

# ***ADbasic***

**Echtzeit-Entwicklungstool für  
*ADwin*-Systeme**

***ADbasic* Version 4.20**

**April 2006**

**License Key:**

*ADwin* - die schnellsten Echtzeitsysteme unter Windows



## Inhaltsverzeichnis

Inhaltsverzeichnis .....	III
Vorwort .....	1
Konventionen .....	2
1 Einführung .....	3
2 Entwicklungsumgebung .....	5
2.1 Grundlegende Schritte .....	5
2.1.1 Entwicklungsumgebung starten .....	5
2.1.2 ADwin-Betriebssystem laden .....	5
2.1.3 Grundelemente der Entwicklungsumgebung .....	6
2.2 Quelltexte und Projekte erstellen .....	9
2.2.1 Strukturierte Befehlsdarstellung .....	9
2.2.2 Kontextmenü im Quelltextfenster .....	10
2.2.3 Projekte verwalten .....	11
2.3 Menüs und Dialogfenster .....	12
2.3.1 Das Menü „File“ .....	13
2.3.2 Das Menü „Edit“ .....	14
2.3.3 Das Menü „View“ .....	15
2.3.4 Das Menü „Build“ .....	15
2.3.5 Das Menü „Options“ .....	17
Dialogfenster „Compiler Options“ .....	17
Dialogfenster „Process Options“ .....	20
Dialogfenster „Settings“ .....	22
2.3.6 Das Menü „Debug“ .....	26
Option „Enable Timing Analyser“ .....	26
Menüeintrag „Show timing information“ .....	26
Menüeintrag „Trace Setup ...“ .....	30
Menüeintrag „Show Trace“ .....	30
Option „Debug mode“ .....	31
Menüeintrag „Show Debug Window“ .....	33
2.3.7 Das Menü „Tools“ .....	34

2.3.8 Das Menü „Window“	34
2.3.9 Das Menü „Help“	35
2.3.10 Das Projektfenster	35
2.3.11 Das Parameterfenster	36
2.3.12 Das Prozessfenster	38
2.3.13 Das Infofenster	39
2.3.14 Die Statusleiste	40
2.4 ADtools	40
3 Prozesse programmieren	43
3.1 Programmaufbau	43
3.1.1 Die 4 Programmabschnitte	45
3.1.2 Weitere Teile eines Programms	45
3.2 Variablen und Felder	47
3.2.1 Übersicht	47
3.2.2 Datenstrukturen	47
3.2.3 Datentypen	48
3.2.4 Zahlenwerte eingeben	49
3.2.5 Globale Variablen (Parameter)	50
3.2.6 Globale Felder (Arrays)	51
3.2.7 System-Variablen	53
3.2.8 Lokale Variablen und Felder	53
3.3 Variablen und Felder – Details	54
3.3.1 Variablen und Felder im Datenspeicher	54
3.3.2 Speicherbereiche	55
3.3.3 2-dimensionale Felder	57
3.3.4 Die Datenstruktur FIFO	58
3.3.5 Strings	61
Normale Zuweisung	62
Zuweisung per Escape-Sequenz	63
Nicht empfohlene Arten der Zuweisung	64
3.4 Berechnungsausdrücke	64
3.4.1 Auswertung von Operatoren	64
3.4.2 Typkonvertierung	65
3.5 Abfragen, Schleifen und Module	68
3.5.1 Unterprogramm- und Funktions-Makros	68
3.5.2 Include-Dateien	69

3.5.3 Bibliotheken (Libraries) . . . . .	69
4 Prozesse optimieren . . . . .	71
4.1 Bearbeitungszeit messen. . . . .	71
4.2 Verschiedene Tipps . . . . .	72
4.2.1 Zugriff auf Hardware-Adressen . . . . .	72
4.2.2 Konstanten anstelle von Variablen . . . . .	73
4.2.3 Schnellere Messfunktion . . . . .	73
4.2.4 Wartezeit genau einstellen . . . . .	74
4.2.5 Wartezeiten nutzen . . . . .	76
4.2.6 Optimierung mit dem Prozessor T11 . . . . .	77
4.3 Debugging und Analyse. . . . .	78
4.3.1 Laufzeitfehler erkennen (Debug-Modus) . . . . .	78
4.3.2 Zeitverhalten prüfen (Timing-Modus) . . . . .	78
Anzahl und Priorität von Prozessen prüfen. . . . .	79
Optimales Zeitverhalten eines Prozesses. . . . .	80
4.3.3 Programmablauf verfolgen (Trace-Modus) . . . . .	81
5 Prozesse im Betriebssystem . . . . .	85
5.1 Prozessverwaltung. . . . .	86
5.1.1 Prozessarten . . . . .	86
5.1.2 Prozesse mit der Priorität „Hoch“ . . . . .	87
5.1.3 Prozesse mit der Priorität „Niedrig“ . . . . .	87
5.1.4 Kommunikationsprozess . . . . .	88
5.2 Zeitverhalten von Prozessen . . . . .	88
5.2.1 Processdelay . . . . .	88
5.2.2 Zeitlich exakter Aufruf von Prozesszyklen . . . . .	90
5.2.3 Niederpriore Prozesse beim T11 . . . . .	91
5.2.4 Auslastung des ADwin-Systems . . . . .	92
5.2.5 Verschiedene Betriebszustände im Betriebssystem . . . . .	92
5.3 Kommunikation . . . . .	94
5.3.1 Datenaustausch zwischen Prozessen . . . . .	94
5.3.2 Kommunikation zwischen PC und ADwin-System . . . . .	94
5.3.3 Die Device No. . . . .	95
5.3.4 Kommunikation mit Entwicklungsumgebungen . . . . .	95

6 Befehlsreferenz . . . . .	97
6.1 Befehlssyntax . . . . .	97
6.2 Befehle L16, Gold, Pro . . . . .	98
6.3 <i>ADwin-Gold</i> und <i>ADwin-light-16</i> . . . . .	235
6.4 <i>ADwin-light-16</i> DIO1/2 / <i>ADwin-Gold</i> CO1 . . . . .	265
6.5 <i>ADwin-Gold</i> -CAN . . . . .	327
6.6 <i>ADwin-L16</i> Rev. B . . . . .	369
6.7 FFT-Library . . . . .	377
7 Was tun bei Problemen? . . . . .	395
Anhang . . . . .	A-1
A.1 Tastaturkürzel in <i>ADbasic</i> . . . . .	A-1
A.2 ASCII-Zeichensatz . . . . .	A-2
A.3 Baudraten für den CAN-Bus . . . . .	A-3
A.4 Lizenzvertrag . . . . .	A-8
A.5 Kommandozeilen-Aufruf . . . . .	A-12
A.6 Obsolete Programmteile . . . . .	A-16
A.7 Index . . . . .	A-22
A.8 Befehle für <i>ADwin-Gold</i> -Systeme . . . . .	A-37
A.9 Befehle für <i>ADwin-light-16</i> -Systeme . . . . .	A-40
A.10 Befehle für <i>ADwin-Pro</i> -Systeme . . . . .	A-43
A.11 Befehle in diesem Handbuch . . . . .	A-45

## Liebe Leserin, lieber Leser!

*ADbasic 4* ist das Werkzeug, mit dem Sie Ihr *ADwin*-System für Ihre spezielle Mess-, Regel- oder Steuer-Aufgabe programmieren. Dieses Handbuch führt Sie einerseits in die Grundlagen der Programmierung von Echtzeit-Prozessen ein, andererseits soll es Ihnen als Nachschlagewerk dienen.

Dies sind die Neuigkeiten in *ADbasic 4*:

Die Entwicklungsumgebung bietet mehr Komfort durch neue, übersichtlichere Fenster, die neue Projektverwaltung von Quelltexten und eine Online-Hilfe. Debug-Funktionen erlauben Ihnen eine einfachere Fehlersuche.

Vor allem aber unterstützt der Compiler nun die Prozessoren T10 und T11.

Auch das Handbuch hat neue Inhalte: Die Befehlsreferenz beinhaltet zusätzlich auch die Befehle der Systeme *ADwin-Gold* und *ADwin-light-16*, die in speziellen Include-Dateien zur Verfügung stehen.

Für Anwender von *ADwin-Pro*-Systemen sind nur die Befehle in Kapitel 6.2 von Bedeutung. Alle weiteren Befehle sind in „*ADwin-Pro*: Systembeschreibung; Programmierung in *ADbasic*“ dokumentiert.

Wenn Sie *ADbasic* das erste Mal verwenden, empfehlen wir Ihnen den schnellen Einstieg mit Kapitel 1 und 3. Wir setzen voraus, dass Sie bereits über grundlegende Kenntnisse des Programmierens z.B. in Basic verfügen. Eine Einführung ins Programmieren von *ADwin*-Systemen und praktische Tipps finden Sie in unserem Tutorial.

Kapitel 2 beschreibt die neue Entwicklungsumgebung und wird für alle Anwender empfohlen.

Sollten Sie Hinweise haben, wo wir unsere Dokumentation verbessern können, bitten wir um Ihre Rückmeldung. Sie helfen uns damit, Ihnen ein gut verständliches und leicht bedienbares Werkzeug an die Hand zu geben.

Wir wünschen Ihnen viel Erfolg beim Programmieren.

Für Rückfragen wenden Sie sich bitte an unseren Support (Adresse in der vorderen Innenseite des Handbuchs).

## Konventionen

Wir verwenden in diesem Handbuch die folgenden typographischen Konventionen und Zeichen:



Dieses Zeichen (Achtung) steht neben einem Absatz mit wichtigen Informationen für eine korrekte Funktion und fehlerfreien Betrieb.



Bei einem „Hinweis“ finden Sie Tipps und Ratschläge für einen effizienten Betrieb.



Das Informations-Zeichen verweist auf weiterführende Informationen im Handbuch oder in anderen Quellen (Dokumentation, Datenblätter, Literatur etc.).



Ein Beispiel verdeutlicht Ihnen, wie Sie das Gelesene einfach in die Praxis umsetzen können.

Am Schrifttyp „Courier“ erkennen Sie Texte, die auf dem Bildschirm erscheinen, z.B. in Fenstern oder Menüs, oder die Sie mit der Tastatur eingeben. In ähnlicher Weise sind die Namen von Menüs und Untermenüs dargestellt: „Menü ► Untermenü“.

Dateinamen und Pfadnamen sind zusätzlich in spitze Klammern gesetzt: <path\xx.ext>.

Elemente eines Quelltextes wie **BEFEHLE**, Variablen, Kommentar und sonstiger Text werden dargestellt wie im Editor der Entwicklungsumgebung.

Tastenbezeichnungen werden in eckigen Klammern und in Kapitälchen angegeben wie [RETURN] oder [CTRL].

Die Bits eines Datenwortes (hier: 16 Bit) werden wie folgt nummeriert:

Bit-Nr.	15	14	13	...	01	00
Wert des Bits	$2^{15}$	$2^{14}$	$2^{13}$	...	$2^1=2$	$2^0=1$
Bezeichnung	MSB	-	-	-	-	LSB

Wenn Zahlen nicht dezimal angegeben werden, erhalten sie als Anhang einen kennzeichnenden Buchstaben; z. B. für die Zahl 17:

- hexadezimale Schreibweise: 11h
- binäre Schreibweise: 10001b



## 1 Einführung

Das *ADwin*-Echtzeitsystem übernimmt alle zeitkritischen Aufgaben in Ihren schnellen dynamischen Prüfständen und Fertigungsanlagen. Für diese Aufgabe programmieren Sie das *ADwin*-System mit dem Entwicklungssystem *ADbasic*.

Damit Sie schnell und effizient mit der Programmierung beginnen können, möchten wir Ihnen zunächst das Konzept eines *ADwin*-Systems kurz erklären.

Alle *ADwin*-Systeme besitzen als zentrale Einheit einen Prozessor, der alle zeitkritischen Aufgaben wie Messdaten-Erfassung, Regelung, Steuerung, Signalgenerierung oder Online-Verarbeitung von einzelnen Messwerten in Echtzeit ausführt. Analoge und digitale Ein- und Ausgänge sowie Erweiterungen wie Zähler und Bussysteme bilden die Verbindung zu Ihrem Prüfstand; die Kommunikation mit dem PC geschieht über Ethernet oder USB.

Der Prozessor des *ADwin*-Systems wird mit dem Echtzeit-Entwicklungs-Tool *ADbasic* programmiert, das die einfache und schnelle Erstellung von zeitkritischen Echtzeit-Prozessen ermöglicht. *ADbasic* ist eine integrierte Entwicklungsumgebung unter Windows mit Möglichkeiten zum Online-Debugging. Die gewohnte BASIC-Befehlssyntax wurde um Funktionen erweitert, mit denen Sie auf Ein- und Ausgänge zugreifen, Echtzeitprozesse steuern und den Datenaustausch mit dem PC vorbereiten können. In Kapitel 3 ist erklärt, wie Sie *ADbasic*-Programme aufbauen.

Mit nur wenigen Programmzeilen können Sie beispielsweise:



- Messgrößen bis zu Abtastfrequenzen von 800kHz erfassen
- schnelle digitale Regler mit Abtastraten bis zu 400kHz entwickeln
- gleichzeitig analoge Signale erzeugen *und* messen, z. B. für dynamische Kennlinien-Messungen

Wenn Sie für einen komplexen Algorithmus mehrere Prozesse verwenden, regelt eine (einstellbare) Hierarchie das zeitliche Zusammenspiel der Prozesse untereinander. Einzelheiten zum Ablauf Ihrer Prozesse im Betriebssystem finden Sie in Kapitel 5.



Die Entwicklungsumgebung *ADbasic* unterstützt Sie bei der Umsetzung Ihrer Aufgabe in einen Prozess. Zunächst erstellen Sie den Quelltext in einer erweiterten Basic-Syntax; mit den Befehlen und Funktionen können Sie die Hardware Ihres *ADwin*-Systems komfortabel programmieren. In Kapitel 3 ist erklärt, wie Sie Programme aufbauen.



Mit dem integrierten Compiler erzeugen Sie aus dem Quelltext lauffähigen Binärcode, der als Prozess auf das *ADwin*-System übertragen und getestet wird. *ADbasic* bietet Ihnen auch die Hilfsmittel, mit denen Sie Ihre Prozesse beobachten, Fehler suchen und die Programme optimieren können (siehe Kapitel 2).

Sobald die Echtzeit-Prozesse zu Ihrer Zufriedenheit laufen, ist Ihre Arbeit mit *ADbasic* bereits beendet.

Mit einer Bedienoberfläche können Sie die Prozesse und Prozessdaten des *ADwin*-Systems vom PC aus steuern und beobachten, d.h. den fertigen Binärcode zum System übertragen sowie die Prozesse starten, steuern und stoppen.

Obwohl das *ADwin*-System autark arbeitet, können Sie aus Ihrer Bedienoberfläche jederzeit auf globale Variablen und Felder zugreifen, ohne zeitkritische Prozesse zu verzögern. Über die globalen Variablen und Felder können alle Prozesse untereinander oder mit dem PC schnell Daten austauschen.

Die klare Trennung von Echtzeit-Prozessen im *ADwin*-System und Bedienoberfläche im PC garantiert Ihnen höchste Betriebssicherheit und zeitlich nachvollziehbare Abläufe.

Unter Windows ermöglicht Ihnen eine DLL- oder ActiveX-Schnittstelle, auf das *ADwin*-System auch aus mehreren Programmen gleichzeitig zuzugreifen.

Darauf basierend gibt es Treiber für .NET sowie für alle gängigen Entwicklungsumgebungen, mit denen Sie Ihre eigene Bedienoberfläche gestalten können, z.B. Delphi, Visual-Basic, C#.NET, Visual-C++. Alternativ können Sie auch Messtechnik-Pakete wie TestPoint, LabVIEW, Diadem, HP-VEE, Intouch und Matlab nutzen.

Schließlich stehen auch Treiber für die Plattformen Linux und Java zur Verfügung.

## 2 Entwicklungsumgebung

Mit der Entwicklungsumgebung *ADbasic* können Sie einfach und schnell Prozesse für *ADwin*-Systeme programmieren und testen. Der *ADbasic*-Compiler verarbeitet eine erweiterte Basic-Syntax und erzeugt Binärdateien, die Sie auch ohne die Entwicklungsumgebung auf das *ADwin*-System übertragen und ausführen können.

### 2.1 Grundlegende Schritte


#### 2.1.1 Entwicklungsumgebung starten

Sie starten die Entwicklungsumgebung *ADbasic*, indem Sie im Windows-Startmenü „Programms ▶ *ADwin* ▶ *ADbasic* 4“ wählen.

Sie sehen nun die *ADbasic*-Entwicklungsumgebung mit den Windows-typischen Elementen wie Fenster, Menü- und Werkzeug-Leiste.

Stellen Sie nun das von Ihnen verwendete *ADwin*-System und den Prozessor im Menü „Options\Compiler“ ein. Die Entwicklungsumgebung speichert Ihre Angaben, so dass Sie sie bei einem erneuten Start von *ADbasic* nicht mehr eingeben müssen (es sei denn, Sie verwenden eine andere *ADwin*-Hardware).

#### 2.1.2 *ADwin*-Betriebssystem laden

Sie übertragen das *ADwin*-Betriebssystem in Ihr *ADwin*-System, indem Sie auf die Schaltfläche  klicken (= booten).

Diesen Boot-Vorgang benötigen Sie immer dann, wenn Sie Ihr *ADwin*-System (nach einer Unterbrechung der Stromversorgung) wieder einschalten oder wenn der PC einen Kommunikationsfehler erkannt und die Verbindung zum System unterbrochen hat.


Wenn Sie das Betriebssystem übertragen, gehen gleichzeitig die Inhalte des Programm- und Datenspeichers auf dem *ADwin*-System verloren und alle globalen Parameter werden auf den Wert 0 gesetzt.



Für jeden Prozessortyp wird das passende Betriebssystem benötigt, das jeweils in einer eigenen Datei „*ADwin*\*.btl“ vorhanden ist („\*“ steht für den Prozessortyp). Aus Ihrer Einstellung im Menü „Options \ Compiler“ entnimmt die Entwicklungsumgebung die Information, welche dieser Dateien beim Boot-Vorgang zu übertragen ist.

Die Dateien *ADwin*\*.btl werden bei der Installation im Verzeichnis <C:\ADwin> abgelegt (Standardinstallation).

### 2.1.3 Grundelemente der Entwicklungsumgebung

Die Entwicklungsumgebung besteht aus mehreren Leisten und Fenstern (siehe Abb. 1); Sie können die Höhe der Fenster frei anpassen. Kontextsensitive Hilfe zu einem Element der Umgebung (Fenster, Schaltfläche, Menüpunkt) erhalten Sie, wenn Sie erst auf die Schaltfläche  klicken und anschließend auf das gewünschte Element.

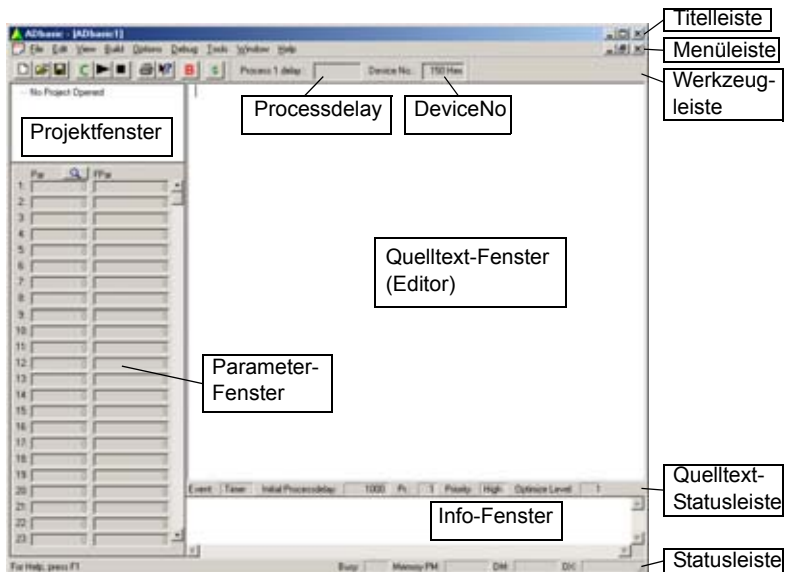


Abb. 1 – Elemente der ADbasic-Entwicklungsumgebung

Sie finden die Befehle der Entwicklungsumgebung über:

- die Werkzeugleiste (siehe Abb. 2)
- die Kontextmenüs der Fenster (rechte Maustaste)
- die Menüleiste. Wenn Sie einen Menübefehl markieren, sehen Sie am linken Rand der Statusleiste eine Befehlserklärung.

