find an overview of all functions. The description of functions is also available using the MATLB help function in the `Command window`:

```
>> help adwin
```

or

```
>> help [function name]
```

Former driver versions used instruction numbers instead of function names. You find the correlation of instruction numbers and function names in annex A.4.

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 described in the *ADwin* driver for MATLAB. They can be programmed in *ADbasic*.

5.1 System control and information

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

ADwin_Init

`ADwin_Init` initializes Matlab for communication with *ADwin* systems.

`ADwin_Init``()`

Notes

During initialization important default values are set among them the following:

- DeviceNo = 1; see also Set_DeviceNo (below).
- Show_Errors = On; see also Show_Errors (page 38).

`ADwin_Init` must be called first, in order to make *ADwin* functions run correctly. If the call misses and an *ADwin* function is being used, the function will call `ADwin_Init` by itself.

Example

```
% Initialize Matlab for communication with ADwin,
% set default device number 1 and show errors.
ADwin_Init();
```

ADwin_Unload

`ADwin_Unload` deletes all *ADwin* functions from PC memory and releases its memory space.

`ADwin_Unload``()`

Set_DeviceNo

`Set_DeviceNo` sets the device number.

`Set_DeviceNo` `(DeviceNo)`

Parameters

`DeviceNo`	board address or DeviceNo in decimal notation. The default setting is 1.

Notes

The PC distinguishes and accesses the *ADwin* systems by the device number. Systems with link adapter are already configured in factory (default setting: 336).

Further information can be found in the online help of the program *ADconfig* or in the manual "*ADwin* Installation".

Example

```
% Set the device number 3
Set_DeviceNo(3);
```

Get_DeviceNo

`Get_DeviceNo` returns the current device number.

```
Get_DeviceNo ()
```

Notes

The PC distinguishes and accesses the *ADwin* systems by the device number. Systems with link adapter are already configured in factory (default setting: 336).

Further information can be found in the online help of the program *ADconfig* or in the manual "*ADwin* Installation".

Example

```
% Query the current device number
num = Get_DeviceNo();
```

Boot

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

```
Boot (Filename, MemSize)
```

Parameters

`Filename` Path and filename of the operating system file (see below).

`MemSize` For processors up from T9: 0 (zero).
For T2, T4, T5, T8: Memory size to be used; the following values are permitted:

10000:	64 KiB
100000:	1 MiB
200000:	2 MiB
400000:	4 MiB
800000:	8 MiB
1000000:	16 MiB
2000000:	32 MiB

Return value Status:
<1000: Error during boot process
8000: Boot process o.k.; up from processor T9.
>8000: Boot process o.k.; for T2…T8 only. The value is the size of physically installed memory.

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\>`.

Processor	Operating System File
T225 (T2)	`ADwin2.btl`
T400 (T4)	`ADwin4.btl`
T450 (T5)	`ADwin5.btl`
T805 (T8)	`ADwin8.btl`

Processor	Operating System File
T9	`ADwin9.btl`
	`ADwin9s.btl` Optimized operating system with smaller memory needs.
T10	`ADwin10.btl`
T11	`ADwin11.btl`
T12	`ADwin12.btl`
T12.1	`ADwin121.btl`

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.

For users of *ADsim* T11:

- As `Filename` you enter the Simulink model being compiled via *ADsimDesk*, which also contains the operating system for the processor. The model file is stored in the model folder in the sub-folder `<model>_ert_rtw/ADwin/` with the name `<model>11c.btl`.
- `<model>` stands for the name of the Simulink model. The notation `11c` refers to the processor type T11 of the *ADwin* hardware.
- Please note that *ADbasic* processes and a compiled Simulink model (from *ADsim* T11) run on the *ADwin* hardware at the same time.

Loading the operating system with `Boot` takes about one second. As an alternative you can also load the operating system via *ADbasic* (icon B) or *ADsimDesk* development environment.

Example

```
% Load the operating system for the T10 processor
ret_val = Boot ('C:\ADwin\ADwin10.btl', 0);

% Load a Simulink model being compiled with ADsim T11
path = 'C:\ADwin\ADsim\Developer\Examples\';
subpath = 'ADsim32_DLL_Example_ert_rtw\ADwin\';
Boot([path,subpath,'ADsim32_DLL_Example11c.btl'], 0);
```

Test_Version

`Test_Version` checks, if the correct operating system for the processor has been loaded and if the processor can be accessed.

```
Test_Version ()
```

Parameters

Return value 0: OK
$\neq$0: Error

Example

```
% Test, if the processor system is loaded
ret_val = Test_Version();
```

Processor_Type

`Processor_Type` returns the processor type of the system.

`Processor_Type ()`

Parameters

Return value — Parameter for the processor type 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

```
% Query the processor type
ret_val = Processor_Type ();
```

Workload

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

`Workload` `(Priority)`

Parameters

`Priority` — 0: Current total workload of the processor.
≠0:is not supported at the moment

Return value — ≠255: 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

```
% Query the processor workload
ret_val = Workload (0);
```

Free_Mem

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

`Free_Mem` `(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: Cacheable: Memory, which can provide data to the cache; T12/T12.1 only.
6: Uncacheable: Memory, which cannot provide data to the cache; T12/T12.1 only.

