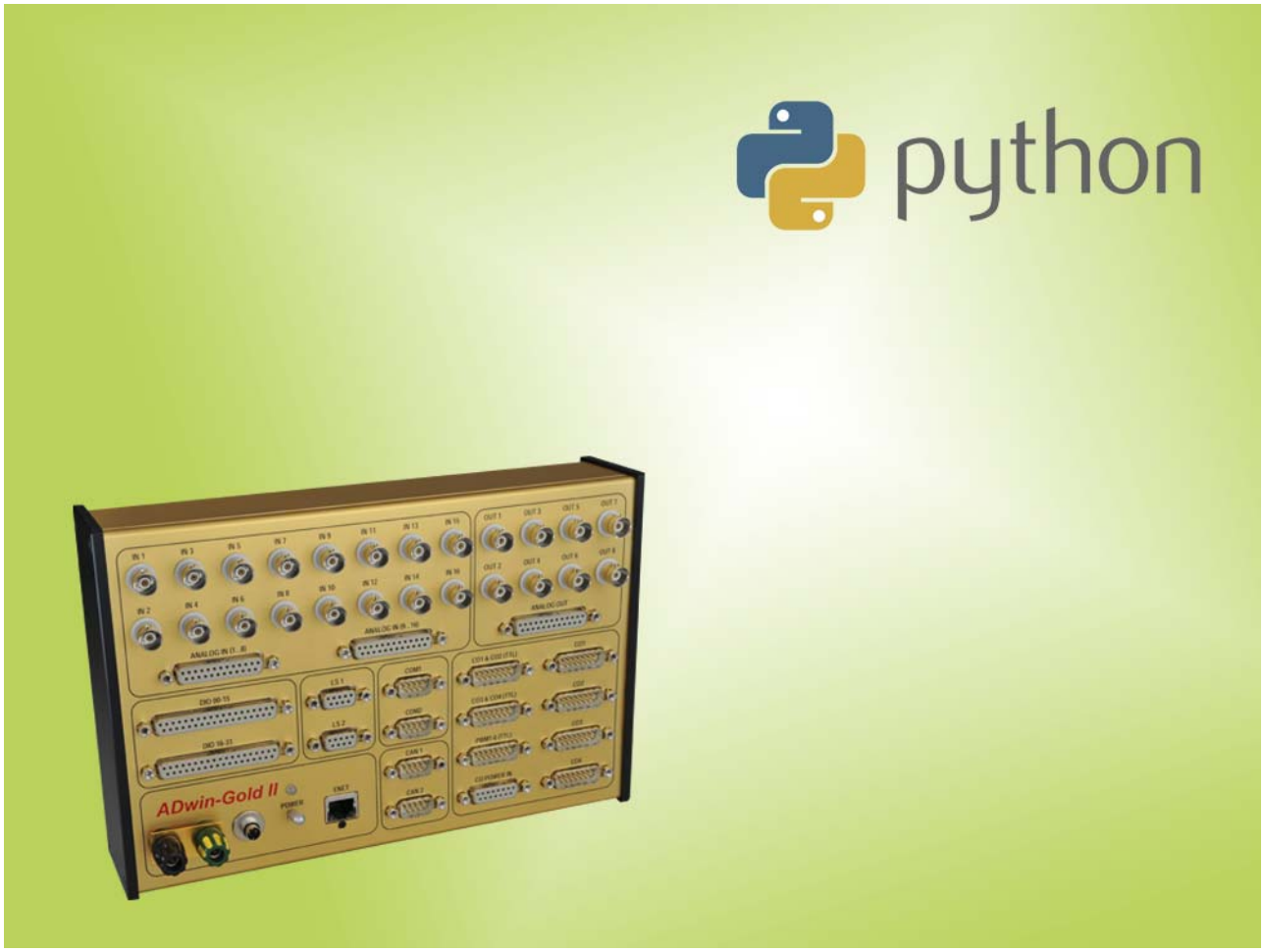
ADwin-Treiber

Treiber für Python



ADwin Treiber Python, Handbuch Jul. 2022

Hier finden Sie immer einen Ansprechpartner für Ihre Fragen:

Hotline:	(0 62 51) 9 63 20
Fax:	(0 62 51) 5 68 19
E-Mail:	info@ADwin.de
Internet	www.ADwin.de

JÄGER
Computergesteuerte
Messtechnik GmbH

Jäger Computergesteuerte
Messtechnik GmbH
Rheinstraße 2-4
D-64653 Lorsch

Inhaltsverzeichnis

Typografische Konventionen

Das „Achtung“-Zeichen steht bei Informationen, die auf Folgeschäden durch Fehlbedienung an der Hard- oder Software, am Messaufbau oder an Personen hinweisen.

Einen „Hinweis“ finden Sie bei

– Informationen, die für einen fehlerfreien Betrieb unbedingt beachtet werden müssen.

– Tipps und Ratschlägen für einen effizienten Betrieb.

Das Zeichen „Information“ verweist auf weiterführende Informationen in dieser Dokumentation oder andere Quellen wie Handbücher, Datenblätter, Literatur etc.

Das Zeichen „Beispiel“ weist auf praktische Beispiele hin.

C:\ADwin\…

Dateinamen und -verzeichnisse sind in spitzen Klammern und im Schrifttyp Courier New angeben.

Programmtext

Programmanweisungen und Benutzer-Eingaben sind durch den Schrifttyp Courier New gekennzeichnet.

```
Var_1
```

Elemente eines Quelltextes wie Befehle, Variablen, Kommentar und sonstiger Text werden im Schrifttyp Courier New und farbig dargestellt.

In einem Datenwort (hier: 16 Bit) werden die Bits wie folgt nummeriert:

Bit-Nr.	15	14	13	…	1	0
Wert des Bits	2^{15}	2^{14}	2^{13}	…	2^1=2	2^0=1
Bezeichnung	MSB	-	-	-	-	LSB

1 Zu diesem Handbuch

Dieses Handbuch enthält umfassende Informationen für den Einsatz des *ADwin*-Python-Treibers.

Folgende Dokumente ergänzen die Treiberbeschreibung:

– Das Handbuch „*ADwin* Installation“ beschreibt die Hardware-Schnittstellen-Installation zu allen *ADwin*-Geräten.

  Beginnen Sie Ihre Installation mit diesem Handbuch.

– Die Handbücher „*ADbasic*“, „*ADsim*“ und „*ADwinC*“. Mit jedem der Programmpakete können Sie Ihre *ADwin*-Hardware programmieren und *ADwin*-Prozesse darauf laufen lassen.
  - *ADbasic* umfasst die Entwicklungsumgebung und die Befehle des Compilers *ADbasic*.
  - Mit dem Programmpaket *ADsim* werden Simulink®-Modelle auf *ADwin*-Systemen lauffähig.
  - Mit ADwinC für Visual Studio können Sie ADwin-Prozesse in der Sprache C entwickeln.

– Die Hardware-Handbücher für Ihre *ADwin*-Hardware.

Es wird vorausgesetzt, dass Sie die Sprache Python beherrschen.

Bitte beachten Sie folgende Hinweise

Damit Ihr *ADwin*-System sicher arbeitet, halten Sie sich an die Informationen dieser und weiterführender Dokumentationen, auf die hier verwiesen wird.

Einschränkung der Anwendergruppe

Der Hersteller des in dieser Dokumentation beschriebenen Systems geht davon aus, dass an dem Gerät nur qualifiziertes Personal arbeitet.

> *Qualifiziertes Personal sind Personen, die aufgrund ihrer Ausbildung, Erfahrung und Unterweisung sowie ihrer Kenntnisse über einschlägige Normen, Bestimmungen, Unfallverhütungsvorschriften und Betriebsverhältnisse von dem für die Sicherheit der Anlage Verantwortlichen berechtigt worden sind, die jeweils erforderlichen Tätigkeiten auszuführen und die dabei mögliche Gefahren erkennen und vermeiden können. (Definition für Fachkräfte nach VDE 105 und IEC 60364).*

Verfügbarkeit der Unterlagen

Diese Produktdokumentation und Unterlagen, auf die verwiesen wird, müssen stets verfügbar sein und konsequent beachtet werden. Für Schäden, die durch Missachtung der Informationen in dieser bzw. der weiterführenden Dokumentation entstehen, übernimmt die Firma *Jäger Computergesteuerte Messtechnik GmbH*, Lorsch, keine Haftung.

Rechtliche Grundlagen

Fremdprodukte werden ohne Vermerk auf mögliche Patentrechte genannt, deren Existenz nicht auszuschließen ist.

Änderungen vorbehalten.

Hotline-Adresse siehe vordere Umschlagseite, innen.

2 *ADwin*-Treiber für Python

Das *ADwin*-System besteht aus einem eigenständigen Messrechner, der Mess- und Regelaufgaben extrem schnell und sicher erledigt, und einer Schnittstelle unter Windows, Linux oder Mac OS, über die Sie mit Python das *ADwin*-System steuern.

Sie verlagern also alle zeitkritischen Prozesse in das *ADwin*-System, haben aber die Steuerung der Prozesse und Verarbeitung der Daten weiterhin mit Python in der Hand.

Beachten Sie: Sie benötigen einen Python-Interpreter in einer Version ab 2.4. Ältere Versionen sind nicht getestet.

Wie Sie das *ADwin*-System programmieren

ADwin-Systeme sind schnell, zuverlässig und flexibel. Sie legen das Verhalten der *ADwin*-Hardware selbst fest. Hierzu gibt es folgende Wege:

- *ADbasic*: Sie erstellen Echtzeit-Prozesse mit der Entwicklungsumgebung *ADbasic*, erzeugen daraus eine Binärdatei und übertragen sie auf das *ADwin*-System (siehe Handbuch oder Online-Hilfe *ADbasic*).
- *ADwinC*: Sie erstellen Echtzeit-Prozesse in Visual Studio in der Sprache C mit dem Programmpaket ADwinC für Visual Studio. Sie erzeugen eine Binärdatei und übertragen sie auf das *ADwin*-System (siehe Handbuch *ADwinC*).
- *ADsim*: Sie erstellen in Simulink ein Modell, exportieren es und kompilieren es mit *ADsim* für die *ADwin*-Hardware (siehe Handbuch *ADsim*).

Bevor Sie die Python-Befehle anwenden können, empfehlen wir Ihnen eine Einarbeitung in *ADbasic*. Hierzu verwenden Sie bitte das *ADbasic*-Handbuch und die Programmieranleitung. Die Beschreibungen werden Ihnen auch das Verständnis des *ADwin*-Systems erleichtern.

ADwin-Systeme mit Python steuern

Jetzt ist der Moment gekommen, mit diesem Handbuch den Schritt in die Praxis zu tun.

Die Abschnitte 2.1 und 2.2 schildern, wie Python und *ADwin* kommunizieren und vertiefen Ihr Verständnis für das *ADwin*-Konzept.

In Kapitel 3 wird die Installation und Einbindung der neuen Befehle beschrieben.

Allgemeines zum Python-Treiber ist in Kapitel 4 erklärt, die einzelnen Funktionen in Form eines als Nachschlagewerks in Kapitel 5.

2.1 Schnittstelle zur Entwicklungsumgebung

Der *ADwin*-Python-Treiber ist die Schnittstelle für die Sprache Python zur Kommunikation mit *ADwin*-Hardware.

Die Kombination der Sprache Python mit einem *ADwin*-Hardware-System bietet Ihnen völlig neue Möglichkeiten. Die Intelligenz und Rechenleistung der *ADwin*-Hardware zum einen und die vielfältigen Python-Funktionen zum Verwalten, Analysieren und Dokumentieren der Messdaten zum anderen bilden ein leistungsstarkes Gespann zum Regeln, Messen und Steuern.

