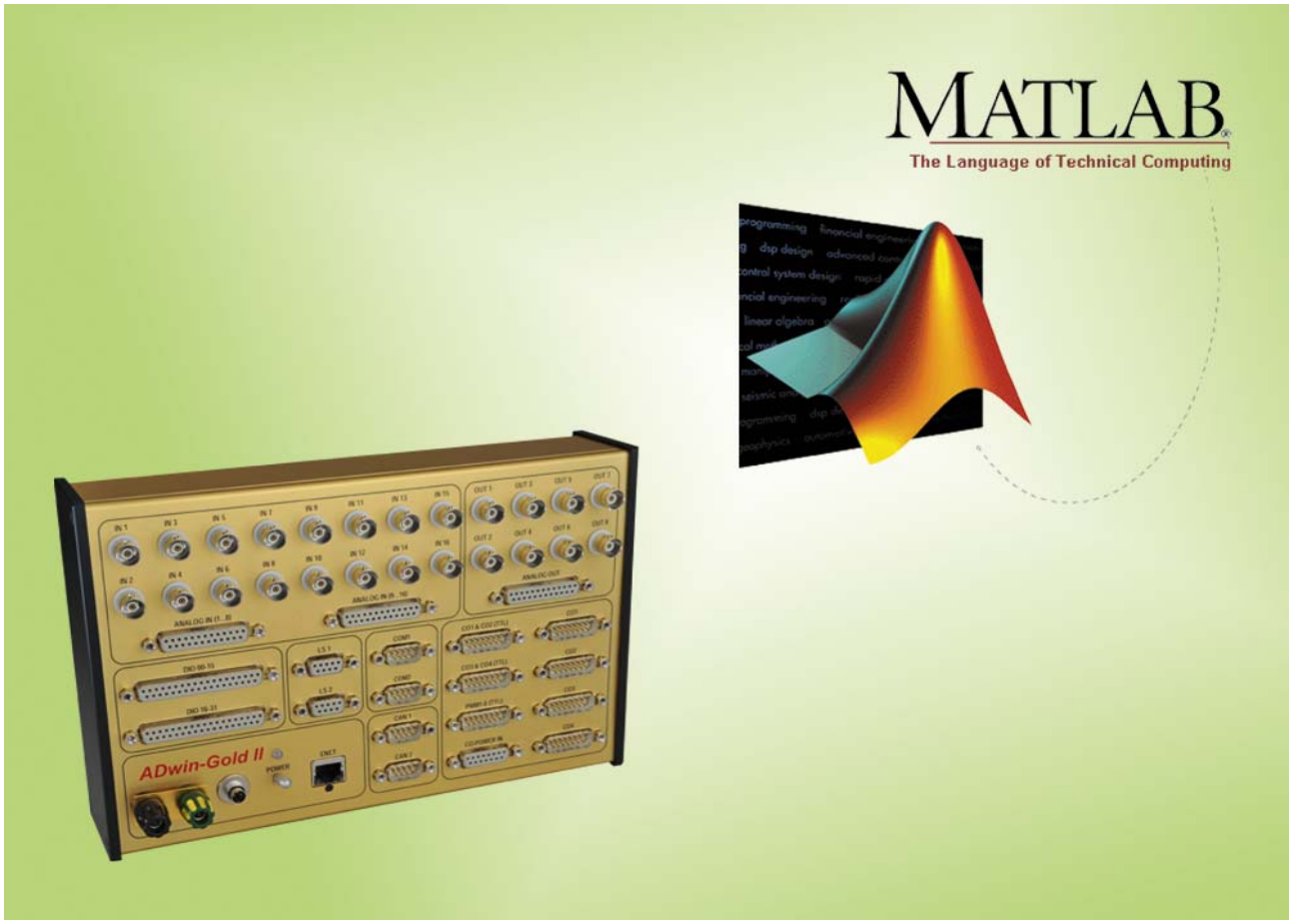
JÄGER
Computergesteuerte
Messtechnik GmbH

ADlab

Treiber für MATLAB 5…9



License Key:

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

1 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.

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*-Treibers für MATLAB® Versionen bis 9.x.

Folgende Dokumente ergänzen die Treiberbeschreibung:

- Das Handbuch „*ADwin* Installation“ beschreibt die Hardware- und Software-Installation für alle *ADwin*-Systeme.
- Falls Sie unter Linux oder MacOS arbeiten: das Handbuch „ADwin für Linux / Mac“, das die Software-Installation und den *ADbasic*-Compiler unter Linux und Mac OS beschreibt.
- Das Handbuch *ADbasic* beschreibt die Entwicklungsumgebung und die Befehle des Compilers *ADbasic*. Mit dem komfortablen Echtzeit-Entwicklungstool *ADbasic* programmieren Sie Ihr *ADwin*-System.
- Das Hardware-Handbuch für Ihr *ADwin*-System.
- Die Handbücher *ADsim* T11 / T12 erklären, wie Sie ein Simulink-Modell exportieren und für die *ADwin*-Hardware als Prozess kompilieren.
- Das Handbuch *ADsim*-Treiber für Matlab beschreibt einen separaten Treiber, der Zugriff auf Parameter, Signale und Blockzustände eines Simulink-Modells erlaubt.

Es wird vorausgesetzt, dass Sie den Umgang mit der Entwicklungsumgebung MATLAB 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 MATLAB®

Dieser Abschnitt führt Sie in die Möglichkeiten des *ADwin*-Treibers für MATLAB ein und erklärt, wie die Kommunikation mit einem *ADwin*-System aus MATLAB unter Windows, Linux oder Mac OS abläuft.

2.1 Schnittstelle zum *ADwin*-System

Der *ADwin*-Treiber für MATLAB ist die Schnittstelle für die Entwicklungsumgebung MATLAB zur Kommunikation mit *ADwin*-Systemen.

Die Kombination der Entwicklungsumgebung MATLAB mit einem *ADwin*-System bietet Ihnen vielfältige Möglichkeiten. Auf der einen Seite nutzen Sie die Intelligenz und Rechenleistung des *ADwin*-Systems zum Regeln, Messen und Steuern. Auf der anderen Seite stehen die vielfältigen MATLAB-Funktionen zum Verwalten, Analysieren und Dokumentieren der Messdaten und für die komfortable Bedienoberfläche.

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

Sie legen das Verhalten der *ADwin*-Hardware selbst fest. Hierzu gibt es 2 mögliche 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*).
– *ADsim* T11: Sie erstellen in Simulink ein Modell, exportieren es und kompilieren es mit *ADsim* für die *ADwin*-Hardware (siehe Handbuch *ADsim*).

2.2 Kommunikation mit dem *ADwin*-System

Mit dem *ADwin*-Treiber für MATLAB greifen Sie aus MATLAB auf globale Variablen und Felder der laufenden *ADwin*-Hardware zu und steuern *ADbasic*-Prozesse.

Daten und Befehle zwischen MATLAB und dem *ADwin*-System durchlaufen den nachfolgend dargestellten Weg.

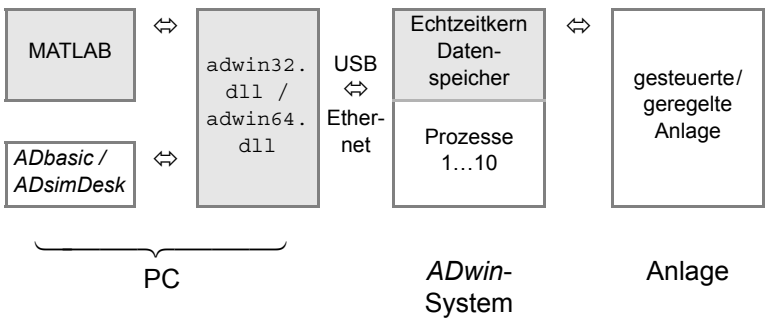


Abb. 1 – *ADwin*-MATLAB Schnittstelle

adwin32.dll

Die `adwin32.dll` (beim 64 Bit-Betriebssystem: `adwin64.dll`) ist die zentrale Schnittstelle für Windows-Anwendungen zum *ADwin*-System und wird daher auch vom *ADwin*-Treiber für MATLAB genutzt. Diese Schnittstelle ermöglicht, dass mehrere Windows-Programme gleichzeitig mit dem *ADwin*-

System kommunizieren: So können z.B. Entwicklungsumgebungen, *ADbasic*, *ADsimDesk* und *ADtools* parallel mit dem *ADwin*-System arbeiten.

Echtzeitkern

Die Schnittstelle `adwin32.dll` / `adwin64.dll` kommuniziert mit dem Echtzeitkern des *ADwin*-Systems, dem Betriebssystem. Deshalb müssen Sie nach jedem Einschalten des Systems zunächst das Betriebssystem – eine Datei `*.btl` wie z.B. `<adwin11.btl>` – dorthin laden.
Nach erfolgreicher Übertragung können Sie *ADbasic*-Prozesse zur *ADwin*-Hardware übertragen und ausführen und auf globale Variablen und Felder zugreifen. Die in *ADbasic* programmierten Prozesse enthalten den Programmcode zur Messung, Steuerung oder Regelung Ihrer Applikation.

Bei *ADsim* enthält die Datei `*.btl` sowohl das Betriebssystem als auch das kompilierte Modell, das einem *ADbasic*-Prozess entspricht. Deswegen sind die Befehle des *ADwin*-Treibers zur Steuerung von Prozessen dort nicht anwendbar.