Return value ≠255: Usable free memory (in bytes).
With Mem_Spec =5/6, the value is given in units of kB.
255: Error

Notes

This function cannot be used in connection with *ADsim* T11.

Example

```
% Query the free memory in the external DRAM
ret_val = Free_Mem (4);
```

5.2 Process control

The control of *ADbasic* and *TiCoBasic* processes is different:

– ADbasic Processes

– TiCoBasic Processes

5.2.1 ADbasic Processes

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

The functions of this sections cannot be used in connection with *ADsim* T11.

There are the processes 1…10 and 15:

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

– 15: Control of the flash LED on *ADwin-Gold* and *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 a process into the *ADwin* system.

```
Load_Process (Filename)
```

Parameters

Filename	Path and filename of the binary file to be loaded
Return value	1 OK ≠1 Error

Notes

This function cannot be used in connection with *ADsim* T11.

You generate binary files in *ADbasic* with "Make > Make Bin file".

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

You can 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 binary file Testprog.T91
% T91 = Processor type T9, process no. 1
Load_Process('C:\MyADbasic\Testprog.T91');
```

Start_Process

`Start_Process` starts a process.

`Start_Process` `(ProcessNo)`

Parameters

`ProcessNo`	Number of the process (1...10, 15).
Return value	≠255: OK 255: Error

Notes

This function cannot be used in connection with *ADsim* T11.

The function has no effect, if you indicate the number of a process, which

- is already running or
- has the same number as the calling process or
- has not yet been loaded to the *ADwin* system.

Example

```
% Start Process 2
Start_Process (2);
```

Stop_Process

`Stop_Process` stops a process.

`Stop_Process` `(ProcessNo)`

Parameters

`ProcessNo`	Process number (1...10, 15).
Return value	≠255: OK 255: Error

Notes

This function cannot be used in connection with *ADsim* T11.

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

```
% stop process 2
ret_val = Stop_Process (2);
```

Clear_Process

`Clear_Process` deletes a process from memory.

`Clear_Process` `(ProcessNo)`

Parameters

`ProcessNo`	Process number (1...10, 15).
Return value	≠1: OK 1: Error

Notes

This function cannot be used in connection with *ADsim* T11.

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.

Example

```
% Delete process 2 from memory.
% Declared DATA and FIFO arrays remain.
ret_val = Clear_Process(2);
```

Process_Status

`Process_Status` returns the status of a process.

`Process_Status` `(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. -1: Process has been stopped, that means, it has received `Stop_Process`, but still waits for the last event.

Notes

This function cannot be used in connection with *ADsim* T11.

Example

```
% Return the status of process 2
ret_val = Process_Status(2);
```

Set_Processdelay

`Set_Processdelay` sets the parameter `Processdelay` for a process

`Set_Processdelay` `(ProcessNo, Processdelay)`

Parameters

`ProcessNo`	Process number (1...10); with ADsim T11: …2.

`Processdelay`	Value (1…2^{31}-1) to be set for the parameter `Processdelay` of the process.
Return value	≠255: OK 255: Error

Notes

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

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

The cycle time is specified in cycles of the *ADwin* processor. The cycle time depends on processor type and process priority:

Processor type	Process priority	
	high	low
T2, T4, T5, T8	1000 ns	64 µs
T9	25 ns	100 µs
T10	25 ns	50 µs
T11	$3.\overline{3}$ ns	$0.00\overline{3}$ µs = $3.\overline{3}$ ns
T12	1 ns	1 ns
T12.1	1.5 ns	1.5 ns

Example

```
% Set Processdelay 2000 of process 1
ret_val = 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).

Get_Processdelay

`Get_Processdelay` returns the parameter `Processdelay` for a process.

`Get_Processdelay` `(ProcessNo)`

Parameters

`ProcessNo`	Process number (1...10); with ADsim T11: …2.
Return value	≠255: The currently set value (1…2^{31}-1) for the parameter `Processdelay`. 255: Error

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*).

For *ADsim* users: The parameter `Processdelay` corresponds to the fixed-step size in Simulink. While the fixed-step size is set in seconds, the `Processdelay` is a multiple of processor cycles, see `Set_Processdelay`.

Example

```
% Get Processdelay of process 1
x = Get_Processdelay(1);
```

5.2.2 TiCoBasic Processes

On an *ADwin* hardware with *TiCo* processor, you can transfer a *TiCoBasic* binary file as process to the *ADwin* hardware and start the process. You can use an *ADbasic* program, the development environment *TiCoBasic*, or Matlab.

To transfer a TiCoBasic process with Matlab, the following steps are necessary:

– Create a binary file with *TiCoBasic*.

  At first, the binary file must be transferred into a global array `Data_x` of the *ADwin* processor, where an *ADbasic* process copies it to the *TiCo* processor.