Typische Anwendungen sind:

– Steuerung schneller Prüfstände
– Signale generieren
– Intelligent messen, Daten mit komplexen Triggerbedingungen erfassen
– Regeln und Steuern
– Online-Verarbeitung, Datenreduzierung
– Hardware-in-the-Loop, Simulation von Sensordaten

2.2 Kommunikation mit der *ADwin*-Hardware

Aus der Entwicklungsumgebung können Sie Prozesse in der *ADwin*-Hardware steuern, Daten von dort auslesen oder dorthin senden. Die Prozesse selbst programmieren Sie mit dem Echtzeit-Entwicklungstool *ADbasic*, erzeugen daraus eine Binärdatei und übertragen sie auf die *ADwin*-Hardware (siehe Handbuch oder Online-Hilfe *ADbasic*).

Daten und Befehle zwischen Python und *ADwin*-Hardware durchlaufen den nachfolgend dargestellten Weg.

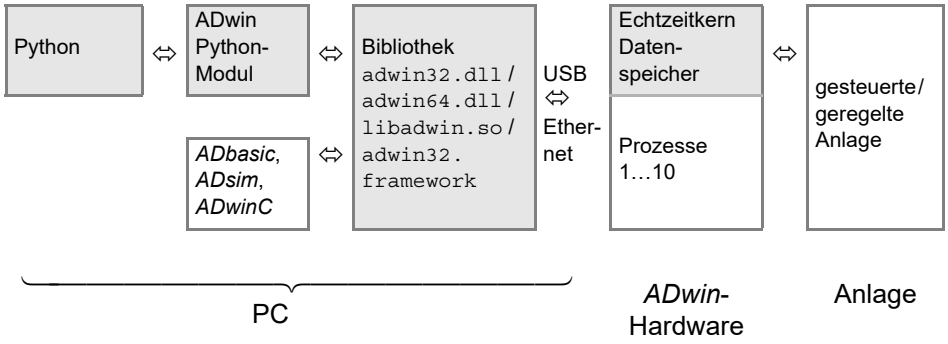


Abb. 1 – *ADwin*-Python Schnittstelle

adwin32.dll

Die Bibliothek (Windows: `adwin32.dll` / `adwin64.dll`, Linux: `libadwin.so`, Mac OS: `adwin32.framework`) ist die zentrale Schnittstelle zur *ADwin*-Hardware für alle Anwendungen und wird daher auch vom *ADwin*-Python-Treiber genutzt. Die Schnittstelle ermöglicht, dass mehrere Programme gleichzeitig mit der *ADwin*-Hardware kommunizieren: So können verschiedene Anwendungen gleichzeitig mit *ADwin*-Hardware arbeiten, unter Windows z.B. Python, *ADbasic* und *ADtools*.

Echtzeitkern

Die Bibliotheksfunktionen kommunizieren mit dem Echtzeitkern der *ADwin*-Hardware, dem Betriebssystem. Deshalb müssen Sie nach jedem Einschalten der Hardware zunächst das Betriebssystem (in Form einer Datei wie `adwin11.btl`) dorthin laden. Nach erfolgreicher Übertragung kann die Hardware Prozesse empfangen und ausführen, Befehle vom PC entgegen nehmen und Daten mit ihm austauschen. Die in *ADbasic* programmierten Prozesse enthalten den Programmcode zur Messung, Steuerung oder Regelung Ihrer Applikation.

Die Aufgaben des Betriebssystems sind:

10 Prozesse

– Verwaltung von bis zu 10 Echtzeit-Prozessen mit niedriger oder hoher Priorität (frei wählbar). Niedrig priorisierte Prozesse können von hoch priorisierten Prozessen unterbrochen werden, letztere können nicht von anderen Prozessen unterbrochen werden.

Datenspeicher

– Bereitstellung von globalen Variablen:
  - 80 Integer-Variablen (`Par_1` … `Par_80`), bereits vordefiniert.
  - 80 Float-Variablen (`FPar_1` … `FPar_80`), bereits vordefiniert.
  - 200 Datenfelder (`DATA_1` … `DATA_200`), Länge und Datentyp sind frei definierbar.

  Sie können die Werte der globalen Variablen und Datenfelder jederzeit lesen und ändern.

Kommunikation

– Kommunikation zwischen *ADwin*-Hardware und PC (via Programmbibliothek).

  Der Kommunikationsprozess läuft mit mittlerer Priorität auf der *ADwin*-Hardware und kann niedrig priorisierte Prozesse für kurze Zeit unterbrechen. Der Kommunikationsprozess interpretiert oder bearbeitet alle Befehle, die Sie vom PC an die *ADwin*-Hardware richten: Steuerbefehle und Befehle für den Datenaustausch.

  Die folgende Tabelle zeigt Beispiele aus jeder Gruppe.

Steuerbefehle, z. B.	
`Load_Process`	überträgt einen Prozess auf die Hardware.
`Start_Process`	startet einen Prozess.
Befehle für den Datenaustausch, z. B.	
`Get_Par`	liefert den aktuellen Wert eines Parameters.
`Set_Par`	ändert den Wert eines Parameters.
`GetData_Long`	liefert die Werte aus einem `DATA`-Feld.

Der Kommunikationsprozess sendet niemals unaufgefordert Daten an den PC. Das stellt sicher, dass nur dann Daten zum PC übertragen werden, wenn Sie diese ausdrücklich angefordert haben.

3 *ADwin* Python-Treiber installieren

3.1 „*ADwin* Installation“ ausführen

Für die Installation benötigen Sie das aktuelle *ADwin*-Software-Paket.

3.1.1 Installation unter Linux oder Mac OS

Folgen Sie der Installationsanleitung im Handbuch „ADwin für Linux / Mac“.

Nach erfolgreicher Installation finden Sie die Dateien in den folgenden Verzeichnissen unterhalb von `</opt/adwin/share>` (Standardinstallation):

Treiber und Beispiele für Python	`./python`
Dokumentation zum *ADwin*-Modul	`./doc/python`
Beispiele für *ADbasic*	`./examples/samples_ADwin`

Fahren Sie fort mit Kapitel 3.2 „ADwin-Modul installieren“.

3.1.2 Installation unter Windows

Falls *ADwin* installiert ist

Wenn Sie bereits *ADwin*-Hardware und -Software installiert haben, können Sie diesen Abschnitt überspringen und mit Kapitel 3.2 weiter arbeiten.

Sonst: Neue Installation

Falls *ADwin*-Hardware neu installiert werden soll, beginnen Sie bitte die Installation mit dem Handbuch „*ADwin* Installation“, das mit der *ADwin*-Hardware ausgeliefert wird. Es beschreibt, wie Sie

– das *ADwin*-Software-Paket installieren.

– die Kommunikations-Treiber unter Windows installieren.

– die Hardware im PC einbauen (falls erforderlich) und die Hardware-Verbindung zwischen PC und *ADwin*-Hardware aufbauen.

Nach erfolgreicher Installation finden Sie die Dateien in den folgenden Ordnern unterhalb von `<C:\ADwin\>` (Standardinstallation):

Treiber und Beispiele für Python	`.\Developer\Python\…`
Beispiele für *ADbasic*	`.\ADbasic\samples_ADwin`
Testprogramm für *ADwin-Gold*, *ADwin-light-16* und Einsteckkarten	`.\Tools\Test\ADtest`
Testprogramm für *ADwin-Pro*	`.\Tools\Test\ADpro`

Fahren Sie fort mit dem nächsten Abschnitt „ADwin-Modul installieren“.

3.2 *ADwin*-Modul installieren

Wenn Sie in Python mit der *ADwin*-Hardware arbeiten wollen, müssen Sie das *ADwin*-Modul installieren.

Folgen Sie diesen Schritten:

– Geben Sie in der Kommandozeile folgenden Aufruf ein:
  - Windows: `$> python setup.py install`
  - Linux / Mac: `$> python ./setup.py install`

  Nun wird das *ADwin*-Modul in das Verzeichnis `site-packages` der Python-Installation kopiert.

– Alternativ können Sie das *ADwin*-Modul aus dem „Python Package Index“ installieren und aktualisieren. Dokumentation und Beispielprogramme sind dabei jedoch nicht enthalten.

  Dazu geben Sie in der Kommandozeile folgenden Aufruf ein:
  - Installieren: `$> pip install adwin`
  - Aktualisieren: `$> pip install adwin --upgrade`

– Das *ADwin*-Modul steht nun zur Verfügung.

  Mit `import ADwin` binden Sie das *ADwin*-Modul ein und mit `adw = ADwin.ADwin(DeviceNo=1, raiseExceptions=1)` erstellen Sie eine neue Instanz der *ADwin*-Klasse mit dem Namen `adw`.

Die Funktionen des Treibers sind in Kapitel 5 beschrieben. Eine alphabetische Liste der Funktionen befindet sich im Abschnitt A.2.

3.3 Zugriff auf die *ADwin*-Hardware testen

Bei der Installation der Hardware und -Software haben Sie bereits den Zugriff auf die *ADwin*-Hardware erfolgreich geprüft. Testen Sie nun mit einem Beispielprogramm aus Kapitel A.1 im Anhang, ob Sie aus Python korrekt auf die *ADwin*-Hardware zugreifen können.

Wenn das Beispielprogramm korrekt funktioniert, können Sie auch mit allen Funktionen des Treibers auf die *ADwin*-Hardware zugreifen.

3.4 *ADwin*-Hardware über andere PCs ansprechen

Wenn *ADwin*-Hardware an einem Host-Rechner angeschlossen, aber in einem Ethernet-Netzwerk nicht direkt ansprechbar ist, können Sie die Verbindung mit dem Programm `ADwinTcpipServer` dennoch herstellen.

Nähere Informationen zur Anwendung von `ADwinTcpipServer` finden Sie in der Online-Hilfe des Programms.

4 Allgemeines zu *ADwin*-Funktionen

4.1 Fehler erkennen

Es gibt folgende Methoden, Fehler während des Programmablaufs zu erkennen und zu verarbeiten:

– Exception auslösen bei Laufzeitfehlern (Ausnahmebehandlung)

  Jeder Fehler löst eine Exception aus, die in einem separaten Programmteil behandelt wird.

  Wir empfehlen, Fehler mit dieser Methode zu verarbeiten.

– Fehlercode explizit abfragen mit `Get_Last_Error` (nächste Seite)

  Sie müssen nach jedem Zugriff auf *ADwin*-Hardware die Fehlernummer abfragen und entsprechend behandeln.

  Diese Methode ist sinnvoll, wenn Sie keine Exceptions verwenden

– Rückgabewert von Funktionen nutzen (nächste Seite)

  Bei einigen Befehlen enthält der Rückgabewert einen Fehlercode, mit dem Sie Fallunterscheidungen im Programmablauf treffen können.

  Da nicht alle Befehle einen Fehlercode zurückgeben, ist eine vollständige Fehlerbehandlung nicht möglich.