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.
  Bei *ADsim* T11 gibt es nur einen einzigen Echtzeit-Prozess mit hoher Priorität, nämlich das kompilierte Simulink-Modell; optional kann zum Modell zusätzlich ein niedrig priorisierter Prozess gehören.

Datenspeicher

– Bereitstellung von globalen Variablen:
  - 80 ganzzahlige Variablen (`PAR_1` … `PAR_80`), bereits vordefiniert.
  - 80 Fließkomma-Variablen (`FPAR_1` … `FPAR_80`), bereits vordefiniert.
  - 200 Datenfelder (`DATA_1` … `DATA_200`), Länge und Datentyp sind frei definierbar.

  Sie können die Werte dieser Variablen bzw. Datenfelder aus MATLAB jederzeit lesen und ändern.

Kommunikation

– Kommunikation zwischen *ADwin*-System und PC (via `adwin32.dll` / `adwin64.dll`).

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

  Die folgende Tabelle zeigt Beispiele aus jeder Gruppe.

Steuerbefehle, z.B.	
`Load_Process`	überträgt einen Prozess auf das System.
`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_Double`	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.

2.3 *ADsim*-Treiber für MATLAB

Ergänzend zum *ADwin*-Treiber für MATLAB gibt es für Simulink-Anwender zusätzlich den *ADsim*-Treiber für MATLAB (siehe Handbuch *ADsim*-Treiber für Matlab).

Der *ADsim*-Treiber arbeitet nur unter Windows.

Während Sie mit dem *ADwin*-Treiber für MATLAB auf *ADwin*-Variablen des Simulink-Modells zugreifen können, ermöglicht der *ADsim*-Treiber den Zugriff auf die übrigen Variablen und Felder des Simulink-Modells, also auf Parameter, Signale und Blockzustände.
Die beiden Treiber können unabhängig voneinander verwendet werden. Beachten Sie, dass sich die Treiber bei der Fehlerverarbeitung sehr unterschiedlich verhalten.

Wir empfehlen, vorwiegend den *ADwin*-Treiber für MATLAB zu verwenden. Im Vergleich bietet er folgende Vorteile:

– Der Treiber unterstützt die kontinuierliche Datenübertragung über Fifo-Felder.
– Kompilierte Simulink-Modelle (mit *ADsim* T11) können mit dem Befehl `Boot` auf die *ADwin*-Hardware übertragen werden.
– Der Treiber arbeitet unter Linux und Mac OS.
– Der Treiber arbeitet auch mit MATLAB-Versionen vor R2007b (7.5).

Anwender von *ADsim* T11 können alle Funktionen des *ADwin*-Treibers für MATLAB nutzen bis auf folgende:

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

2.4 Weitergabe einer Stand-alone Application

Mit dem MATLAB-Compiler können Stand-alone Applications erzeugt werden, also eigenständig lauffähige Programme. In eine Stand-alone Application kann auch der *ADwin*-Treiber für MATLAB integriert werden.

Beachten Sie folgende Hinweise, wenn Sie eine Stand-alone Application an einen Endnutzer weitergeben:

– Der Endnutzer benötigt neben der Stand-alone Application auch ein *ADwin*-Software-Paket, und er muss die Software installlieren.
– Die Installation des *ADwin*-Software-Pakets ist in Kapitel 3.1 beschrieben. Weitere Installationsschritte entfallen beim Endkunden.
– Der Endkunde benötigt weder einen Lizenzschlüssel für den *ADwin*-Treiber für MATLAB, der in der Application enthalten ist, noch für *ADbasic*.

3 *ADwin*-Treiber für MATLAB® installieren

3.1 „*ADwin* Installation“ ausführen

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

3.1.1 Installation unter Linux oder Mac

Folgen Sie der Installationsanleitung im Handbuch „ADwin für Linux / Mac“. Achten Sie darauf, auch das Archiv `adwin-matlab-x.y.tar.gz` zum Schluss zu installieren.

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

Treiber und Beispiele für MATLAB	`./Developer/Matlab`
Beispiele für *ADbasic*	`./samples_ADwin`

Fahren Sie fort mit Kapitel 3.2 „ADwin-Treiber für MATLAB® einbinden“.

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*-System aufbauen.

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

Treiber und Beispiele für MATLAB	`.\Developer\Matlab\…`
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`

64 Bit-Version benötigt Visual Studio

Beachten Sie bei Matlab in der 64 Bit-Version: Um den *ADwin*-Treiber nutzen zu können, benötigen Sie laut MathWorks zusätzlich einen C-Compiler. MathWorks gibt auf seiner Homepage `www.mathworks.de` unter dem Stichwort „Supported and Compatible Compilers“ eine Liste geeigneter C-Compiler für das jeweilige `MATLAB`-Release an, meist mit Download-Link.

Der Zwang zur Nutzung eines externen C-Compilers besteht in Matlab (64 Bit) für alle externen Programmpakete (DLLs). Matlab erzeugt mit dem Compiler eine interne Datei, über die es eine DLL erst verwenden kann.

32 Bit-Version: Compiler konfigurieren

Wenn in einer 32 Bit-Version der C-Compiler nicht funktioniert wie erwartet, muss der Compiler möglicherweise noch konfiguriert werden. Zum Konfigurieren geben Sie in Matlab den Befehl `mex -setup` ein.

3.2 *ADwin*-Treiber für MATLAB® einbinden

Wenn Sie in MATLAB mit dem *ADwin*-System arbeiten wollen, müssen Sie den Treiber in die Entwicklungsumgebung einbinden.

Neuer „license key“

– Geben Sie den *ADwin*-Treiber frei, indem Sie Ihren neuen „license key“ in *ADbasic* eingeben (Menu: „`Help ▸ About ADbasic`“).

  Sie finden den „license key“ auf der ersten Seite im Handbuch *ADbasic*.

  Sie müssen zum Registrieren des Lizenzschlüssels unter Windows NT, 2000, XP, Vista, 7, 8 und 10 zur Benutzergruppe „Administratoren“ gehören. Es genügt nicht, volle Zugriffsrechte auf dem PC zu haben. Fragen Sie hierzu Ihren System-Administrator.

– Starten Sie den „`Path Browser`“ in MATLAB. Fügen Sie den Pfad `<C:\ADwin\Developer\Matlab\ADwin>` (Standardpfad) zu der Liste verwendeter Verzeichnisse hinzu und speichern sie sie.

  Wenn eine ältere Version der Datei `<ADlab.DLL>` im MATLAB-Verzeichnis existiert, löschen Sie diese Datei. Anderenfalls verwendet MATLAB die ältere Dateiversion.

– Aus Ihrer Matlab-Version ergibt sich das erforderliche *ADwin*-Treiberpaket. Das Paket gilt jeweils für 32 Bit- und 64 Bit-Versionen.

  Für Matlab-Versionen ab R2007b (*ADwin*-Treiberpaket P3) sind alle weiteren Einstellungen bereits voreingestellt.

Matlab-Version	*ADwin*-Treiberpaket		
	P1	P2	P3
7.0 oder früher	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) oder später			x

– Wenn Sie eines der älteren *ADwin*-Treiberpakete P1 oder P2 benötigen:
  - Löschen Sie den Ordner `<.\ADwin>` im Verzeichnis `<C:\ADwin\Developer\Matlab>`.
  - Erzeugen Sie je nach gewählter Matlab-Version eine Kopie des Ordners `<.\ADwin_P1>` oder `<.\ADwin_P2>`.
  - Benennen Sie den kopierten Ordner um in `<.\ADwin>`.

Der *ADwin*-Treiber ist nun in MATLAB eingebunden.

3.3 Zugriff auf das *ADwin*-System testen

Bei der Installation der Hardware und -Software haben Sie bereits den Zugriff auf das *ADwin*-System erfolgreich geprüft. Der folgende Test zeigt Ihnen, ob Sie auch aus MATLAB korrekt auf das *ADwin*-System zugreifen können.

Geben Sie im „Command window“ die folgenden Zeilen ein:

```
>> ADwin_Init(); %erst ab Matlab-Version 7.1 erforderlich
>> Boot('C:\ADwin\ADwin9.btl', 0);
>> Get_Last_Error()
ans =
       0
```

Mit den Zeilen führen Sie Folgendes aus:

– Sie initialisieren MATLAB für die Kommunikation mit *ADwin*-Systemen; dabei wird automatisch die Device No. 1 als Zielgerät festgelegt.

  Wenn Sie bei der Installation in *ADconfig* eine andere Gerätenummer eingestellt haben, stellen Sie sie separat mit `Set_DeviceNo` ein.

– Sie übertragen das Betriebssystem für den Prozessor T9 auf das *ADwin*-System (= Boot-Vorgang).

  Die Dateinamen für andere Prozessoren als T9 finden Sie auf Seite 14.

– Sie fragen den Fehlercode ab, der beim Booten erzeugt wurde. Der Wert 0 bestätigt Ihnen die Betriebsbereitschaft des *ADwin*-Systems.

  Ein Fehlercode > 0 zeigt einen Fehler beim Booten an. Sie finden eine Liste der Fehlernummern in Kapitel A.2 im Anhang.

Sie können jetzt mit allen Funktionen des Treibers auf das *ADwin*-System zugreifen.

Zur Einführung empfehlen wir Ihnen, mit den Beispielprogrammen aus dem Anhang, Abschnitt A.1 zu arbeiten.