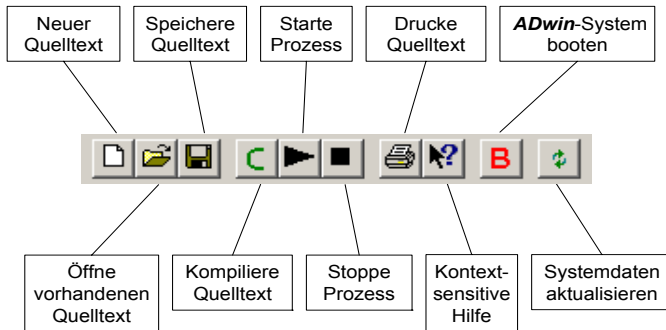



Abb. 2 – Die Werkzeugleiste

Sie wählen ein Menü (und dann einen Befehl) aus, indem Sie das Menüfeld mit der linken Maustaste anklicken, oder indem Sie die Tastenkombination [ALT] + [ANFANGSBUCHSTABE] des entsprechenden Menüs eingeben. Manche Befehle besitzen auch Tastatur-Kürzel (Short-Cuts, siehe Anhang A.1), die in den Menüs angezeigt werden.

Im aktiven Quelltext-Fenster (Editor) bearbeiten Sie den Quelltext für je einen Prozess. Sie können mehrere Fenster parallel geöffnet haben; die Fenstergrößen sind frei einstellbar. Weitere Informationen zum jeweiligen Quelltextfenster werden an verschiedenen Stellen angezeigt:

- Die Titelleiste zeigt den Namen des aktiven Quelltext-Fensters.
- Am unteren Rand zeigt die Quelltext-Statusleiste die von Ihnen eingestellten Prozess-Optionen.  
Ein Rechtsklick auf die Statusleiste öffnet das Dialogfenster „Process Options“.
- Im Parameter-Fenster (s. Kapitel 2.3.11, Seite 36) können Sie durch Drücken der Schaltfläche  (einmalig) markieren lassen, welche globalen Parameter Sie im aktiven Quelltext oder Projekt verwenden.
- Im Info-Fenster werden Fehlermeldungen (rot hinterlegt) und Warnungen des Compilers angezeigt (siehe Kapitel 2.3.13 „Das Infofenster“).

Im Projektfenster werden der Name des offenen Projekts und die zugehörigen Quelltext-Dateien angezeigt, anderenfalls bleibt es leer.

Einige Parameter Ihres *ADwin*-Systems werden kontinuierlich aktualisiert und angezeigt (nur, wenn Ihr PC mit einem System kommuniziert):

- Das Processdelay (Prozess-Zykluszeit) zu der Prozessnummer, die Sie für das aktive Quelltextfenster eingestellt haben, dargestellt am rechten Rand der Werkzeugleiste.
- Die Werte der globalen Variablen im Parameterfenster; wenn Sie einen dieser Werte verändern, wird er sofort zum *ADwin*-System übertragen.
- Informationen zur Speicherauslastung in der Statusleiste (siehe Kapitel 2.3.14).

Informationen laufender Prozesse werden in separaten Fenstern angezeigt:

- Zeitverhalten: Timing-Fenster (Seite 26)
- Laufzeitfehler: Debug-Fenster (Seite 31)
- Programmablauf: Trace-Fenster (Seite 30)

## 2.2 Quelltexte und Projekte erstellen

Öffnen Sie für jeden Quelltext eines Prozesses ein eigenes Fenster (mit „File ▶ New“).

Der Editor und der Compiler unterscheiden nicht zwischen Groß- und Kleinschreibung. In unseren Beispielen verwenden wir allerdings zur besseren Unterscheidung Großbuchstaben für Befehle und globalen Variablen und Kleinbuchstaben für lokale Variablen und Kommentare.

Wenn Sie Hilfe zu einem *ADbasic*-Befehl benötigen: Markieren Sie den Befehl im Quelltext und drücken die Taste [F1], um die Online-Hilfe mit der passenden Information zu öffnen.

Sie können Zahlenwerte nicht nur dezimal, sondern auch in hexadezimaler, binärer und Exponential-Schreibweise eingeben (siehe Kapitel 3.2.4 „Zahlenwerte eingeben“).

### 2.2.1 Strukturierte Befehlsdarstellung

Wenn Sie eine Befehlszeile eingegeben haben, ändert der Editor automatisch die Farbe der eingegebenen Befehlswörter, Variablen- und Feldnamen, und er rückt die Zeilen so ein, dass sich ein übersichtliches Bild ergibt. Beides zusammen soll Ihnen – vor allem in größeren Quelltexten – bei der Suche nach bestimmten Textstellen helfen.

Der Editor unterscheidet die von Ihnen eingegebenen Zeichenketten in folgende Syntax-Kategorien:

- **Standard**: allgemeiner Programmtext
- **Comment**: Kommentar
- **KEYWORD**: *ADbasic*-Befehle
- **EXTERNAL KEYWORD**: Include- und Library-Befehle
- **Identifizier**: Namen von Variablen und Feldern

Sie können die Farbgebung und die Einrückung nach eigener Vorstellung verändern oder ganz abschalten. Hierzu rufen Sie den Menüpunkt „Options ▶ Settings“ auf und wählen im Dialogfenster den Reiter „Syntax Color“ oder den Reiter „Editor“.

### 2.2.2 Kontextmenü im Quelltextfenster

Im Quelltextfenster können Sie verschiedene Hilfsfunktionen über das Kontextmenü erreichen (Klick mit rechter Maustaste, siehe unten).

Cut	Ctrl+X
Copy	Ctrl+C
Paste	Ctrl+V
Comment Block	
Uncomment Block	
Indent	
Outdent	
Mark Controlblock	
Unmark Controlblock	
Add to Project	
Enable Trace	
Disable Trace	

Abb. 3 – Kontextmenü zum Quelltext-Fenster

Für die folgenden Befehle müssen Sie jeweils vorher ein Zeichen, eine Zeile oder mehrere Zeilen markieren:

- „Comment Block“ fügt an jedem Zeilenanfang ein Kommentarzeichen ein, so dass der Compiler diese Zeilen überspringt.
- „Indent“ rückt die Zeilen um einen Tabulator-Schritt nach rechts, „Outdent“ nach links. Einrückungen können den Quelltext übersichtlicher gestalten.
- „Mark Control block“ markiert die vollständige Kontrollstruktur, wenn Sie vorher eines der Kennwörter der Struktur markiert hatten. „Unmark Control block“ löscht die Markierung wieder. Die Markierung dient zur optischen Prüfung verschachtelter Strukturen; sie kann nicht für eine Bearbeitung der Kontrollstruktur genutzt werden.



Die folgenden Kontrollstrukturen werden erkannt:

- **DO ... UNTIL**
  - **FOR ... TO ... {STEP} ... NEXT**
  - **IF ... THEN ... {ELSE} ... ENDIF**
  - **SELECTCASE**
- „Enable Trace“ aktiviert die markierten Zeilen für den Trace-Modus (siehe auch „Programmablauf verfolgen (Trace-Modus)“ auf Seite 81) und kennzeichnet sie mit einem Fragezeichen „?“ am Zeilenanfang.
  - „Disable Trace“ deaktiviert die Zeilen und entfernt das Fragezeichen.

### 2.2.3 Projekte verwalten

Sie können beliebig viele Quelltexte gemeinsam als Projekt verwalten, beispielsweise wenn Sie eine Anwendung mit mehreren Prozessen programmieren. Es kann jeweils nur ein einziges Projekt geöffnet sein.

Wenn Sie ein Projekt verwenden, können Sie:

- Quelltext-Dateien in ein geöffnetes Projekt einbinden und daraus löschen.
- mit dem Projekt gleichzeitig alle eingebundenen Quelltext-Dateien öffnen und die zuletzt genutzten Fenster-Einstellung.
- alle in einem Projekt benutzten globalen Variablen anzeigen (siehe Kapitel 2.3.11 auf Seite 36).
- mit dem Projekt die zuletzt genutzten Fenster-Einstellung speichern.

Einige projektbezogene Befehle können Sie über das Kontextmenü erreichen (mit der rechten Maustaste in das Projektfenster klicken, siehe auch „Das Projektfenster“ auf Seite 35). Alle weiteren Projektbefehle sind über das Menü „File“ zugänglich.

Beachten Sie bitte, dass das Öffnen eines Projekts dazu führt, dass geöffnete Quelltexte automatisch geschlossen werden. Wenn ungesicherte Dateien vorhanden sind, werden Sie zum Speichern dieser Dateien aufgefordert.

## 2.3 Menüs und Dialogfenster

Sie finden in der Menüleiste folgende Funktionen:

- File:       Dateien und Projekte verwalten.       (Seite 13)
- Edit:       Quelltexte editieren.       (Seite 14)
- View:       (Seite 15)
- Build:      Ausführbare Programme erzeugen.       (Seite 15)
- Options:    Optionen aller Art einstellen.       (Seite 17)
- Debug:      Hilfe zur Fehlersuche.       (Seite 26)
- Tools:      Verschiedene Hilfsfunktionen.       (Seite 34)
- Window:     Quelltextfenster anordnen.       (Seite 34)
- Help:       Hilfe, Versions- und Lizenz-Informationen.       (Seite 35)

## 2.3.1 Das Menü „File“

Das Menü „File“ enthält Befehle zum Verwalten von Dateien und Projekten.

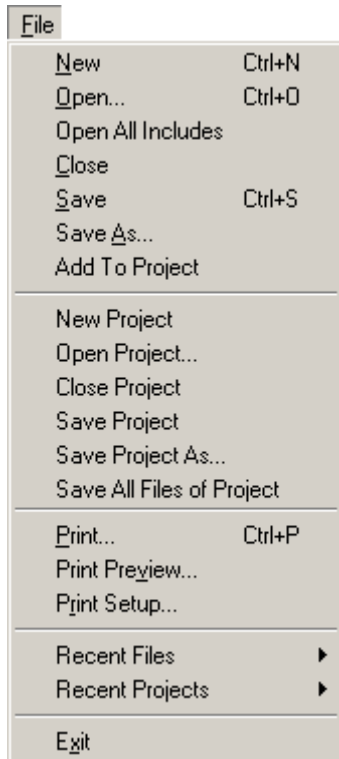
Mit den Befehlen können Sie Dateien öffnen, speichern und neue (Quelltext-Fenster) erzeugen. Sie können beliebig viele Quelltexte bearbeiten, aber höchstens 10 Prozesse gleichzeitig auf Ihr ADwin-System laden.

Der Befehl „Open all Includes“ öffnet alle Dateien, die Sie im aktiven Quelltext mit dem Befehl **#INCLUDE** eingebunden haben.

In gleicher Weise wie Dateien können Sie auch Projekte öffnen, speichern und neue erzeugen; mit dem Unterschied, dass nur ein einziges Projekt gleichzeitig geöffnet sein kann. Weitere Befehle sind über das Projektfenster zugänglich (siehe Kapitel 2.3.10).

Das Menü enthält auch die Druckfunktionen (Drucken, Druckvorschau und Drucker-einrichtung).

Unter „Recent Files“ und „Recent Projects“ wird eine Liste der 10 zuletzt geöffneten Dateien und Projekte angezeigt.



### 2.3.2 Das Menü „Edit“

Das Menü „Edit“ enthält die nach den Windows-Konventionen üblichen Editier-Funktionen.

Darüber hinaus enthält das Menü eine Such-Funktion (Find) sowie eine Ersetzen-Funktion (Replace).

Wir empfehlen Ihnen nicht, mit „Cut and Paste“ Zeichen oder Programmzeilen aus anderen Programmen in einen Quelltext einzufügen. Es kann hierbei zu unvorhergesehenen Fehlfunktionen kommen.

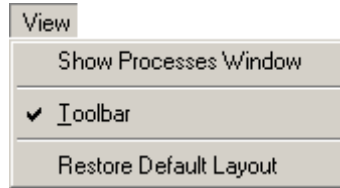
Edit	
<u>U</u> ndo	Ctrl+Z
<u>R</u> edo	Ctrl+Y
<u>C</u> ut	Ctrl+X
<u>C</u> opy	Ctrl+C
<u>P</u> aste	Ctrl+V
S <u>e</u> lect A <u>l</u> l	Ctrl+A
<u>F</u> ind...	Ctrl+F
F <u>i</u> nd Next	F3
R <u>e</u> place...	Ctrl+H

## 2.3.3 Das Menü „View“

Im Menü „View“ können Sie

- das Prozessfenster
  - die Werkzeugleiste (Toolbar)
- ein- und ausschalten.

Näheres zum Prozessfenster finden Sie in , zur Werkzeugleiste siehe Seite 7.

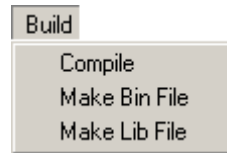


Mit „Restore Default Layout“ stellen Sie mit einem Tastendruck die Standard-Ansicht wieder her, die beim ersten Öffnen des Programms *ADbasic* aktiv war.


## 2.3.4 Das Menü „Build“

Im Menü „Build“ können Sie den aktiven Quelltext übersetzen:

- mit „Compile“ in einen Prozess.
- mit „Make Bin File“ in eine Binärdatei.
- mit „Make Lib File“ in eine Library.

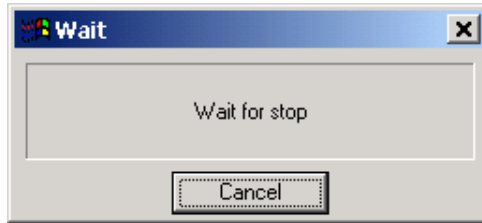


„Compile“ ist der mächtigste Befehl des Menüs: Er übersetzt den aktiven Quelltext, überträgt den erzeugten Binärcode als Prozess auf das *ADwin*-System und startet den Prozess.

Der Prozess wird allerdings nur dann automatisch gestartet, wenn Sie im Menü „Options\Compiler“ den Eintrag „Autostart“ auf „Yes“ gesetzt haben. Anderenfalls können Sie den Prozess mit der Schaltfläche  aus der Werkzeugleiste oder im Prozessfenster starten (siehe Seite 38).

Wenn der Quelltext Fehler oder kritische Sequenzen enthält, wird die erste betreffende Zeile rot markiert.

Beim Kompilieren erscheint manchmal die Meldung „Wait for stop“:



Die Meldung „Wait for stop“ erscheint, wenn ein auf dem *ADwin*-System laufender Prozess gestoppt werden muss, damit der kompilierte Prozess geladen werden kann. Mit CANCEL brechen Sie das Laden des kompilierten Prozesses ab; einen neuen Ladeversuch starten Sie mit „Compile“.

„Make Bin File“ ist nur für lizenzierte *ADbasic*-Benutzer verfügbar. Der Befehl übersetzt den aktiven Quelltext in Binärcode und speichert diesen automatisch in einer Datei ab. Die Binärdatei wird mit dem Namen und im Verzeichnis der Quelltext-Datei abgelegt, jedoch mit der Dateiendung „.Txn“. Hierbei steht „x“ für den Prozessortyp und „n“ für die Prozessnummer (siehe Das Menü „Options“, Dialogfenster „Process Options“).

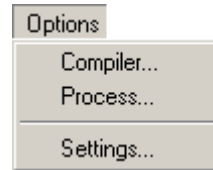


Eine Binärdatei mit der Endung <\*.TA3> können Sie beispielsweise als Prozess auf ein *ADwin*-System mit dem Prozessor T10 übertragen, der den Prozess unter der Nummer 3 lädt. Sie können Binärdateien aus allen Entwicklungsumgebungen (wie C oder Visual Basic) an das *ADwin*-System übertragen (siehe Kapitel 5.3.4 auf Seite 95).

„Make Lib File“ ist ebenfalls nur für lizenzierte *ADbasic*-Benutzer verfügbar. Der Befehl übersetzt den aktiven Quelltext in Binärcode und speichert diesen automatisch als Library-Datei ab. Die Library wird mit dem Namen und im Verzeichnis der Quelltext-Datei abgelegt, jedoch mit der Dateiendung „.Lib“. Hierbei steht „x“ wieder für den Prozessortyp (s.o.). Anschließend können Sie die Library in andere Quelltexte einbinden und auf deren Funktionen und Unterprogramme zugreifen (siehe Kapitel 3.5.3 auf Seite 69).

## 2.3.5 Das Menü „Options“

Im Menü „Options“ können Sie eine Reihe von Optionen einstellen, die unmittelbar wirksam werden. Zu jedem Menüpunkt wird ein eigenes Dialogfenster aufgerufen, in dem Sie Ihre Einstellungen vornehmen.



### Dialogfenster „Compiler Options“

Die Einstellungen, die Sie in diesem Dialogfenster machen (bitte von oben nach unten), werden für jedes Kompilieren eines Quelltextes verwendet. Insbesondere sind dies Informationen über das ADwin-System, auf dem die übersetzten Quelltexte als Prozess laufen sollen.

Wenn Sie Quelltexte für verschiedene ADwin-Systeme kompilieren möchten, müssen Sie für jedes System neu die Einstellungen in diesem Dialogfenster anpassen.

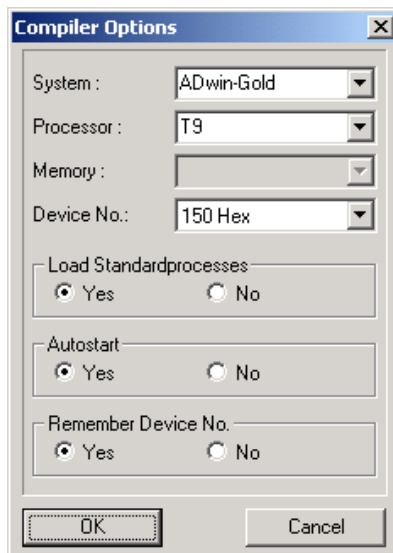


Abb. 4 – Das Dialogfenster „Compiler Options“

- „System“: Wählen Sie den Eintrag aus, der Ihrem *ADwin*-System entspricht.
- „Processor“: Wählen Sie den Eintrag aus, der Ihrem *ADwin*-System entspricht.
- Die Kurzbezeichnungen der Prozessor-Typen entsprechen den folgenden Vollbezeichnungen:

Kurzbezeichnung	T11	T10	T9	T8	T5	T4	T2
Vollbezeichnung	ADSP TS101S	ADSP 21160	ADSP 21062	T805	T450	T400	T225

Abb. 5 – Prozessorbezeichnungen

- „Memory“: Diese Einstellung ist für die Systeme *ADwin-Gold*, *ADwin-light-16* und *ADwin-Pro* ab dem Prozessor T9 nicht sinnvoll und daher abgeblendet.


Für Transputer-Prozessoren (T2...T8) lesen Sie die entsprechenden Informationen bitte im Hardware-Handbuch nach.

- „Device No.“: Wählen Sie die Gerätenummer, mit der das gewünschte *ADwin*-System angesprochen werden kann. Sie vergeben diese Gerätenummer mit dem Programm <ADconfig.exe>. Die Werkseinstellung ist „150 Hex“.