– Create an *ADbasic* process, which fulfills the following tasks:
  - Dimensioning a global array `Data_x` of data type `Long`. Please make sure that the array size exceeds the size of the *TiCoBasic* binary file.
  - Transferring the data from `Data_x` to the *TiCo* processor using the instruction **`TiCo_Load`** / **`P2_TiCo_Load`**. The process starts automatically.
  - Saving the instruction's return value to a global variable `Par_x`.
  - Exiting the *ADbasic* proces with `Exit`.

  You find example *ADbasic* processes in the installation directory (see chapter 3.1) under
  `<.\ADbasic\samples_ADwin_GoldII>`
  `<.\ADbasic\samples_ADwin_ProII>`

– Create the binary file of the process in *ADbasic*.

– Do the following steps in Matlab:
  - Transfer the *ADbasic* binary file with `Load_Process` as process to the *ADwin* hardware, but do not start the process yet.
  - Transfer the *TiCoBasic* binary file with `Data2File` to the correct array `Data_x` of the *ADwin* processor.
  - Start the *ADbasic* process with `Start_Process`.
  - Read the global variable `Par_x` and check if transferring has been successful.

5.3 Transfer of Global Variables

Instructions for data transfer between PC and *ADwin* system with the predefined global variables of the ADwin hardware `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` on the *ADwin* system have the following range of values:

`PAR_1 ... PAR_80`: -2147483648 … +2147483647
= $-2^{31} \ldots +2^{31}-1$

The functions transfer integer values with 32-bit width. With functions returning only one value, MATLAB saves the value in a variable of data type double.

If a global variable contains a counter value or a bit pattern the value must be processed in MATLAB with data type `int32` or `uint32`; find more details in Data Types on page 6.

Set_Par

`Set_Par` sets a global variable `PAR` to the specified value.

`Set_Par` `(Index, Value)`

Parameters

`Index`	Number (1 … 80) of the global variable `PAR_1 … PAR_80`.
`Value`	Value to be set for the LONG variable.
Return value	≠255: OK 255: Error

Example

```
% Set LONG variable PAR_1 to 2000
ret_val = Set_Par(1,2000);
```

Get_Par

`Get_Par` returns the value of a global variable `PAR`.

`Get_Par` `(Index)`

Parameters

`Index`	Number (1 … 80) of the global variable `PAR_1 … PAR_80`.
return value	≠255: Current value of the variable, data type int32. 255: Error

Example

```
% Read value of the LONG variable PAR_1
x = Get_Par(1);
```

Get_Par_Block

`Get_Par_Block` transfers a specified number of consecutive global variables `PAR` into a row vector (data type int32).

Get_Par_Block (StartIndex, Count)

Parameters

StartIndex	Number (1 … 80) of the first global variable PAR_1 … PAR_80 to be transferred.
Count	Number (≥1) of values to be transferred.
Return value	Row vector with transferred values.

Example

Read values of variables `PAR_10…PAR_39` and write the values to the row vector `v`:

```
v = Get_Par_Block(10, 30);
```

Get_Par_All

`Get_Par_All` transfers all 80 global variables `PAR_1… PAR_80` into a row vector (data type int32).

Get_Par_All ()

Parameters

Return value	Row vector with transferred values (PAR_1…PAR_80).

Example

Read values of variables `PAR_1…PAR_80` and write the values to the row vector `v`:

```
v = Get_Par_All();
```

5.3.2 Global Variables FPAR_1...FPAR_80

The global variables FPAR_1...FPAR_80 on the *ADwin* system have the following range of values, depending on the processor type (see also section Data Types):

- FLOAT until T11 — 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 with T12/T12.1 — negative: $-1{,}797693134862315 \cdot 10^{+308} \ldots -2{,}225073858507201 4 \cdot 10^{-308}$
  positive: $+2{,}2250738585072014 \cdot 10^{-308} \ldots +1{,}797693134862315 \cdot 10^{+308}$

Set_FPar

Set_FPar sets a global variable FPAR to a specified single value.

Set_FPar (Index, Value)

Parameters

Index	Number (1 … 80) of the global variable FPAR_1 … FPAR_80.
Value	Value of data type single to be set for the variable.
Return value	≠255: OK 255: Error

Notes

Set_FPar always transfers a 32-bit float value even though FPAR may have 64-bit precision.

Example

```
% set variable FPAR_6 to 34.7
ret_val = Set_FPar(6, 34.7);
```

Set_FPar_Double

Set_FPar_Double sets a global variable FPAR to a specified double value.

Set_FPar_Double (Index, Value)

Parameters

Index	Number (1 … 80) of the global variable FPAR_1 … FPAR_80.
Value	Value of data type double to be set for the FPAR variable.
Return value	≠255: OK 255: Error

Notes

With processors until T11, the destination variable on the ADwin system has single precision only.

Example

```
% set variable FPAR_6 to 34.7
ret_val = Set_FPar_Double(6, 34.7);
```

Get_FPar

Get_FPar returns the single value of a global variable FPAR.