3.4 Ein *ADwin*-System über andere PCs ansprechen

Wenn ein *ADwin*-System 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 2 Möglichkeiten, einen Fehler bei der Ausführung einer *ADwin*-Funktion zu erkennen:

Rückgabewert der Funktion

1. Der Rückgabewert einer Funktion gibt an, ob ein Fehler aufgetreten ist.

   Beachten Sie bitte:
   - Die Funktionen verwenden unterschiedliche Werte, um einen Fehler anzuzeigen.
   - Bei Funktionen, die mehrere Werte auf einmal lesen, ist der Rückgabewert nicht die Fehlernummer:
     `Get_Par_Block`, `Get_Par_All`, `Get_FPar_Block`, `Get_FPar_All`, `Get_FPar_Block_Double`, `Get_FPar_All_Double`, `GetData_Double`, `GetFifo_Double`.
   - 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_FPar_Double`, `Get_Processdelay`, `Free_Mem`.

Get_Last_Error

2. Die Funktion `Get_Last_Error` (siehe Seite 42) gibt die Nummer des zuletzt aufgetretenen Fehlers zurück.

   Um jeden auftretenden Fehler zu behandeln, rufen Sie `Get_Last_Error` nach jedem Zugriff auf ein *ADwin*-System auf.

Wegen der Nachteile der 1. Variante (Rückgabewert der Funktion) empfehlen wir, Fehler grundsätzlich mit `Get_Last_Error` abzufragen.

Zum Beispiel kann ein Fehler bei der Funktion `Get_Processdelay` nur mit `Get_Last_Error` erkannt werden.

Zunächst die unsichere Variante (Rückgabewert der Funktion):

```
gd_2 = Get_Processdelay(2); % Processdelay von Prozess 2
if (gd_2 ~= 255)
 … % kein Fehler
end
```

Wenn `Get_Processdelay` den Wert 255 zurückgibt, ist unklar, ob ein Fehler aufgetreten ist oder ob der Parameter `Processdelay` den Wert 255 enthält. Deshalb muss `Get_Last_Error` verwendet werden:

```
gd_2 = Get_Processdelay(2); % Processdelay von Prozess 2
if (Get_Last_Error() == 0)
 … % kein Fehler
end
```

4.2 Die „DeviceNo.“

Eine „Device No.“ ist die Gerätenummer eines bestimmten *ADwin*-Systems an einem PC. Ein *ADwin*-System wird immer über eine „Device No.“ angesprochen.

Sie legen die „Device No.“ für jedes *ADwin*-System mit dem Programm *ADconfig* an. Nähere Informationen zur Programmbedienung finden Sie in der Online-Hilfe von *ADconfig*.

Alle Funktionen des *ADwin*-Treibers für MATLAB verwenden eine interne Variable `DeviceNo`, um auf ein *ADwin*-System zuzugreifen. Mit der Funktion

`Set_DeviceNo` stellen Sie ein, auf welches *ADwin*-System die Funktionen des *ADwin*-Treibers zugreifen. Die Voreinstellung ist die Nummer 1.

4.3 Datentypen

Die Funktionen und Parameter des *ADwin*-Treibers für MATLAB verwenden folgende Datentypen:

ADwin-Treiber für MATLAB		*ADwin*-Hardware
Datentyp	Definition	Datentyp
char	unsigned integer 8 Bit	String
int32	signed integer 32 Bit	Long
single	float 32 Bit	Float / Float32
double	float 64 Bit	Float64

Als Funktionsparameter können Sie einfache Variablen (1×1 Matrix) und Reihenvektoren verwenden.

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
+∞ (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

4.3.1 Ganzzahlige Datentypen beim Datenaustausch wandeln

Beim Lesen von ganzzahligen Werten geben die *ADwin*-Funktionen (z.B. mit `Get_Par`...) Werte mit dem Datentyp `double` zurück. Umgekehrt werden beim Schreiben von Modellvariablen in der Regel Werte mit dem Datentyp `double` übergeben.

Bei ganzzahligen Datentypen ist es stattdessen bei Zählerständen und Bitmustern notwendig, in MATLAB den Datentyp `int32` oder `uint32` zu verwenden. Dazu muss der Datentyp gewandelt werden. Zu unterscheiden sind:

– Zählerstände: Um mit Zählerständen zu rechnen – typischerweise Differenzen zwischen 2 Zählerständen – müssen Werte mit dem Datentyp `int32` verwendet werden.
  Das betrifft sowohl Ereigniszähler als auch Zeitzähler.
– Bitmuster: Bitmuster müssen im Datentyp `uint32` bearbeitet werden, auch um die MATLAB-Funktionen für Bitoperationen nutzen zu können.

  Bitmuster werden verwendet bei:
  - Status von 16/32 Digitalkanälen
  - Datenbytes von CAN-Nachrichten
  - Signale eines SSI-Encoders (Gray-codiert)
  - gepackte Werte: 2 Werte zu 16 Bit in einem 32 Bit-Wert
– Boolesche Werte: Der Datentyp `double` stellt die Booleschen Werte 0 und 1 richtig dar, eine Typwandlung ist nicht erforderlich.

  Boolesche Werte kommen z.B. als Status einzelner Digitalkanäle vor.
– Zahlenwerte: Der Datentyp `double` stellt Zahlenwerte richtig dar, eine Typwandlung ist nicht erforderlich.

Datentyp int32

So wandeln Sie Daten in den Datentyp `int32`:

– Variable lesen und in Datentyp `int32` wandeln:
```
value = Get_Par(5);
val_int32 = cast(value, 'int32');
```
– Beim Schreiben muss der Datentyp nicht gewandelt werden.

  Der Ablauf ist nachfolgend der Vollständigkeit halber dargestellt, d.h. Wert in Datentyp `int32` wandeln und in Variable schreiben:
```
val_int32 = cast(value, 'int32')
Set_Par(5, val_int32);
```

Datentyp uint32

So wandeln Sie Daten in den Datentyp `uint32`:

– Wert lesen und in Datentyp `uint32` wandeln:
```
value = Get_Par(5);
val_int32 = cast(value, 'int32');
val_uint32 = typecast(val_int32, 'uint32')
```
– Wert in Datentyp `int32` wandeln und in Variable schreiben:
```
val_int32 = typecast(val_uint32, 'int32')
Set_Par(5, val_int32);
```

4.4 Daten von 2-dimensionalen Feldern austauschen

In *ADbasic* können globale DATA-Felder 2-dimensional (2D) deklariert werden. Die Funktionen des *ADwin*-Treibers für MATLAB verwenden an dieser Stelle jedoch nur Zeilenvektoren.
Mit Hilfe der MATLAB-Funktion `reshape` kann ein Zeilenvektor sehr einfach zu einer 2-dimensionalen Matrix gewandelt werden.

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

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

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

Beachten Sie auch die Hinweise zu 2-dimensionalen Feldern im *ADbasic*-Handbuch.

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 MATLAB mit `GetData` gelesen:

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

Die Daten werden in folgender Reihenfolge übertragen:

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

Die Funktion `GetData` legt das Element `DATA_8[7][2]` also in `vector[20]` ab.

Die allgemeine Formel ergibt mit `s=3`:

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

5 Beschreibung der *ADwin*-Funktionen

Die Beschreibung der Funktionen ist in folgende Abschnitte unterteilt:

Rufen Sie die Funktion `ADwin_Init` als erstes auf, bevor Sie mit anderen Funktionen auf das *ADwin*-System zugreifen.

Im Anhang A.3 finden Sie eine Übersicht aller Funktionen. Sie können die Beschreibung der *ADwin*-Funktionen auch über die MATLAB-Hilfefunktion im „`Command window`“ aufrufen:

```
>> help adwin
```

oder

```
>> help [Funktionsname]
```

In früheren Versionen des *ADwin*-Treibers für MATLAB wurden anstelle der Funktionsnamen Befehlsnummern verwendet. Eine Zuordnung zwischen Nummern und Funktionsnamen finden Sie im Anhang A.4.

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*-Treiber nicht enthalten. Sie können solche Anwendungen in *ADbasic* (bzw. mit *ADsim* T11) programmieren.

5.1 Systemsteuerung und -information

Initialisierung des *ADwin*-Systems und Information über den Betriebszustand.

ADwin_Init

`ADwin_Init` initialisiert Matlab für die Kommunikation mit *ADwin*-Systemen.

```
ADwin_Init()
```

Bemerkungen

Bei der Initialisierung werden wichtige Vorgabewerte eingestellt, darunter auch die beiden folgenden:

- DeviceNo = 1; siehe auch Set_DeviceNo (unten).
- Show_Errors = On; siehe auch Show_Errors (Seite 42).

`ADwin_Init` muss als erstes aufgerufen werden, damit *ADwin*-Funktionen korrekt arbeiten. Fehlt der Aufruf und eine *ADwin*-Funktion wird genutzt, dann ruft die Funktion `ADwin_Init` selbst auf.

Beispiel

```
% Matlab für Kommunikation mit ADwin initialisieren,
% Vorgabewerte für DeviceNo und Fehleranzeige setzen.
ADwin_Init();
```

ADwin_Unload

`ADwin_Unload` löscht die *ADwin*-Funktionen aus dem PC-Speicher und gibt den Speicherplatz frei.

```
ADwin_Unload()
```

Set_DeviceNo

`Set_DeviceNo` setzt die „Device-Nummer“.

```
Set_DeviceNo(DeviceNo)
```

Parameter

`DeviceNo`	Karten-Adresse oder Device-Nummer in dezimaler Schreibweise. Die Voreinstellung ist 1.