4.1.1 Exception auslösen

Sie können das *ADwin*-Python-Modul so einstellen, dass es bei Laufzeitfehlern eine Exception mit dem Namen `ADwinError` auslöst. Strukturieren Sie dazu das Programm wie folgt:

```
# ADwin-Funktionen und Fehlerroutinen bereitstellen
from ADwin import ADwin, ADwinError

# eine Instanz der ADwin-Klasse anlegen,
#  Exception auslösen
adw = ADwin(DeviceNo=1, raiseExceptions=1)

# Fehlerbehandlung mit try / except
try:
    …                   # das python-Programm

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

Das Klassenattribut `raiseExceptions` bestimmt das Verhalten des Moduls bei einem Laufzeitfehler. Mit dem Wert 1 löst das Modul bei einem Laufzeitfehler eine Exception aus. Wenn Sie mit dem Wert 0 Exceptions unterdrücken, müssen Sie nach jedem Zugriff auf *ADwin*-Hardware den Fehlercode explizit abfragen (siehe unten).

Im Programmabschnitt `try` geben Sie das Python-Programm ein, das auch die Zugriffe auf *ADwin*-Hardware enthält. Tritt ein Fehler auf, wird der Programmzweig mit Exception-Typ `ADwinError` ausgeführt.

4.1.2 Fehlercode explizit abfragen

Wenn Sie das Klassenattribut `raiseExceptions` so einstellen, dass es bei Laufzeitfehlern keine Exception auslöst, müssen Sie nach jedem Zugriff auf *ADwin*-Hardware die Fehlernummer mit der Funktion `Get_Last_Error` abfragen und entsprechend behandeln.

Zu jeder Fehlernummer erhalten Sie den zugehörigen Klartext mit der Funktion `Get_Error_Text`. Eine Liste aller Fehlermeldungen finden Sie im Abschnitt A.2 im Anhang.

Die Funktionen `Get_Last_Error` und `Get_Error_Text` sind ab Seite 41 beschrieben.

Im folgenden Beispiel wird auf das undefinierte Feld `DATA_1` zugegriffen; der auftretende Fehler löst (wegen der Einstellung von `raiseExceptions` = 0) keine Exception aus. Stattdessen wird der Fehler mit `Get_Last_Error` explizit abgefragt:

```
>>> 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 Rückgabewert von Funktionen nutzen

Bei einigen Funktionen enthält der Rückgabewert einen Fehlercode, den Sie für Fallunterscheidungen im Programmablauf nutzen können.

Beachten Sie bitte:

– Wenn Laufzeitfehler eine Exception auslösen (siehe Kapitel 4.1.1), geben die Funktionen keinen Fehlercode zurück (Typ `noneType`).

– Die Funktionen verwenden unterschiedliche Werte, um einen Fehler anzuzeigen.

– Der zurückgegebene Fehlercode hat nichts mit der Liste der Fehlermeldungen im Anhang zu tun.

– Der Rückgabewert ist nicht immer eindeutig. Wenn z.B. `Get_Processdelay` den Wert 255 zurückgibt, ist unklar, ob ein Fehler aufgetreten ist oder ob der Parameter `Processdelay` den Wert 255 enthält.

  Bei den folgenden Funktionen ist der Rückgabewert nicht eindeutig, d.h. er kann als Fehler oder als Wert verstanden werden:

  - `Fifo_Empty`
  - `Fifo_Full`
  - `Get_Par`
  - `Get_FPar`
  - `Get_Processdelay`
  - `Free_Mem`

  Für eine eindeutige Fehlerbehandlung müssen Sie eine Exception auslösen oder den Fehlercode explizit abfragen.

4.2 Die „DeviceNo.“

Eine „Device No.“ ist die Gerätenummer einer bestimmten *ADwin*-Hardware an einem PC. *ADwin*-Hardware wird immer über die zugehörige „Device No.“ angesprochen.

Sie legen die „Device No.“ für jede *ADwin*-Hardware mit dem Programm *ADconfig* an. Nähere Informationen zur Programmbedienung finden Sie in der Hilfe von *ADconfig*. Unter Windows ist eine Online-Hilfe verfügbar; unter Linux rufen Sie die Hilfe auf mit:

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

Sie geben die verwendete `DeviceNo` beim Anlegen einer Instanz einer *ADwin*-Klasse an; der Standardwert ist 1. Sie legen also für jede *ADwin*-Hardware eine eigene Instanz an.

4.3 Datentypen

Die Funktionen und Parameter des *ADwin*-Python-Treibers verwenden die Python Standard-Typen `int`, `float` und `str` sowie verschiedene C-kompatible Typen der Bibliothek `ctypes`.

Wenn Sie mit `numpy`-Feldern arbeiten möchten, konvertieren Sie die `ctype`-Felder mit der Funktion `numpy.ctypeslib.as_array()`.

Im Unterschied zu Python verwendet *ADbasic* üblicherweise folgende Datentypen:

Datentyp	Definition
`String`	unsigned integer 32 Bit
`Long`	signed integer 32 Bit
`Float` (bis T11) `Float32`	float 32 Bit
`Float64`	float 64 Bit

Bei 32 Bit-Fließkommazahlen bis Prozessor T11 werden Bitmuster von ungültigen oder grenzwertigen Werten in der *ADwin*-Hardware bei der Übertragung auf den PC in andere Werte gewandelt, siehe folgende Tabelle. Zahlen im gültigen Wertebereich (normalized numbers) bleiben unverändert.
Beim Prozessor T12 werden dagegen die IEEE-Bezeichnungen wie `#INF` verwendet.

Bezeichnung gemäß IEEE	Bitmuster-Bereich	32 Bit-Wert auf dem PC
+0	`00000000h`	0
Positive denormalized numbers	`00000001h` `007FFFFFh`	0
Positive normalized numbers	`00800000h` `7F7FFFFFh`	+1,175494 · 10-38 +3,402823 · 10+38

Bezeichnung gemäß IEEE	Bitmuster-Bereich	32 Bit-Wert auf dem PC
+∞ (Infinity, #INF)	7F800000h	3.402823E+38
Signalling Not a number (SNaN)	7F800001h 7FBFFFFFh	3.402823E+38
Quiet 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
Signalling Not a number (SNAN)	FF800001h FFBFFFFFh	3.402823E+38
Indeterminate	FFC00000h	3.402823E+38
Quiet Not a number (QNAN)	FFC00001h FFFFFFFFh	3.402823E+38

4.4 2-dimensionale Felder

In *ADbasic* können globale `Data`-Felder 2-dimensional (2D) deklariert werden. Die Funktionen des *ADwin*-Treibers für Python verwenden an dieser Stelle jedoch nur eindimensionale Felder.

Allgemein gilt für die Zuordnung eines Elements in einem 2D-Feld aus *ADbasic* zu einem Element in einem 1D-Feld aus Python:

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

Hierbei ist `s` die 2. Dimension von `DATA_n` bei der Deklaration in *ADbasic*.

Als Beispiel sei ein 2D-Feld in *ADbasic* deklariert mit

```
DIM DATA_8[7][3] AS FLOAT 'd.h. s=3
```

Die 7×3 Elemente des Felds werden in Python mit `GetData_Float` gelesen:

```
# Elemente 1…21 aus DATA_8 in array übertragen
array = adw.GetData_Float(8,1,21)
```

Die Daten werden in der folgender Reihenfolge übertragen. Bitte beachten Sie, dass in Python Feld-Indizes bei 0 beginnen, in *ADbasic* aber bei 1:

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

Die Funktion `GetData_Float` legt das Element `DATA_8[7][2]` also in `array[19]` ab.

Die allgemeine Formel ergibt mit `s=3`:

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 Beschreibung der *ADwin*-Funktionen

Die Beschreibung der Funktionen ist in folgende Abschnitte unterteilt:

Im Anhang A.3 finden Sie eine Übersicht aller Funktionen.

Beachten Sie auf jeden Fall das Kapitel 4, in dem allgemeine Hinweise zur Verwendung der *ADwin*-Funktionen beschrieben sind.

Befehle zum Ansprechen analoger und digitaler Ein- und Ausgänge sind im *ADwin*-Python-Treiber nicht enthalten. Sie können solche Anwendungen in *ADbasic* programmieren.

Programmstruktur

Zu jedem Python-Programm gehört folgende Programmstruktur:

```
# ADwin-Funktionen und Fehlerroutinen bereitstellen
from ADwin import ADwin, ADwinError

# eine Instanz der ADwin-Klasse anlegen, Exception
auslösen
adw = ADwin(DeviceNo=1, raiseExceptions=1)

# Fehlerbehandlung mit try / except (siehe Kapitel 4.1)
try:
  …                    # zu überwachender Programmzweig

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

Im folgenden wird bei den Beispielen der *ADwin*-Funktionen davon ausgegangen, dass mit `adw` bereits eine Instanz der *ADwin*-Klasse angelegt wurde.

5.1 Hardwaresteuerung und -information

Initialisierung der *ADwin*-Hardware und Information über den Betriebszustand.

Boot

`Boot` intialisiert die *ADwin*-Hardware und lädt die Betriebssystem-Datei dorthin.

```
Boot(str Filename)
```

Parameter

`Filename` Pfad und Dateiname der Betriebssystemdatei (siehe Tabelle unten).

Bemerkungen

Die Initialisierung löscht alle Prozesse auf der Hardware und setzt alle globalen Variablen auf den Wert 0.

Die zu ladende Betriebssystemdatei ist abhängig vom Prozessortyp der angesprochenen Hardware. Die folgende Tabelle zeigt die Dateinamen für die verschiedenen Prozessoren.

Die Dateien sind im Verzeichnis `<C:\ADwin\>` bzw. `/opt/adwin/share/btl/` abgelegt, auf das Sie mit dem Klassenattribut `adw.ADwindir` zugreifen können.

Prozessor	Betriebssystemdatei
T9	`ADwin9.btl`
	`ADwin9s.btl`[1]
T10	`ADwin10.btl`
T11	`ADwin11.btl`
T12	`ADwin12.btl`
T12.1	`ADwin121.btl`

Sie können auch Prozessoren vom Typ T2...T8 verwenden; wenden Sie sich hierzu an unseren Support (Adresse siehe vordere Umschlagseite, innen).

Der PC kann erst mit der *ADwin*-Hardware kommunizieren, nachdem das Betriebssystem geladen ist. Laden Sie das Betriebssystem nach jedem Aus- und Einschalten der *ADwin*-Hardware neu.

Das erfolgreiche Laden des Betriebssystems mit `Boot` dauert etwa eine Sekunde, bei T12.1 etwa sechs Sekunden. Alternativ können Sie das Betriebssystem auch über die *ADbasic* Entwicklungsumgebung laden (Schaltfläche B).

Beispiel

```
# Betriebssystem für Prozessor T10 laden
adw.Boot(adw.ADwindir + '\\ADwin10.btl')
```

1. Optimiertes Betriebssystem mit etwas geringerem Speicherbedarf.

Test_Version

`Test_Version` prüft, ob das richtige Betriebssystem für den Prozessor geladen wurde, und ob der Prozessor ansprechbar ist.

```
int Test_Version()
```

Parameter

Rückgabewert 0: OK
≠0:Fehler.

Beispiel

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

Processor_Type

`Processor_Type` gibt den Prozessortyp des Hardwares zurück.

```
str Processor_Type()
```

Parameter

Rückgabewert String für den Prozessortyp der Hardware.

unknown	T9
T2	T10
T4	T11
T5	T12
T8	T12.1

Beispiel

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

Workload

`Workload` gibt die durchschnittliche Prozessor-Auslastung seit dem vorigen Aufruf von `Workload` zurück.