Get_FPar (Index)

Parameters

Index	Number (1 … 80) of the global variable FPAR_1 … FPAR_80.
Return value	≠255: Current single value of the variable 255: Error

Notes

Since processor T12, FPAR variables in the *ADwin* system have 64-bit precision. Nevertheless, Get_FPar will return a value of data type single.

Example

```
% Read the value of the variable FPAR_56
ret_val = Get_FPar(56);
```

Get_FPar_Block

Get_FPar_Block transfers a specified number of consecutive global variables FPAR into a row vector (data type single).

Get_FPar_Block (StartIndex, Count)

Parameters

StartIndex	Number (1 … 80) of the first global variable FPAR_1… FPAR_80 to be transferred.
Count	Number (≥1) of variables to be transferred.
Return value	Row vector with transferred values of data type single.

Example

Read values of variables PAR_10 … PAR_34 and store in a row vector v:

```
v = Get_FPar_Block(10,25);
```

Get_FPar_All

Get_FPar_All transfers all global variables FPAR_1…FPAR_80 into a row vector (data type single).

Get_FPar_All ()

Parameters

Return value	Row vector with transferred values of data type single.

Example

Read values of variables PAR_1 … PAR_80 and store in a row vector v:

```
v = Get_FPar_All();
```

Get_FPar_Double

`Get_FPar_Double` returns the double value of a global variable `FPAR`.

`Get_FPar_Double` `(Index)`

Parameters

`Index`	Number (1 … 80) of the global variable `FPAR_1` … `FPAR_80`.
Return value	≠255: Current double value of the variable 255: Error

Notes

Until T11, please note: float values in the *ADwin* system have 32-bit precision. Nevertheless, `Get_FPar_Double` will return a value of data type double.

Example

```
% Read the value of the variable FPAR_56
ret_val = Get_FPar_Double(56);
```

Get_FPar_Block_Double

`Get_FPar_Block_Double` transfers the specified number of global variables `FPAR` into a row vector (data type double).

`Get_FPar_Block_Double` `(StartIndex, 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	Row vector with transferred values of data type double.

Notes

Until T11, please note: floating-point values in the *ADwin* system have 32-bit precision. You should therefore display `FPAR` values only with single precision to avoid misunderstandings.

Example

Read the values of the variables `PAR_10` … `PAR_34` and store in a row vector `v`:

```
v = Get_FPar_Block_Double(10,25);
```

Get_FPar_All_Double

`Get_FPar_All_Double` transfers all global variables `FPAR_1…FPAR_80` into a row vector (data type double).

`Get_FPar_All_Double` `()`

Parameters

Return value	Row vector with transferred values of data type double.

Notes

Until T11, please note: floating-point values in the *ADwin* system have 32-bit 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 a row vector `v`:

```
v = 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 in *ADbasic* before using it in MATLAB (see "*ADbasic*" manual).

5.4.1 Data arrays

You have to declare each array in *ADbasic* before using it in MATLAB with
`DIM DATA_x AS LONG/FLOAT/FLOAT32/FLOAT64`

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

– `LONG`	-2147483648 … +2147483647
– `FLOAT` (until 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: $+2{,}2250738585072014 \cdot 10^{-308}$ … $+1{,}797693134862315 \cdot 10^{+308}$

With data type `LONG`, the functions transfer integer values with 32-bit width. Return values in MATLAB have the data type `double`.

If integer values contain a counter value or a bit pattern the value must be processed in MATLAB with data type `int32` or `uint32`; find more details in Data Types on page 6.

Data_Length

`Data_Length` returns the length of an *ADbasic* array of data type `LONG`, `FLOAT`, `FLOAT32`, or `FLOAT64`, that is the number of elements.

```
Data_Length (DataNo)
```

Parameters

`DataNo`	Array number (1...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 `DATA` array of the type `STRING` you use the instruction `String_Length`.

Example

In *ADbasic*, `DATA_2` is dimensioned as:
```
DIM DATA_2[2000] AS LONG
```

In MATLAB, you will have the length of the array `DATA_2`:
```
>> Data_Length(2)
ans =
    2000
```

SetData_Double

SetData_Double transfers data from a row vector of data type double into a DATA array of the *ADwin* system.

```
SetData_Double (DataNo, Vector, Startindex)
```

Parameters

DataNo	Number (1...200) of destination array DATA_1 … DATA_200. DATA may have data type Long, Float, Float32, or Float64.
Vector	Row vector, from which data are transferred.
StartIndex	Number (≥1) of the first element in the destination array, into which data is transferred.
Return value	≠255: OK 255: Error or array is not declared.

Notes

The DATA array must be greater than the number of values in the MATLAB vector plus Startindex.

If the data type of the DATA array has 32-bit precision, the 64-bit double values from Vector are converted, which causes a loss of decimal places.

Until T11, please note: float values in the *ADwin* system have 32-bit precision. You should therefore display data of Vector only with single precision to avoid misunderstandings.

If MATLAB data from more dimensional matrices is to be transferred the data has to be copied into a row vector first. In a column vector, the first data element will be transferred only.

The function SetData_Double replaces the function Set_Data, which was used with former driver versions.

Example

Write the complete row vector x into DATA_1, beginning at the array element DATA_1[100]:

```
SetData_Double(1,x,100);
```

GetData_Double

`GetData_Double` transfers parts of a `DATA` array from an *ADwin* system into a row vector of data type double.