Bemerkungen

ADwin-Systeme werden vom PC durch die sogenannte „Device-Nummer“ unterschieden und angesprochen (siehe Kapitel 4.2 auf Seite 8). Systeme mit Link-Adapter sind bereits werkseitig eingestellt (auf 336).

Weitere Informationen finden Sie in der Online-Hilfe des Programms *ADconfig* oder im Handbuch „*ADwin* Installation“.

Beispiel

```
% Die Device-Nummer 3 setzen
Set_DeviceNo(3);
```

Get_DeviceNo

`Get_DeviceNo` gibt die aktuelle „Device-Nummer“ zurück.

```
Get_DeviceNo ()
```

Bemerkungen

ADwin-Systeme werden vom PC durch die sogenannte „Device-Nummer“ unterschieden und angesprochen (siehe Kapitel 4.2 auf Seite 8). Systeme mit Link-Adapter sind bereits werkseitig eingestellt (auf 336).

Weitere Informationen finden Sie in der Online-Hilfe des Programms *ADconfig* oder im Handbuch „*ADwin* Installation“.

Beispiel

```
% Die aktuelle Device-Nummer abfragen
num = Get_DeviceNo();
```

Boot

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

```
Boot (Filename, Memsize)
```

Parameter

`Filename`	Pfad und Dateiname der Betriebssystem-Datei (siehe Liste unten).
`MemSize`	Für Prozessoren ab T9: 0 (Null). Für T2, T4, T5, T8: Speichergröße; die folgenden Werte sind zulässig: 10000: 64 KiB 100000: 1 MiB 200000: 2 MiB 400000: 4 MiB 800000: 8 MiB 1000000: 16 MiB 2000000: 32 MiB
Rückgabewert	Statusmeldung: <1000: Fehler beim Booten 8000: Booten o.k.; ab T9. >8000: Booten o.k.; nur für T2…T8. Der Wert steht für die installierte Speichergröße.

Bemerkungen

Die Initialisierung löscht alle Prozesse auf dem System und setzt alle globalen Variablen auf den Wert 0. Der Parameter `Processdelay` wird auf den Wert 1000 voreingestellt (siehe auch Seite 22).

Für *ADbasic*-Nutzer: Die zu ladende Betriebssystemdatei ist abhängig vom Prozessortyp des angesprochenen Systems. Die folgende Tabelle zeigt die Dateinamen für die verschiedenen Prozessoren. Die Dateien sind im Verzeichnis `<C:\ADwin\>` abgelegt.

Prozessor	Betriebssystemdatei
T225 (T2)	ADwin2.btl
T400 (T4)	ADwin4.btl
T450 (T5)	ADwin5.btl
T805 (T8)	ADwin8.btl
T9	ADwin9.btl
	ADwin9s.btl Optimiertes Betriebssystem mit etwas geringerem Speicherbedarf.
T10	ADwin10.btl
T11	ADwin11.btl
T12	ADwin12.btl
T12.1	ADwin121.btl

Der PC kann erst mit dem *ADwin*-System kommunizieren, nachdem Sie das Betriebssystem geladen haben. Laden Sie das Betriebssystem nach jedem Aus- und Einschalten des *ADwin*-Systems neu.

Für Nutzer von *ADsim* T11:

- Sie geben unter `Filename` das mit *ADsimDesk* kompilierte Simulink-Modell an, in das auch das Betriebssystem für den Prozessor integriert ist. Die Datei ist im Modellordner abgelegt, im Unterordner `<model>_ert_rtw/ADwin/` unter `<model>11c.btl`.
- `<model>` steht für den Namen des Simulink-Modells. Die Bezeichnung `11c` steht für den Prozessortyp T11 der *ADwin*-Hardware.
- Beachten Sie, dass Sie *ADbasic*-Prozesse und mit *ADsim* T11 kompilierte Simulink-Modelle nicht gleichzeitig auf *ADwin*-Hardware ausführen können.

Das erfolgreiche Laden des Betriebssystems mit `Boot` nimmt bis zu einer Sekunde in Anspruch. Alternativ können Sie das Betriebssystem auch über die Entwicklungsumgebung *ADbasic* (Schaltfläche `B`) bzw. über *ADsimDesk* laden.

Beispiel

```
% Betriebssystem für den Prozessor T10 laden
Boot('C:\ADwin\ADwin10.btl', 0);

% mit ADsim T11 kompiliertes Simulink-Modell laden
Boot('C:\ADwin\ADsim\Developer\Examples\ADsim32_DLL_Example_
ert_rtw\ADwin\ADsim32_DLL_Example11c.btl', 0);
```

Test_Version

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

```
Test_Version()
```

Parameter

Rückgabewert 0: OK
≠0: Fehler

Beispiel

```
% Test, ob das Betriebssystem geladen ist
ret_val = Test_Version();
```

Processor_Type

`Processor_Type` gibt den Prozessortyp des Systems zurück.

`Processor_Type``()`

Parameter

Rückgabewert Prozessor-Kennziffer des Systems

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

Beispiel

```
% Prozessortyp abfragen
ret_val = Processor_Type();
```

Workload

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

`Workload` `(Priority)`

Parameter

`Priority` 0: aktuelle Gesamtauslastung des Prozessors
≠0:wird derzeit nicht unterstützt

Rückgabewert ≠255: 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 1 ms) aufrufen.

Beispiel

```
% Prozessorauslastung abfragen
ret_val = Workload(0);
```

Free_Mem

`Free_Mem` ermittelt den auf dem System verfügbaren freien Speicher für verschiedene Speicherarten.

```
Free_Mem (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 Datenspeicher (EM_LOCAL); nur T11
- 3 : interner Datenspeicher (DM_LOCAL); T9…T11
- 4 : externer DRAM-Speicher (DRAM_EXTERN); T9…T11
- 5: Cacheable: Speicher, der Daten an den Cache liefern kann; nur T12/T12.1.
- 6: Uncacheable: Speicher, der keine Daten an den Cache liefern kann; nur T12/T12.1.

Rückgabewert Zusammenhängender freier Speicher (in Byte)
Bei `Mem_Spec` = 5/6 ist der Wert in Einheiten von kBytes angegeben.
255: (nicht eindeutig).

Bemerkungen

Die Funktion ist in Verbindung mit *ADsim* T11 nicht verwendbar.

Beispiel

```
% Abfrage des freien Speichers im externen DRAM
ret_val = Free_Mem(4);
```

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*-System.

Die Funktionen dieses Abschnitts sind in Verbindung mit *ADsim* T11 nicht verwendbar.

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 die Binärdatei eines Prozesses ins *ADwin*-System.

```
Load_Process (Filename)
```

Parameter

Filename	Pfad und Dateinamen der zu ladenden Binärdatei.
Rückgabewert	1 OK ≠1 Error

Bemerkungen

Die Funktion ist in Verbindung mit *ADsim* T11 nicht verwendbar.

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

Das Ausschalten eines Systems löscht geladene Prozesse. Sie müssen deshalb nach dem Einschalten die von Ihnen benötigten Prozesse erneut laden.

Sie können bis zu 10 Prozesse auf ein *ADwin*-System laden. Laufende Prozesse werden durch das Laden weiterer Prozesse (mit unterschiedlicher Prozessnummer) auf das System 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

```
% Binärdatei Testprog.T91 laden
% T91 = Prozessortyp T9, Prozessnr. 1
ret_val = Load_Process('C:\MyADbasic\Testprog.T91');
```

Start_Process

`Start_Process` startet einen Prozess.

```
Start_Process (ProcessNo)
```

Parameter

`ProcessNo`	Nummer des Prozesses (1…10, 15)
Rückgabewert	≠255: OK 255: Fehler

Bemerkungen

Die Funktion ist in Verbindung mit *ADsim* T11 nicht verwendbar.

Die Funktion hat keine Auswirkung, wenn Sie die Nummer eines Prozesses angeben, der

- bereits läuft oder
- gleich der Nummer des aufrufenden Prozesses ist oder
- noch nicht auf das *ADwin*-System geladen ist.

Beispiel

```
% Prozess 1 starten
ret_val = Start_Process (2);
```

Stop_Process

`Stop_Process` stoppt einen Prozess.

```
Stop_Process (ProcessNo)
```

Parameter

`ProcessNo`	Nummer des Prozesses (1…10, 15)
Rückgabewert	≠255: OK 255: Fehler

Bemerkungen

Die Funktion ist in Verbindung mit *ADsim* T11 nicht verwendbar.

Die Funktion hat keine Auswirkung, wenn Sie die Nummer eines Prozesses angeben, der

- bereits gestoppt ist oder
- noch nicht auf das *ADwin*-System geladen ist.

Beispiel

```
% Prozess 2 stoppen
ret_val = Stop_Process(2);
```

Clear_Process

`Clear_Process` löscht einen Prozess aus dem Speicher.

```
Clear_Process (ProcessNo)
```

Parameter

`ProcessNo`	Nummer des Prozesses (1…10, 15)
Rückgabewert	≠1: OK 1: Fehler

Bemerkungen

Die Funktion ist in Verbindung mit *ADsim* T11 nicht verwendbar.

Geladene Prozesse belegen Speicherplatz im System. Sie können mit `Clear_Process` Prozesse aus dem Speicher entfernen, um für andere Prozesse mehr Platz 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-Systemen sorgt der Prozess 15 für das Blinken der LED; nach dem Entfernen blinkt die LED nicht mehr.

Beispiel

```
% Freigeben des von Prozess 2 belegten Speichers.
% Deklarierte Datas und FIFOs bleiben erhalten.
ret_val = Clear_Process(2);
```

Process_Status

`Process_Status` liefert den Status eines Prozesses.