```
int Workload()
```

Parameter

Rückgabewert 0…100: Prozessor-Auslastung (in Prozent)
255: Fehler

Bemerkungen

Die Prozessorauslastung wird für den Zeitraum zwischen dem vorherigen und dem aktuellen Aufruf von `Workload` ermittelt. Wenn Sie die aktuelle Prozessorauslastung benötigen, müssen Sie die Funktion 2fach und mit einem geringen Zeitabstand (etwa 1ms) aufrufen.

Beispiel

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

Free_Mem

`Free_Mem` ermittelt den auf der Hardware verfügbaren freien Speicher für verschiedene Speicherarten.

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

Parameter

`Mem_Spec`	Speicherart: 0 : alle Speicherarten gemeinsam (nur für T2, T4, T5, T8) 1 : interner Programmspeicher (PM_LOCAL); T9…T11 2: interner Zusatzspeicher (EM_LOCAL); nur T11 3 : interner Datenspeicher (DM_LOCAL); T9…T11 4 : externer DRAM-Speicher (DRAM_EXTERN); T9…T11 5 : Speicher, der Daten an den Cache liefern kann (CACHEABLE); nur T12/T12.1. 6 : Speicher, der keine Daten an den Cache liefern kann (UNCACHEABLE); nur T12/T12.1.
Rückgabewert	≠255: Zusammenhängender freier Speicher in Byte, bei `Mem_Spec`=5/6 in kByte. 255: möglicherweise Fehler

Beispiel

```
# Abfrage des freien Speichers (Cacheable)
print('Free_Mem:', adw.Free_Mem(5), 'kBytes')
```

5.2 Prozess-Steuerung

Die Steuerung von *ADbasic*- und *TiCoBasic*-Prozessen ist grundsätzlich verschieden:

– ADbasic-Prozesse
– TiCoBasic-Prozesse

5.2.1 ADbasic-Prozesse

Befehle zur Steuerung einzelner *ADbasic*-Prozesse auf dem *ADwin*-Hardware.

Es gibt die Prozesse 1…10 und 15. Die Prozesse haben folgende Funktion:

– 1…10: Sie selbst programmieren die Funktion in *ADbasic*.
– 15: Steuerung der Blink-LED bei *ADwin-Gold* und *ADwin-Pro*.

Der Prozess 15 ist Bestandteil des Betriebssystems und wird nach dem Booten automatisch gestartet. Weitere Informationen finden Sie im *ADbasic*-Handbuch, Kapitel „Prozessverwaltung“.

Load_Process

`Load_Process` lädt eine Binärdatei als Prozess in die *ADwin*-Hardware.

```
Load_Process(str Filename)
```

Parameter

`Filename`	Pfad und Dateiname der zu ladenden Binärdatei.

Bemerkungen

Sie erzeugen Binärdateien in *ADbasic* mit „`Build` ▸ `Make Bin file`“.

Aus- und Einschalten der Hardware löscht geladene Prozesse. Sie müssen nach dem Einschalten die benötigten Prozesse erneut laden.

Sie können bis zu 10 Prozesse auf die *ADwin*-Hardware übertragen. Laufende Prozesse werden durch das Laden weiterer Prozesse (mit unterschiedlicher Prozessnummer) in die Hardware nicht beeinflusst.

Bevor Sie den Prozess ins *ADwin*-System laden, müssen Sie sicherstellen, dass dort nicht bereits ein Prozess mit der gleichen Prozessnummer läuft. Wenn das doch der Fall ist, müssen Sie den laufenden Prozess zuerst mit `Stop_Process` stoppen.

Wenn Sie mehrmals Prozesse laden, kann es zu einer Speicherfragmentierung kommen. Beachten Sie die entsprechenden Hinweise im Handbuch *ADbasic*.

Beispiel

```
# Datei Testprg.T91 laden: Prozessor T9, Prozessnr. 1
# die Datei Testprg.T91 liegt im aktuellen Verzeichnis
adw.Load_Process('Testprg.T91')
```

Start_Process

`Start_Process` startet einen Prozess.

```
Start_Process(int ProcessNo)
```

Parameter

`ProcessNo`	Nummer des Prozesses (1...10, 15).

Bemerkungen

`Start_Process` hat keine Auswirkung, wenn die Prozessnummer
- zu einem bereits laufenden Prozess gehört oder
- gleich der Nummer des aufrufenden Prozesses ist oder
- zu einem Prozess gehört, der noch nicht auf die *ADwin*-Hardware geladen ist.

Beispiel

```
adw.Start_Process(1)# Prozess 1 starten
```

Stop_Process

`Stop_Process` stoppt einen Prozess.

```
Stop_Process(int ProcessNo)
```

Parameter

`ProcessNo`	Nummer des Prozesses (1...10, 15).

Bemerkungen

Die Funktion hat keine Auswirkung, wenn Sie die Nummer eines Prozesses angeben, der
- bereits gestoppt ist oder
- noch nicht auf die *ADwin*-Hardware geladen ist.

Beispiel

```
adw.Stop_Process(2)# Prozess 2 stoppen
```

Clear_Process

`Clear_Process` löscht einen Prozess aus dem Speicher.

```
Clear_Process(int ProcessNo)
```

Parameter

`ProcessNo`	Nummer des Prozesses (1...10, 15).

Bemerkungen

Geladene Prozesse belegen Speicherplatz im Programmspeicher. Sie können mit `Clear_Process` Prozesse aus dem Programmspeicher löschen, um mehr Platz für andere Prozesse zu erhalten.

Wenn Sie einen Prozess löschen möchten, gehen Sie folgendermaßen vor:

- Stoppen Sie den laufenden Prozess mit `Stop_Process`. Ein laufender Prozess kann nicht gelöscht werden.
- Prüfen Sie mit `Process_Status`, ob der Prozess tatsächlich gestoppt ist.
- Löschen Sie den Prozess mit `Clear_Process` aus dem Speicher.

Auf Gold- und Pro-Hardwareen sorgt der Prozess 15 für das Blinken der LED; nach dem Entfernen blinkt die LED nicht mehr.

Bitte beachten Sie, dass das Löschen von Prozessen zu Speicherfragmentierung führen kann. Sie finden weitere Informationen im Handbuch *ADbasic*, Abschnitt „Speicherfragmentierung“.

Beispiel

```
# Freigeben des von Prozess 2 belegten Speichers.
# Deklarierte DATA- und FIFO-Felder bleiben erhalten.
adw.Stop_Process(2)
while adw.Process_Status(2) <> 0:
    pass
adw.Clear_Process(2)
```

Process_Status

`Process_Status` liefert den Status eines Prozesses.

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

Parameter

`ProcessNo`	Nummer des Prozesses (1...10, 15).
Rückgabewert	Status des Prozesses: 1 : Prozess läuft. 0 : Prozess läuft nicht, d.h. er ist nicht geladen, nicht gestartet oder gestoppt. <0:Prozess wird gestoppt, d.h. er hat ein `Stop_Process` erhalten, wartet aber noch auf den letzten Event.

Beispiel

```
# Status von Prozess 2 zurückgeben
print('Process_Status 2:', adw.Process_Status(2))
```

Set_Processdelay

`Set_Processdelay` stellt den Parameter `Processdelay` für einen Prozess ein.

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

Parameter

`ProcessNo`	Nummer des Prozesses (1...10).
`Process-delay`	Einzustellender Wert (1…2^{31}-1) für den Parameter `Processdelay` des Prozesses (siehe Tabelle unten).

Bemerkungen

Der Parameter `Processdelay` steuert die Zeitspanne zwischen zwei Event-Aufrufen eines zeitgesteuerten Prozesses (siehe Handbuch oder Online-Hilfe *ADbasic*).

Zu jedem Prozess gibt es eine minimale Zeitspanne: Ein Unterschreiten der minimalen Zeitspanne führt zu einer Überlastung des Prozessors und die Kommunikation zur *ADwin*-Hardware bricht ab.

Die Zeitspanne wird in einer Zeiteinheit angegeben, die vom Prozessortyp und von der Priorität des Prozesses abhängt:

Prozessortyp	Prozesspriorität	
	hoch	niedrig
T9	25ns	100µs
T10	25ns	50µs
T11	3,3ns	0,003µs = 3,3ns

Prozessortyp	Prozesspriorität	
	hoch	niedrig
T12	1ns	1ns
T12.1	1,5ns	1,5ns

Beispiel

```
# Processdelay 2000 bei Prozess 1 einstellen.
adw.Set_Processdelay(1,2000)
```

Bei einem hochprioren, zeitgesteuerten Prozess und dem Prozessor T9 wird der Prozess alle 50µs (=2000*25ns) aufgerufen.

Get_Processdelay

`Get_Processdelay` gibt den Parameter Processdelay für einen Prozess zurück.

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

Parameter

`ProcessNo` Nummer des Prozesses (1...10).

Rückgabewert ≠255: Aktuell eingestellter Wert (1…2^{31}-1) für den Parameter `Processdelay` des Prozesses.
255: möglicher Fehler oder Wert von `Processdelay`. Bitte beachten Sie Kapitel 4.1.3.

Bemerkungen

Der Parameter `Processdelay` steuert die Zeitspanne zwischen zwei Event-Aufrufen eines zeitgesteuerten Prozesses (siehe `Set_Processdelay` sowie Handbuch oder Online-Hilfe *ADbasic*).

Der Parameter `Processdelay` ersetzt den früheren Parameter `Globaldelay`.

Beispiel

```
# Processdelay  des ADbasic-Prozesses 1 abfragen
print('Processdelay 1:', adw.Get_Processdelay(1))
```

5.2.2 TiCoBasic-Prozesse

Bei einer *ADwin*-Hardware mit *TiCo*-Prozessor können Sie eine *TiCoBasic*-Binärdatei als Prozess auf die *ADwin*-Hardware übertragen und starten. In python sind dafür folgende Schritte notwendig:

– Erzeugen Sie die Binärdatei mit *TiCoBasic*.

  Die Binärdatei muss zunächst in ein globales Feld `Data_x` des *ADwin*-Prozessors übertragen werden und gelangt dann mit Hilfe eines *ADbasic*-Prozesses weiter zum *TiCo*-Prozessor.

– Erzeugen Sie mit *ADbasic* einen Prozess, der folgende Aufgaben erfüllt:
  - Dimensionierung eines globalen Felds `Data_x` vom Datentyp `Long`. Achten Sie darauf, dass das Feld größer ist als die Binärdatei.
  - Übertragen der Daten aus `Data_x` zum *TiCo*-Prozessor mit dem Befehl **`TiCo_Load`** / **`P2_TiCo_Load`**. Der Prozess startet automatisch.
  - Speichern des Befehls-Rückgabewerts in einer globalen Variablen `Par_x`.
  - Beenden des *ADbasic*-Prozesses mit `Exit`.

  Sie finden Beispiele für solche *ADbasic*-Prozesse im Installationsverzeichnis (siehe Kapitel 3.1) unter
  `<.\ADbasic\samples_ADwin_GoldII>`
  `<.\ADbasic\samples_ADwin_ProII>`

– Erzeugen Sie in *ADbasic* die Binärdatei des Prozesses.

– In python sind folgende Schritte notwendig:
  - Laden Sie die *ADbasic*-Binärdatei mit `Load_Process.VI` als Prozess auf die *ADwin*-Hardware, starten den Prozess aber noch nicht.
  - Übertragen Sie die *TiCoBasic*-Binärdatei mit `File2Data.VI` in das richtige Feld `Data_x` des *ADwin*-Prozessors.
  - Starten Sie den *ADbasic*-Prozess mit `Start_Process.VI`.
  - Lesen Sie die globale Variable `Par_x` und prüfen, ob die Übertragung erfolgreich war.

5.3 Übertragung von globalen Variablen

Befehle zur Datenübertragung zwischen PC und *ADwin*-Hardware mit den vordefinierten globalen Variablen Par_1 ... Par_80 und FPar_1 ... FPar_80.

5.3.1 Globale Variablen Par_1 ... Par_80

Die globalen Variablen Par_1 ... Par_80 haben folgenden Wertebereich:

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

Set_Par

Set_Par überträgt einen ganzzahligen Wert mit 32 Bit Genauigkeit in eine globale Variable Par.

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

Parameter

Index	Nummer (1 ... 80) der globalen Variablen Par_1 ... Par_80.
Value	Zu setzender Wert für die Variable.

Beispiel