```
GetData_Double (DataNo, Startindex, Count)
```

Parameters

`DataNo`	Number (1...200) of the source array `DATA_1` … `DATA_200`. `DATA` may have data type `Long`, `Float`, `Float32`, or `Float64`.
`StartIndex`	Number (≥1) of the first element in the source array to be transferred.
`Count`	Number (≥1) of the data to be transferred.
Return value	Row vector with transferred values of data type double.

Notes

Until T11, please note: float values in the *ADwin* system have 32-bit precision. You should therefore display data of the returned row vector only with single precision to avoid misunderstandings.

Even though an *ADbasic* array may be dimensioned 2-dimensional, the return value is always a row vector. If needed, the vector may be transformed into a matrix in MATLAB, e.g. using `reshape`.

There is more information about 2-dimensional arrays in chapter 4.4 on page 11.

The function `GetData_Double` replaces the function `Get_Data`, which was used with former driver versions.

Example

Transfer 1000 values from `DATA_1` starting from index 100 into row vector `x`:

```
x = GetData_Double(1, 100, 1000);
```

Data2File

`Data2File` saves data of type Long, Float/Float32, or Float64 from a `DATA` array of the *ADwin* system into a file (on the hard disk).

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

Parameters

`Filename`	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 first element in the source array to be transferred.
`Count`	Number (≥1) of the first data to be transferred.
`Mode`	Write mode: 0: File will be overwritten. 1: Data is appended to an existing file.
Return value	0: OK ≠0: Error

Notes

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

The data are saved as binary file in the appropriate MATLAB data type (see table). If not existing, the file will be created.

Data type of `DATA` array	Saved data type
Long	`int32`
Float (until processor T11) Float32 (processor T12/T12.1)	`single`
Float64 (Prozessor T12/T12.1)	`double`

Example

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

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

File2Data

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

File2Data (Filename, DataType, DataNo, 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 the values saved in the file. Select one of the following contants: 'type_integer' : Values of type int32 (32 bit). 'type_single' : Values of type single (32 bit). 'type_double' : Values of type double (64 bit).
DataNo	Number (1…200) of the destination array DATA_1 … DATA_200.
Startindex	Index (≥1) of the first element in the destination array to be written.
Return value	0: OK ≠0: Error

Notes

The file values are expected to be saved as binary in one of the formats int32, single or double.

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

If required, the values of the source file are automatically converted into the data type of the destination DATA array. There are the destination data types Long, Float/Float32, and Float64 (see table).

Saved data type	Data type of DATA array
int32	Long
single	Float (until processor T11) Float32 (processor T12/T12.1)
double	Float64 (processor T12/T12.1)

Example

In *ADbasic*, DATA_1 is dimensioned as:

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

In Matlab:
Copy values of type integer 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.

```
ret_val = File2Data('Test.dat', 'type_integer', 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 an FIFO array element depends on the data type:

– LONG	-2147483648 … +2147483647
– FLOAT (until 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: $-1{,}797693134862315 \cdot 10^{+308} \ldots -2{,}2250738585072014 \cdot 10^{-308}$ 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 always be used to check if there are values, which have not yet been read, before reading from the FIFO.

With data type LONG, the functions transfer integer values with 32-bit width. Return values in MATLAB have the data type double.

If integer values contain a counter value or a bit pattern the data type in MATLAB must be converted to int32 or uint32; find more details in Data Types on page 6.

Fifo_Empty

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

Fifo_Empty (FifoNo)

Parameters

FifoNo	Number (1...200) of the FIFO array DATA_1 … DATA_200.
Return value	≠255: Number of empty elements in the FIFO array. 255: Error

Example

In *ADbasic*, DATA_5 is dimensioned as:

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

In MATLAB, you will get the number of empty elements in DATA_5:

```
>> Fifo_Empty(5)
ans =
    68
```

Fifo_Full

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

Fifo_Full (FifoNo)

Parameters

FifoNo	Number (1...200) of the FIFO array DATA_1 … DATA_200.
Return value	≠255: Number of the used elements in the FIFO array. 255: Error

Example

In *ADbasic*, DATA_12 is dimensioned as:

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

In MATLAB, you will get the number of used elements in DATA_12:

```
>> Fifo_Full(12)
ans =
    2105
```

Fifo_Clear

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

Fifo_Clear (FifoNo)

Parameters

FifoNo	Number (1...200) of the FIFO array DATA_1 … DATA_200.
Return value	≠255: OK 255: Error