Die Einstellung „NONE“ erlaubt es Ihnen, auch dann Quelltexte für die oben eingestellte *ADwin*-Hardware zu kompilieren, wenn diese nicht mit Ihrem PC verbunden ist.

- „Load standard processes“: Diese Einstellung ist nur für die Systeme *ADwin-Gold*, *ADwin-light-16* verfügbar.

Bei der Einstellung „Yes“ (Standard) werden während des Boot-Vorgangs die Standard-Prozesse 11, 12 und 15 (siehe Kapitel 5.1.1 auf Seite 86) in das *ADwin*-System übertragen. Mit „No“ wird die Übertragung der Prozesse 11 und 12 unterdrückt.

- „Autostart“: Wenn Sie die Einstellung „Yes“ verwenden, wird die beim Kompilieren erzeugte und ins *ADwin*-System übertragene Binärdatei sofort als Prozess gestartet. Mit der Einstellung „No“ müssen Sie den Prozess mit der Schaltfläche  aus der Werkzeugleiste oder im Prozessfenster starten.



- „Remember Device No.“: Die Einstellung „Yes“ speichert die zuletzt verwendete Device No. (siehe oben) beim Schließen von *ADbasic*; beim Neustart wird diese Nummer automatisch eingestellt. Mit der Einstellung „No“ startet *ADbasic* mit der Einstellung „NONE“.

**Dialogfenster „Process Options“**

In diesem Dialogfenster legen Sie Compiler-Optionen für das gerade aktive Quelltextfenster fest, d.h. Sie bestimmen Eigenschaften desjenigen Prozesses, der aus dem aktiven Quelltext übersetzt und ins *ADwin*-System übertragen wird.

Sie müssen für jedes Quelltextfenster separat die nötigen Einstellungen vornehmen, indem Sie das Dialogfenster jeweils neu aufrufen (es sei denn, Sie möchten die Voreinstellung verwenden).

Wenn Sie im Dialogfenster „Compiler Options“ den Prozessortyp T9, T10 oder T11 eingestellt haben, wird das in Abb. 6 gezeigte Dialogfenster geöffnet. Das Dialogfenster für die Prozessortypen T4, T5 oder T8 weicht hiervon in wenigen Punkten ab und ist im Anhang A.6.1 beschrieben.

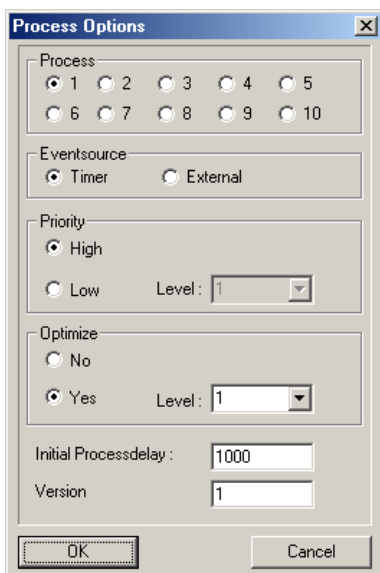


Abb. 6 – Das Dialogfenster „Process Options“

- „Process“: Stellen Sie die Nummer ein, unter der der übertragene Prozess auf dem System angesprochen wird.

Wenn Sie mehrere Prozesse gleichzeitig auf einem *ADwin*-System ablaufen lassen, müssen Sie jedem Prozess eine eigene Nummer zuweisen.

- „Eventsource“: Stellen Sie ein, welches Event-Signal den Abschnitt **EVENT**: Ihres Prozesses starten soll.

Mit der Einstellung „Timer“ definieren Sie Impulse des internen Zählers als Event-Signal. In diesem Fall legen Sie mit der Systemvariablen **PROCESSDELAY** fest, in welchen Zeitabständen der Zähler ein Event-Signal auslöst.

Mit „External“ legen Sie fest, dass ein Signal am Event-Eingang Ihrer ADwin-Hardware den Prozess startet. Dies könnte beispielsweise ein Impuls eines Messaufnehmers sein. Ein solcher Prozess muss mit hoher Priorität ablaufen. Stellen Sie für diesen Fall die Option „Priority“ auf „High“.

Wie Sie bei einem ADwin-Pro-System einen externen Event-Eingang nutzen können, lesen Sie bitte in der zugehörigen Software-Dokumentation unter dem Befehl **EVENTENABLE** nach.

- „Priority“: Stellen Sie hier die Priorität des Prozesses ein, mit er im System laufen soll. Weitere Informationen zu diesem Thema finden Sie in Kapitel 5.1.1 „Prozessarten“.

Der Eintrag „Level“ (-10...+10) definiert die Priorität innerhalb der *niederpriorien* Prozesse, d.h. ein Prozess mit höherem „Level“ kann solche mit niedrigem Level unterbrechen, nicht aber umgekehrt. Eine höhere Zahl steht für höhere Priorität.

- „Optimize“: Die wahlweise einschaltbare Optimierung kann die Prozess-Ausführungszeit (je nach Programmierung) um bis zu 20% verkürzen sowie den benötigten Speicherplatz verringern. Eine höhere Einstellung unter „Level“ führt zu kürzeren Ausführungszeiten.

Wenn Sie unerwartete Compiler- oder Laufzeitfehler eines Prozesses feststellen, können Sie dies in Ausnahmefällen durch die erneute Einstellung eines niedrigeren „Level“ beheben.

- „Initial Processdelay“: Stellen Sie hier das Processdelay (Zykluszeit) ein, mit dem der Prozess beginnen soll.
- „Version“: Hier können Sie einen ganzzahligen Wert eingeben, um verschiedene Versionen Ihres Programms zu unterscheiden.

**Dialogfenster „Settings“**

Dieses Dialogfenster hat mehrere Blätter, die Sie über die Reiter am oberen Rand aktivieren.

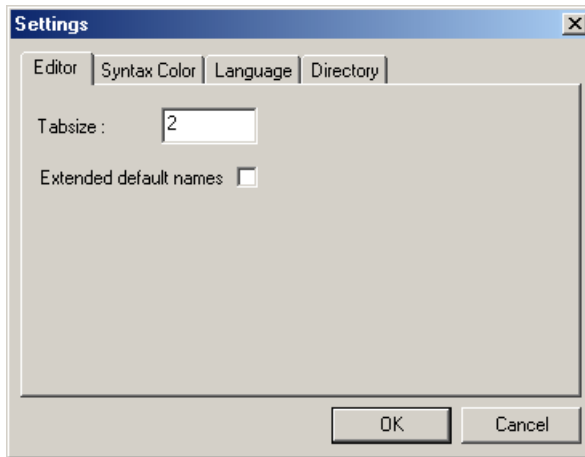
**Blatt „Editor“**

Abb. 7 – Das Dialogfenster „Settings“, Blatt „Editor“

Geben Sie bei „Tabsize“ ein, um wieviele Zeichen ein einzelner Tabulator-sprung eine Zeile einrückt. Die automatische Einrückung wird gemeinsam mit der farbigen Befehlsmarkierung auf dem Blatt „Syntax Color“ ein- und ausgeschaltet.

Wenn Sie die Option „Extended default names“ aktivieren, wird, sobald Sie einen neuen Quelltext erzeugen, diesem automatisch ein Dateiname im Format „ADbJJMMTT\_nn.bas“ zugewiesen. Dabei ist „JJMMTT“ das aktuelle Datum und „nn“ eine 2stellige Zählnummer (nn wird nach einer Änderung des Systemdatums auf Null gesetzt).

Diese Option soll Ihnen helfen, einer Vielzahl von neu erzeugten Quelltexten einen eindeutigen Dateinamen zu geben.

## Blatt „Syntax Color“

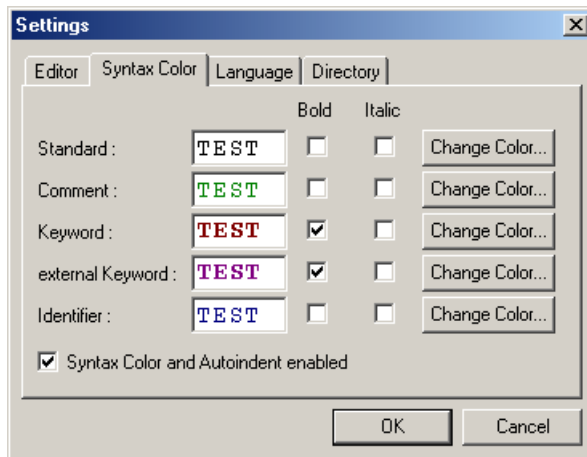


Abb. 8 – Das Dialogfenster „Settings“, Blatt „Syntax Color“

Stellen Sie hier ein, ob die strukturierte Befehlsdarstellung ausgeführt wird und falls ja, wie die Syntax-Kategorien (siehe Kapitel 2.2.1 „Strukturierte Befehlsdarstellung“ auf Seite 9) hervorgehoben werden.

Sie können für jede Kategorie eine beliebige Farbe („Change Color“) und die Schriftschnitte „Bold“ (fett) oder „Italic“ (kursiv) einstellen.

Die strukturierte Befehlsdarstellung „Syntax Color and Autoindent enabled“ ist als Standardeinstellung aktiviert. Die automatische Einrückung verwendet als Tabulatorbreite die Einstellung aus dem Blatt „Editor“.

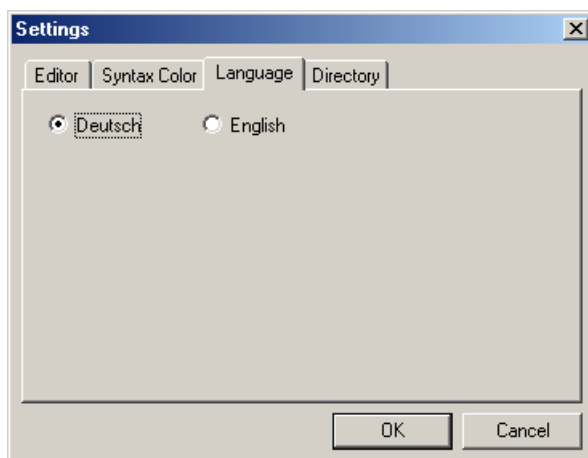
**Blatt „Language“**

Abb. 9 – Das Dialogfenster „Settings“, Blatt „Language“

Hier können Sie wählen, in welcher Landessprache die Fehlermeldungen des Compilers ausgegeben werden. Zur Wahl stehen *Deutsch* oder *English*.

## Blatt „Directory“

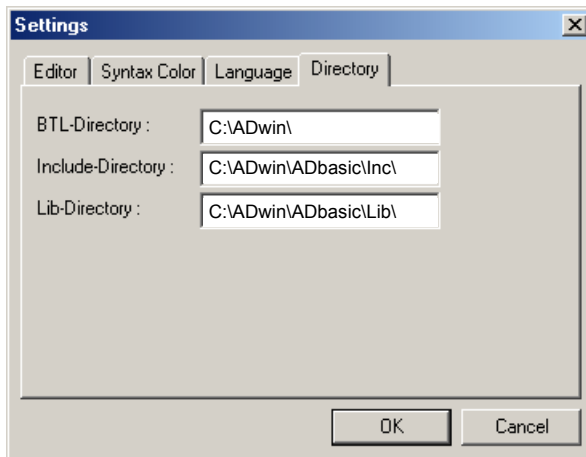



Abb. 10 – Das Dialogfenster „Settings“, Blatt „Directory“

In diesem Dialogfenster legen Sie die Verzeichnisse fest, in denen das Betriebssystem und der Compiler nach Dateien suchen:

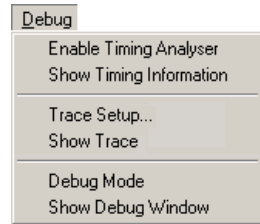
- „BTL-Directory“: Hier sucht die Entwicklungsumgebung nach den Betriebssystem-Dateien `<*.bt1>`, die beim Boot-Vorgang in das ADwin-System übertragen werden (siehe Kapitel 2.1.2).
- „Include-Directory“: In diesem Verzeichnis sucht der Compiler nach Dateien `<*.inc>`, die Sie mit **#INCLUDE** (ohne Pfadangabe) in einen Quelltext einbinden.
- „Lib-Directory“: In diesem Verzeichnis sucht der Compiler nach Library-Dateien `<*.lib>`, die Sie mit **IMPORT** (ohne Pfadangabe) in einen Quelltext einbinden.

Beachten Sie bitte, dass der Pfadname immer mit einem „\“ (Backslash) enden muss. 

Wir empfehlen, die voreingestellten Verzeichnisse nicht zu verändern. Wenn Sie Include- und Library-Dateien aus anderen Verzeichnissen einbinden, können Sie im Einbinde-Befehl den korrekten Pfadnamen angeben.

### 2.3.6 Das Menü „Debug“

Im Menü „Debug“ stellen Sie Optionen ein, die Ihnen beim Auffinden von Laufzeit- oder Symantik-Fehlern helfen. Beachten Sie bitte, dass alle Einstellungen erst nach dem nächsten Kompilieren wirksam werden.



#### Option „Enable Timing Analyser“

Wenn Sie die Option „Enable Timing Analyser“ aktivieren und einen Quelltext kompilieren, werden zusätzliche Informationen über das Zeitverhalten des Prozesses verfügbar (Anzeigen der Informationen: siehe Menüeintrag „Show timing information“). Diese Option benötigt etwa 60 Taktzyklen (bei den Prozessoren T9, T10 und T11) pro Event und Prozess zusätzlich und beeinflusst damit das Zeitverhalten geringfügig. Wir empfehlen daher, die Option nur für das Kompilieren eines oder weniger Prozesse zu aktivieren und sie dann wieder zu deaktivieren. Diese Options-Einstellungen der Prozesse werden beim Verlassen von *ADbasic* nicht gespeichert.

#### Menüeintrag „Show timing information“

Mit dem Menüeintrag „Show timing information“ öffnen Sie das Fenster „Timing Information“ (nur bei aktivierter Option „Enable timing analyser“). Es zeigt 7 Kennwerte für die Prozesse 1...10, die das Zeitverhalten der Prozesse seit dem Zeitpunkt des letzten Starts beschreiben. Nähere Hinweise über die Auswertung dieser Informationen gibt Ihnen Kapitel 4.3.2 „Zeitverhalten prüfen (Timing-Modus)“.

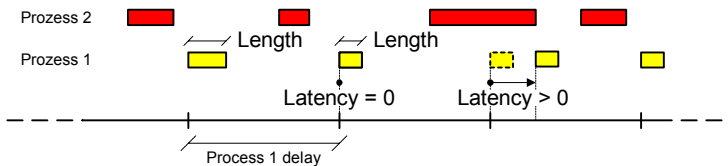
Die Kennwerte sind nur für hoch priorisierte Prozesse verwendbar. Bei einem extern gesteuerten Prozess sind die Werte in den Zeilen 4-6 nicht sinnvoll und werden als 0 (Null) angezeigt.



Timing Information							
Process No.:	1	2	3	4	9	10	
min. Length :		14	22				
max. Length :		15	30				
Ø Length :		14.4	22				
max. Latency :		0	16				
max (Latency + Length) :		15	38				
count (Length > Delay) :		0	0				
Critical timings :		0	0				

Abb. 11 – Das Fenster „Timing information“

Alle Zeitangaben sind in Taktzyklen zu 25ns angegeben. „Length“ bezeichnet die Dauer eines Prozesszyklus (Abschnitt **EVENT**); diese Bearbeitungsdauer kann man auch ermitteln wie in Kapitel 4.1 „Bearbeitungszeit messen“ beschrieben. „Latency“ ist die Zeitdauer zwischen dem Auftreten eines Event-Signals (extern oder vom internen Timer) und dem tatsächlichen Start des Prozesszyklus, im Bild am zeitgesteuerten Prozess 1 dargestellt.



Die Kennwerte des Fensters haben folgende Bedeutung:

- „min. Length“: Minimale gemessene Dauer eines Prozesszyklus
- „max. Length“: Maximale gemessene Dauer eines Prozesszyklus
- „Ø Length“: Durchschnittliche Dauer des Prozesszyklus, berechnet als Mittelwert aus den 1000 letzten „length“-Werten.

Gemeinsam mit min. Length und max. Length zeigt dieser Kennwert, wie groß und wie gleichmäßig die Bearbeitungsdauer eines Prozesszyklus ist. Unterschiedliche Bearbeitungsdauern kommen zustande, wenn große Datenmengen erst nach längerer Zeit ausgewertet werden oder wenn fallweise Unterscheidungen (**IF**, **CASE**) verschieden lang dauernde Programmabschnitte (Schleifen) beinhalten.

- „max. Latency“: Größte gemessene Startverzögerung eines Prozesszyklus; nur für zeitgesteuerte Prozesse verfügbar.

Eine Startverzögerung entsteht, wenn beim Auftreten eines Event-Signals gerade ein hochpriorer Prozess bearbeitet wird. Dies tritt auf, wenn die Bearbeitungsdauer eines Prozesszyklus das (zu diesem Prozess gehörende) Processdelay überschreitet. Bei mehreren hochprioreren Prozessen sind gelegentliche Startverzögerungen unvermeidbar, es sei denn, deren Processdelays sind ganzzahlige Vielfache voneinander.

Die Summe über alle Verzögerungen sollte im Mittel immer 0 sein; dies entspricht dem Einhalten einer mittleren Frequenz. Daneben ist der Kennwert wichtig für Prozesse, deren Prozesszyklen sehr genau zum vorbestimmten Zeitpunkt ablaufen müssen.

- „max. (Latency+Length)“: Maximum aus der Summe von Startverzögerung und Dauer eines Prozesszyklus; nur für zeitgesteuerte Prozesse verfügbar.

Für ein optimales Zeitverhalten sollte dieser Wert kleiner als das Processdelay sein; wenn dies eingehalten werden kann, verursacht der Prozess keine Startverzögerung bei sich selbst (bei anderen Prozessen ist das aber möglich).

- „count (Length > Delay)“: Wert, der angibt, wie oft ein Prozesszyklus das Processdelay überschritten hat; nur für zeitgesteuerte Prozesse verfügbar. Dieser Wert sollte möglichst Null sein.

Je größer dieser Wert ist, umso häufiger hat der Prozess eine Startverzögerung bei sich selbst (und vielleicht auch bei anderen Prozessen) hervorgerufen. Das Betriebssystem versucht kontinuierlich, diese Verzögerungen wieder aufzuholen. Über den Verlust von Events sagt die Anzahl der Überschreitungen dagegen nichts aus.

- „Critical timings“: Anzahl des Zutreffens einer Bedingung, die ein möglicherweise verlorenes Event-Signal bedeutet. Dieser Wert sollte unbedingt Null sein.

Dieser Kennwert hat je nach Art und Anzahl der Prozesse eine unterschiedliche Bedeutung (siehe auch Kapitel 5.2.5 „Verschiedene Betriebszustände im Betriebssystem“, Seite 92).

Event-Signale können verloren gehen bei:

- einem einzigen zeitgesteuerten, hochpriorigen Prozess (auch in Kombination mit dem extern gesteuerten Prozess)
- dem extern gesteuerten Prozess (auch in Kombination mit einem oder mehreren zeitgesteuerten Prozessen)

Bei mehreren zeitgesteuerten Prozessen können Event-Signale *nicht* verloren gehen, das Eintreten der nachstehenden Bedingung wird aber dennoch gezählt. Hier ist der Kennwert zu interpretieren als Auftreten eines sehr schlechten Zeitverhaltens, das auf jeden Fall verbessert werden muss.

Eine Anzahl verlorener Event-Signale bedeutet, dass (seit dem letzten Start des Prozesses) weniger Prozesszyklen ausgeführt wurden als Event-Signale aufgetreten sind, nämlich wahrscheinlich um die angegebene Anzahl weniger. Dieser Verlust kann vom Betriebssystem nicht ausgeglichen werden.