```
# Werte aller PAR-Variablen setzen
for i in range(1, 81): adw.Set_Par(i, i)
```

Get_Par

Get_Par gibt den Wert einer globalen Variablen Par als ganzzahligen Wert mit 32 Bit Genauigkeit zurück.

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

Parameter

Index	Nummer (1 ... 80) der globalen Variablen Par_1 ... Par_80.
Rückgabewert	≠255: Aktueller Wert der Variablen 255: möglicherweise Fehler

Beispiel

```
# Werte der Variablen Par_1..Par_10 lesen
for i in range(1,11): print(adw.Get_Par(i))
```

Get_Par_Block

Get_Par_Block gibt die Werte von mehreren globalen Variablen Par als ganzzahlige Werte mit 32 Bit Genauigkeit in einem Feld zurück.

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

Parameter

StartIndex	Nummer (1 … 80) der globalen Variablen Par_1 … Par_80, die zuerst übertragen wird.
Count	Anzahl (≥1) der zu übertragenden Werte.
Rückgabewert	Zielfeld für die Variablenwerte.

Beispiel

Werte der Variablen Par_10 … Par_39 lesen und im Feld ArrayLong ab Element 1 speichern:

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

Get_Par_All

Get_Par_All gibt die Werte von allen globalen Variablen Par_1 ... Par_80 als ganzzahlige Werte mit 32 Bit Genauigkeit in einem Feld zurück.

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

Parameter

Rückgabewert	Zielfeld für die Variablenwerte.

Beispiel

Werte der Variablen Par_1 … Par_80 lesen und im Feld ArrayLong speichern:

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

Beachten Sie: Da die Indizierung von Python-Feldern bei 0 beginnt, findet sich beispielsweise Par_9 in ArrayLong[8] wieder.

5.3.2 Globale Variablen FPar_1 … FPar_80

Die globalen Variablen FPar_1 ... FPar_80 haben je nach Prozessor folgenden Wertebereich:

T9…T11 (32 Bit):	negativ: $-3{,}402823 \cdot 10^{+38}$ … $-1{,}175494 \cdot 10^{-38}$ positiv: $+1{,}175494 \cdot 10^{-38}$ … $+3{,}402823 \cdot 10^{+38}$
T12/T12.1 (64 Bit):	negativ: $-1{,}7976931348623 15 \cdot 10^{+308}$ … $-2{,}2250738585072014 \cdot 10^{-308}$ positiv: $+2{,}2250738585072014 \cdot 10^{-308}$ … $+1{,}7976931348623 15 \cdot 10^{+308}$

Set_FPar

Set_FPar überträgt einen Fließkommawert mit 32 Bit Genauigkeit in eine globale Variable FPar.

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

Parameter

Index	Nummer (1 … 80) der globalen Variablen FPar_1 … FPar_80.
Value	Zu setzender Wert für die Variable.

Beispiel

```
# Variable FPar_6 auf 34.7 setzen
adw.Set_FPar(6, 34.7)
```

Set_FPar_Double

Set_FPar_Double überträgt einen Fließkommawert mit 64 Bit Genauigkeit in eine globale Variable FPar.

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

Parameter

Index	Nummer (1…80) der globalen Variablen FPAR_1 … FPAR_80.
Value	Zu setzender Wert für die Variable.

Bemerkungen

Bei Prozessoren bis T11 hat die Zielvariable auf dem *ADwin*-System nur einfache Genauigkeit.

Beispiel

```
// Variable FPAR_6 auf 34.7 setzen
Set_FPar_Double(6, 34.7);
```

Get_FPar

Get_Par gibt den Wert einer globalen Variablen FPar als Fließkommawert mit 32 Bit Genauigkeit zurück.

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

Parameter

Index	Nummer (1 … 80) der globalen Variablen FPar_1 … FPar_80.
Rückgabewert	≠255.0: Aktueller Wert der Variablen 255.0: möglicherweise Fehler

Beispiel

```
# Wert der Variablen FPar_56 lesen
print('FPar_56:', adw.Get_FPar(56))
```

Get_FPar_Block

Get_FPar_Block gibt die Werte von mehreren globalen Variablen FPar als Fließkommawerte mit 32 Bit Genauigkeit in einem Feld zurück.

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

Parameter

StartIndex	Nummer (1 … 80) der ersten globalen Variablen FPar_1 … FPar_80, die übertragen wird.
Count	Anzahl (≥1) der zu übertragenden Werte.
Rückgabewert	Zielfeld für die Variablenwerte.

Beispiel

Werte der Variablen FPar_10 … FPar_34 lesen und im Feld Array-Float ab Element 1 speichern:

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

Get_FPar_All

Get_FPar_All gibt die Werte von aller 80 globalen Variablen FPar_1 … FPar_80 als Fließkommawerte mit 32 Bit Genauigkeit in einem Feld zurück.

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

Parameter

Rückgabewert	Zielfeld für die Variablenwerte.

Beispiel

Werte der Variablen FPar_1 … FPar_80 lesen und im Feld Array-Float ab Element 1 speichern:

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

Get_FPar_Double

Get_FPar_Double gibt den Wert einer globalen Variablen FPar als Fließkommawert mit 64 Bit Genauigkeit zurück.

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

Parameter

Index	Nummer (1 ... 80) der globalen Variablen FPAR_1 ... FPAR_80.
Rückgabewert	≠255.0: Aktueller Wert der Variablen 255.0: möglicherweise Fehler

Bemerkungen

Bis T11 gilt: Beachten Sie, dass Fließkommawerte im *ADwin*-System einfache Genauigkeit (single precision) haben. Sie sollten daher Werte der globalen Variablen FPar nur mit einfacher Genauigkeit anzeigen lassen, um Missverständnisse bezüglich der Genauigkeit zu vermeiden.

Beispiel

```
# Wert der Variablen FPar_56 lesen
print('FPar_56:', adw.Get_FPar_Double(56))
```

Get_FPar_Block_Double

Get_FPar_Block_Double gibt die Werte von mehreren globalen Variablen FPar als Fließkommawerte mit 64 Bit Genauigkeit in einem Feld zurück.

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

Parameter

StartIndex	Nummer (1 ... 80) der globalen Variablen FPAR_1 ... FPAR_80, die zuerst übertragen wird.
Count	Anzahl (≥1) der zu übertragenden Variablenwerte.
Rückgabewert	Zielfeld für die Variablenwerte.

Bemerkungen

Bis T11 gilt: Beachten Sie, dass Fließkommawerte im *ADwin*-System einfache Genauigkeit (single precision) haben. Sie sollten daher die Rückgabewerte nur mit einfacher Genauigkeit anzeigen lassen, um Missverständnisse bezüglich der Genauigkeit zu vermeiden.

Beispiel

Werte der Variablen FPAR_10 ... FPAR_34 lesen und im Feld Array-Double ab Element 0 speichern:

```
ArrayDouble = adw.Get_FPar_Block_Double(10,25)
```

Get_FPar_All_Double

`Get_FPar_All_Double` gibt die Werte von aller 80 globalen Variablen `FPar_1` ... `FPar_80` als Fließkommawerte mit 64 Bit Genauigkeit in einem Feld zurück.

```
ctypes.c_double_Array Get_FPar_All_Double()
```

Parameter

`DeviceNo`	Device No. (1…65535), siehe Kapitel 4.3.
`ErrorNo`	Variable für die Rückgabe des Fehlerstatus: 0 = OK; ≠0:Fehler (siehe Kapitel A.2).
Rückgabewert	Zielfeld für die Variablenwerte.

Bemerkungen

Bis T11 gilt: Beachten Sie, dass Fließkommawerte im *ADwin*-System einfache Genauigkeit (single precision) haben. Sie sollten daher die Rückgabewerte nur mit einfacher Genauigkeit anzeigen lassen, um Missverständnisse bezüglich der Genauigkeit zu vermeiden.

Beispiel

Werte der Variablen `FPAR_1` … `FPAR_80` lesen und im Feld `Array-Double` ab Element 0 speichern:

```
ArrayDouble = adw.Get_FPar_All_Double()
```

5.4 Übertragung von Datenfeldern (Arrays)

Befehle zur Datenübertragung zwischen PC und *ADwin*-Hardware mit globalen **DATA**-Feldern (DATA_1…DATA_200):

– Einfache Datenfelder

– FIFO-Felder

– Datenfelder mit String-Daten

Sie müssen jedes Feld vor seiner Verwendung unter *ADbasic* deklarieren (vgl. Handbuch *ADbasic*).

5.4.1 Einfache Datenfelder

Deklarieren Sie einfache Felder vor der Verwendung unter *ADbasic* mit
DIM DATA_x **AS LONG**/**FLOAT**/**FLOAT32**/**FLOAT64**

Ein Feldelement hat je nach Datentyp folgenden Wertebereich:

–	**LONG**	-2147483648 … +2147483647
–	**FLOAT** (bis T11), **FLOAT32**	negativ: $-3{,}402823 \cdot 10^{+38}$ … $-1{,}175494 \cdot 10^{-38}$ positiv: $+1{,}175494 \cdot 10^{-38}$ … $+3{,}402823 \cdot 10^{+38}$
–	**FLOAT64**	negativ: $-1{,}7976931348623 15 \cdot 10^{+308}$ … $-2{,}2250738585072014 \cdot 10^{-308}$ positiv: $+2{,}2250738585072014 \cdot 10^{-308}$ … $+1{,}7976931348623 15 \cdot 10^{+308}$

Data_Length

Data_Length gibt die in *ADbasic* deklarierte Länge eines Felds vom Typ **LONG**, **FLOAT**/**FLOAT64** oder **STRING** zurück, d.h. die Anzahl der Elemente.

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

Parameter

DataNo	Nummer (1...200) des Felds Data_1…Data_200.
Rückgabewert	>0: Deklarierte Länge des Felds (=Anzahl der Elemente) 0: Fehler, das Feld ist nicht deklariert. -1: Sonstiger Fehler.

Bemerkungen

Bei einem DATA-Feld vom Typ **STRING** stellen Sie die Länge der Zeichenfolge mit dem Befehl String_Length fest.

Beispiel

In *ADbasic* ist DATA_2 dimensioniert als:

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

In Python bestimmt man die Länge des Felds DATA_2:

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

Data_Type

`Data_Type` gibt den Datentyp eines in *ADbasic* deklarierten `DATA`-Felds zurück.

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

Parameter

`DataNo` Nummer des Felds (1...200).

Rückgabewert Tupel mit 2 Werten (siehe Liste unten):

- Kennzahl (0…8) für den *ADbasic*-Datentyp des Felds, abhängig vom Prozessortyp.
- String für den *ADbasic*-Datentyp des Felds, abhängig vom Prozessortyp.

Bemerkungen

Die Datentypen in ADbasic sind abhängig vom Prozessortyp:

Datentyp des Felds in *ADbasic*	Prozessortyp				
	T9	T10	T11	T12	T12.1
undefined	0	0	0	0	0
(Byte)	–	–	–	(1)	(1)
(Short)	(2)	(2)	(2)	(2)	(2)
(Integer)	(3)	(3)	(3)	(4)	(4)
Long	4	4	4	4	4
Float (bis T11) / Float32 (nur T12/T12.1)	5	5	5	5	5
Float64	–	–	–	6	6
String	3	3	3	8	8

Die Datentypen **`BYTE`**, **`SHORT`** und **`INTEGER`** sind derzeit nur eingeschränkt in *ADbasic* einsetzbar und nur der Vollständigkeit halber aufgeführt; die Datentypen stehen deswegen in der Tabelle in Klammern.

Beispiel

In *ADbasic* ist `DATA_2` dimensioniert als:

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

In Python erhält man den Datentyp des Felds `DATA_2`:

```
print('type of Data_2:', adw.Data_Type(2))
```

SetData_Long

SetData_Long überträgt ganzzahlige Werte mit 32 Bit Genauigkeit (int32) in ein DATA-Feld der *ADwin*-Hardware.

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

Parameter

PC_Array	Quellfeld, aus dem Werte übertragen werden.
DataNo	Nummer (1...200) des Zielfelds DATA_1 … DATA_200.
StartIndex	Nummer (≥1) des ersten Elements im Zielfeld, das beschrieben wird.
Count	Anzahl (≥1) der zu übertragenden Werte.

Beispiel

100 Elemente des Quellfelds ArrayLong (ab Element 0) in die Elemente 30…129 des Zielfelds DATA_3 übertragen:

```
dataType = ctypes.c_int32 * 100
ArrayLong = dataType(0)
for i in range(100): ArrayLong[i] = i+100
adw.SetData_Long(ArrayLong, 3, 30, 100)
```

GetData_Long

GetData_Long überträgt Werte aus einem DATA-Feld der *ADwin*-Hardware als ganzzahlige Werte mit 32 Bit Genauigkeit (int32) in ein Feld.

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

Parameter

DataNo	Nummer (1...200) des Quellfelds DATA_1 … DATA_200.
StartIndex	Nummer (≥1) des ersten Elements im Quellfeld, das übertragen wird.
Count	Anzahl (≥1) der zu übertragenden Werte.
Rückgabewert	Neu erzeugtes Zielfeld, das die übertragenen Werte enthält.

Beispiel

Die Elemente 1…100 aus DATA_2 werden in das Feld ArrayLong ab Index 0 übertragen:

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

SetData_Float

`SetData_Float` überträgt Fließkommawerte mit 32 Bit Genauigkeit in ein `DATA`-Feld der *ADwin*-Hardware.