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
ret_val = Fifo_Clear(45);
```

SetFifo_Double

`SetFifo_Double` transfers data from a row vector into a FIFO array.

SetFifo_Double (FifoNo, Vector)

Parameters

FifoNo	Number (1...200) of the FIFO array DATA_1 … DATA_200.
Data	Row vector with values to be transferred.
Return value	≠255: OK 255: Error

Notes

You should first use the function `Fifo_Empty` to check, if the FIFO array has enough empty elements to hold all data of the row vector. If more data are transferred into the FIFO array than empty elements are given, the surplus data are overwritten and are definitively lost.

Until T11, please note: float values in the *ADwin* system have 32-bit precision. You should therefore display data of `Vector` only with single precision to avoid misunderstandings.

The function `SetFifo_Double` replaces the function `Set_Fifo`, which was used with former driver versions.

Example

Check FIFO array `DATA_12` for empty elements and transfer all elements of the row vector `vector` into the FIFO array:

```
num_fifo = Fifo_Empty(12);
num_vector = length(vector);
if num_fifo >= num_vector
 SetFifo_Double(12, vector);
end
```

GetFifo_Double

`GetFifo_Double` transfers FIFO data from a FIFO array to a row vector.

`GetFifo_Double` `(FifoNo, Count)`

Parameters

`FifoNo`	Number (1...200) of the FIFO array `DATA_1` … `DATA_200`.
`Count`	Number (≥1) of elements to be transferred.
Return value	Row vector with transferred values

Notes

You should first use the function `Fifo_Empty` to check, how much used elements the FIFO array has. If more data are read from the FIFO array than used elements are given, the surplus data is erroneous.

Until T11, please note: float values in the *ADwin* system have 32-bit precision. You should therefore display data of the returned row vector only with single precision to avoid misunderstandings.

The function `GetFifo_Double` replaces the function `Get_Fifo`, which was used with former driver versions.

Example

Query the number of used elements in the FIFO array `DATA_12` and transfer 200 values into the row vector `v`:

```
num_fifo = Fifo_Full(12);
if num_fifo >= 200
 v = GetFifo_Double(12, 200);
end
```

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.

The functions of this sections cannot be used in connection with *ADsim* T11.

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 number 0 … 127. The termination (ASCII character 0) 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.

`String_Length` `(DataNo)`

Parameters

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

Notes

This function cannot be used in connection with *ADsim* T11.

`String_Length` counts the characters in a `DATA` array up to the termination char (ASCII character 0). The termination char is not counted as character.

Example

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

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

In MATLAB, you will get the length of the array `DATA_2`:

```
>> String_Length(2)
ans =
    11
```

SetData_String

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

SetData_String (DataNo, String)

Parameters

DataNo	Number (1...200) of the FIFO array DATA_1 … DATA_200.
String	String variable or text in quotes, which is to be transferred.
Return value	≠-1: OK -1: Error

Notes

This function cannot be used in connection with *ADsim* T11.

`SetData_String` appends the termination char (ASCII character 0) to each transferred string.

Example

```
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 into a string variable.

GetData_String (DataNo, MaxCount)

Parameters

DataNo	Number (1...200) of the array DATA_1 … DATA_200.
MaxCount	Max. number (≥1) of the transferred characters without termination char.
Return value	String variable with the transferred chars.

Notes

This function cannot be used in connection with *ADsim* T11.

If the string in the `DATA` array contains a termination char, the transfer stops exactly there, that is the termination char will not be transferred.

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 a string of max. 100 characters from `DATA_2`:

```
string = GetData_String(2,100);
```

If the `DATA` array in the *ADwin* system has the termination char at position 9, then 8 characters are read.

5.5 Error handling

Show_Errors

Show_Errors enables or disables the display of error messages in a message box.

```
Show_Errors (OnOff)
```

Parameters

OnOff	0: Do not show any error messages. 1: Show error messages in a message box (default).

Notes

The function Show_Errors refers to all functions that may display error messages in a message box. These are:

- Boot
- Test_Version
- Load_Process

If message boxes are disabled with Show_Errors, the program keeps on running when an error occurs. The user cannot and does not have to confirm any error messages.

Example

```
% Show error messages
Show_Errors(1);
```

Get_Last_Error

Get_Last_Error returns the number of the error that occurred last in the interface adwin32.dll / adwin64.dll.

```
Get_Last_Error ()
```

Parameters

return value	0: no error ≠0: error number

Notes

To each error number you will get the text with the function Get_Last_Error_Text. You will find a list of all error messages in chapter A.2 of the Appendix.

After the function call the error number is automatically reset to 0.

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

Example

```
% Reading the previous error number
Error = Get_Last_Error();
```

Get_Last_Error_Text

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

`Get_Last_Error_Text` `(Last_Error)`

Parameters

`Last_Error`	Error number
Return value	Error text

Notes

Usually, the return value of the function `Get_Last_Error` is used as error number `Last_Error`.

Example

```
errnum = Get_Last_Error();
if errnum!=0
 pErrText = Get_Last_Error_Text(errnum);
end
```

Set_Language

`Set_Language` sets the language for the error messages.