Ein Event-Verlust wird gleich gesetzt mit dem Eintreten der Bedingung:

- bei zeitgesteuerten Prozessen:  
 $\text{max. latency} + \text{length} > 2 \times \text{Processdelay}$
- bei extern gesteuerten Prozessen:  
Wenn die Bearbeitung des Abschnitts **EVENT**: gerade beendet ist, steht schon ein neues externes Event-Signal an. Falls während dieser Bearbeitungszeit noch weitere Event-Signale aufgetreten sind, sind diese weiteren Events verloren.

Manchmal ist es möglich, dass trotz zutreffender Bedingung *kein* Event verloren geht. Daher liegen Sie auf der sicheren Seite, wenn Sie die Anzahl zutreffender Bedingungen so weit wie möglich reduzieren.

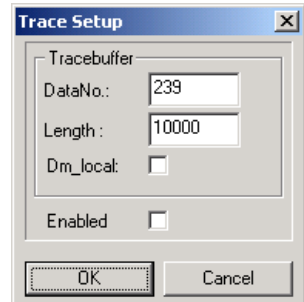


**Menüeintrag „Trace Setup ...“**

Mit dem Menüeintrag „Trace Setup ...“ öffnen Sie das gleichnamige Einstellungsfenster für den Trace-Modus. Für die Anwendung des Trace-Modus finden Sie weitere Informationen in Kapitel 4.3.3 auf Seite 81.

Der Trace-Modus wird mit der Option `Enabled` aktiviert.

Das Eingabefeld `DataNo` gibt an, in welchem globalen Feld die gewünschten Prozess-Informationen abgelegt werden. Lassen Sie die Einstellung `239` (für `DATA_239`) unverändert, wenn Sie Informationen zu einem einzigen Prozess benötigen.



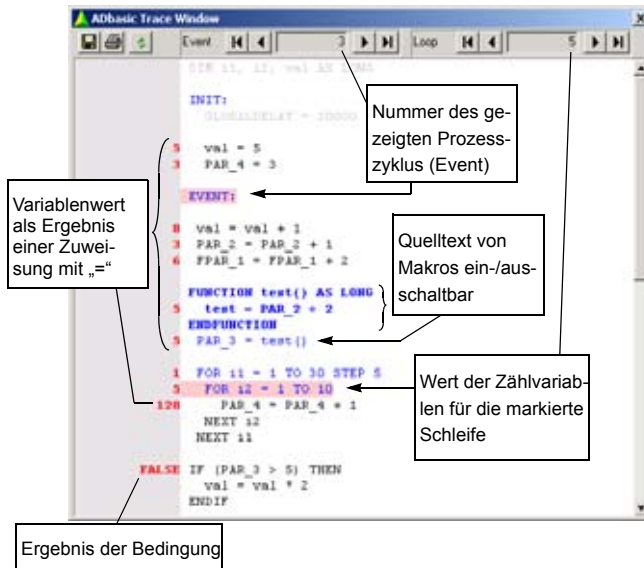
Geben Sie unter `Length` die Größe des globalen Felds (in Long-Werten) so an, dass einerseits das Feld genügend Platz für die Trace-Informationen hat und andererseits im ADwin-System genug Speicher für die Variablen Ihrer Prozesse bleibt.

Bei aktivierter Option `DM_Local` wird das globale Feld im lokalen Speicher anstatt im externen Speicher abgelegt (siehe auch „Speicherbereiche“ auf Seite 55). Auf Daten im kleineren (!) lokalen Speicher kann der Prozessor wesentlich schneller zugreifen.

**Menüeintrag „Show Trace“**



Mit dem Menüeintrag „Show trace“ öffnen Sie das Fenster „ADbasic Trace Window“ (nur bei aktivem Trace-Modus).

Im Trace-Fenster sehen Sie links neben den (für den Trace-Modus aktivierten) Quelltextzeilen die Prozess-Informationen. Die wesentlichen Informationen im Fenster sind:



Die angezeigten Informationen werden zur Laufzeit, also während Ihr Programm läuft, in ein globales Feld gespeichert (normalerweise [DATA\\_239](#), siehe Menüeintrag „Trace Setup ...“). Die Entwicklungsumgebung kopiert den Feldinhalt anschließend zum PC und zeigt ihn an. Je nach Feldgröße können die Informationen mehr oder weniger Event-Durchläufe umfassen.

Durch die Schaltfläche **New Values**  in der Kopfzeile werden aktuelle Prozess-Informationen in das globale Feld gespeichert und dann an den PC übertragen. Die vorherigen Prozess-Informationen gehen damit verloren.

Für einen späteren Vergleich können Sie entweder alle Prozessinformationen speichern  oder den aktuellen Bildschirminhalt drucken .

## Option „Debug mode“

Wenn Sie die Option „Debug mode“ aktivieren und anschließend einen Quelltext kompilieren, werden zusätzliche Sicherheitsabfragen in den Prozess eingebaut (siehe auch Kapitel 4.3.1 auf Seite 78).

Das Einschalten der Option verlängert die Programm-Ausführungszeit und vergrößert den Speicherbedarf. In der Regel liegt dies in einer Größenordnung von ca. 20%, aber auch größere Werte sind möglich. Sie sollten daher diese Option nur während der Programmentwicklung nutzen.

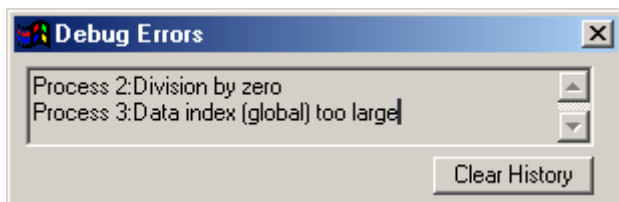


Abb. 12 – Das Fenster „Debug Errors“

Wenn ein Laufzeitfehler im ADwin-System auftritt, wird das Fenster „Debug Errors“ automatisch geöffnet. Wenn Sie das Fenster geschlossen haben, können Sie es über den Menüeintrag „Show Debug Window“ wieder aktivieren.

Das Betriebssystem korrigiert Laufzeitfehler so, dass ein stabiler Betriebszustand erhalten bleibt; dies kann zu unerwarteten Programm-Ergebnissen führen. Bei Laufzeitfehlern mit Pro II-Modulen wird der Prozess gestoppt.

Die folgende Tabelle zeigt, welche Fehler angezeigt werden und welche Korrektur dazu jeweils erfolgt.

Laufzeitfehler	Korrekturmaßnahme
Division durch Null	Das Ergebnis einer Float-Division wird durch +3.40282E+38 ersetzt, das einer Long-Division durch +2147483647.
Wurzel aus negativer Zahl	Das Ergebnis des Wurzelziehens wird durch den Wert 0 ersetzt.
Data index zu groß / <1 Array index zu groß / <1 Zugriff auf nicht deklarierte Elemente eines lokalen oder globalen Felds, d. h. auf eine zu große oder zu kleine Elementnummer.	Eine zu kleine Elementnummer (<1) wird durch 1 ersetzt, eine zu große Elementnummer durch die größte dimensionierte Elementnummer.
Fifo-Index ist kein Fifo Fifo-Nummer ist nicht im gültigen Bereich 1...200	Befehl ( <b>FIFO_CLEAR</b> , <b>FIFO_FULL</b> , <b>FIFO_EMPTY</b> ) wird nicht ausgeführt.
Adresse des Pro II Moduls ist >15 oder <1	Der Prozess wird beendet.

Laufzeitfehler	Korrekturmaßnahme
P2_BURST_xxx <sup>1</sup> : "startadr" ist nicht durch 4 teilbar	Der Prozess wird beendet.
P2_BURST_xxx <sup>1</sup> : Anzahl der Werte nicht durch 4 teilbar	Der Prozess wird beendet.
P2_BURST_INIT: Number of values is not divisable by 4 / by 8	Der Prozess wird beendet.
P2_BURST_READ_UNPACKED1: Anzahl der Werte ist nicht durch 8 teilbar	Der Prozess wird beendet.
P2_BURST_READ_UNPACKED2: Anzahl der Werte ist nicht durch 4 teilbar	Der Prozess wird beendet.
P2_BURST_READ_UNPACKED8: Anzahl der Werte ist nicht durch 2 teilbar	Der Prozess wird beendet.
P2_BURST_READ: Anzahl der Werte ist kleiner als 1 / als 4	Der Prozess wird beendet.

Es wird für jeden Prozess nur ein einzelner Fehler angezeigt (in der Regel der zuletzt aufgetretene), auch wenn der Prozess mehrere Laufzeitfehler erzeugt.

Bitte beachten Sie: Beim Befehl **MEMCPY** wird nur der Zugriff auf das Zielfeld geprüft und korrigiert; ein Zugriff auf nicht deklarierte Elemente des Quellfelds wird nicht erkannt.

## Menüeintrag „Show Debug Window“

Mit dem Menüeintrag „Show Debug Window“ können Sie das Fenster „Debug Errors“ nur (nach Schließen von Hand) **erneut** öffnen. Das erste Öffnen erfolgt automatisch, wenn ein Laufzeitfehler im ADwin-System auftritt.

In beiden Fällen muss die Option „Debug mode“ aktiviert sein.

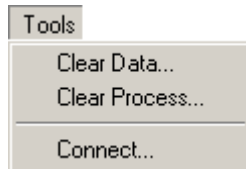
---

1. Gilt für P2\_BURST\_INIT, P2\_BURST\_READ, P2\_BURST\_WRITE

### 2.3.7 Das Menü „Tools“

Im Menü „Tools“ rufen Sie verschiedene Hilfsprogramme auf.

Der Menüeintrag „Clear Data“ gibt den Speicher im angeschlossenen *ADwin*-System frei, der von einem bestimmten *DATA*-Feld belegt wird (Gegenstück zum Befehl **DIM**). Die im Feld enthaltenen Daten gehen damit verloren.



Geben Sie im folgenden Dialogfenster die Nummer des gewünschten *Data*-Felds ein, z. B. „3“ für *Data\_3* und bestätigen Sie die Freigabe des Felds.

Der Menüeintrag „Clear Process“ löscht einen bestimmten Prozess aus dem Speicher. Beachten Sie, dass ein Prozess erst gelöscht werden kann, wenn Sie ihn vorher gestoppt haben.

Der Menüeintrag „Connect“ ermöglicht Einstellungen für das (von uns nicht mehr aktualisierte) Programm *ADserver*, mit dem Sie eine Netzwerk-Verbindung zu einem *ADwin*-System einrichten können. Eine Beschreibung hierzu finden Sie im Anhang A.6.2.



Wir empfehlen Ihnen, das Programm *ADwin TCP/IPserver* anstelle des Programms *ADserver* zu verwenden. Nehmen Sie für diesen Fall keine Einstellungen im Dialogfenster vor, sondern schließen Sie es!

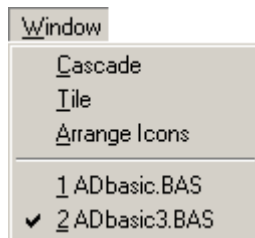
Sie finden nähere Informationen z. B. in der Online-Hilfe von *ADwin TCP/IPserver*.

### 2.3.8 Das Menü „Window“

Mit dem Menü „Window“ können Sie zwischen verschiedenen Quelltext-Fenstern umschalten und diese am Bildschirm arrangieren.


Der Menüpunkt *Arrange Icons* ordnet die Symbole verkleinerter Dateien neu an, was Sie z. B. nach einer Änderung der Bildschirmauflösung verwenden können.

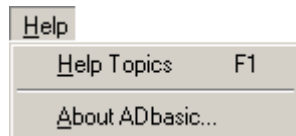
Unten im Menü können Sie über die Dateinamen einen der geöffneten Quelltexte zum aktiven Fenster machen. Der aktive Quelltext ist mit einem Haken gekennzeichnet; im Beispiel rechts ist dies „ADbasic3.bas“.





### 2.3.9 Das Menü „Help“

Mit dem Menü „Help“ rufen Sie die Online-Hilfe zu *ADbasic* auf. Alternativ können Sie auch die Schaltfläche  oder die Taste [F1] verwenden.



Mit dem Menüeintrag „About *ADbasic*“ öffnen Sie ein Fenster, das die Version der Entwicklungsumgebung und den verwendeten „License key“ angibt. Wenn Sie die Schaltfläche „Change License“ drücken, können Sie den „License key“ ändern.

Sie finden den License key auf dem Deckblatt dieses *ADbasic*-Handbuchs. Ohne Eingabe eines gültigen License key befindet sich *ADbasic* im Demo-Modus. In diesem Modus ist das Arbeiten mit der Entwicklungsumgebung nur zu Prüfungs-, Demonstrations- und Bewertungszwecken erlaubt.

### 2.3.10 Das Projektfenster

Das Projektfenster zeigt an, ob ein Projekt geöffnet und welche Quelltextdateien eingebunden sind.

Sie können im Projektfenster folgende Aktionen ausführen:

- Eine Quelltext-Datei öffnen und zum aktiven Quelltext machen:
  - Klicken Sie doppelt (linke Maustaste) auf die Datei oder
  - Markieren Sie eine Datei (linke Maustaste) und wählen „Open“ aus dem Kontextmenü (rechte Maustaste).
- Eine Quelltext-Datei speichern:  
Markieren Sie eine Datei und wählen „Save“ aus dem Kontextmenü.
- Eine Quelltext-Datei aus dem Projekt löschen:  
Markieren Sie die Quelltextdatei mit der Maus und
  - drücken die Taste [ENTF] oder
  - wählen „Remove from Project“ aus dem Kontextmenü.
- Die Anzeige der eingebundenen Dateien verbergen:  
Doppelklick auf den Projektnamen; es erscheint das Zeichen [+] links vom Projektnamen.

Folgende Aktionen sind nur im Kontextmenü wählbar:

- Eine Quelltextdatei auf einem Speichermedium in das Projekt einbinden:  
Wählen Sie „Add to Project“ aus dem Kontextmenü.
- Alle offenen Quelltextdateien werden in das Projekt einbinden:  
Wählen Sie „Add Open Files to Project“ aus dem Kontextmenü.
- Alle offenen Quelltextdateien des Projekts speichern:  
Wählen Sie „Save All Files of Project“ aus dem Kontextmenü.

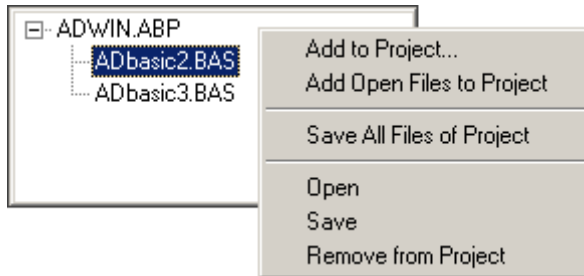




Abb. 13 – Das Projekt-Fenster mit Kontextmenü

### 2.3.11 Das Parameterfenster


Das Parameterfenster zeigt Ihnen in einer Tabelle einen Teil der globalen Parameter `PAR_1...PAR_80` und `FPAR_1...FPAR_80`. Mit dem Schiebepfeil am rechten Rand können Sie den angezeigten Parameter-Bereich auswählen.

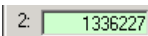
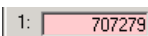
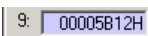
Wenn die Kommunikation zwischen PC und ADwin-System aktiv ist (Schaltfläche  in der Werkzeugleiste), sind die Tabellenfelder weiß hinterlegt und zeigen die Werte der globalen Parameter an. Die Werte werden kontinuierlich vom System ausgelesen und angezeigt. Grau hinterlegte Felder zeigen eine inaktive Kommunikation an (Schaltfläche .

Par	FPAr
1: 707279	0
2: 1336227	0
3: 628948	3.46212e+028
4: 0	-2.66234
5: 00000011H	0
6: 0	0
7: 0	0
8: 707279	0
9: 0	0
10: 0	0

Abb. 14 – Das Parameter-Fenster


Die Werteanzeige eines Parameters (`PAR_1...PAR_80`) kann zwischen dezimal und hexadezimal umgestellt werden (siehe `PAR_5` in Abb. 14). Klicken Sie hierzu mit der rechten Maustaste auf die Nummer der betreffenden Variable (links vom Tabellenfeld) und wählen die Option `Hexadezimal`.

Wenn Sie die Schaltfläche  drücken, werden die Tabellenfelder der Parameter farbig markiert, die im aktiven Quelltext und (falls vorhanden) im zugehörigen Projekt verwendet sind. Die Farben haben folgende Bedeutung:

- Grün: Parameter wird nur im aktiven Quelltext verwendet. 
- Rot: Parameter wird im aktiven Quelltext verwendet, aber auch in einem anderen Quelltext des Projekts. 
- Blau: Parameter wird im Projekt verwendet, jedoch nicht im aktiven Quelltext. 

Die Markierungs-Funktion ist nur verfügbar, wenn die Kommunikation zwischen PC und *ADwin*-System aktiv ist. Außerdem greift die Funktion nur auf gespeicherte Quelltexte zu; Sie werden daher zum Speichern eines Quelltextes aufgerufen, wenn dieser nach dem Öffnen verändert wurde.

### 2.3.12 Das Prozessfenster

Das Prozess-Fenster zeigt Informationen über die Prozesse 1...10 auf einem ADwin-System an, wenn die Kommunikation zwischen PC und System aktiv ist (Schaltfläche  in der Werkzeugleiste). Anderenfalls sind die Felder grau hinterlegt

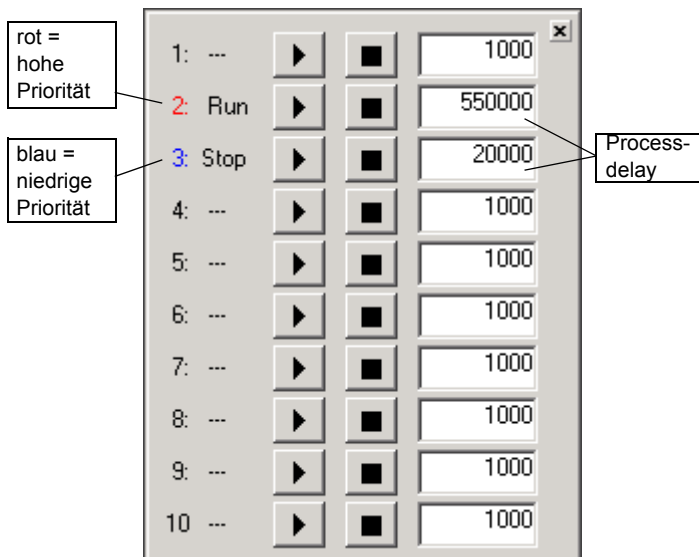




Abb. 15 – Das Prozess-Fenster

Für jeden Prozess 1...10 werden der Status („Run“ oder „Stop“) und das Processdelay (Prozess-Zykluszeit) angezeigt; das zum aktiven Quelltext gehörige Processdelay ist auch in der Werkzeugleiste zu sehen. Die Priorität eines Prozesses erkennen Sie an der Farbe der Prozessnummer: rot = hochprior, blau = niederprior. Die Bedeutung und die Zeiteinheiten des Processdelay sind in Kapitel 5.2.1, Seite 88 erläutert.

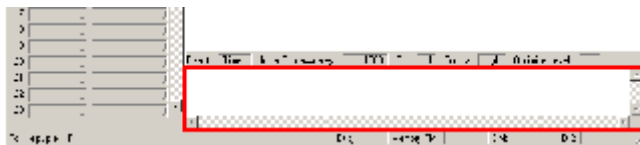
Sie können das Processdelay im Prozess-Fenster auch verändern; nach der Eingabe wird dieser Wert automatisch an das ADwin-System übertragen. Achten Sie darauf, Ihr System nicht durch zu kleine Werte überlasten.

Mit den Schaltflächen  und  können Sie die Ausführung eines Prozesses beenden und wieder starten. Die Schaltflächen in der Werkzeugleiste haben die gleiche Funktion; sie beziehen sich auf den zum aktiven Quelltext gehörigen Prozess.