```
Process_Status (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. -1: Prozess wird gestoppt, d.h. er hat ein `Stop_Process`, erhalten, wartet aber noch auf den letzten Event.

Bemerkungen

Die Funktion ist in Verbindung mit *ADsim* T11 nicht verwendbar.

Beispiel

```
% Status von Prozess 2 zurückgeben
ret_val = Process_Status(2);
```

Set_Processdelay

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

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

Parameter

`ProcessNo`	Nummer des Prozesses: 1…10; mit *ADsim* T11: 1…2.
`Process-delay`	Einzustellender Wert (1…2^{31}-1) für den Parameter `Processdelay` des Prozesses.
Rückgabewert	≠255: OK 255: Fehler

Bemerkungen

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

Zu jedem Prozess gibt es eine minimale Zykluszeit: Ein Unterschreiten des Minimalwerts führt zu einer Überlastung des Prozessors und die Kommunikation zum *ADwin*-System bricht ab.

Die Zykluszeit wird in Taktzyklen des *ADwin*-Prozessors angegeben. Der Taktzyklus hängt vom Prozessortyp und von der Priorität des Prozesses ab:

Prozessortyp	Prozesspriorität	
	hoch	niedrig
T2, T4, T5, T8	1000 ns	64 µs
T9	25 ns	100 µs
T10	25 ns	50 µs
T11	3,3 ns	0,003 µs = 3,3 ns
T12	1 ns	1 ns
T12.1	1,5 ns	1,5 ns

Beispiel

```
% Processdelay 2000 bei Prozess 1 einstellen.
ret_val = Set_Processdelay(1,2000);
```

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

Get_Processdelay gibt den Parameter Processdelay für einen Prozess zurück.

```
Get_Processdelay (ProcessNo)
```

Parameter

ProcessNo	Nummer des Prozesses: 1…10; mit *ADsim* T11: 1…2.
Rückgabewert	≠255: Aktuell eingestellter Wert (1…2^{31}-1) für den Parameter Processdelay des Prozesses. 255: Fehler

Bemerkungen

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

Für *ADsim*-Nutzer: Der Parameter Processdelay entspricht der Basis-Abtastzeit (fixed-step size) in Simulink. Während die Basis-Abtastzeit in Sekunden angegeben wird, ist Processdelay ein Vielfaches der Prozessor-Tyktzyklus, siehe Set_Processdelay.

Beispiel

```
% Processdelay des ADbasic-Prozesses 1 abfragen
x = 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 Matlab 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 Matlab 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*-System 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 auf dem *ADwin*-System haben folgenden Wertebereich:

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

Die Befehle übertragen ganzzahlige Werte mit 32 Bit Breite. Bei der Rückgabe von Einzelwerten speichert MATLAB den Wert in einer Variablen vom Datentyp Double.

Wenn die globale Variable einen Zählerstand oder ein Bitmuster enthält, muss der Wert in MATLAB mit dem Datentyp int32 oder uint32 verarbeitet werden; Näheres siehe Abschnitt Datentypen auf Seite 6.

Set_Par

Set_Par setzt eine globale Variable PAR auf den gewünschten Wert.

Set_Par (Index, Value)

Parameter

Index	Nummer (1 ... 80) der globalen Variablen PAR_1 ... PAR_80.
Value	Zu setzender Wert für die Long-Variable.
Rückgabewert	≠255: OK 255: Fehler

Beispiel

```
% LONG-Variable PAR_1 auf den Wert 2000 setzen
ret_val = Set_Par(1, 2000);
```

Get_Par

Get_Par gibt den Wert einer globalen Variablen PAR zurück.

Get_Par (Index)

Parameter

Index	Nummer (1 ... 80) der globalen Variablen PAR_1 ... PAR_80.
Rückgabewert	≠255: Aktueller Wert der Variablen, Datentyp int32. 255: Fehler

Beispiel

```
% LONG-Variable PAR_1 lesen und in x schreiben
x = Get_Par (1);
```

Get_Par_Block

Get_Par_Block überträgt eine anzugebende Anzahl an globalen Variablen PAR in einen Zeilenvektor (Datentyp int32).

```
Get_Par_Block (StartIndex, 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	Zeilenvektor mit den übertragenen Werten.

Beispiel

Werte der Variablen PAR_10 ... PAR_39 lesen und im Zeilenvektor v speichern:

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

Get_Par_All

Get_Par_All überträgt alle globalen Variablen PAR_1 ... PAR_80 in einen Zeilenvektor (Datentyp int32).

```
Get_Par_All ()
```

Parameter

Rückgabewert	Zeilenvektor mit übertragenen Werten (PAR_1...PAR_80)

Beispiel

Werte der Variablen PAR_1 ... PAR_80 lesen und im Zeilenvektor v speichern:

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

5.3.2 Globale Variablen FPAR_1...FPAR_80

Die globalen Variablen FPAR_1 ... FPAR_80 auf dem *ADwin*-System haben folgenden Wertebereich, je nach Prozessortyp (siehe auch Abschnitt Datentypen):

- FLOAT (bis 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 (T12/T12.1)
  negative: $-1{,}797693134862315 \cdot 10^{+308} \ldots -2{,}225073858507201 4 \cdot 10^{-308}$
  positive: $+2{,}225073858507201 4 \cdot 10^{-308} \ldots +1{,}797693134862315 \cdot 10^{+308}$

Set_FPar

Set_FPar setzt eine globale Variable FPAR auf den gewünschten Single-Wert.

```
Set_FPar (Index, Value)
```

Parameter

Index	Nummer (1 ... 80) der globalen Variablen FPAR_1 ... FPAR_80.
Value	Zu setzender Wert vom Datentyp single für die Variable.
Rückgabewert	≠255: OK 255: Fehler

Bemerkungen

Set_FPar überträgt immer einen Fließkommawert mit 32 Bit, auch wenn FPAR eine Genauigkeit von 64 Bit haben kann.

Beispiel

```
% Variable FPAR_6 auf 34.7 setzen
Set_FPar(6, 34.7);
```

Set_FPar_Double

Set_FPar_Double setzt eine globale Variable FPAR auf den gewünschten Double-Wert.

```
Set_FPar_Double (Index, Value)
```

Parameter

Index	Nummer (1 ... 80) der globalen Variablen FPAR_1 ... FPAR_80.
Value	Zu setzender Wert vom Datentyp Double für die Variable FPAR.
Rückgabewert	≠255: OK 255: Fehler

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_FPar gibt den Single-Wert einer globalen Variablen FPAR zurück.

```
Get_FPar (Index)
```

Parameter

Index	Nummer (1 … 80) der globalen Variablen FPAR_1 … FPAR_80.
Rückgabewert	≠255: Aktueller Wert der Variablen, Datentyp single. 255: Fehler

Bemerkungen

Ab Prozessor T12 haben FPAR-Variablen im *ADwin*-System doppelte Genauigkeit (64 Bit). Get_FPar liefert auch dann einen Wert vom Datentyp Single zurück.

Beispiel

```
% FPAR_56 lesen und den Wert in die MATLAB-Variable x schreiben
x = Get_FPar(56);
```

Get_FPar_Block

Get_FPar_Block überträgt die angegebene Anzahl an aufeinander folgenden globalen Variablen FPAR in einen Zeilenvektor (Datentyp single).

```
Get_FPar_Block (StartIndex, 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	Zeilenvektor mit den übertragenen Werten vom Datentyp single.

Beispiel

Werte der Variablen FPAR_10 … FPAR_34 lesen und im Zeilenvektor v speichern:

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

Get_FPar_All

Get_FPar_All überträgt alle 80 globalen Variablen FPAR_1 … FPAR_80 in einen Zeilenvektor (Datentyp single).

```
Get_FPar_All ()
```

Parameter

Rückgabewert	Zeilenvektor mit den übertragenen Werten vom Datentyp single.

Beispiel

Werte der Variablen FPAR_1 … FPAR_80 lesen und im Zeilenvektor v speichern:

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

Get_FPar_Double

Get_FPar_Double gibt den Double-Wert einer globalen Variablen FPAR zurück.

Get_FPar_Double (Index)

Parameter

Index	Nummer (1 … 80) der globalen Variablen FPAR_1 … FPAR_80.
Rückgabewert	≠255: Aktueller Wert der Variablen 255: Fehler

Bemerkungen

Bis T11 gilt: Beachten Sie, dass Fließkommawerte im *ADwin*-System einfache Genauigkeit (32 Bit) haben. Get_FPar_Double liefert auch dann einen Wert vom Datentyp Double zurück.

Beispiel

```
% FPAR_56 lesen und den Wert in die MATLAB-Variable x schreiben
x = Get_FPar_Double(56);
```

Get_FPar_Block_Double

Get_FPar_Block_Double überträgt die angegebene Anzahl an aufeinander folgenden globalen Variablen FPAR in einen Zeilenvektor (Datentyp double).

Get_FPar_Block_Double (StartIndex, 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	Zeilenvektor mit den übertragenen Werten vom Datentyp double.

Bemerkungen

Bis T11 gilt: Beachten Sie, dass Fließkommawerte im *ADwin*-System einfache Genauigkeit (32 Bit) haben. Sie sollten daher Werte aus dem Feld Array nur mit einfacher Genauigkeit anzeigen lassen, um Missverständnisse bezüglich der Genauigkeit zu vermeiden.

Beispiel

Werte der Variablen FPAR_10 … FPAR_34 lesen und im Zeilenvektor v speichern:

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

Get_FPar_All_Double

Get_FPar_All_Double überträgt alle 80 globalen Variablen FPAR_1 … FPAR_80 in einen Zeilenvektor (Datentyp double).