`Set_Language` `(language)`

Parameters

`language`	Languages for error messages: 0: Language set in Windows 1: English 2: German
Return value	0

Notes

The instruction changes the language setting for the error messages of the interface `adwin32.dll` / `adwin64.dll` and for the function `Get_Last_Error_Text`.

If a different language than English and German is set under Windows, the error messages are displayed in English.

Example

```
% set english language for error messages
Set_Language(1);
```

ADwin_Debug_Mode_On

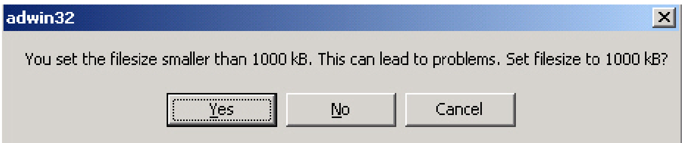
`ADwin_Debug_Mode_On` activates the debug mode. In debug mode, all function calls are logged in log files.

```
ADwin_Debug_Mode_On (Filename, Size)
```

Parameters

`Filename` Path and name of the log file. Enter the file name *without* extension but with the absolute path name!

`Size` Max file size in kByte (1000 = 1 MiB). If the given size is smaller than 1000 a warning is shown.



Return value
0 : OK
-1: File name > 255 characters (cannot be processed).
-2: Debug mode already activated (no effect)
-3: Access to the registry is needed, but impossible. Probably the user does not have the necessary administration rights for access or the max. size of the registry is exceeded. Please call your administrator.

Notes

We recommend *not* using this function in your application software. Instead, run the tool `<C:\ADwin\Tools\Test\DebugMode.exe>`, which has the same function.

If the debug mode is active the function calls to all *ADwin* systems and the answers are logged. The log may be useful for error handling (please contact the support division of Jäger Computergesteuerte Messtechnik GmbH).

If the size of the log file exceeds `Size`, additional files will be generated. The file extension is a consecutive number (001...nnn), which is automatically generated.

Please note:
- Set the file size to at least 1000 kByte.
- Deactive the debug mode with `ADwin_Debug_Mode_Off`, when you don't need it any more.

Otherwise, you will get a lot of log files, which slows down file management under Windows.

Example

```
ADwin_Debug_Mode_On('C:\temp\log', 1000)
```

In the `<C:\temp>` directory log files with the name `<log.nnn>` are generated.

ADwin_Debug_Mode_Off

`ADwin_Debug_Mode_Off` deactivates the debug mode.

```
ADwin_Debug_Mode_Off ()
```

Example

```
% Deactivate debug mode
ADwin_Debug_Mode_Off();
```

Annex

A.1 Program Examples

The following examples are written for a ***ADwin-Gold*** system, which is accessed via the Device No. `1`. You find the corresponding source files (and the appropriate binary files for ***ADbasic***) in the following directories:

– ***ADbasic***: `C:\ADwin\ADbasic\Samples_ADwin`

– MATLAB®: `C:\ADwin\Developer\Matlab\Samples`

We assume, that you load the process to your ***ADwin*** systen from ***ADbasic*** (source file). Alternatively, you can load the binary file from MATLAB® to the system with the instruction `Load_Process`.

If you use an other system (than ***ADwin-Gold***), you have to adjust the instruction `ADC` in the ***ADbasic*** programs. If you use a different Device No. than `1`, you have to set it in MATLAB® with the instruction `Set_DeviceNo`.

Online Evaluation of Measurement Data

BAS_DMO1

The ***ADbasic*** program described below writes the lowest and highest measurement values of the analog input channel 1 to the parameters `Par_1` and `Par_2`.

```
REM The program BAS_DMO1 searches the maximum and
REM minimum values out of 1000 measurements of ADC1
REM and writes the result to Par_1 and Par_2

DIM i1, iw, max, min AS LONG
INIT:
 i1 = 1
 max = 0
 min = 65535

EVENT:
 iw = adc(1)
 IF (iw>max) THEN max = iw
 IF (iw<min) THEN min = iw
 i1 = i1+1
 IF (i1>1000) THEN
  i1 = 1
  Par_1 = min : REM Write minimum value to Par_1
  Par_2 = max : REM Write maximum value to Par_2
  max = 0
  min = 65535
 ENDIF
```

From MATLAB® these data can be read out with the function `Get_Par(1)`:

```
% mat_dmo1.m
% Queries 5 times PAR_1 and PAR_2
Start_Process(1)
for i=1:5,
 min = Get_Par(1) % query Par_1 (minimum value)
 max = Get_Par(2) % query Par_2 (maximum value)
end ;
Stop_Process(1)
```

BAS_DMO2

Digital Proportional Controller

The ***ADbasic*** program described below is a digital proportional controller, which reads the setpoint from `PAR_1` and the gain factor from `PAR_2`.

```
REM The program BAS_DMO2 is a digital proportional
REM controller. The setpoint is defined by Par_1,
REM the gain by Par_2.

DIM deviation, actval AS LONG

EVENT:
 deviation = PAR_1 - ADC(1)
 actval = deviation * PAR_2 + 32768
 DAC(1, actval)
```