```
SetData_Float(list|array|ctypes.c_float_Array PC_
Array, int DataNo, int Startindex, int Count)
```

Parameter

`PC_Array`	Quellfeld, aus dem Werte übertragen werden.
`DataNo`	Nummer(1...200) des Zielfelds `DATA_1` … `DATA_200`.
`StartIndex`	Nummer (≥1) des ersten Elements im Zielfeld, das beschrieben wird.
`Count`	Anzahl (≥1) der zu übertragenden Werte.

Beispiel

Die ersten 80 Elemente des Quellfelds `ArrayFloat` in die Elemente 20…99 des Zielfelds `DATA_3` übertragen:

```
dataType = ctypes.c_float * 80
ArrayFloat = dataType(0)
for i in range(80): ArrayFloat[i] = i+100.1234
adw.SetData_Float(ArrayFloat, 3, 1, 80)
```

GetData_Float

`GetData_Float` überträgt Werte aus einem `DATA`-Feld der *ADwin*-Hardware als Fließkommawerte mit 32 Bit Genauigkeit in ein Feld.

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

Parameter

`DataNo`	Nummer (1...200) des Quellfelds `DATA_1` … `DATA_200`.
`StartIndex`	Nummer (≥1) des ersten Elements im Quellfeld, das übertragen wird.
`Count`	Anzahl (≥1) der zu übertragenden Werte.
Rückgabewert	Neu erzeugtes Zielfeld, das die übertragenen Werte enthält.

Beispiel

Die Elemente 1…100 aus `DATA_2` werden in das Feld `ArrayFloat` ab Index 0 übertragen:

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

SetData_Double

SetData_Double überträgt Fließkommawerte mit 64 Bit Genauigkeit in ein DATA-Feld der *ADwin*-Hardware.

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

Parameter

PC_Array	Quellfeld, aus dem Werte übertragen werden.
DataNo	Nummer(1...200) des Zielfelds DATA_1 ... DATA_200.
StartIndex	Nummer (≥1) des ersten Elements im Zielfeld, das beschrieben wird.
Count	Anzahl (≥1) der zu übertragenden Werte.

Bemerkungen

Bis T11 gilt: Bitte beachten Sie, dass Fließkomma-Werte im *ADwin*-System einfache Genauigkeit (single precision) haben. Sie sollten daher Daten aus dem Feld PC_Array() nur mit einfacher Genauigkeit anzeigen lassen, um Missverständnisse bezüglich der Genauigkeit zu vermeiden.

Beispiel

Die ersten 80 Elemente des Quellfelds ArrayDouble in die Elemente 20…99 des Zielfelds DATA_3 übertragen:

```
dataType = ctypes.c_double * 80
ArrayDouble = dataType(0)
for i in range(80): ArrayDouble[i] = i+100.1234
adw.SetData_Double(ArrayDouble, 3, 1, 80)
```

GetData_Double

GetData_Double überträgt Werte aus einem DATA-Feld der *ADwin*-Hardware als Fließkommawerte mit 64 Bit Genauigkeit in ein Feld.

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

Parameter

DataNo	Nummer (1...200) des Quellfelds DATA_1 ... DATA_200.
StartIndex	Nummer (≥1) des ersten Elements im Quellfeld, das übertragen wird.
Count	Anzahl (≥1) der zu übertragenden Werte.
Rückgabewert	Neu erzeugtes Zielfeld, das die übertragenen Werte enthält.

Bemerkungen

Bis T11 gilt: Bitte beachten Sie, dass Fließkomma-Werte im *ADwin*-System einfache Genauigkeit (single precision) haben. Sie sollten daher Daten aus dem Zielfeld nur mit einfacher Genauigkeit anzeigen lassen, um Missverständnisse bezüglich der Genauigkeit zu vermeiden.

Beispiel

Elemente 1…100 aus DATA_2 in ArrayDouble ab Index 0 übertragen:

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

Data2File

`Data2File` speichert Werte aus einem `DATA`-Feld des *ADwin*-Systems in einer Datei (auf der Festplatte).

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

Parameter

`Filename`	Pfad und Dateiname. Wenn kein Pfad angegeben ist, wird die Datei im Projektverzeichnis[?] gespeichert.
`DataNo`	Nummer (1...200) des Quellfelds `DATA_1` … `DATA_200`.
`Startindex`	Nummer (≥1) des Elements im Quellfeld, das zuerst übertragen wird.
`Count`	Anzahl (≥1) der zu übertragenden Werte.
`Mode`	Schreibmodus: 0: Datei wird überschrieben (falls vorhanden) 1: Daten werden an eine vorhandene Datei angehängt

Bemerkungen

Das `DATA`-Feld darf nicht als **`FIFO`** deklariert sein.

Die Daten werden in der Datei binär gespeichert. Der Datentyp (Long, Float32, Float64) wird an das gelesene `DATA`-Feld angepasst.

Wenn die Datei nicht vorhanden ist, wird sie neu angelegt.

Beispiel

Die Elemente 1…1000 aus dem *ADbasic*-Feld `DATA_1` in der Datei `<Test.dat>` im Projektverzeichnis speichern:

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

File2Data überträgt eine Datei (aus dem Dateisystem) in ein DATA-Feld des *ADwin*-Systems.

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

Parameter

Filename	Zeiger auf Pfad und Namen der Quelldatei. Wenn kein Pfad angegeben ist, wird die Datei im Projektverzeichnis gesucht.
DataType	Datentyp der Werte in der Quelldatei. Wählen Sie eine der folgenden Konstanten: 2: Ganzzahlige Werte (32 Bit, signed). 5: Fließkommawerte (32 Bit). 6: Fließkommawerte (64 Bit).
DataNo	Nummer (1...200) des Zielfelds DATA_1 ... DATA_200.
Startindex	Nummer (≥1) des Elements im Zielfeld, das zuerst beschrieben wird.

Bemerkungen

Alle Werte in der Datei Filename müssen binär in einem der Formate Long, Float oder Double vorliegen, und DataType muss entsprechend angegeben sein.

Das Zielfeld DATA darf nicht als **FIFO** deklariert sein. Das Feld muss so groß dimensioniert sein, dass alle Werte aus der Datei aufgenommen werden können.

Wenn das Zielfeld einen anderen Datentyp hat als DataType, werden die Werte aus der Quelldatei in das Zielformat gewandelt. Es gibt je nach Prozessor die Zielformate Long, Float, Float32 und Float64.

Beispiel

In *ADbasic* sei DATA_1 dimensioniert als:

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

Werte vom Typ Long aus der Datei <Test.dat> im Projektverzeichnis in das *ADbasic*-Feld DATA_1 übertragen, beginnend mit dem Feldelement DATA_1[20]. In der Datei dürfen höchstens 980 Werte enthalten sein, damit die Feldgröße von DATA_1 nicht überschritten wird.

```
File2Data('Test.dat', 2, 1, 20)
```

5.4.2 FIFO-Felder

Befehle zur Datenübertragung zwischen PC und *ADwin*-Hardware mit globalen DATA-Feldern (DATA_1…DATA_200), die als FIFO deklariert sind.

Sie müssen jedes FIFO-Feld vor seiner Verwendung unter *ADbasic* deklarieren (vgl. Handbuch „*ADbasic*“): **DIM** DATA_x[n] **AS** TYPE **AS FIFO**

Ein FIFO-Feldelement hat je nach Datentyp und Prozessor folgenden Wertebereich:

- **LONG** -2147483648 … +2147483647
- **FLOAT** (bis T11), **FLOAT32** negativ: $-3{,}402823 \cdot 10^{+38}$ … $-1{,}175494 \cdot 10^{-38}$; positiv: $+1{,}175494 \cdot 10^{-38}$ … $+3{,}402823 \cdot 10^{+38}$
- **FLOAT64** negativ: $-1{,}7976931348623 15 \cdot 10^{+308}$ … $-2{,}2250738585072014 \cdot 10^{-308}$; positiv: $+2{,}2250738585072014 \cdot 10^{-308}$ … $+1{,}7976931348623 15 \cdot 10^{+308}$

Um sicherzustellen, dass noch Platz im FIFO ist, sollten Sie vor dem Schreiben die Funktion **FIFO_EMPTY** verwenden. In gleicher Weise prüft die Funktion **FIFO_FULL** vor dem Lesen, ob noch nicht gelesene Werte vorhanden sind.

Fifo_Empty

Fifo_Empty liefert die Anzahl der freien Elemente eines Fifo-Felds.

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

Parameter

FifoNo	Nummer (1...200) des Fifo-Felds DATA_1 … DATA_200.
Rückgabewert	≠255: Anzahl der freien Elemente im Fifo-Feld. 255: möglicherweise Fehler

Beispiel

In *ADbasic* sei DATA_5 dimensioniert als:

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

In Python erhalten Sie die Anzahl der freien Elemente (≤100) in DATA_5:

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

Fifo_Full

Fifo_Full liefert die Anzahl der belegten Elemente eines Fifo-Felds.

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

Parameter

FifoNo	Nummer (1...200) des Fifo-Felds DATA_1 … DATA_200.
Rückgabewert	≠255: Anzahl der belegten Elemente im Fifo-Feld 255: möglicherweise Fehler

Beispiel

In *ADbasic* sei DATA_12 dimensioniert als:

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

In Python erhalten Sie die Anzahl der belegten Elemente (≤2500) in DATA_12:

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

Fifo_Clear

`Fifo_Clear` initialisiert den Schreib- und Lesezeiger eines Fifo-Felds. Die Daten des FIFO-Felds sind anschließend nicht mehr verfügbar.