### 2.3.13 Das Infofenster

Im Info-Fenster werden Meldungen des Compilers zum jeweils aktiven Quelltext dargestellt:

- Fehlermeldungen (in rot)
- Warnungen
- Statusmeldung nach dem Kompilieren



Eine erfolgreiche Statusmeldung nach dem Kompilieren sieht z.B. folgendermaßen aus:

```
0 error(s), 0 warning(s)
Process compiled. Codesize: 836 Workspacesize: 8
Stacksize: 20 Byte
```

Die Werte geben Hinweise, wieviel Speicherplatz der Prozess benötigen wird:

- **Codesize:** Größe der erzeugten Binärdatei in Bytes; die Datei wird als Prozess im Programmspeicher (PM) abgelegt.
- **Workspacesize:** Benötigter Speicherplatz in Bytes im lokalen Datenspeicher (DM) für
  - lokale Variablen und Felder
  - interne Zwecke ( $2 \times 4$  Byte)

Darüber hinaus wird weiterer Platz im Datenspeicher benötigt, der manuell berechnet werden kann:

- Jedes globale Feld benötigt etwa vierzig Byte im lokalen Datenspeicher (für interne Zwecke).
- Jedes Element eines globalen Felds benötigt 4 Byte (im externen Datenspeicher; nur wenn das Feld **AT DM\_LOCAL** deklariert ist, liegen die Daten im lokalen Datenspeicher).
- **Stacksize:** Größe des internen Stapels, der für Libraries verwendet wird.

Es wird nicht angezeigt, wieviel Speicherplatz im externen Datenspeicher (DX) benötigt wird.

### 2.3.14 Die Statusleiste

Die Statusleiste befindet sich am unteren Rand des *ADbasic*-Fensters.



Letzte Aktion CPU- und Speicherauslastung im *ADwin*-System Cursor-Position und in *ADbasic* Tastatureinstellungen

- Links: Informationen zur zuletzt ausgeführten Aktion.
- Mitte: Die aktuelle Prozessor- und Speicherauslastung des *ADwin*-Systems (bei aktivierter Verbindung zwischen PC und *ADwin*-System).
- Rechts: Die aktuelle Cursor-Position im Quelltextfenster (Zeile und Spalte); daneben die Tastatureinstellungen CAPS LOCK, NUM LOCK und SCROLL LOCK.

Die Angaben zur Prozessor-/Speicherauslastung bedeuten:

**Busy:** Zeitliche Auslastung des Prozessors in Prozent, berechnet als: Rechenzeit / (Rechenzeit + Leerlaufzeit).

**PM:** Freier Programmspeicher in Bytes.

**EM:** Freier Zusatzspeicher in Bytes (nur für T11).

**DM:** Freier interner Datenspeicher in Bytes.

**DX:** Freier externer Datenspeicher in Bytes.

## 2.4 ADtools

*ADtools* ist eine Sammlung kleiner Hilfsprogramme, mit denen Sie die globalen Variablen (*Par*, *FPar*) und Felder (*Data*) von *ADwin*-Systemen anzeigen und auch ändern können. Die Programme unterstützen Sie beim Entwickeln von Prozessen für Ihr *ADwin*-System, indem sie z.B. Werte und Zustände anzeigen, diese mit praktischen Werkzeugen verändern oder einfache Messwert-Verläufe darstellen.

Starten Sie eines der *ADtools*, indem Sie im Windows-Startmenü „Programme ▶ *ADwin* ▶ *ADtools* ▶ <Toolname>“ wählen. Mit der rechten Maustaste öffnen Sie das Konfigurationsmenü, um die Art der Anzeige und die darzustellenden Variablen auszuwählen.

Jedes *ADtool* ist ein eigenständiges Windows-Programm, das Sie auch mehrfach starten können: Lassen Sie sich alle interessanten Parameter auf dem

Bildschirm anzeigen. Wenn Sie eine passende Bildschirmanzeige zusammengestellt haben, können Sie die Gesamt-Konfiguration speichern und später erneut verwenden.

Folgende *ADtools* stehen Ihnen zur Verfügung:



**TDigit** ermöglicht das Anzeigen und Eingeben von Werten für globale Variablen und Felder.



**TGraph** stellt den Inhalt globaler Felder in Kurvenform dar.



**TButton** löst beim Druck auf die Schaltfläche eine von Ihnen definierte Aktion aus, wie z. B.: Booten, Variable ändern, Prozess laden, starten oder stoppen.



**TLed** zeigt einen Variableninhalt an durch Leuchten, Blinken oder Flackern und kann zur Überwachung dienen. Auch ein akustischer Alarm ist möglich.



**TMeter** Analog-Zeigerinstrument zur Anzeige von globalen Variablen und Feldern.



**TPoti** Potentiometer zur Anzeige und zum Verändern von globalen Variablen und Feldern.



**TProcess** zeigt Informationen der laufenden Prozesse, kann diese starten, stoppen und das Timing verändern.



**TPar\_FPar** ermöglicht das Anzeigen und Eingeben von allen oder ausgewählten Variablen.



**TFifo** ermöglicht das kontinuierliche Speichern von Daten eines FIFO-Feldes in eine Datei.



**TBin** zeigt bis zu 5 ganzzahlige Variablen binär (als DIL-Schalter) und hexadezimal an. Die Variablenwerte können auch geändert werden.



**ADtools** speichert und lädt eine von Ihnen erstellte Gesamt-Konfigurationen aus mehreren *ADtools*.

Alle weiteren Informationen zu den Hilfsprogrammen entnehmen Sie bitte der Online-Hilfe, die Sie im Programm *ADtools.exe* aufrufen.





### 3 Prozesse programmieren


In diesem Kapitel zeigen wir Ihnen, wie Sie ein *ADbasic*-Programm aufbauen, strukturieren und welche Variablen Ihnen dabei zur Verfügung stehen.

#### 3.1 Programmaufbau

Sie geben ein *ADbasic*-Programm als ASCII-Text mit dem Editor der Entwicklungsumgebung ein; dabei verwenden Sie eine erweiterte Basic-Syntax. Diesen Quelltext übersetzt der Compiler in einen ausführbaren Prozess für Ihr spezielles *ADwin*-System.

Ein Quelltext besteht aus einer beliebigen Anzahl von Befehlszeilen, die jeweils einen Befehl oder eine Zuweisung enthalten (Ausnahme siehe : (Doppelpunkt)). In einer Zeile können Sie bis zu 255 (ASCII-)Zeichen eingeben.

*ADbasic* akzeptiert bei Befehlen und Variablennamen Groß- und Kleinschreibung (in Beispielen haben wir zur Übersichtlichkeit Befehle und globale Variablen groß geschrieben).

Ein Programm besteht aus bis zu 4 Abschnitten, die bei der Ausführung auf dem *ADwin*-System unterschiedliche Aufgaben übernehmen, sowie aus den erforderlichen Deklarationen. Halten Sie die Reihenfolge der Abb. 16 beim Programmaufbau ein. 

Jedes Programm muss zumindest den Abschnitt **EVENT** : enthalten.

Optional können Sie Funktionen und Unterprogramme definieren, und „Include“-Dateien und Libraries einbinden.

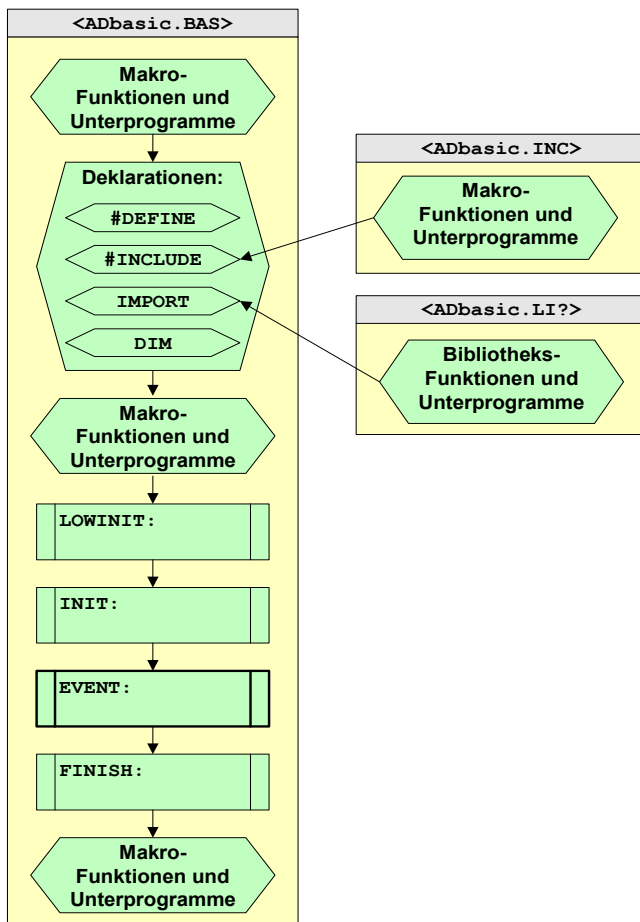



Abb. 16 – Aufbau eines ADbasic-Programms

### 3.1.1 Die 4 Programmabschnitte


Die 4 Programmabschnitte (Abb. 16) beginnen jeweils mit einem Kennwort und haben folgende Funktion:

- **LOWINIT**: ist nur bei hochprioren Prozessen einsetzbar.

Dieser Abschnitt wird bei jedem Start des Prozesses einmalig durchlaufen und dient zur Initialisierung, z.B. von Variablen oder Datenleitungen. Er wird immer vor dem Abschnitt **INIT**: ausgeführt (falls vorhanden) und immer mit niedriger Priorität, Stufe 1.

Dieser Abschnitt ist für umfangreiche Initialisierungs-Sequenzen geeignet, da er (wegen seiner niedriger Priorität) unterbrochen werden kann. 

- **INIT**: entspricht dem Abschnitt **LOWINIT**: (Initialisierung, einmaliger Durchlauf). Er wird aber mit der von Ihnen eingestellten Priorität bearbeitet (Menüpunkt „Options / Process“).

Dieser Abschnitt kann, wenn Sie hohe Priorität eingestellt haben, nicht unterbrochen werden und sollte dann möglichst kurz sein. 

- **EVENT**: ist der zentrale Funktionsabschnitt, der (typischerweise) in regelmäßigen Abständen aufgerufen wird, bis er gestoppt wird. Je nach Einstellung wird der Aufruf durch einen zyklischen Timer-Event oder durch einen externen Event ausgelöst.
- **FINISH**: wird nach dem Stoppen eines Prozesses einmalig durchlaufen und ist daher das „Gegenstück“ zu den Initialisierungsabschnitten. Dieser Abschnitt wird immer mit niedriger Priorität, Stufe 1 ausgeführt.

Die Abschnitte **LOWINIT**:, **INIT**: und **FINISH**: sind optional, der Abschnitt **EVENT**: muss immer vorhanden sein. 

### 3.1.2 Weitere Teile eines Programms

#### Symbolische Definitionen

Mit der Anweisung **#DEFINE** können Sie symbolische Namen definieren. Gruppieren Sie alle diese Definitionen am Beginn der Datei und vor dem Start der 4 Programmabschnitte.

#### Deklarationen

In *ADbasic* müssen Sie nur lokale Variablen und alle Felder vor der Benutzung deklarieren. Die globalen Variablen **PAR\_n** und **FPAR\_n** sind bereits vordefi- 

niert und müssen nicht deklariert werden. Nach der Deklaration haben Variablen und Felder keinen definierten Inhalt, sollten also von Ihnen initialisiert werden.

Alle diese Variablen und Felder sind innerhalb des Prozesses in allen Programmabschnitten verfügbar. Auf die globalen Variablen und Felder können Sie darüber hinaus auch aus anderen Prozessen und vom PC aus zugreifen, z.B. um zwischen den Prozessen oder zwischen Prozessen und PC Daten auszutauschen.

### **Makros**

Makro-Funktionen und -Unterprogramme werden bei einem Aufruf im Programmtext an der aufrufenden Stelle eingefügt. Sie müssen in jedem Fall außerhalb der 4 Programmabschnitte definiert werden (siehe Abb. 16 auf Seite 44).

### **Libraries**

Das Einbinden von Libraries muss vor dem Beginn der Programmabschnitte erfolgen. Library-Funktionen und -Unterprogramme sind Programm-Module mit geringerem Speicherbedarf (bei mehrfachem Aufruf) als die o.g. Makro-Funktionen und -Unterprogramme.

## 3.2 Variablen und Felder

### 3.2.1 Übersicht

Datenstruktur	Name	Datentyp	Bemerkung
Globale Variablen und Felder			
Variable (Skalar)	PAR_1...PAR_80	LONG	Vordefiniert,
	FPAR_1...FPAR_80	FLOAT	nicht deklarierbar,
System-Variable	PROCESSDELAY	LONG	Speicherbereich DM
	PROZESS <sub>n</sub> _RUNNING	LONG	
Ein- oder zweidi- mensionales Feld (Vektor)	DATA_1 [ ] [ ] ...	LONG,	Name DATA_ nicht änderbar, nur Deklara- tion von Feldnummer und Dimension.
	DATA_200 [ ] [ ]	FLOAT,	
		STRING,	
		FIFO	
Lokale Variablen und Felder			
Variable (Skalar)	frei wählbar	LONG, FLOAT	muss deklariert wer- den
Ein-dimensiona- les Feld (Vektor)	frei wählbar	LONG, FLOAT, STRING	muss deklariert wer- den

Wenn Sie bei der Deklaration den Speicherbereich nicht explizit festlegen, werden Variablen standardmäßig im internen Speicher DM angelegt sowie Felder im externen Speicher DX (Speicherbelegung siehe Kapitel 3.3.1).

Alle Datentypen haben eine Länge von 32 Bit.

### 3.2.2 Datenstrukturen

In *ADbasic* stehen Ihnen vor allem 2 Datenstrukturen zur Verfügung:

- Variablen (Skalare)

**VAR**

Eine Variable kann einen einzelnen Wert enthalten.

- Ein- und zweidimensionale Felder.

**ARRAY**

Ein Feld besteht aus einer frei definierbaren Zahl von Feldelementen, die je einen Wert enthalten können.

Sie können eindimensionale globale Felder `DATA_n` auch als FIFO verwenden (Ringspeicher nach dem Prinzip: First in, first out, siehe Kapitel 3.3.4 auf Seite 58).

Die maximale Anzahl an Variablen bzw. die Größe eines Felds ist nur durch die Speichergröße des ADwin-Systems begrenzt.

Der Compiler unterscheidet

- globale Variablen und Felder:

Auf globale Datenstrukturen können sowohl alle Prozesse als auch PC-Anwendungen zugreifen, z. B. zum Austausch von Daten.

Die Systemvariablen zählen zu den globalen Variablen .

- lokale Variablen und Felder:

Lokale Datenstrukturen sind nur innerhalb des Prozesses verfügbar, in dem sie deklariert wurden. In gleicher Weise können Sie lokale Datenstrukturen innerhalb einer Funktion oder eines Unterprogramms deklarieren.

Sie deklarieren (fast) alle Variablen und Felder mit der Anweisung **DIM**; dadurch werden der Datentyp bestimmt sowie der erforderliche Platz im Speicher belegt und dem Variablennamen fest zugeordnet.

Zur Vereinfachung für Sie sind die globalen Variablen `PAR_1 ... PAR_80` und `FPAR_1 ... FPAR_80` bereits vordefiniert; Sie müssen (und können) diese Variablen also nicht deklarieren.

Die Deklaration globaler Felder erkennt der Compiler am Namen „`DATA_n`“, wobei „`DATA_`“ ein festgelegter Text ist und „`n`“ die von Ihnen festgelegte Nummer des Felds (1...200).



Variablen und Feldelemente haben nach der Deklaration keinen definierten Wert und sollten deshalb mit einem sinnvollen Wert (z. B. Null) initialisiert werden. Ausnahme: Nach dem Einschalten des ADwin-Systems werden die globalen Variablen automatisch mit Null initialisiert.

### 3.2.3 Datentypen

Der Compiler verarbeitet folgende Datentypen:

- `LONG`: Ganzzahliger 32 Bit-Wert mit dem Wertebereich:

$$-2147483648 \dots +2147483647 = -2^{31} \dots +2^{31}-1$$

- `FLOAT`: Fließkomma-Wert mit dem Wertebereich:

$$-3,402823 \cdot 10^{+38} \dots -1,175494 \cdot 10^{-38} \text{ (negative Werte, 32 Bit)}$$

$$+1,175494 \cdot 10^{-38} \dots +3,402823 \cdot 10^{+38} \text{ (positive Werte, 32 Bit)}$$

Der Wertebereich entspricht nicht dem „IEEE-Floating point“-Format.

Ab dem Prozessor T11 beträgt die Rechengenauigkeit 40 Bit (Wertebereich siehe unten) und zwar ausschließlich in folgenden Bereichen:



- Berechnungen innerhalb des *ADwin*-Systems.
- Auswertung von Konstanten durch den Compiler.

Die 40 Bit-Genauigkeit kann auf dem PC nicht genutzt oder angezeigt werden, weil zwischen PC und *ADwin*-System – aus Gründen der Geschwindigkeit – nur 32 Bit-Werte übertragen werden.

Der Wertebereich für 40 Bit-Fließkommawerte lautet:

$-3,402823668 \cdot 10^{+38} \dots -1,175494351 \cdot 10^{-38}$  (negative Werte)

$+1,175494351 \cdot 10^{-38} \dots +3,402823669 \cdot 10^{+38}$  (positive Werte)

- **STRING**: ASCII-Zeichenfolge, die in Feldern so gespeichert wird, dass jedes Feldelement ein Zeichen enthält (Details siehe Kapitel 3.3.5). Ein einzelnes Zeichen entspricht einem ganzzahligen 8 Bit-Wert im Bereich 0...255.

Geben Sie bei der Deklaration von Variablen und Feldern an, welchen Datentyp sie haben sollen.

Bei der Kombination von ganzzahligen und Fließkomma-Werten kommt es zu einer Typkonvertierung. Dies kann unter bestimmten Umständen zu einem anderen als dem erwarteten Berechnungsergebnis führen. Näheres finden Sie im Abschnitt „Typkonvertierung“.



Im nächsten Abschnitt ist dargestellt, mit welchen Schreibweisen Sie einen Zahlenwert eingeben können.

### 3.2.4 Zahlenwerte eingeben

Sie können 4 verschiedene Schreibweisen benutzen, wenn Sie einen Zahlenwert angeben möchten. Die folgenden Beispiele weisen einer Variablen *x* den Wert 93 zu.

Bei Fließkomma-Werten wird der Punkt „.“ als Dezimaltrennzeichen verwendet (englische Schreibweise).



1. Dezimale Schreibweise:

*x* = 93                      ganzzahliger Wert **LONG** oder

*x* = 93.0                    Fließkomma-Wert **FLOAT**

Beachten Sie den Unterschied: Die Zahl „93“ hat den Datentyp **LONG**, die Zahl „93.0“ dagegen den Datentyp **FLOAT**. Dies ist wichtig, wenn



Sie beide Datentypen in einem Berechnungsausdruck miteinander verwenden (siehe Kapitel 3.4.2).

2. Exponential-Schreibweise:

`x = 93E0` ganzzahliger Wert `LONG` oder

`x = 9.3E1` Fließkomma-Wert `FLOAT`

Hierbei steht „9.3E1“ für  $9,3 \times 10^1$ , d.h. nach dem „E“ folgt der (max. 2-stellige) Exponent zur Basis 10.

3. Binäre Schreibweise:

`x = 1011010b` angehängtes „b“; nur `LONG`

4. Hexadezimale Schreibweise:

`x = 5Ah` angehängtes „h“, nur `LONG`

Wenn der hexadezimale Wert mit einem Buchstaben (A-F) beginnt, müssen Sie eine Null voranstellen: Anstelle von „F6h“ also „0F6h“. Anderenfalls interpretiert der Compiler ihren Wert als den Namen einer lokalen Variablen.