```
Get_FPar_All_Double ()
```

Parameter

Rückgabewert Zeilenvektor mit den übertragenen Werten vom Datentyp double.

Bemerkungen

Bis T11 gilt: Beachten Sie, dass Fließkommawerte im *ADwin*-System einfache Genauigkeit (32 Bit) haben. Sie sollten daher Werte aus dem Feld Array nur mit einfacher Genauigkeit anzeigen lassen, um Missverständnisse bezüglich der Genauigkeit zu vermeiden.

Beispiel

Werte der Variablen FPAR_1 … FPAR_80 lesen und im Zeilenvektor v speichern:

```
v=Get_FPar_All_Double();
```

5.4 Übertragung von Datenfeldern (Arrays)

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

- Einfache Datenfelder
- FIFO-Felder
- Datenfelder mit String-Daten

Sie müssen jedes Feld vor seiner Verwendung in *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	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: $+2{,}2250738585072014 \cdot 10^{-308}$ … $+1{,}797693134862315 \cdot 10^{+308}$

Beim Datentyp LONG übertragen die Befehle ganzzahlige Werte mit 32 Bit Breite. Rückgabewerte in MATLAB haben den Datentyp double.

Wenn ganzzahlige Werte einen Zählerstand oder ein Bitmuster enthalten, muss der Wert in MATLAB mit dem Datentyp int32 oder uint32 verarbeitet werden; Näheres siehe Abschnitt Datentypen auf Seite 6.

Data_Length

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

```
Data_Length (DataNo)
```

Parameter

DataNo	Nummer des Felds (1...200).
Rückgabewert	>0: Deklarierte Länge des Felds (=Anzahl der Elemente) 0: Fehler - 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 MATLAB bestimmt man die Länge des Felds DATA_2:

```
>> Data_Length(2)
ans =
    2000
```

SetData_Double

SetData_Double überträgt alle Daten von einem Zeilenvektor mit Datentyp double in ein DATA-Feld des *ADwin*-Systems.

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

Parameter

DataNo	Nummer (1...200) des Zielfelds DATA_1 ... DATA_200. DATA kann den Datentyp Long, Float, Float32 oder Float64 haben.
Vector	Zeilenvektor, aus dem Daten übertragen werden.
StartIndex	Nummer (≥1) des ersten Elements im Zielfeld, das beschrieben wird.
Rückgabewert	≠255: OK 255: Fehler oder Feld ist nicht deklariert.

Bemerkung

Das DATA-Feld muß mindestens so groß sein wie die Anzahl der Werte im MATLAB-Vektor zuzüglich StartIndex.

Wenn der Datentyp des DATA-Felds 32 Bit Genauigkeit hat, werden die Double-Werte aus Vector in diesen Datentyp gewandelt. Dadurch können Informationen verloren gehen.

Bis T11 gilt: Bitte beachten Sie, dass Fließkomma-Werte im *ADwin*-System einfache Genauigkeit (32 Bit) haben. Sie sollten daher Daten aus Vector nur mit einfacher Genauigkeit anzeigen lassen, um Missverständnisse bezüglich der Genauigkeit zu vermeiden.

Um MATLAB-Daten aus mehrdimensionalen Matrizen zu übertragen, müssen Sie die Daten erst in einen Zeilenvektor übertragen. Bei einem Spaltenvektor wird nur das erste Daten-Element übertragen.

Die Funktion SetData_Double ersetzt die Funktion Set_Data, die bei früheren Treiberversionen verwendte wurde.

Beispiel

Den vollständigen Zeilenvektor x in das Feld DATA_1 übertragen, beginnend ab dem Element DATA_1[100]:

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

GetData_Double überträgt Teile eines DATA-Felds vom *ADwin*-System in einen Zeilenvektor (Datentyp double).

GetData_Double (DataNo, Startindex, Count)

Parameter

DataNo	Nummer (1...200) des Quellfelds DATA_1 … DATA_200. DATA kann den Datentyp Long, Float, Float32 oder Float64 haben.
StartIndex	Nummer (≥1) des ersten Elements im Quellfeld, das übertragen wird.
Count	Anzahl (≥1) der zu übertragenden Daten.
Rückgabewert	Zeilenvektor mit den übertragenen Werten vom Datentyp double.

Bemerkungen

Bis T11 gilt: Bitte beachten Sie, dass Fließkomma-Werte im *ADwin*-System einfache Genauigkeit (32 Bit) haben. Sie sollten daher Daten aus dem zurückgegebenen Zeilenvektor nur mit einfacher Genauigkeit anzeigen lassen, um Missverständnisse bezüglich der Genauigkeit zu vermeiden.

Auch wenn ein Feld in *ADbasic* 2-dimensional angelegt ist, gibt die Anweisung einen Zeilenvektor zurück. Falls gewünscht, kann der Vektor in MATLAB in eine Matrix umformatiert werden, z. B. mit reshape.

Näheres zum Umgang mit 2-dimensionalen Feldern finden Sie in Kapitel 4.4 auf Seite 11.

Die Funktion GetData_Double ersetzt die Funktion Get_Data, die bei früheren Treiberversionen verwendte wurde.

Beispiel

1000 Werte aus DATA_1 ab Element 100 in den Zeilenvektor x lesen:

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

Data2File

`Data2File` speichert Daten vom Typ `Long`, `Float`/`Float32` oder `Float64` aus einem `DATA`-Feld des *ADwin*-Systems in einer Datei (auf der Festplatte).

`Data2File` (`Filename`, `DataNo`, `Startindex`, `Count`, `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 ersten Elements im Quellfeld, das übertragen wird.
`Count`	Anzahl (≥1) der zu übertragenden Daten
`Mode`	Schreibmodus: 0: Datei wird überschrieben (falls vorhanden) 1: Daten werden an eine vorhandene Datei angehängt
Rückgabewert	0: OK ≠0: Fehler

Bemerkungen

Das `DATA`-Feld darf nicht als `FIFO` definiert sein.

Die Daten werden binär gespeichert und zwar entsprechend dem Datentyp, mit dem das `DATA`-Feld auf dem ADwin-System angelegt ist (siehe Tabelle). Wenn die Datei nicht vorhanden ist, wird sie neu angelegt.

Datentyp des `DATA`-Felds	gespeicherter Datentyp
Long	`int32`
Float (bis Prozessor T11) Float32 (Prozessor T12/T12.1)	`single`
Float64 (Prozessor T12/T12.1)	`double`

Beispiel

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

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

File2Data

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

```
File2Data (Filename, DataType, DataNo, 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 String-Konstanten: 'type_integer': Werte vom Typ int32 (32 Bit). 'type_single': Werte vom Typ single (32 Bit). 'type_double': Werte vom Typ double (64 Bit).
DataNo	Nummer (1...200) des Zielfelds DATA_1 ... DATA_200.
Startindex	Nummer (≥1) des ersten Elements im Zielfeld, das beschrieben wird.
Rückgabewert	0: OK ≠0: Fehler

Bemerkungen

Alle Werte in der Datei Filename müssen binär in einem der Formate int32, single oder double vorliegen.

Das DATA-Feld darf nicht als FIFO definiert 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 die Zielformate Long, Float/Float32 und Float64.

Gespeicherter Datentyp	Datentyp des DATA-Felds
int32	Long
single	Float (bis Prozessor T11) Float32 (Prozessor T12/T12.1)
double	Float64 (Prozessor T12/T12.1)

Beispiel

In *ADbasic* sei DATA_1 dimensioniert als:

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

In Matlab:
Werte vom Typ integer 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.

```
ret_val = File2Data('Test.dat', 'type_integer', 1, 20);
```

5.4.2 FIFO-Felder

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

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

Ein FIFO-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: $-1{,}7976931348623150 \cdot 10^{+308}$ … $-2{,}2250738585072014 \cdot 10^{-308}$ positiv: $+2{,}2250738585072014 \cdot 10^{-308}$ … $+1{,}7976931348623150 \cdot 10^{+308}$

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.

Beim Datentyp LONG übertragen die Befehle ganzzahlige Werte mit 32 Bit Breite. Rückgabewerte in MATLAB haben den Datentyp double.

Wenn ganzzahlige Werte einen Zählerstand oder ein Bitmuster enthalten, muss der Wert in MATLAB mit dem Datentyp int32 oder uint32 verarbeitet werden; Näheres siehe Abschnitt Datentypen auf Seite 6.

Fifo_Empty

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

Fifo_Empty (FifoNo)

Parameter

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

Beispiel

In *ADbasic* sei DATA_5 dimensioniert als:

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

In MATLAB erhalten Sie die Anzahl der freien Elemente in DATA_5:

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

Fifo_Full

`Fifo_Full` liefert die Anzahl der belegten Elemente eines FIFO-Felds.

```
Fifo_Full (FifoNo)
```

Parameter

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

Beispiel

In *ADbasic* sei `DATA_12` dimensioniert als:

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

In MATLAB erhalten Sie die Anzahl der belegten Elemente in `DATA_12`:

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

Fifo_Clear

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

```
Fifo_Clear (FifoNo)
```

Parameter

`FifoNo`	Nummer (1...200) des Fifo-Felds `DATA_1` … `DATA_200`.
Rückgabewert	≠255: OK 255: Fehler

Bemerkungen

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

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
Fifo_Clear(45);
```

SetFifo_Double

SetFifo_Double überträgt Daten aus einem Zeilenvektor in ein FIFO-Feld.