```
Fifo_Clear(int FifoNo)
```

Parameter

`FifoNo`	Nummer (1...200) des Fifo-Felds `DATA_1` … `DATA_200`.

Bemerkungen

Beim Start eines *ADwin*-Prozesses werden die FIFO-Zeiger eines Felds nicht automatisch initialisiert. Wir empfehlen deshalb für *ADbasic*, `Fifo_Clear` gleich zu Beginn Ihres Programms aufzurufen.

Beachten Sie bei einem *ADsim*-Prozess: Innerhalb des *ADsim*-Prozesses kann keine Initialisierung stattfinden, daher kann eine Initialisierung mit `Fifo_Clear` sinnvoll sein.

Das Initialisieren der FIFO-Zeiger im Programmablauf ist sinnvoll, wenn alle beschriebenen Elemente verworfen werden sollen, z.B. wegen eines Fehlers.

Beispiel

```
# Daten im FIFO-Feld DATA_45 verwerfen
adw.Fifo_Clear(45)
```

SetFifo_Long

`SetFifo_Long` überträgt ganzzahlige Werte mit 32 Bit Genauigkeit in ein Fifo-Feld der *ADwin*-Hardware.

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

Parameter

`FifoNo`	Nummer (1...200) des Fifo-Felds `DATA_1` … `DATA_200`.
`PC_Array`	Quellfeld, aus dem Werte übertragen werden.
`Count`	Anzahl (≥1) der zu übertragenden Werte.

Beispiel

FIFO-Zielfeld `DATA_12` auf genügend freie Elemente prüfen und 1000 Werte des Quellfelds `ArrayLong` in das Zielfeld übertragen:

```
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 überträgt Werte aus einem FIFO-Feld der *ADwin*-Hardware als ganzzahlige Werte mit 32 Bit Genauigkeit in ein Feld auf dem PC.

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

Parameter

FifoNo	Nummer (1...200) des Fifo-Felds DATA_1 ... DATA_200.
Count	Anzahl (≥1) der zu übertragenden Werte.
Rückgabewert	Neu erzeugtes Zielfeld, das die übertragenen Werte enthält.

Beispiel

FIFO-Quellfeld DATA_2 auf genügend belegte Elemente prüfen und 3000 Werte des Quellfelds in das Zielfeld ArrayLong ab Index 0 übertragen:

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

SetFifo_Float

SetFifo_Float überträgt Fließkommawerte mit 32 Bit Genauigkeit in ein Fifo-Feld der *ADwin*-Hardware.

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

Parameter

FifoNo	Nummer (1...200) des Fifo-Felds DATA_1 ... DATA_200.
PC_Array	Zeiger auf das Quellfeld, aus dem Werte übertragen werden.
Count	Anzahl (≥1) der zu übertragenden Werte.

Beispiel

FIFO-Zielfeld DATA_12 auf genügend freie Elemente prüfen und 1000 Werte des Quellfelds ArrayFloat in das Zielfeld übertragen:

```
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` überträgt Werte aus einem FIFO-Feld der *ADwin*-Hardware als Fließkommawerte mit 32 Bit Genauigkeit in ein Feld auf dem PC.

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

Parameter

`FifoNo`	Nummer (1...200) des Fifo-Felds `DATA_1` ... `DATA_200`.
`Count`	Anzahl (≥1) der zu übertragenden Werte.
Rückgabewert	Neu erzeugtes Zielfeld, das die übertragenen Werte enthält.

Beispiel

FIFO-Quellfeld `DATA_12` auf genügend belegte Elemente prüfen und 200 Werte des Quellfelds in das Zielfeld `ArrayFloat` ab Index 0 übertragen:

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

SetFifo_Double

`SetFifo_Double` überträgt Fließkommawerte mit 64 Bit Genauigkeit in ein Fifo-Feld der *ADwin*-Hardware.

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

Parameter

`FifoNo`	Nummer (1...200) des Fifo-Felds `DATA_1` ... `DATA_200`.
`PC_Array`	Zeiger auf das Quellfeld, aus dem Werte übertragen werden.
`Count`	Anzahl (≥1) der zu übertragenden Werte.

Bemerkungen

Bis T11 gilt: Bitte beachten Sie, dass Fließkomma-Werte im *ADwin*-System einfache Genauigkeit (single precision) haben. Sie sollten daher Daten aus dem Feld `PC_Array()` nur mit einfacher Genauigkeit anzeigen lassen, um Missverständnisse bezüglich der Genauigkeit zu vermeiden.

Beispiel

FIFO-Zielfeld `DATA_12` auf genügend freie Elemente prüfen und 1000 Werte des Quellfelds `ArrayDouble` in das Zielfeld übertragen:

```
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` überträgt Werte aus einem FIFO-Feld der *ADwin*-Hardware als Fließkommawerte mit 64 Bit Genauigkeit in ein Feld auf dem PC.

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

Parameter

`FifoNo`	Nummer (1...200) des Fifo-Felds `DATA_1` ... `DATA_200`.
`Count`	Anzahl (≥1) der zu übertragenden Werte.
Rückgabewert	Neu erzeugtes Zielfeld, das die übertragenen Werte enthält.

Bemerkungen

Bis T11 gilt: Bitte beachten Sie, dass Fließkomma-Werte im *ADwin*-System einfache Genauigkeit (single precision) haben. Sie sollten daher Rückgabewerte nur mit einfacher Genauigkeit anzeigen lassen, um Missverständnisse bezüglich der Genauigkeit zu vermeiden.

Beispiel

FIFO-Quellfeld `DATA_12` auf genügend belegte Elemente prüfen und 200 Werte des Quellfelds in das Zielfeld `ArrayDouble` ab Index 0 übertragen:

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

5.4.3 Datenfelder mit String-Daten

Befehle zur Datenübertragung zwischen PC und *ADwin*-Hardware mit globalen DATA-Feldern (DATA_1…DATA_200), die String-Daten enthalten.

Sie müssen jedes DATA-Feld vor seiner Verwendung unter *ADbasic* deklarieren (vgl. Handbuch „*ADbasic*“): **DIM** DATA_x[n] **AS STRING**.

Ein Feldelement in einem DATA-Feld vom Typ **STRING** kann ein Zeichen mit dem ASCII-Wert 0…127 enthalten. Der ASCII-Wert 0 (Endekennung oder NULL) markiert das Ende einer Zeichenkette in einem DATA-Feld .

String_Length

String_Length gibt die Länge eines Datenstrings in einem DATA-Feld zurück.

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

Parameter

DataNo	Nummer (1...200) des Felds DATA_1 … DATA_200.
Rückgabewert	≠-1: Länge des Strings = Anzahl der Zeichen. -1: Fehler

Bemerkungen

String_Length zählt die Zeichen im DATA-Feld bis zum ersten Auftreten der Endekennung (NULL). Die Endekennung selbst wird nicht als Zeichen gezählt.

Die in *ADbasic* deklarierte Länge eines DATA-Felds stellen Sie mit dem Befehl Data_Length fest.

Beispiel

In *ADbasic* ist DATA_2 dimensioniert als:

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

In Python bestimmt man die Länge des Felds DATA_2:

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

SetData_String

SetData_String überträgt einen String in ein DATA-Feld.

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

Parameter

DataNo	Nummer (1...200) des Fifo-Felds DATA_1 … DATA_200.
String	Zu übertragender String.

Bemerkungen

SetData_String hängt jedem übertragenen String als letztes Zeichen die Endekennung (NULL) an.

Beispiel

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

Der String „Hello World“ wird in das Feld DATA_2 geschrieben und die Endekennung angehängt.

GetData_String

GetData_String überträgt einen String aus einem DATA-Feld zum PC.

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

Parameter

DataNo	Nummer (1...200) des Quellfelds DATA_1 ... DATA_200.
MaxCount	Max. Anzahl (≥1) der übertragenen Zeichen ohne Endekennung.
Rückgabewert	Feld mit dem übertragenen String aus dem Quellfeld.

Bemerkungen

Wenn der String im DATA-Feld eine Endekennung (ASCII-Nummer 0) enthält, stoppt die Übertragung genau dort, d.h. die Endekennung wird nicht übertragen. Die Anzahl der bis dahin gelesenen Zeichen ohne die Endekennung ist der Rückgabewert.

Wenn MaxCount größer ist als die in *ADbasic* definierte Zeichenzahl des Strings, erhalten Sie über Get_Last_Error() den Fehler "Data too small".

Wenn Sie einen großen Wert für MaxCount angeben, hat die Funktion eine entsprechend lange Ausführungszeit, selbst wenn der übertragene String nur kurz ist.
Bei zeitkritischen Anwendungen mit großen Strings kann es günstiger sein, wie folgt vorzugehen:

- Sie stellen die tatsächliche Anzahl der Zeichen im String mit String_Length() fest.
- Sie lesen den String mit Getdata_String() und übergeben die tatsächliche Zeichnanzahl als MaxCount.

Beispiel

Aktuellen String aus DATA_2 holen und in ArrayString kopieren:

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

5.5 Fehlercode abfragen

Die folgenden Befehle sind nur in bestimmten Fällen sinnvoll einsetzbar. Beachten Sie daher bitte Kapitel 4.1 „Fehler erkennen“.

Get_Last_Error

`Get_Last_Error` gibt die Nummer des zuletzt in der *ADwin*-Programmbibliothek aufgetretenen Fehlers zurück.

```
int Get_Last_Error()
```

Bemerkungen

Die Funktion ist nur sinnvoll, wenn Sie für die Fehlerbehandlung keine Exceptions verwenden (siehe Kapitel 4.1 auf Seite 7). In diesem Fall müssen Sie nach jedem Zugriff auf *ADwin*-Hardware die Fehlernummer abfragen und entsprechend behandeln; nach einem erfolgreichen Zugriff wird die Fehlernummer automatisch auf 0 gesetzt.

Auch beim Auftreten mehrerer Fehler nacheinander enthält `Last_Error` nur die Nummer des zuletzt aufgetretenen Fehlers.

Zu jeder Fehlernummer erhalten Sie den zugehörigen Klartext mit der Funktion `Get_Error_Text`. Eine Liste aller Fehlermeldungen finden Sie im Abschnitt A.2 im Anhang.

Der Aufruf von `Get_Last_Error` beeinflusst die Fehlernummer nicht.

Beispiel

```
# raiseExceptions = 0: Bei einem Laufzeitfehler wird
# keine Exception ausgelöst
adw.raiseExceptions = 0
Status = adw.Process_Status(2)
# Fehlernummer lesen nach jedem Zugriff
# auf ADwin-Hardware
ErrNum = Get_Last_Error()
if ErrNum: print('Fehler: ', \)
    adw.Get_Error_Text(ErrNum)
```

Get_Error_Text

`Get_Error_Text` gibt einen Fehlertext zu einer vorhandenen Fehlernummer zurück.

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

Parameter

`ErrorNumber` Fehlernummer = Rückgabewert der Funktion `Get_Last_Error`.

Rückgabewert Fehlertext.

Bemerkungen

Die Funktion kann in Verbindung mit `Get_Last_Error` aufgerufen werden.

Beispiel

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

Anhang

A.1 Beispielprogramme

Zum ***ADwin***-Treiber für Python gehören einfache Beispielprogramme, die das Zusammenspiel von Python mit dem ***ADwin***-System verdeutlichen. Zu jedem Beispiel gehören die ausführbaren Dateien und die Quelltexte.

Die Python-Beispiele verwenden das Qt-Toolkit, Version 5, und das Python-Modul PyQt5. Toolkit und Modul sind im Internet verfügbar von Riverbank: `https://www.riverbankcomputing.co.uk/software/pyqt/download`.

Beachten Sie bitte, dass ein vollständiges Beispiel aus einem Python- und einem ***ADbasic***-Programm besteht. Beide Programme übernehmen typischerweise die folgenden, unterschiedlichen Aufgaben:

– Das Python-Programm startet, überwacht und stoppt einen Prozess auf dem ***ADwin***-System und zeigt die übertragenen Daten an.

  Die Dateien liegen im ***ADwin***-Verzeichnis, siehe Kapitel 3.1 auf Seite 5.