### 3.2.5 Globale Variablen (Parameter)

Auf globale Variablen (und Felder) können alle laufenden Prozesse und der PC zugreifen; daher eignen sie sich gut zum Datenaustausch zwischen den Prozessen oder zwischen den Prozessen und dem PC. Ihnen stehen 80 ganzzahlige Variablen, 80 Fließkomma-Variablen sowie bis zu 200 Felder (Arrays) vom Datentyp **LONG** oder **FLOAT** zur Verfügung. Alle Variablen und Feldelemente haben eine Länge von 32 Bit.

Die ebenfalls global verfügbaren Systemvariablen sind auf Seite 53 beschrieben.

Sie können die globalen Variablen in Ihren Programmen an beliebiger Stelle verwenden, ohne sie zu deklarieren. Die Variablen haben jedoch keinen definierten Wert und sollten deshalb mit einem sinnvollen Wert (z. B. Null) initialisiert werden. Ausnahme: Nach dem Booten des ADwin-Systems werden die Variablen automatisch mit Null initialisiert.



Die globalen Variablen werden auch als Parameter bezeichnet und haben die Namen:

- `PAR_1`, `PAR_2`, ..., `PAR_80` mit dem Datentyp **LONG** für ganzzahlige 32Bit-Werte.
- `FPAR_1`, `FPAR_2`, ..., `FPAR_80` mit dem Datentyp **FLOAT** für Fließkommawerte.

### Beispiel

```
PAR_5 = 700           'Parameter 5 erhält den
                      'Wert 700.
PAR_72 = ADC(1)       'Die Spannung am analogen
                      'Eingang 1 wird gemessen
                      'und in Parameter 72
                      'abgelegt.
```



Im Gegensatz zu den sonstigen Variablen dürfen Sie die globalen Variablen `PAR_n` und `FPAR_n` nicht deklarieren, da sie vordefiniert und dem Compiler bereits bekannt sind.



### 3.2.6 Globale Felder (Arrays)

Die globalen Felder ermöglichen Ihnen, große Datenmengen zwischen den Prozessen auf dem ADwin-System oder dem PC auszutauschen (siehe auch Kapitel 5.3.1 „Datenaustausch zwischen Prozessen“). Ihnen stehen bis zu 200 globale Felder (Arrays) vom Datentyp **LONG** oder **FLOAT** zur Verfügung.

Da Größe und Datentyp wählbar sind, müssen Sie globale Felder am Anfang Ihres Programms deklarieren und möglichst auch initialisieren (die Feldelemente haben sonst keinen definierten Wert).



Die Deklaration eines globalen Felds erkennt der Compiler am Namen „`DATA_n`“, wobei „`DATA_`“ ein festgelegter Text ist und „`n`“ die von Ihnen festgelegte Nummer des Felds (1...200). Die Namen für `DATA`-Felder sind also:

```
DATA_1, DATA_2, ..., DATA_200.
```

Andere Feldnummern sind unzulässig. Sie können jedoch die Feldnummern frei wählen, auch die Deklaration von z. B. `DATA_5` (ohne `DATA_1 ... DATA_4`) ist gültig. In Ihrem Programm werden die Felder vom Compiler anhand ihrer Nummer unterschieden.



### Beispiel

```
DIM DATA_5[20000] AS LONG
REM Deklariert das Feld 5 mit 20000 Elementen vom Typ LONG.
DIM DATA_3[7][5] AS FLOAT
REM Deklariert das Feld 3 mit 7x5 Elementen vom Typ FLOAT.
```

Näheres zu 2-dimensionalen Feldern steht in Kapitel 3.3.3.

Die maximale Größe der Felder richtet sich nur nach dem verfügbaren Speicherplatz. Beispielsweise kann auf einem ADwin-System mit 16MB Speicher ein Feld mit bis zu 4 Millionen Elementen vom Typ **LONG** deklariert werden.

Nachdem das Feld deklariert ist, können Sie auf jedes einzelne Element zugreifen. Das erste Element eines Felds besitzt den Index 1.



Weisen Sie *auf keinen Fall* dem Element 0 eines Felds einen Wert zu, z. B. mit  
DATA\_1[0] = ...



### Beispiel

```
REM Der globalen, ganzzahligen Variablen PAR_1 wird der Wert
REM des 200. Elements aus dem Feld 5 zugewiesen.
PAR_1 = DATA_5[200]

REM Durch diese Anweisung erhält das 345. Element aus dem Feld
REM DATA_5 den Wert 4000.
DATA_5[345] = 4000

REM Diese Anweisung weist dem 1. (!) Element aus dem
REM 2dimensionalen Feld DATA_3 den Wert 300 zu.
DATA_3[1][1] = 300
```

Sie können den Index eines *Feldelements* auch über eine Variable übergeben:

```
REM Auch hier wird, wie im Beispiel davor, dem 345. Element des
REM Felds DATA_5 der Wert 4000 zugewiesen.
nummer1 = 345
DATA_5[nummer1] = 4000
```



Dagegen darf die Nummer eines *Felds* nicht durch eine Variable übergeben werden. Die folgende Anweisung führt zu einer Fehlermeldung des ADbasic-Compilers:

```
num = 2
DATA_num[300] = 20      'FALSCH !!
DATA_2[300] = 20        'RICHTIG
```

Der Compiler interpretiert `DATA_num` als Namen eines lokalen Felds, das (wahrscheinlich) nicht deklariert wurde und daher nicht verfügbar ist. Verwenden Sie statt dessen die Schreibweise `DATA_2`.

### 3.2.7 System-Variablen

Um Informationen über den Status des *ADwin*-Systems zu erhalten, stehen Ihnen die folgenden System-Variablen zur Verfügung. Diese Variablen sind global, d.h. für alle Prozesse und vom PC aus verfügbar. Weitere Informationen finden Sie bei den Befehlsbeschreibungen.

#### **PROZESS<sub>n</sub>\_RUNNING**

Zeigt den Status des Prozesses `n` an (mit `n = 1...10`), d.h. ob der Prozess läuft, gerade angehalten wird oder gestoppt ist (siehe Seite 193). Die Variable kann nur gelesen werden.

#### **PROCESSDELAY**

Der Soll-Zeitabstand, in dem zeitgesteuerte Prozesse vom Zähler aufgerufen werden, ist das `Processdelay`, auch Zykluszeit genannt. Mit der Systemvariablen **PROCESSDELAY** können Sie die Zykluszeit abfragen oder einstellen. Die Zykluszeit wird in Taktzyklen des Zählers gemessen (siehe Kapitel 5.2.1).

Sie können die Variable **PROCESSDELAY** nur innerhalb der Abschnitte **INIT**: und **EVENT**: lesen und beschreiben. Das Beschreiben der Variablen ist jedoch nur 1mal pro Abschnitt erlaubt, weil sonst das *ADwin*-System in einen instabilen Zustand geraten kann.

Ein Beschreiben im Abschnitt **EVENT**: sollte gleich am Abschnittsanfang geschehen, weil sich das Ändern der Variablen sofort auf den Aufruf des nächsten Prozesszyklus auswirkt. Anderenfalls kann die zeitlich exakte Bearbeitung der Prozesszyklen außer Takt geraten.

Achten Sie darauf, dass die Auslastung des Prozessors möglichst weniger als 90% beträgt, keinesfalls aber 100% übersteigen darf.



### 3.2.8 Lokale Variablen und Felder

Alle lokalen Variablen und Felder, die Sie für Ihren Prozess benötigen, müssen Sie vor dem Beginn des ersten Abschnitts in Ihrem *ADbasic*-Programm deklarieren und möglichst auch initialisieren (sie haben sonst keinen definierten Wert).



Namen müssen mit einem Buchstaben beginnen und dürfen nur aus Buchstaben (a-z, A-Z), Ziffern (0-9) und dem Zeichen „\_“ (Underscore) bestehen.

Umlaute sind nicht erlaubt, Groß- und Kleinschreibung wird nicht unterschieden. Die Länge von Variablennamen ist nur begrenzt durch die max. Zeilenlänge (255 Zeichen).

Bei (skalaren) Variablen sind als Datentypen ganzzahlige Werte (**LONG**) und Fließkomma-Werte (**FLOAT**) verfügbar, jeweils in 32 Bit.



### Beispiel

```
DIM wert AS LONG           'Definiert die Variable 'wert'
                                'mit dem Datentyp LONG
DIM wert1, wert2 AS FLOAT 'Definiert die
                                'Variablen 'wert1' und 'wert2'
                                'mit dem Datentyp FLOAT
```

Variablen können Sie nicht nur als skalare Größe, sondern auch als eindimensionales Feld deklarieren, das heißt, Sie können Felder von Variablen erzeugen und verarbeiten. Die Anzahl der zu dimensionierenden Elemente im Feld wird in eckigen Klammern nach dem Namen eingegeben.



### Beispiel

```
DIM wert[100] AS FLOAT 'Definiert ein Feld der
                        'Länge 100 mit Namen 'wert'
                        'und dem Datentyp FLOAT
```



Das erste Element eines Felds besitzt den Index 1, im Beispiel: `wert[1]`. Auf das Element mit dem Index 0 dürfen Sie nicht zugreifen.

## 3.3 Variablen und Felder – Details

### 3.3.1 Variablen und Felder im Datenspeicher

Sie können für Felder und lokale Variablen explizit festlegen, in welchem Speicherbereich (siehe Kapitel 3.3.2) sie angelegt werden. Diese Festlegung geschieht bei der Deklaration mit **DIM** im Quelltext durch die Zusätze **AT DM\_LOCAL** oder **AT DRAM\_EXTERN**. Bei dem Prozessor T11 ist ein zusätzlicher Speicherbereich mit **AT EM\_LOCAL** verfügbar.

Wenn Sie bei der Deklaration keinen Zusatz verwenden, werden Variablen im internen Speicher DM angelegt sowie Felder im externen Speicher DX.

Wie empfehlen Ihnen die Verwendung des internen Speichers für Variablen und (kleine) Felder, auf die Sie sehr schnell zugreifen möchten. Der langsamere externe Speicher ist wegen seiner Größe vorwiegend für Felder geeignet.

In Abb. 17 sehen Sie Beispiele für Deklarationen, um Variablen und Felder in den verschiedenen Speicherbereichen anzulegen.

Variable / Feld	Speicherbereich	Deklaration im Quelltext
Lokale Variable	intern (DM)	<b>DIM</b> var <b>AS</b> <VARTYPE> oder <b>DIM</b> var <b>AS</b> ... <b>AT</b> DM_LOCAL
	Zusatz (EM)	<b>DIM</b> var <b>AS</b> ... <b>AT</b> EM_LOCAL
	extern (DX)	<b>DIM</b> var <b>AS</b> ... <b>AT</b> DRAM_EXTERN
Feld	intern (DM)	<b>DIM</b> array[5] <b>AS</b> ... <b>AT</b> DM_LOCAL
	Zusatz (EM)	<b>DIM</b> array[5] <b>AS</b> ... <b>AT</b> EM_LOCAL
(global oder lokal)	extern (DX)	<b>DIM</b> array[5] <b>AS</b> ... oder <b>DIM</b> array[5] <b>AS</b> ... <b>AT</b> DRAM_EXTERN

Abb. 17 – Festlegung des Speicherbereichs bei Deklarationen

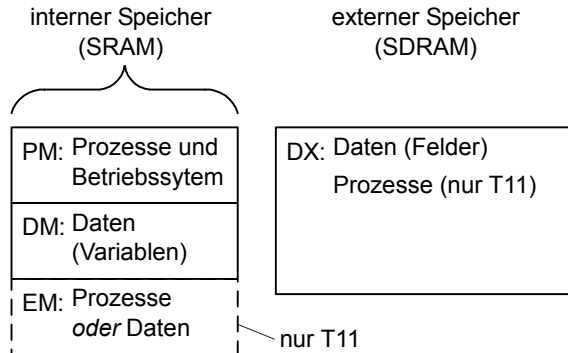
Die globalen Variablen `PAR_1...PAR_80` und `FPAR_1...FPAR_80` sind vordefiniert und stehen immer im internen Speicher DM zur Verfügung. Sie brauchen und können diese daher nicht neu (z.B. für den externen Speicher) deklarieren.



### 3.3.2 Speicherbereiche

Der Prozessor des ADwin-Systems verwendet einen schnellen internen Speicher (SRAM) und einen großen externen Speicher (SDRAM).

Je die Hälfte des internen Speichers steht als Programmspeicher PM und als Datenspeicher DM zur Verfügung. Der Prozessor T11 besitzt außerdem einen internen Zusatzspeicher EM, der entweder als Programm- oder als Datenspeicher genutzt werden kann.



- Programmspeicher (PM)  
Der Programmspeicher belegt die Hälfte des internen SRAM und nimmt das Betriebssystem und Ihre Prozesse auf.
- Zusatzspeicher intern (EM)  
Der interne Zusatzspeicher EM ist nur beim Prozessor T11 vorhanden. Der Zusatzspeicher kann entweder als Datenspeicher oder als Programmspeicher genutzt werden.
- Datenspeicher intern (DM)  
Der interne Datenspeicher belegt die Hälfte des internen SRAM und nimmt die globalen und lokalen Variablen auf).
- Externer Speicher (DX)  
Der externe Speicher belegt das externe SDRAM und nimmt die globalen und lokalen Felder auf .  
Beim T11 kann der externe Speicher auch Prozesse bis zu einem Megabyte Größe aufnehmen.

Sie können auf Daten im internen Speicher DM deutlich schneller zugreifen (etwa Faktor fünf) als auf Daten im externen Speicher DX.

Die Speichergröße (SRAM, SDRAM) ist eine Bestelloption und kann nicht nachträglich vergrößert werden.

Die Größe der Speicherbereiche ist der einzige Faktor, der die Größe von Prozessen und die Zahl der deklarierbaren Variablen und Felder begrenzt (indirekt auch die Größe der Quelltext-Dateien). Sie sehen in der Statusleiste der

Entwicklungsumgebung, wieviel Speicher Ihnen in PM, DM, EM und DX noch zur Verfügung steht (angegeben in Bytes).

### 3.3.3 2-dimensionale Felder

Globale Felder `DATA_n` können mit 1 oder 2 Dimensionen deklariert werden. Die grundsätzlichen Eigenschaften sind in Kapitel 3.2.6 „Globale Felder (Arrays)“ beschrieben.

Die 2-dimensionale Schreibweise kann (im Vergleich zu 1-dimensionalen Feldern) eine Problemlösung vereinfachen. Gleichzeitig aber verlangsamt sie den Datenzugriff und sie erfordert zusätzlichen Programmspeicher.



Der Geschwindigkeitsverlust und der zusätzliche Speicherbedarf sind umso größer, je öfter ein Programm auf 2-dimensionale Felder zugreift.

In folgenden Fällen ist es erforderlich, auf die Daten eines 2-dimensionalen Felds so zuzugreifen, als wäre es 1-dimensional deklariert:

- Im PC, wenn die Daten eines 2D-Felds vom ADwin-System dorthin übertragen und verarbeitet werden.  
Umgekehrt können Daten eines 1D-Felds vom PC ins ADwin-System übertragen werden, auch wenn das Zielfeld in ADbasic 2-dimensional deklariert ist.
- Innerhalb eines Library-Moduls (`LIB_SUB`, `LIB_FUNCTION`), dem ein 2D-Feld als Argument übergeben wird.

Bei der beschriebenen Art des Zugriffs ist die Reihenfolge der Daten im Speicher von Bedeutung. Als Beispiel sei ein 2D-Feld deklariert mit

```
DIM DATA_1[3][2] AS FLOAT
```

Im Datenspeicher werden diese 3×2 Elemente linear abgelegt. Die folgende Tabelle zeigt, mit welchem Element-Index der 1D-Zugriff auf die Elemente des Beispiel-Felds erfolgt.

Feld-Index 2D	[1][1]	[1][2]	[2][1]	[2][2]	[3][1]	[3][2]
Feld-Index 1D	[1]	[2]	[3]	[4]	[5]	[6]
Speicherstelle	n	n+1	n+2	n+3	n+4	n+5

Dem Element `DATA_1[3][1]`, das im Hauptprogramm verwendet wird, würde z.B. in einem Library-Modul das 5. Element des übergebenen Felds entsprechen:

```
REM Hauptprogramm
DATA_1[3][1] = 17
setpar1(DATA_1)           'setzt PAR_1 = 17

REM Library-Modul
LIB_SUB setpar1(BYREF array[] AS LONG)
    PAR_1 = array[5]       'entspricht DATA_1[3][1]
LIB_ENDSUB
```

Beachten Sie bitte: Diese Art des Zugriffs ist nur in den oben genannten beiden Fällen zulässig. In allen anderen Fällen muss die 2-dimensionale Schreibweise angewendet werden.



Für die Zuordnung von 2D-Elementen zu 1D-Elementen gilt allgemein:

$$\text{DATA\_n}[i][j] \hat{=} \text{DATA\_n}[s \cdot (i - 1) + j]$$

wobei `s` die 2. Dimension von `DATA_n` bei der Deklaration ist. Im obigen Beispiel ist `s=2`.

### 3.3.4 Die Datenstruktur FIFO

Für Anwendungen, in denen große Datenmengen kontinuierlich übertragen werden sollen, empfehlen wir die Verwendung eines globalen Felds `DATA_n` mit der Datenstruktur FIFO: Ein Ringspeicher, der nach dem Prinzip „First In, First Out“ verwaltet wird.

In einem Ringspeicher werden die Daten auf besondere Weise verwaltet. Sie können sich die Daten als eine Kette vorstellen, an deren Ende Sie neue Daten einzeln anhängen und an deren Spitze Sie einzelne Daten abholen können. Sie greifen also – im Unterschied zu einem „einfachen“ Feld – nicht auf beliebige Feldelemente zu, sondern immer nur auf das erste oder letzte (über je einen Datenzeiger). Dadurch lesen Sie die Daten in der gleichen Reihenfolge aus, wie sie in das Feld geschrieben wurden (=First In, First Out).

Sie können nur eindimensionale globale Felder (`DATA_n`) als Ringspeicher deklarieren. Mögliche Datentypen sind **LONG** oder **FLOAT**.




### Beispiel

```
DIM DATA_5[1000] AS LONG AS FIFO
```




Diese Anweisung deklariert das globale Feld mit der Nummer 5 als FIFO-Ringspeicher mit 1000 Elementen vom Typ **LONG**.

Beachten Sie bitte, dass Sie ein globales Feld **DATA** nicht gleichzeitig als „einfaches“ Feld und als Ringspeicher verwenden können. 

Da ein FIFO-Feld eine endliche (von Ihnen deklarierte) Zahl von Elementen besitzt, bildet die Kette aus benutzten und unbenutzten Feldelementen einen Ring, den Ringspeicher. Die Datenzeiger auf das erste und das letzte benutzte Feldelement werden automatisch verwaltet, wenn Sie dem Feld einen neuen Wert zuweisen oder einen Wert auslesen.

Nach der Deklaration eines FIFO-Felds sollten Sie die Zeiger mit dem Befehl **FIFO\_CLEAR** initialisieren.

Aus der Ringstruktur des FIFO-Felds ist ersichtlich, dass die Spitze der Datenkette das Datenende „überholen“ kann. Dies ist möglich, wenn Sie Daten schneller in den FIFO schreiben als Sie sie auslesen. Dadurch werden alte gespeicherte Daten überschrieben und gehen somit verloren. 

Auf ein bestimmtes FIFO-Feld greifen Sie zu, indem Sie dessen Feldnamen (mit der entsprechenden Feldnummer) angeben.

### Beispiel



```
DIM DATA_5[1000] AS LONG AS FIFO
DATA_5 = 95           'Schreibt in den FIFO mit der
                       'Nummer 5 den Wert 95.
PAR_7 = DATA_5       'Liest einen Wert aus dem FIFO und
                       'speichert ihn in der globalen
                       'Variablen PAR_7
```

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.