```
SetFifo_Double (FifoNo, Vector)
```

Parameter

FifoNo	Nummer (1...200) des Fifo-Felds DATA_1 … DATA_200.
Vector	Zeilenvektor mit den zu übertragenden Werten.
Rückgabewert	≠255: OK 255: Fehler

Bemerkungen

Prüfen Sie vor der Übertragung mit der Funktion Fifo_Empty, ob das FIFO-Feld genügend freie Elemente enthält, um alle Daten des Zeilenvektors aufzunehmen. Wenn Sie mehr Daten in ein FIFO-Feld übertragen als dort freie Elemente vorhanden sind, werden überzählige Daten überschrieben und sind unwiderruflich verloren.

Bis T11 gilt: Bitte beachten Sie, dass Fließkomma-Werte im *ADwin*-System einfache Genauigkeit (32 Bit) haben. Sie sollten daher Daten aus Vector nur mit einfacher Genauigkeit anzeigen lassen, um Missverständnisse bezüglich der Genauigkeit zu vermeiden.

Die Funktion SetFifo_Double ersetzt die Funktion Set_Fifo, die bei früheren Treiberversionen verwendte wurde.

Beispiel

FIFO-Zielfeld DATA_12 auf genügend freie Elemente prüfen und alle Elemente des Zeilenvektors vector in das Zielfeld übertragen:

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

GetFifo_Double

GetFifo_Double überträgt FIFO-Daten vom *ADwin*-System in einen MATLAB-Vektor.

```
GetFifo_Double (FifoNo, Count)
```

Parameter

FifoNo	Nummer (1...200) des Fifo-Felds DATA_1 … DATA_200.
Count	Anzahl (≥1) der zu übertragenden Elemente.
Rückgabewert	Zeilenvektor mit übertragenen Werten

Bemerkungen

Prüfen Sie vor der Übertragung mit der Funktion Fifo_Full, ob und wieviele belegte Elemente das FIFO-Feld enthält. Wenn Sie mehr Daten aus einem FIFO-Feld auslesen als dort belegte Elemente vorhanden sind, erhalten Sie fehlerhafte Daten.

Bis T11 gilt: Bitte beachten Sie, dass Fließkomma-Werte im *ADwin*-System einfache Genauigkeit (32 Bit) haben. Sie sollten daher Daten aus dem zurückgegebenen Zeilenvektor nur mit einfacher Genauigkeit anzeigen lassen, um Missverständnisse bezüglich der Genauigkeit zu vermeiden.

Die Funktion GetFifo_Double ersetzt die Funktion Get_Fifo, die bei früheren Treiberversionen verwendte wurde.

Beispiel

Anzahl der belegten Elemente im FIFO-Quellfeld DATA_12 abfragen und 200 Werte in den Zeilenvektor v übertragen:

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

5.4.3 Datenfelder mit String-Daten

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

Die Funktionen in diesem Abschnitt sind in Verbindung mit *ADsim* T11 nicht verwendbar.

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

Ein Feldelement in einem DATA-Feld vom Typ STRING kann ein Zeichen mit der ASCII-Nummer 0…127 enthalten. Die Endekennung (ASCII-Nummer 0) markiert das Ende einer Zeichenkette in einem DATA-Feld .

String_Length

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

```
String_Length (DataNo)
```

Parameter

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

Bemerkungen

Die Funktion ist in Verbindung mit *ADsim* T11 nicht verwendbar.

String_Length zählt die Zeichen im DATA-Feld bis zum ersten Auftreten der Endekennung (ASCII-Nummer 0). Die Endekennung selbst wird nicht als Zeichen gezählt.

Beispiel

In *ADbasic* ist DATA_2 dimensioniert als:

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

In MATLAB bestimmt man die Länge des Felds DATA_2:

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

SetData_String

SetData_String überträgt einen String in ein DATA-Feld.

```
SetData_String (DataNo, String)
```

Parameter

DataNo	Nummer (1...200) des Fifo-Felds DATA_1 … DATA_200.
String	String-Variable oder zu übertragende Zeichenkette in Hochkommata.
Rückgabewert	≠-1: OK -1: Fehler

Bemerkungen

Die Funktion ist in Verbindung mit *ADsim* T11 nicht verwendbar.

SetData_String hängt jedem übertragenen String als letztes Zeichen die Endekennung (ASCII-Nummer 0) an.

Beispiel

```
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 in eine String-Variable.

GetData_String (DataNo, MaxCount)

Parameter

DataNo	Nummer (1...200) des Felds DATA_1 ... DATA_200.
MaxCount	Max. Anzahl (≥1) der übertragenen Zeichen ohne Endekennung.
Rückgabewert	String-Variable mit den übertragenen Zeichen.

Bemerkungen

Die Funktion ist in Verbindung mit *ADsim* T11 nicht verwendbar.

Wenn der String im DATA-Feld eine Endekennung enthält, stoppt die Übertragung genau dort, d.h. die Endekennung wird nicht übertragen.

Wenn MaxCount größer ist als die in *ADbasic* definierte Zeichenzahl des Strings, erhälten 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

String mit max. 100 Zeichen aus DATA_2 holen:

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

Enthält das DATA-Feld im *ADwin*-System z.B. an Position 9 eine Endekennung, so werden 8 Zeichen gelesen.

5.5 Fehlerbehandlung

Show_Errors

Show_Errors aktiviert oder deaktiviert die Ausgabe von Fehlermeldungen in Meldungsfenstern.

```
Show_Errors (OnOff)
```

Parameter

OnOff	0: Keine Fehlermeldungen ausgeben. 1: Fehlermeldungen in einer Messagebox ausgeben (Default).

Bemerkungen

Die Funktion Show_Errors bezieht sich auf alle Funktionen, die eine Fehlermeldung in einem Meldungsfenster ausgeben können. Dies sind:

- Boot
- Test_Version
- Load_Process

Wenn Meldungsfenster mit Show_Errors deaktiviert sind, läuft das Programm beim Auftreten eines Fehlers ohne Unterbrechung weiter. Der Benutzer kann und muss Fehlermeldungen nicht mehr quittieren.

Beispiel

```
% Fehlermeldungen anzeigen
Show_Errors(1);
```

Get_Last_Error

Get_Last_Error gibt die Nummer des zuletzt in der Schnittstelle adwin32.dll / adwin64.dllaufgetretenen Fehlers zurück.

```
Get_Last_Error ()
```

Parameter

Rückgabewert	0: kein Fehler ≠0: Nummer des Fehlers

Bemerkungen

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

Nach dem Funktionsaufruf wird die Fehlernummer automatisch auf 0 zurückgesetzt.

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

Beispiel

```
% Die letzte Fehlernummer lesen
Error = Get_Last_Error();
```

Get_Last_Error_Text

Get_Last_Error_Text gibt einen Fehlertext zu einer vorhandenen Fehlernummer zurück.

Get_Last_Error_Text (Last_Error)

Parameter

Last_Error Fehlernummer

Rückgabewert Fehlertext

Bemerkungen

Als Fehlernummer Last_Error wird üblicherweise der Rückgabewert der Funktion Get_Last_Error verwendet.

Beispiel

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

Set_Language

Set_Language stellt die Sprache für Fehlermeldungen ein

Set_Language (language)

Parameter

language Sprache für Fehlermeldungen:
0 : in Windows eingestellte Sprache.
1 : Englisch
2 : Deutsch

Rückgabewert 0

Bemerkungen

Der Befehl ändert die Spracheinstellung für Fehlermeldungen der Schnittstelle adwin32.dll / adwin64.dll und für die Funktion Get_Last_Error_Text.

Ist unter Windows eine andere Sprache als Englisch oder Deutsch eingestellt, werden Fehlermeldungen in Englisch ausgegeben.

Beispiel

```
% englische Sprache für Fehlermeldungen einstellen
Set_Language(1);
```

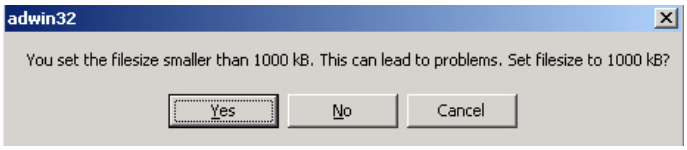
ADwin_Debug_Mode_On

ADwin_Debug_Mode_On schaltet den Debug-Modus ein. Im Debug-Modus wird ein Protokoll aller Funktionsaufrufe in Dateien gespeichert.

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

Parameter

Filename	Pfad und Name der Datei, in die das Protokoll geschrieben wird. Geben Sie den Dateinamen *ohne* Endung und mit dem absoluten Pfadnamen an!
Size	max. Dateigröße in kByte (1000 = 1MiB). Bei Angabe eines Werts kleiner 1000 wird eine Warnung ausgegeben:



Rückgabewert
0 : OK
-1: Dateiname > 255 Zeichen (kann nicht verarbeitet werden)
-2: Debug Mode bereits eingeschaltet (ohne Auswirkung)
-3: Zugriff auf Registry erforderlich, aber nicht möglich. Wahrscheinlich fehlen die für den Zugriff nötigen Benutzerrechte; ggf. wurde auch die max. Größe der Registry überschritten. Bitte wenden Sie sich an Ihren Administrator.