– Das ***ADbasic***-Programm definiert den Ablauf des Prozesses auf dem ***ADwin***-System, also das Messen, Steuern, Regeln und die zeitkritische Auswertung.

Folgende Beispiele stehen Ihnen zur Verfügung:

– BAS_DMO1: **Messwerte online auswerten**

– BAS_DMO2: **Regelgrößen online vorgeben**

– BAS_DMO3: **Eine Messreihe darstellen**

– BAS_DMO7: **Signale generieren**

Alle Beispielprogramme sind für ***ADwin-Gold*** mit der Device No. 1 geschrieben. Für andere Einstellungen oder andere ***ADwin***-Hardware müssen Sie die Quelltexte anpassen und neu kompilieren.

Für Windows folgen Sie diesen Anweisungen:

– Öffnen Sie den ***ADbasic***-Quelltext `<BAS_DMOx.bas>` und passen die Einstellungen unter „`Options` ▸ `Compiler`“ an.

  Für ***ADwin-Pro I*** gibt es separate Beispielprogramme im Verzeichnis `<C:\ADwin\ADbasic\samples_ADwin_Pro\>`.

– Für ***ADwin-light-16*** kann der Quelltext unverändert stehen bleiben.

  Wenn Sie ***ADwin-Gold II*** oder ***ADwin-Pro II*** verwenden, müssen Sie die passenden Include-Dateien einbinden und jeweils die Befehle `ADC` und `DAC` anpassen.

– Erzeugen Sie eine neue Binärdatei mit „`Build` ▸ `Make Bin File`“.

– Im Python-Quelltext
  - ändern Sie das Klassenattribut `DeviceNo` auf die Nummer des gewünschten ***ADwin***-Systems
  - passen Sie den Namen der Betriebssystemdatei `<ADwin9.btl>` und den Namen der Binärdatei `<BAS_DMOx.T91>` an.

Für Linux gelten die Anweisungen entsprechend; die Compiler-Anwendung unter Linux im Handbuch „ADwin für Linux / Mac“ beschrieben.

Messwerte online auswerten **BAS_DMO1**

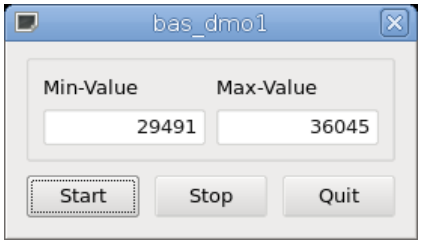
Das Beispiel `<BAS_DMO1>` erfasst Messwerte in Zyklen, wertet sie online aus und zeigt das Ergebnis an.

Das Beispiel führt folgende Aufgaben aus:

– Das Python-Programm lädt das ***ADwin***-Betriebssystem für den Prozessor T9: `<ADwin9.btl>`.

– Das Python-Programm lädt die ***ADbasic***-Binärdatei für T9 als Prozess 1: `<BAS_DMO1.T91>`.

– Mit der Schaltfläche `Start` starten Sie
  • den geladenen ***ADbasic***-Prozess 1 und
  • den Timer.



Der ***ADbasic***-Prozess erfasst 1000 Messwerte am Analogeingang 1, ermittelt daraus den Minimal- und Maximalwert und speichert die beiden Werte in den globalen Variablen *PAR_1* und *PAR_2*. Dieser Messzyklus wird solange wiederholt, bis der Prozess gestoppt wird. Nach dem ersten Messzyklus setzt der Prozess die globale Variable *PAR_10* auf den Wert 1.

Der Timer prüft fünfmal pro Sekunde, ob die globale Variable *PAR_10* den Wert 1 hat. Sobald dies der Fall ist, liest die Funktion die Werte der globalen Variablen *PAR_1* und *PAR_2* und zeigt sie in den Fenstern für Minimalwert und Maximalwert an.
Wenn Sie am Analogeingang 1 kein Signal anlegen, schwanken die angezeigten Werte um den Nullpunkt, also um den Wert 32768.

– Mit der Schaltfläche `Stop` stoppen Sie
  • den ***ADbasic***-Prozess 1 und
  • den Timer.

BAS_DMO2

Regelgrößen online vorgeben

Das Beispiel `<BAS_DMO2>` gibt einem laufenden Regelprozess auf dem ***ADwin***-System, einem digitalen P-Regler, die Regelparameter vor. Sie können die Regelparameter online verändern.

Das Beispiel führt folgende Aufgaben aus:

– Das Python-Programm lädt das ***ADwin***-Betriebssystem für den Prozessor T9: `<ADwin9.btl>`.

– Das Python-Programm lädt die ***ADbasic***-Binärdatei für T9 als Prozess 1: `<BAS_DMO2.T91>`.

– Mit der Schaltfläche `Start` starten Sie den geladenen ***ADbasic***-Prozess 1.

– Mit den beiden Schiebereglern stellen Sie online den Sollwert (`Setpoint`) und die Verstärkung (`Gain`) des digitalen P-Reglers ein.

  Bei jeder neuen Einstellung schreibt das Python-Programm die Werte der Schieberegler in die globalen Variablen `PAR_1` und `PAR_2`.
  Das ***ADbasic***-Programm liest die globalen Variablen aus und verwendet die Werte als Regelparameter.

– Mit der Schaltfläche `Stop` beenden Sie den ***ADbasic***-Prozess 1.

Eine nähere Beschreibung des ***ADbasic***-Prozesses finden Sie im Tutorial.

Eine Messreihe darstellen **BAS_DMO3**

Das Beispiel `<BAS_DMO3>` erfasst eine Reihe von Messwerten und zeigt die Messreihe als Kurve an.

Das Beispiel führt folgende Aufgaben aus:

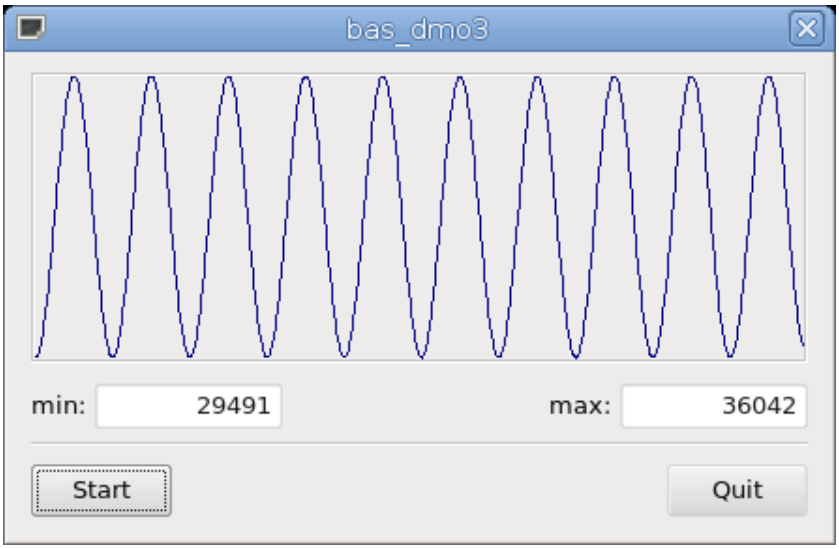
– Das Python-Programm lädt das ***ADwin***-Betriebssystem für den Prozessor T9: `<ADwin9.btl>`.

– Das Python-Programm lädt die ***ADbasic***-Binärdatei für T9 als Prozess 1: `<BAS_DMO3.T91>`.

– Mit der Schaltfläche `Start` starten Sie
  • den geladenen ***ADbasic***-Prozess und
  • die Python-Funktion `Timer1`.

  Der ***ADbasic***-Prozess `BAS_DMO3` erfasst 1000 Messwerte am Analogeingang 1 und speichert sie in dem globalen Feld *`DATA_1`*. Anschließend setzt der Prozess die globale Variable *`PAR_10`* auf den Wert 1 und stoppt.

  Die Python-Funktion `Timer1` prüft zehnmal pro Sekunde, ob die globale Variable *`PAR_10`* den Wert 1 hat. Sobald dies der Fall ist, liest die Funktion 1000 Werte aus dem globalen Feld *`DATA_1`* und zeigt sie als Kurve an.



  Die angezeigte Kurve hängt davon ab, welchen Signalverlauf Sie am Analogeingang 1 anlegen.

– Sie können die Erfassung einer Messreihe beliebig oft wiederholen.

BAS_DMO7

Signale generieren

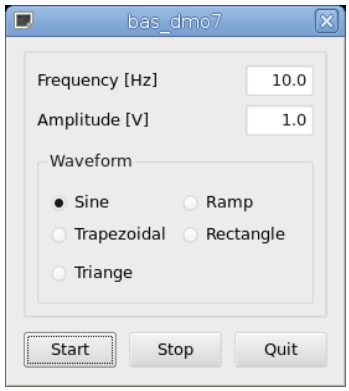
Im Beispielprogramm `<BAS_DMO7>` arbeitet der ***ADbasic***-Prozess als Signalgenerator. Sie können in der Bedienoberfläche online die Signalform, die Frequenz und die Amplitude des Ausgabesignals einstellen.

Das Beispielprogramm führt folgende Aufgaben aus:

– Das Python-Programm lädt das ***ADwin***-Betriebssystem für den Prozessor T9: `<ADwin9.btl>`.

– Das Python-Programm lädt die ***ADbasic***-Binärdatei für T9 als Prozess 1: `<BAS_DMO7.T91>`.

– Mit der Schaltfläche `Start` starten Sie den geladenen ***ADbasic***-Prozess 1.



– Geben Sie die Parameter für das Ausgangssignal ein:
  - Frequenz: 0…1000 Hz
  - Amplitude: 0…10 V
  - Signalform: Sinus, Rampe, Trapez, Rechteck, Dreieck.

  Sobald Sie einen der Parameter ändern, schreibt das Python-Programm die gewählten Parameter in die globalen Variablen `PAR_1`, `PAR_2` und `PAR_3`.

  Das ***ADbasic***-Programm liest die globalen Variablen aus und generiert das entsprechende Ausgangssignal.

– Mit der Schaltfläche `Stop` beenden Sie den ***ADbasic***-Prozess 1.

A.2 Fehlermeldungen

Fehler-Nr.	Fehlermeldung
0	Kein Fehler
1	Timeout Fehler beim Schreiben zum ADwin-System.
2	Timeout Fehler beim Lesen vom ADwin-System.
10	Die Device-Nummer ist nicht erlaubt.
11	Die Device-Nummer ist nicht konfiguriert.
15	Funktion für dieses Device nicht erlaubt.
20	Inkompatible Versionen von ADwin-Betriebssystem, Treiberdatei adwin32.dll und / oder ADbasic-Binärdatei.
100	Das Data-Feld ist zu klein.
101	Das Fifo-Feld ist zu klein oder nicht genug Daten.
102	Das Fifo-Feld hat nicht genug Daten.
103	Das Data-Feld ist nicht deklariert.
150	Nicht genug Speicher oder Speicherzugriffsfehler.
200	Datei konnte nicht gefunden werden.
201	Temporäre Datei konnte nicht erstellt werden.
202	Die Datei ist keine ADbasic Binärdatei.
203	Die Datei ist ungültig.[1]
204	Die Datei ist keine BTL.
205	ADbasic Binärdatei ist für den falschen Prozessor oder beschädigt.
2000	Netzwerk Fehler (TCP/IP).
2001	Netzwerk timeout.
2002	Falsches Passwort.
3000	USB Gerät nicht gefunden.
3001	Gerät nicht gefunden.

1. Möglicherweise fehlt bei `<ADwin5.btl>` die „memory table“, eine andere Datei wurde zu `<ADwin5.btl>` umbenannt oder diese ist beschädigt

A.3 Index der Funktionen