**Beispiel**

```
DIM free,used,value1 AS LONG
DIM DATA_1[1000] AS LONG AS FIFO
REM Sind noch Elemente zum Beschreiben frei?
free = FIFO_EMPTY(1)
IF (free > 0) THEN
    DATA_1 = wert1
ENDIF
REM Können noch benutzte Elemente gelesen werden?
used = FIFO_FULL(1)
IF (used > 0) THEN
    PAR_7 = DATA_1
ENDIF
```

### 3.3.5 Strings

Mit Zeichenketten (Strings) lassen sich Steuerzeichen und Texte z.B. von anderen Prozessüberwachungs-Geräten über eine RS-232-Schnittstelle zum ADwin-System transferieren, umwandeln und verarbeiten.

Zur String-Verarbeitung stehen eine Reihe von Befehlen zu Verfügung:

<b>ASC</b>	ASCII-Nummer eines Zeichens bestimmen
<b>CHR</b>	Zeichen zu einer ASCII-Nummer bestimmen
<b>FLOTOSTR</b>	Float-Wert in einen String wandeln
<b>FLO40TOSTR</b>	40 Bit Float-Wert in einen String wandeln
<b>LNGTOSTR</b>	Long-Wert in einen String wandeln
<b>STRCOMP</b>	2 Strings auf Gleichheit prüfen
<b>STRLEFT</b>	Zeichen linksbündig aus einem String kopieren
<b>STRLEN</b>	Länge eines Strings bestimmen
<b>STRMID</b>	Zeichenfolge aus einem String kopieren
<b>STRRIGHT</b>	Zeichen rechtsbündig aus einem String kopieren
<b>VALF</b>	String in einen Float-Wert wandeln
<b>VALI</b>	String in einen Long-Wert wandeln
<b>+</b> (String-Addition)	Operator, um Strings aneinander zu hängen

Für die meisten Befehle müssen Sie die Library-Datei `<STRING.LI*>` einbinden (das Zeichen `*` bezeichnet den Prozessortyp: `9` für T9, `A` für T10, `B` für T11). Die Library-Datei finden Sie nach der Installation im Library-Verzeichnis (Standard: `<C:\ADwin\ADbasic\LIB>`).

Eine String-Variable hat einen ähnlichen Aufbau wie ein Feld, d.h. jedes Feld-element enthält ein Zeichen. Die Dimensionierung eines Strings für 5 Zeichen lautet:

```
IMPORT STRING.LI9
DIM text[5] AS STRING
```

Durch die Dimensionierung wird im Speicher ein Feld für den String reserviert, das wie folgt aufgebaut ist:

```
text[1]   Länge des Strings in Zeichen (5)
text[2]   Das 1. Zeichen
text[3]   Das 2. Zeichen
text[4]   Das 3. Zeichen
text[5]   Das 4. Zeichen
text[6]   Das 5. Zeichen
text[7]   Das String-Ende-Zeichen, abschl./term. Null (00h)
```

Jedes Element belegt im Speicher 4 Bytes. Das erste und das letzte Element des Strings reserviert der *ADbasic*-Compiler automatisch.

Nach der Dimensionierung sind die Elemente nicht initialisiert. Ihren Inhalt erhalten sie erst bei einer Zuweisung.

### Normale Zuweisung

Die übliche Zuweisung von Werten an eine String-Variable erfolgt durch eine Zeichenkette in doppelten Hochkommas (") und wird vorwiegend für Texte verwendet. Für jedes der Zeichen legt *ADbasic* im Speicher die entsprechende ASCII-Nummer ab (Tabelle im Anhang).



### Beispiel

```
text = "HALLO"
```

Element-Index	Speicher-inhalt	Bedeutung
text[1]	05h	Länge des Strings in Zeichen (5)
text[2]	48h	ASCII-Nummer für "H"
text[3]	41h	ASCII-Nummer für "A"
text[4]	4Ch	ASCII-Nummer für "L"
text[5]	4Ch	ASCII-Nummer für "L"
text[6]	4Fh	ASCII-Nummer für "O"
text[7]	00h	String-Ende-Zeichen

Die Wertzuweisung in Hochkommas sollten Sie nur für die Zeichen mit den ASCII-Nummern 20h...7Fh (= sichtbare Zeichen im einfachen ASCII-Zei-

chensatz, siehe auch Anhang) einsetzen, mit Ausnahme der folgenden Zeichen. Diese müssen Sie per Escape-Sequenz zuweisen (s.u.):

- einfaches Hochkomma ('): \x39
- doppeltes Hochkomma ("): \x34
- Backslash (\): \x5C

## Zuweisung per Escape-Sequenz

Wenn Sie nicht nur Texte verarbeiten, sondern auch Zahlenwerte oder Steuerzeichen in einen String einbinden möchten, sollten Sie dafür Escape-Sequenzen einsetzen. Mit jeder Escape-Sequenz übergeben Sie eine einzelne Zahl an den *ADbasic*-Compiler, der sie unverändert im Speicher ablegt. Geben Sie die Escape-Sequenz innerhalb von doppelten Hochkommas in der Schreibweise "\xhh" an, wobei hh die zu übergebende Zahl in hexadezimaler Schreibweise ist. Jede solche Sequenz besteht aus genau 4 Zeichen.

## Beispiel

```
text = "\x48\x41\x4C\x4C\x4F"
```

Der Speicherinhalt ist identisch mit dem vorherigen Beispiel.

Da Sie auf diese Weise einem String beliebige Zahlen von 00h bis FFh hinzufügen können, eignen sich Escape-Sequenzen besonders für die Zuweisung nicht darstellbarer Zeichen (wie Line Feed, Carriage Return, ...).

Zusätzlich zur Schreibweise \xhh gibt es für häufig verwendete (Steuer-) Zeichen spezielle Escape-Sequenzen:

Sequenz	ASCII-Nummer	Bedeutung
\\	5C	Backslash (\)
\t	09	Tabulator (TAB)
\n	0A	Line Feed / Zeilenvorschub (LF)
\r	0D	Carriage Return / Wagenrücklauf (CR)

Sie können bei der Zuweisung von Werten an eine String-Variable die oben vorgestellten Schreibweisen beliebig kombinieren.

## Beispiel

```
text = "HA\x4C\x4C"
```

Der Speicherinhalt ist wieder identisch mit dem bekannten Beispiel.



Sie dürfen in einen String kein String-Ende-Zeichen - z.B. mit der Escape-Sequenz `\x00` - einfügen (Beispiel: `text = "HA\x00LLO"`). Der *ADbasic*-Compiler verarbeitet dies zwar korrekt, jedoch können sich Fehler bei der weiteren Verarbeitung des Strings ergeben.

### Nicht empfohlene Arten der Zuweisung

Sie können auch Zeichen mit den ASCII-Nummern `00h...1Fh` oder `80h...FFh` in eine String-Zuweisung einfügen, z.B. Umlaute wie `[ß]`, `[Ö]` oder `[?]`, mit "copy and paste" aus anderen Anwendungen oder durch die Tastenkombination `[ALT] + Nummer`. Wir empfehlen stattdessen ausdrücklich die Zuweisung per Escape-Sequenz!

Der Compiler ist zwar in der Lage, solche Zeichen zu verarbeiten. Die Zeichen haben aber entweder keine eindeutige ASCII-Nummer (länderspezifisch) oder sie können - möglicherweise schon im *ADbasic*-Editor - unerwünschte Aktionen (Zeilenumbruch o.ä.) und Programmfehler verursachen.



Wir empfehlen Ihnen, jegliche Steuer- und Sonderzeichen nur als Escape-Sequenzen in einen String einzufügen.

## 3.4 Berechnungsausdrücke

### 3.4.1 Auswertung von Operatoren

Ein Berechnungsausdruck ist das, was Sie einer Variablen zuweisen oder einem Befehl als Argument übergeben. Er besteht aus einer beliebigen Kombination von:

- einfachen Daten: Konstante, Variable oder Feldelement
- Operatoren, die auf Argumente angewendet werden, die wieder Berechnungsausdrücke sind.

Für die Auswertung eines Berechnungsausdruck ist es wesentlich, in welcher Reihenfolge die Operatoren angewendet werden. Hierzu werden die Operatoren in Kategorien eingeteilt, die nach Prioritäten geordnet sind: Eine Kategorie höherer Priorität wird vor einer Kategorie niedriger Priorität bearbeitet (siehe Abb. 18).

Bitte beachten Sie, dass die automatische Typkonvertierung in manchen Fällen die Auswertung des Berechnungsausdrucks beeinflusst (siehe Seite 65).

Operator	Kategorie
" "	Begrenzer von Zeichenketten
Kennwort in <i>ADbasic</i>	Befehl, Funktion, Variable, etc.
=	Zuweisung
( )	Klammern
-	Vorzeichen einer <i>Konstanten</i>
^	Potenz
* /	Punkt-Operatoren
+ -	Strich-Operatoren
AND OR XOR	Binär-Operatoren
< > =	Vergleichs-Operatoren
AND OR	Boolesche Operatoren

Abb. 18 – Prioritäten von Operatoren-Kategorien  
(von oben nach unten absteigende Priorität)

Wenn sich 2 Operatoren in der gleichen Kategorie befinden (oder gleiche Operatoren vorhanden sind), dann verarbeitet der Compiler diese wie sie erscheinen, von links nach rechts.

### Beispiel

```
var = PAR_1 + PAR_2 * PAR_1^3 / 4
```

entspricht

```
var = PAR_1 + (PAR_2 * (PAR_1^3) / 4)
```



Wenn Sie Variablen mit negativem Vorzeichen verwenden, kann dies in manchen Fällen zu unerwarteten Ergebnissen führen, die Sie durch Klammersetzung vermeiden.



### Beispiel

```
var = 1/-x      'nicht empfohlene Schreibweise
var = 1/ (-x)   'Korrekt: Negativer Umkehrwert
```



### 3.4.2 Typkonvertierung

Sie können in *ADbasic* im allgemeinen die Datentypen **LONG** und **FLOAT** (siehe auch Kapitel 3.2.3 „Datentypen“) gemeinsam verwenden, ohne auf

passende Datentypen zu achten. Die Daten vom Typ **LONG** werden, falls erforderlich, automatisch in den Typ **FLOAT** konvertiert.

Beachten Sie bitte folgende Besonderheiten:

- Abschneiden von Nachkommastellen



Wenn ein Fließkomma-Wert einer ganzzahligen Variablen zugewiesen wird, dann werden die Nachkommastellen abgeschnitten und gehen verloren.

- Konvertierung *aller* ganzzahligen Werte

Wenn in einem Berechnungsausdruck ein Fließkomma-Wert enthalten ist, werden *vor* der Auswertung des Ausdrucks *alle* ganzzahligen Werte des Ausdrucks automatisch konvertiert. Dies gilt auch dann, wenn ein ganzzahliger Berechnungsausdruck

- einer Fließkomma-Variablen zugewiesen wird oder
- als Argument eines *ADbasic*-Befehls dient, der einen Fließkomma-Wert erwartet.



### Beispiel

```
PAR_1 = 2 / 4 * 3      'Ergebnis: PAR_1=0, weil 2/4 = 0
```



Nachkommastellen werden bei rein ganzzahligen Berechnungen immer abgeschnitten und gehen verloren.

aber:

```
FPAR_1 = 2 / 4 * 3      'Ergebnis: FPAR_1=1,5
PAR_1 = 2 / 4.0 * 3     'Ergebnis: PAR_1=1 (abgeschnitten!)
```

Hier erzwingen die Fließkomma-Variable `FPAR_1` und der Fließkomma-Wert `4.0` die Konvertierung aller ganzzahligen Werte.

- Automatische Konvertierung verhindern



Auch das Setzen von Klammern verhindert nicht die automatische Konvertierung in **FLOAT**. Sollen Berechnungen unbedingt in **LONG** durchgeführt werden, so müssen Sie hierfür eine eigene Zeile programmieren.



## Beispiel



```
PAR_1 = 2
PAR_2 = 5
REM hier wird konvertiert:
FPAR_3 = (PAR_2 / PAR_1) + 0.2 'FPAR_3 = 2.7
REM hier dagegen nicht:
PAR_9 = PAR_2 / PAR_1 'PAR_9 = 2 (abgeschnitten)
FPAR_4 = PAR_9 + 0.2 'Ergebnis: FPAR_1 = 2.2
```

## – Konvertierung von Argumenten

Folgende Ausdrücke werden immer getrennt ausgewertet (und ggf. konvertiert):

- Jeder Parameter eines Befehls.  
Zusätzlich kann durch den Datentyp des Parameters ein Abschneiden von Nachkommastellen auftreten (Parameter-Datentyp siehe jeweilige Befehlsbeschreibung).
- Jedes an Funktionen und Unterprogramme übergebene Argument.
- Jede einzelne Bedingung (Boolescher Ausdruck) bei **IF...THEN** oder **DO...UNTIL**, die mit **AND** und **OR** verknüpft werden kann.

## Beispiel



```
PAR_1 = 2 : FPAR_2 = 5.5

REM Beide Bedingungen sind erfüllt, PAR_1 wird nicht in
REM FLOAT gewandelt: PAR_3 = 1
IF ((PAR_1 / 4 * 3 = 0) AND (FPAR_2 * 1.1 > 5.5)) THEN
    PAR_3 = 1
ENDIF

REM Die Bedingung mit FLOAT beeinflusst die
REM LONG-Berechnung nicht: PAR_3 = 0
IF (FPAR_2 * 1.1 > 5.5) THEN PAR_3 = PAR_1 / 4 * 3
```

### 3.5 Abfragen, Schleifen und Module

Wenn Sie umfangreiche Programme schreiben, können Sie diese mit in *ADbasic* gut strukturieren. Ihnen stehen folgende Strukturelemente zur Verfügung:

- Kontrollstrukturen verkürzen aufwändige Abschnitte.
  - Schleifen für oft wiederholte Abschnitte:  
**DO ... UNTIL** oder  
**FOR ... TO ... {STEP} ... NEXT.**
  - Abfragen für fallweise Unterscheidungen:  
**IF ... THEN ... {ELSE} ... ENDIF** oder  
**SELECTCASE ... ENDSELECT.**
- Unterprogramm- und Funktions-Makros ermöglichen Ihnen, häufig benutzte Programmabschnitte zu definieren als
  - Unterprogramm-Makros mit **SUB ... ENDSUB**
  - Funktions-Makros mit **FUNCTION ... ENDFUNCTION**
- Sammlungen von Quelltext-Abschnitten und Programm-Makros in Include-Dateien, die Sie komplett in Ihren Quelltext einbinden mit  
**#INCLUDE** filename.inc
- Bibliotheken (Libraries) von kompilierten Funktionen und Unterprogrammen, die Sie nach Bedarf in Ihren Quelltext einbinden:
  - Library-Unterprogramme: **LIB\_SUB ... LIB\_ENDSUB**
  - Library-Funktionen: **LIB\_FUNCTION ... LIB\_ENDFUNCTION**

Sie finden nähere Erklärungen und Beispiele der Befehle in Kapitel 6 „Befehlsreferenz“.

#### 3.5.1 Unterprogramm- und Funktions-Makros

Die Syntax von Unterprogramm- und Funktions-Makros ist sehr einfach, Sie müssen lediglich die Begriffe **SUB ... ENDSUB** und **FUNCTION ... ENDFUNCTION** wie eine Klammer um die jeweiligen Programmabschnitte legen. Funktionen geben – im Unterschied zu Unterprogrammen – einen Wert zurück.

Unterprogramme und Funktionen definieren Makros, d.h. deren vollständiger Anweisungsblock wird (noch vor dem Kompilieren) an der aufrufenden Stelle in den Quelltext eingefügt.

Makros erhöhen die Übersichtlichkeit Ihres Quelltextes. Beachten Sie aber auch, dass jeder Aufruf die erzeugte Binärdatei vergrößert. Sie können alternativ auch Library-Funktionen oder -Unterprogramme verwenden (siehe unten).

Sie finden nähere Informationen zum Aufbau von Makros in der Befehlsreferenz (Seite 151: FUNCTION ... ENDFUNCTION; Seite 223: SUB ... END-SUB).

### 3.5.2 Include-Dateien

Sie können eine Sammlung von Quelltext-Abschnitten erstellen und in einer sogenannten „Include-Datei“ speichern. Solche Dateien (bzw. den darin enthaltenen Quelltext) können Sie sehr einfach mit dem Befehl **#INCLUDE** in Ihren aktuellen Quelltext einbinden.

Der Inhalt von Include-Dateien unterliegt den gleichen Regeln wie der von normalen Quelltext-Dateien, vorwiegend enthalten sie jedoch nur Unterprogramm- und Funktions-Makros.

Zum Erstellen einer Include-Datei geben Sie, wie bei einer „normalen“ *ADbasic*-Datei, den gewünschten Quelltext ein und speichern diesen mit „File / Save as“ als Dateityp „Include file \*.inc“.

Je nach enthaltenem Quelltext müssen Sie darauf achten, an welcher Stelle Sie die Include-Datei in Ihren aktuellen Quelltext einbinden, damit die korrekte Programmstruktur gewahrt bleibt. Wenn die Include-Datei Unterprogramm- und Funktions-Makros enthält, muss sie beispielsweise vor dem Abschnitt **INIT**: oder nach dem Abschnitt **FINISH**: eingebunden werden.

Sie können Include-Dateien auch in Quelltexte von Library-Dateien oder anderen Include-Dateien einbinden.

Die Include-Dateien, die mit *ADbasic* geliefert werden, enthalten nur Unterprogramm- und Funktions-Makros, die Befehle für den Hardware-Zugriff definieren. Aus diesem Grund ist die korrekte Stelle für das Einbinden dieser Dateien der Anfang des Quelltexts (siehe Seite 44).



### 3.5.3 Bibliotheken (Libraries)

In einer Bibliothek können Sie kompilierte Library-Unterprogramme und -Funktionen (Module) zusammenfassen. Mit dem Befehl **IMPORT** binden Sie diejenigen Module einer Library in einen Prozess ein, die dort tatsächlich aufgerufen werden.

Die Library-Module sind den Funktions- und Unterprogramm-Makros ähnlich. Sie erstellen diese in einem Quelltext mit den Befehlen **LIB SUB ... LIB ENDSUB** und **LIB FUNCTION ... LIB ENDFUNCTION** und kompilieren daraus die Library-Datei mit „Build / Make lib file“.

Wenn Sie einen Quelltext kompilieren, in dem eine Library importiert wird, werden nur die im Quelltext aufgerufenen Library-Module zu der Binärdatei hinzugefügt. Ein mehrfacher Aufruf im Quelltext vergrößert die Binärdatei

nicht (im Gegensatz dazu siehe auch Kapitel 3.5.1 „Unterprogramm- und Funktions-Makros“), jedoch benötigt jeder einzelne Aufruf auch zusätzliche Ausführungszeit.



Beachten Sie bitte, dass ein Library-Modul kein Library-Modul innerhalb der gleichen Library-Datei aufrufen kann. Wir empfehlen Ihnen, statt dessen Funktions- und Unterprogramm-Makros zu verwenden. Alternativ können Sie auch eine zusätzliche Library erstellen (oder mehrere).

Wenn Sie Libraries verschachtelt einbinden (d.h. in einer Library eine weitere Library einbinden), müssen Sie im aufrufenden Quelltext die Libraries aller Schachtelungsstufen einbinden (siehe Abb. 19), sonst erhalten Sie eine Fehlermeldung des Compilers.



Rekursive Aufrufe von Library-Funktionen oder Unterprogrammen sind nicht erlaubt.

Sie finden nähere Informationen zum Aufbau von Library-Modulen in der Befehlsreferenz (Seite 164: **LIB\_FUNCTION ... LIB\_ENDFUNCTION**; Seite 168: **LIB\_SUB ... LIB\_ENDSUB**).