Bemerkungen

Wir empfehlen, diese Funktion *nicht* in Ihrer Anwendung zu benutzen. Verwenden Sie stattdessen das Programm <C:\ADwin\Tools\Test\DebugMode.exe>, das die gleiche Funktion hat.

Bei eingeschaltetem Debug-Modus werden alle Funktionsaufrufe von allen und an alle *ADwin*-Systeme aufgezeichnet. Das Protokoll kann für eine eventuelle Fehlerauswertung hilfreich sein (wenden Sie sich hierfür an den Support der Jäger Computergesteuerte Messtechnik GmbH).

Überschreitet das Protokoll die Dateigröße Size, werden zusätzliche Dateien der gleichen Größe erzeugt. Zur Unterscheidung wird die Dateiendung automatisch als fortlaufende Zahl erzeugt (001…nnn).

Beachten Sie auf jeden Fall:

- Geben Sie als Dateigröße immer mindestens 1.000 kByte an.
- Schalten Sie den Debug-Mode mit ADwin_Debug_Mode_Off aus, wenn Sie ihn nicht mehr benötigen.

Anderenfalls werden sehr viele Protokolldateien erzeugt, was die Dateiverwaltung unter Windows erheblich verlangsamt.

Beispiel

```
ADwin_Debug_On('C:\temp\log', 1000);
```

Im Verzeichnis <C:\temp> werden Protokolldateien mit dem Namen <log.nnn> angelegt:

ADwin_Debug_Mode_Off

`ADwin_Debug_Mode_Off` schaltet den Debug Modus aus.

```
ADwin_Debug_Mode_Off ()
```

Beispiel

```
% Debug-Modus ausschalten
ADwin_Debug_Mode_Off();
```

Anhang

A.1 Beispielprogramme

Die folgenden Beispiele sind für ein ***ADwin-Gold***-System geschrieben, das mit der Gerätenummer (Device No.) `1` angesprochen wird. Sie finden die entsprechenden Quelltexte (und die zugehörigen Binärdateien für ***ADbasic***) in den folgenden Verzeichnissen:

– ***ADbasic***: `C:\ADwin\ADbasic\Samples_ADwin`
– MATLAB®: `C:\ADwin\Developer\Matlab\Samples`

Wir gehen davon aus, dass Sie den Prozess aus ***ADbasic*** (Quelltext) in Ihr ***ADwin***-System übertragen. Sie können alternativ auch aus MATLAB® die Binärdatei mit dem Befehl `Load_Process` ins System übertragen.

Wenn Sie ein anderes ***ADwin***-System verwenden, müssen Sie im jeweiligen ***ADbasic***-Programm den Befehl `ADC` anpassen. Wenn Sie eine andere Device No. als `1` verwenden, müssen Sie diese in MATLAB® mit dem Befehl `Set_DeviceNo` einstellen.

BAS_DMO1

Online Evaluation of Measurement Data

Das ***ADbasic***-Programm schreibt den minimalen und den maximalen Messwert vom analogen Eingangskanal 1 in die globalen Variablen `Par_1` und `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
```

Aus MATLAB® können die Werte mit der Funktion `Get_Par` gelesen werden:

```
% 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)
```

Digitaler P-Regler

BAS_DMO2

Das ***ADbasic***-Programm ist ein digitaler P-Regler, das den Sollwert in `PAR_1` und die Regelverstärkung in `PAR_2` ablegt.

```
rem Das Programm BAS_DMO2 ist ein digitaler
rem P-Regler. Der Sollwert wird extern mit PAR_1
rem vorgegeben, die Verstärkung mit PAR_2.

DIM abweichung, stell AS LONG

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

Von MATLAB® aus können Sollwert und Regelverstärkung mit den folgenden Anweisungen geändert werden:

```
Set_Par(1, 17) ; % Sollwert auf den Wert 17 setzen.
Set_Par(2, 3) ; % Verstärkung auf Wert 3 setzen.
```

Beispiel zur Datenübertragung

BAS_DMO3

Das ***ADbasic*** Programm erfasst Daten vom Analogeingang 1 und schreibt sie in das Feld `DATA_1`.

```
rem Das Programm BAS_DMO3 misst den analogen Eingang 1
rem und schreibt 1000 Daten in ein DATA-Feld.
rem Die Daten werden mit Hilfe des DATA-Felds übertragen.

DIM DATA_1[1000] AS LONG
DIM index AS LONG

INIT:
 Par_10 = 0
 index = 0              'Zählvariable initialisieren
 PROCESSDELAY = 40000   'Zykluszeit = 1ms (ADSP)

EVENT:
 index = index + 1      'Zählvariable erhöhen
 IF (index > 1000) THEN '1000 Messungen fertig?
  Par_10 = 1            'Ende-Flag setzen
  END                   'Prozess beenden
 ENDIF
 DATA_1[index] = ADC(1)'Messung durchführen und speichern
```

In MATLAB werden die Daten aus `DATA_1` gelesen und als Kurve dargestellt.

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

A.2 Liste der 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

A.4 Zuordnung alte Befehlsnummern zu Funktionen

In früheren Versionen des ***ADwin***-Treibers für MATLAB® wurden Befehlsnummern anstelle der Funktionsnamen verwendet. Die folgende Liste ordnet den alten Befehlsnummern die aktuellen Funktionsnamen zu, die die gleiche oder eine ähnliche Aufgabe erfüllen.

Beachten Sie bitte, dass die Parameter und die Parameter-Reihenfolge beim Aufruf mit den alten Befehlsnummern teilweise unterschiedlich zum Aufruf mit Funktionsnamen.

Nr.	Bedeutung	Aufruf	Funktionsname
38	globale Variable oder Status-Parameter lesen	ADlab(38, Index)	Get_Par (Index) Get_FPar (Index) Process_Status (ProcessNo) Get_Processdelay (ProcessNo)
34	globale Variable oder Status-Parameter setzen	ADlab(34, Index, Value)	Set_Par (Index, Value) Set_FPar (Index, Value) Set_Processdelay (ProcessNo, Processdelay)
50	Prozess starten	ADlab(50, ProcessNo)	Start_Process (ProcessNo)
32	Prozess starten	ADlab(32, ProcessNo)	Start_Process (ProcessNo)
51	Prozess stoppen	ADlab(51, ProcessNo)	Stop_Process (ProcessNo)
33	Prozess stoppen	ADlab(33, ProcessNo)	Stop_Process (ProcessNo)
104	Data-Feld in einen Vektor übernehmen	ADlab(104, DataNo, Count)	GetData_Double (DataNo, Startindex, Count)
105	Vektor in ein Data-Feld übertragen	ADlab(105, DataNo, Vector)	SetData_Double (DataNo, Vector, Startindex)
106	Teile eines Data-Felds in einen Vektor übernehmen	ADlab(106, DataNo, Count, Startindex)	GetData_Double (DataNo, Startindex, Count)
107	Vektor ab Startindex in ein Data-Feld übertragen	ADlab(107, DataNo, Vector, Startindex)	SetData_Double (DataNo, Vector, Startindex)
110	FIFO-Feld in einen Vektor übernehmen	ADlab(110, DataNo, Count)	GetFifo_Double (FifoNo, Count)
111	Vektor in ein FIFO-Feld übertragen	ADlab(111, FifoNo, Vector)	SetFifo_Double (FifoNo, Vector)
112	Anzahl der belegten Elemente eines FIFO-Felds ermitteln	ADlab(112, FifoNo)	Fifo_Full (FifoNo)
114	Inhalt eines FIFO-Felds löschen	ADlab(114, FifoNo)	Fifo_Clear (FifoNo)
113	Anzahl der freien Elemente eines FIFO-Felds ermitteln	ADlab(113, FifoNo)	Fifo_Empty (FifoNo)
120	Teil eines Data-Felds in eine Datei speichern	ADlab(120, Filename, DataNo, Count, Startindex)	Data2File (Filename, DataNo, Startindex, Count, Mode)
121	Teil eines Data-Felds an eine Datei anhängen	ADlab(121, Filename, DataNo, Count, Startindex)	Data2File (Filename, DataNo, Startindex, Count, Mode)
138	Activate-PC Flag lesen	ADlab(138, ProzessNr)	Für das Flag Activate-PC gibt es keine Funktion; als Ersatz sollte *PAR_10* verwendet werden.
200	DeviceNo umstellen	ADlab(200, DeviceNo)	Get_DeviceNo ()
253	Freien Speicher des ADwin-Systems abfragen	x = ADlab(253)	Free_Mem (Mem_Spec)
254	Auslastung des ADwin-Systems abfragen	x = ADlab(254)	Workload (Priority)
300	Betriebssystem laden (booten)	ADlab(300, Filename, Memsize)	Boot (Filename, Memsize)

Nr.	Bedeutung	Aufruf	Funktionsname
310	Prozess laden	ADlab(310, Filename)	Load_Process (Filename)
350	Alle 80 globalen Integer-Variablen lesen	ADlab(350)	Get_Par_All ()
351	Mehrere globale Integer-Variablen lesen	ADlab(351, Startindex, Anzahl)	Get_Par_Block (StartIndex, Count)
352	Alle 80 globalen Float-Variablen lesen	ADlab(352)	Get_FPar_All ()
353	Mehrere globalen Float-Variablen lesen	ADlab(353, Startindex, Count)	Get_FPar_Block (StartIndex, Count)
400	Letzte Fehlernummer lesen	ADlab(400)	Get_Last_Error ()