From MATLAB® the setpoint and the gain factor can be changed with the following instructions:

```
Set_Par(1, 17) ; % Change setpoint to 17
Set_Par(2, 3) ; % Change gain factor to 3
```

BAS_DMO3

Data Transfer

The ***ADbasic*** program described below writes measurement data into a DATA-array.

```
REM The program BAS_DMO3 measures the analog input 1
REM and writes the data to a DATA array
REM The data are transferred by using a DATA-array

DIM DATA_1[1000] AS LONG
DIM index AS LONG

INIT:
 Par_10 = 0
 index = 0               'reset array pointer
 Processdelay = 40000    'cycle-time of 1ms (T9)

EVENT:
 index = index + 1       'increment array pointer
 IF (index > 1000) THEN'1000 samples done?
  Par_10 = 1                 'set End-Flag
  END                    'terminate process
 ENDIF
 DATA_1[index] = ADC(1)'acquire sample and save in array
```

From MATLAB® the saved DATA array can be read:

```
% mat_dmo3.m
% liest den Datensatz DATA_1 ein.
Start_Process(1)
while x~=1
 x = Get_Par(10)
end
y1 = GetData_Double(1,1,1000); % Read DATA_1
plot(y1) ;
```

A.2 List of 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

A.4 Mapping of old function numbers to function names

In previous versions of the ***ADwin***-MATLAB® driver, function numbers were used instead of function names. The following table maps the old numbers to current function names, which perform the same or a similiar task.

Please note, that a function call with an old number may in parts use other parameters or in different order compared to the call with function names.

Nr.	task	calling syntax	function name
38	read global variable or status variable	ADlab(38, Index)	Get_Par (Index) Get_FPar (Index) Process_Status (ProcessNo) Get_Processdelay (ProcessNo)
34	set global variable or status variable	ADlab(34, Index, Value)	Get_Par (Index) Set_FPar (Index, Value) Set_Processdelay (ProcessNo, Processdelay)
50	start a process	ADlab(50, ProcessNo)	Start_Process (ProcessNo)
32	start a process	ADlab(32, ProcessNo)	Start_Process (ProcessNo)
51	stop a process	ADlab(51, ProcessNo)	Stop_Process (ProcessNo)
33	stop a process	ADlab(33, ProcessNo)	Stop_Process (ProcessNo)
104	read a Data-field into a vector	ADlab(104, DataNo, Count)	GetData_Double (DataNo, Startindex, Count)
105	write a vector into a Data-field	ADlab(105, DataNo, Vector)	SetData_Double (DataNo, Vector, Startindex)
106	read part of a Data-field into a vector	ADlab(106, DataNo, Count, Startindex)	GetData_Double (DataNo, Startindex, Count)
107	write a vector into a Data-field, starting from StartIndex	ADlab(107, DataNo, Vector, Startindex)	SetData_Double (DataNo, Vector, Startindex)
110	read a FIFO-field into a vector	ADlab(110, DataNo, Count)	GetFifo_Double (FifoNo, Count)
111	write a vector into a FIFO-field	ADlab(111, FifoNo, Vector)	SetFifo_Double (FifoNo, Vector)
112	query the number of used elements in a FIFO-field	ADlab(112, FifoNo)	Fifo_Full (FifoNo)
114	clear the contents of a FIFO-field	ADlab(114, FifoNo)	Fifo_Clear (FifoNo)
113	query the number of free elements in a FIFO-field	ADlab(113, FifoNo)	Fifo_Empty (FifoNo)
120	save part of a Data-field in a file	ADlab(120, Filename, DataNo, Count, Startindex)	Data2File (Filename, DataNo, Startindex, Count, Mode)
121	append part of a Data-field to a file	ADlab(121, Filename, DataNo, Count, Startindex)	Data2File (Filename, DataNo, Startindex, Count, Mode)
138	read Activate-PC flag	ADlab(138, ProzessNr)	No function name is available for the Activate-PC flag. *PAR_10* should be used alternatively.
200	set a DeviceNo	ADlab(200, DeviceNo)	Set_DeviceNo (DeviceNo)
253	query free memory of the ADwin system	x = ADlab(253)	Free_Mem (Mem_Spec)
254	query the workload of the ADwin system	x = ADlab(254)	Workload (Priority)
300	load the operating system (boot)	ADlab(300, Filename, Memsize)	Boot (Filename, MemSize)
310	load a process	ADlab(310, Filename)	Load_Process (Filename)

Nr.	task	calling syntax	function name
350	read all 80 global integer variables	ADlab(350)	Get_Par_All ()
351	read some global integer variables	ADlab(351, Startindex, Anzahl)	Get_Par_Block (StartIndex, Count)
352	read all 80 global float variables	ADlab(352)	Get_FPar_All ()
353	read some global float variables	ADlab(353, Startindex, Count)	Get_FPar_Block (StartIndex, Count)
400	read last error number	ADlab(400)	Get_Last_Error ()