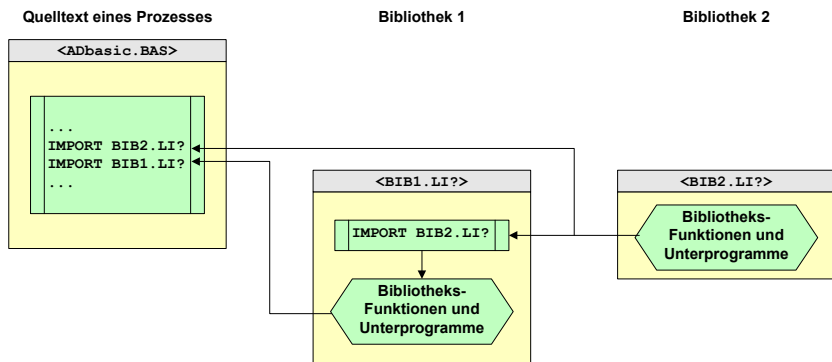


Abb. 19 – Verschachteltes Einbinden von Bibliotheken

## 4 Prozesse optimieren


Ihr *ADwin*-System ist dafür ausgelegt, Ihre Regel-, Steuer- und Messaufgabe schnell und präzise auszuführen. Je nach Anforderung kann es erforderlich werden, dass Sie Ihre *ADbasic*-Programm für eine schnellere Bearbeitungszeit optimieren.

Im folgenden zeigen wir beispielhaft, mit welchen Mitteln und an welchen Stellen Sie bei einer Optimierung ansetzen können. Die Vorgehensweise hängt von vielen Faktoren ab und ist daher auf den Einzelfall abzustimmen. Wir verweisen an dieser Stelle auch auf das „*ADbasic* Tutorial“, in dem Sie weitere Beispiele für die Optimierung von Prozessen finden.

### 4.1 Bearbeitungszeit messen

Als Grundlage für eine Optimierung ist es wichtig, die Bearbeitungszeit eines Prozesszyklus oder von Programmabschnitten zu messen. Sie verwenden hierzu die internen Zähler Ihres *ADwin*-Systems.

Der Prozessor des *ADwin*-Systems verfügt über zwei interne Zähler, jeweils einen für hochpriorie und für niederpriorie Prozesse, die in unterschiedlichen Zeittakten hochgezählt werden. Mit dem Befehl **READ\_TIMER** () können Sie den aktuellen Zählerstand feststellen; es wird automatisch der Zähler ausgelesen, der der Priorität des laufenden Prozesses entspricht.

Nach dem Einschalten der Spannungsversorgung werden beide Zähler auf den Wert 0 (Null) gesetzt und anschließend in festen Zeittakten (siehe Abb. 21) kontinuierlich hochgezählt. 

Sie messen die Bearbeitungszeit von Programmen als Zeitdifferenz. Im folgenden Beispiel wird die Bearbeitungszeit eines zeitkritischen Abschnitts (abzüglich eines Offsets) in der globalen Variablen **PAR\_1** gespeichert.

Sie erhalten den Offset, wenn Sie die beiden **READ\_TIMER** () -Zeilen nacheinander – ohne dazwischen liegende Programmzeilen – ausführen und die Differenz dieser Werte bilden. Der Offset muss für das betrachtete Programm nur ein Mal ermittelt werden.

**Beispiel**

REM WICHTIG: Keinesfalls Float-Variablen verwenden!

```
DIM t1, t2 AS LONG
```

EVENT:

```
...
t1 = READ_TIMER()
...
t2 = READ_TIMER()
PAR_1 = t2 - t1 - 4
```

'zeitkritischer Abschnitt  
'Bearb.zeit des zeitkritischen  
'Abschnitts in Zeittakten  
' (Offset = 4 Zeittakte)

Wenn `PAR_1` im obigen Beispiel den Wert 37 erhält, hat der zeitkritische Abschnitt bei einem hochprioren Prozess  $37 \times 25\text{ns} = 925\text{ns}$  benötigt.

Sie können die Zeitmessung auch benutzen, um beispielsweise die Zeitdifferenz zwischen zwei externen Events zu messen. Im Beispiel wird diese bei jedem Aufruf in der globalen Variablen `PAR_1` gespeichert.

**Beispiel**

```
DIM oldtime, time AS LONG
```

INIT:

```
oldtime = READ_TIMER()
```

EVENT:

```
time = READ_TIMER()
PAR_1 = time - oldtime
oldtime = time
```

## 4.2 Verschiedene Tipps

### 4.2.1 Zugriff auf Hardware-Adressen

Viele Funktionen Ihres ADwin-Systems werden über dessen Steuer- und Datenregister kontrolliert. Diese Funktionen können Sie sehr schnell ausführen lassen, wenn Sie mit den Befehlen **PEEK** und **POKE** *direkt* auf die entsprechenden Register zugreifen. Direkt bedeutet, dass Sie im Prozesszyklus die Adressen nicht berechnen, sondern als konstante Werte übergeben: Sie sparen die Berechnungszeit ein.

Die Adressen der Steuer- und Datenregister finden Sie in Ihrem jeweiligen Hardware-Handbuch.

## 4.2.2 Konstanten anstelle von Variablen

Eine Berechnung kann deutlich schneller ausgeführt werden, wenn Sie Werte als Konstanten und nicht mit Variablen angeben.

### Beispiel



```
PAR_1 = SQRT(PAR_2)      'mit PAR_2=17
PAR_1 = SQRT(17)
```

Für die erste Berechnung muss zur Laufzeit der Wert der Variablen `PAR_2` ermittelt, die Wurzel berechnet und `PAR_1` zugewiesen werden.

In der zweiten Berechnung kann schon der Compiler den Wert ermitteln. Zur Laufzeit wird dieser nur noch zugewiesen.

## 4.2.3 Schnellere Messfunktion

Mit dem Befehl **ADC** wird eine A/D-Wandlung für einen Kanal mit bestimmter Verstärkung vorgenommen. Der Befehl ist sehr einfach gehalten, um Ihnen die Anwendung zu erleichtern, denn er fasst mehrere Ablaufschritte zusammen (siehe Kapitel 6.3 „ADwin-Gold und ADwin-light-16“, Seite 235 oder „ADwin-Pro Programmierung in ADbasic“).

Es gibt verschiedene Situationen, in denen Sie mit den einzelnen Ablaufschritten einen schnelleren Ablauf erzielen können als mit dem Befehl **ADC**.

Beispielsweise wird mit dem Befehl **ADC** nicht ausgenutzt, dass sich auf einem *ADwin-Gold*-System zwei ADC befinden, die gleichzeitig zwei verschiedene Kanäle konvertieren können. Dies geschieht im folgenden Beispiel:

### Beispiel



```
REM Beispiel für Gold
REM Beide Multiplexer der ADC auf Kanal 1 setzen
SET_MUX(000000b)
...
START_CONV(11b)      'Einschwingzeit abwarten
WAIT_EOC(11b)        'Wandlung an beiden ADC starten
PAR_1 = READADC(1)    'Wandlungsende abwarten
PAR_2 = READADC(2)    'Auslesen von ADC1
                     'Auslesen von ADC2
```

Auf dem *ADwin-light-16*-System befindet sich nur ein ADC.



#### 4.2.4 Wartezeit genau einstellen

Mit einer Wartezeit kann man leicht einen genauen Zeitabstand zwischen 2 Befehlen einstellen, z. B. um zwischen **SET\_MUX** und **START\_CONV** die Einschwingzeit des Multiplexers zu überbrücken. Beachten Sie auch Kapitel 4.2.5 „Wartezeiten nutzen“.

Der Befehl zum Einstellen der Wartezeit ist abhängig vom Prozessortyp:

- Prozessoren T9 und T10:

Der Befehl **SLEEP** stellt die genaue Wartezeit ein: Der Prozessor stoppt für die eingestellte Zeit, so dass der folgende Befehl entsprechend verzögert startet.

Das Überbrücken einer Multiplexer-Einschwingzeit von 14µs auf einem Pro I-Modul könnte folgendermaßen aussehen:

```
SET_MUX(2,00000b)      'Mux auf Kanal 1 setzen
REM Hier z.B. wird eine Berechnung eingeschoben, die
REM 8µs dauert, um die freie Prozessorzeit zu nutzen.
SLEEP(60)              'Restliche Zeit (6µs) bis 14µs warten
START_CONV(2)          'Wandlung starten
```

- Prozessor T11:

Für die Wartezeit gibt es 3 mögliche Befehle:

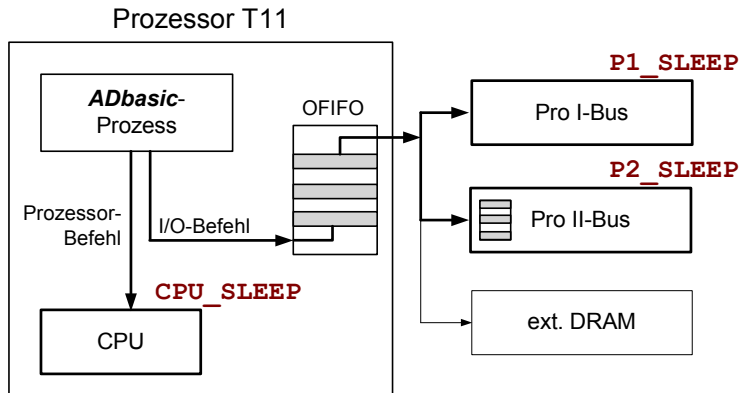
- **P1\_SLEEP** lässt den Pro I-Bus warten.
- **P2\_SLEEP** lässt den Pro II-Bus warten.
- **CPU\_SLEEP** lässt den Prozessor warten (entspricht **SLEEP**).

Wenn die Wartezeit einen Zeitabstand zwischen Ein- / Ausgabebefehlen für Pro I-Module überbrücken soll, ist **P1\_SLEEP** der richtige Befehl; für Pro II-Module ist es **P2\_SLEEP**. Der Befehl **CPU\_SLEEP** kann nur in wenigen Fällen sinnvoll eingesetzt werden.

Das Überbrücken einer Multiplexer-Einschwingzeit von 14µs auf einem Pro I-Modul könnte folgendermaßen aussehen:

```
SET_MUX(2,00000b)      'Mux auf Kanal 1 setzen
P1_SLEEP(1400)          '14µs auf dem Pro I-Bus warten.
                        'Beachten Sie die Zeiteinheit.
START_CONV(2)          'Wandlung starten
REM Die Berechnung folgt erst jetzt und wird beim T11
REM automatisch parallel zu den I/O-Befehlen durchgeführt
REM Achtung: Verwenden Sie für die Berechnung möglichst nur
REM Variablen aus dem internen Speicher. Anderenfalls ist es
REM möglich, dass die Berechnung doch erst ausgeführt wird,
REM wenn die I/O-Befehle abgearbeitet sind.
```





Warum gibt es mehrere Befehle für die Wartezeit? Der Prozessor T11 bearbeitet Prozessor-Befehle und I/O-Befehle<sup>2</sup> quasi-parallel (siehe Skizze oben). Das ist besonders schnell, führt aber auch zu paralleler, also getrennter Zeitsteuerung und damit zu den 3 Befehlen für die Wartezeit.

Die quasi-parallele Bearbeitung wird durch den 5-stufigen Zwischenspeicher **OFIFO** möglich: Das Betriebssystem legt die I/O-Befehle in **OFIFO** ab (falls dort noch Platz ist) und kann sofort den nächsten Befehl verarbeiten. Im obigen Beispiel werden auf diese Weise die Befehle **SET\_MUX**, **P1\_SLEEP** und **START\_CONV** an **OFIFO** übergeben; die nachfolgende Berechnung kann bereits in der CPU ablaufen, während z. B. die Wartezeit auf dem Pro I-Bus noch aktiv ist.

Beachten Sie bitte: Eine Berechnung, die quasi-parallel in der CPU ablaufen soll, darf nur Variablen aus dem internen Speicher verwenden. Ein Zugriff auf das externe DRAM, den üblichen Speicherort für Felder, ist für das Betriebssystem ein I/O-Befehl und durchläuft daher den Zwischenspeicher **OFIFO**.



- 
2. I/O-Befehle sind Befehle, die über den Zwischenspeicher **OFIFO** auf externe Geräte zugreifen. Externe Geräte sind (für die CPU) die Module am Pro I- oder Pro II-Bus und der externe Speicher.

#### 4.2.5 Wartezeiten nutzen

Manche Befehle erfordern nach ihrem Aufruf eine bestimmte Wartezeit, ohne dabei den Prozessor zu nutzen. Diese Zeit können Sie für andere Berechnungen nutzen.

Solche Befehle sind **SET\_MUX** und **START\_CONV**, nach denen Wartezeiten nötig sind, um das Einschwingen des Multiplexers und die Konvertierung der ADC abzuwarten. Während dieser Wartezeit ist aber der Prozessor nicht beschäftigt, könnte also andere Aufgaben übernehmen.

Genauere Angaben über die erforderlichen Wartezeiten bei der Datenwandlung finden Sie in Ihrem Hardware-Handbuch.

Als praktische Anwendung wird das Beispiel aus dem Abschnitt „Schnellere Messfunktion“ nun so erweitert, dass in einem Prozesszyklus 2 Messungen an je 2 ADC durchgeführt werden. Dadurch können Sie im Vergleich zum Befehl **ADC** in der gleichen Zeit die 4fache Zahl an Messungen durchführen.

Der wesentliche Effekt beruht darauf, dass die einzelnen Schritte der 2 Messungen nicht nacheinander folgen, sondern das Setzen des Multiplexers in die Wartezeit der jeweils anderen Messung verschoben wird. Die Abläufe der beiden Messungen überlagern sich: Auf den Wandlungsstart für die Kanäle 1+2 folgt das Setzen des Multiplexers für die Kanäle 3+4.



#### Beispiel

```
REM Beispiel für Gold Rev. B (für T11 nicht geeignet)
INIT:
    SET_MUX(000000b)      'Mux für 1. Messung setzen, Kanäle 1+2
    SLEEP(140)            '14 µs warten

EVENT:
    START_CONV(11b)       'Wandlung starten (Kanäle 1+2)
    SET_MUX(001001b)      'Mux setzen, Kanäle 3+4
    WAIT_EOC(11b)         'Wandlungsende abwarten (Kanäle 1+2)
    PAR_1 = READADC(1)     'Auslesen von ADC1, Kanal 1
    PAR_2 = READADC(2)     'Auslesen von ADC2, Kanal 2

    START_CONV(11b)       'Wandlung starten (Kanäle 3+4)
    SET_MUX(000000b)      'Mux setzen, Kanäle 1+2
    WAIT_EOC(11b)         'Wandlungsende abwarten (Kanäle 3+4)
    PAR_3 = READADC(1)     'Auslesen von ADC1, Kanal 3
    PAR_4 = READADC(2)     'Auslesen von ADC2, Kanal 4
```

Für die erste Messung ist nun erforderlich, den Multiplexer bereits im Abschnitt **INIT**: zu setzen, damit der erste Start der Wandlung definiert ist.

Achten Sie streng darauf, dass Sie die Mindest-Wartezeiten für das Einschwingen des Multiplexers und für die Konvertierung der ADC nicht unterschreiten, da sonst die A/D-Wandlung nicht funktioniert und falsche Ergebnisse liefert. Hinweise bietet das Kapitel 4.2.4 „Wartezeit genau einstellen“.



### 4.2.6 Optimierung mit dem Prozessor T11

Dieser Abschnitt beschreibt, wie Sie die besonderen Eigenschaften des Prozessors T11 nutzen, um einen Prozess schneller ablaufen zu lassen, insbesondere durch optimierten Speicherzugriff.

Falls Sie dennoch an die Leistungsgrenzen des Prozessors T11 stoßen, sind weitere Optimierungen möglich, allerdings nur im Zusammenhang mit Ihrer Anwendung. Bitte wenden Sie sich dazu an unseren Support (siehe Innenseite des Handbuchs).

#### Internen Speicher verwenden

Verwenden Sie für zeitkritische Vorgänge möglichst nur Variablen und Felder im internen Speicher (DM oder EM). Während normal deklarierte Variablen automatisch im internen Speicher abgelegt werden, müssen Felder (ob lokal oder global) dafür wie folgt deklariert werden:

```
DIM DataLocal[100] AS LONG AT DM_LOCAL  
DIM DATA_5[2000] AS FLOAT AT DM_LOCAL
```

Im Vergleich zum internen Speicher ist ein Zugriff auf den externen Speicher beim Prozessor T11 aus 2 Gründen wesentlich langsamer. Zum einen wird der Zugriff als I/O-Befehl über den Zwischenspeicher **OFIFO** übertragen (siehe Seite 75), wobei Verzögerungen auftreten können. Zum anderen ist die Verwaltung des externen Speichers langsamer als die Verwaltung des internen Speichers.

#### Zugriff auf externen Speicher

Verwenden Sie beim Datenzugriff auf den externen Speicher – soweit im Programm machbar – möglichst immer Datenblöcke, greifen Sie also nicht einzeln auf Werte zu. Bei blockweiser Datenübertragung kann der Prozessor eine beschleunigte Zugriffsart einsetzen, so dass er z.B. einen Block von 20 Werten schneller übertragen kann als 3 einzelne Werte.

Die Blockdatenübertragung ist beispielsweise sinnvoll einsetzbar, wenn viele Messwerte in kurzer Zeit eingelesen werden: Zunächst wird ein Datenpaket im schnellen, internen Speicher abgelegt. Sobald der Messvorgang eine nicht zeitkritische Phase erreicht, werden die Daten mit dem Befehl **MEMCPY** in den

externen Speicher übertragen und der interne Speicher steht wieder bereit für das nächste Datenpaket.

### 4.3 Debugging und Analyse

Die Bedienoberfläche beinhaltet mit Debug-, Timing- und Trace-Modus praktische Hilfsmittel zur Fehlersuche und Programmanalyse. Alle Modi werden über das Menü „Debug“ aktiviert (siehe Seite 26) und entfalten ihre Hilfsfähigkeit für solche Prozesse, die bei aktiviertem Modus kompiliert werden.



Beachten Sie: Das Aktivieren der Hilfs-Modi erzeugt zusätzlichen Programm-Code. Dadurch verlängert sich die Ausführungszeit des Programms und der Speicherbedarf wird erhöht – zum Teil beträchtlich. Nutzen Sie diese Hilfsmittel daher nur zum Entwickeln und Testen von Programmen.

#### 4.3.1 Laufzeitfehler erkennen (Debug-Modus)

Der Debug-Modus ist ein Hilfsmittel zum Aufspüren von folgenden Laufzeitfehlern in *ADbasic*-Programmen:

- Division durch Null
- Wurzel aus einer negativen Zahl
- Zugriff auf zu große / zu kleine Elementnummern eines Felds

Ohne Debug-Modus werden diese Laufzeitfehler ignoriert, d.h. das Ergebnis der entsprechenden Zeile ist undefiniert, wird aber dennoch für den weiteren Programmablauf verwendet. Dies kann, je nach Programm, unerwünschte Folgen haben, im schlimmsten Fall bis zum Absturz des *ADwin*-Systems.

Sie aktivieren die Option „Debug mode“ im Menü „Debug“; kompilieren Sie anschließend den zu prüfenden Quelltext. Sobald ein Laufzeitfehler auftritt, wird er automatisch im Fenster „Debug Errors“ angezeigt. Außerdem wird der Fehler so korrigiert, dass ein stabiler Betriebszustand erhalten bleibt.



Gefundene Fehler sollten in jedem Fall bereinigt werden; auch die automatische Fehlerkorrektur des Debug-Modus ist nur ein Hilfsmittel für die Fehlersuche, nicht für den Dauerbetrieb.

Die Einzelheiten zum Aktivieren und zur Anzeige der Laufzeitfehler sind im Abschnitt „Option „Debug mode““ beschrieben.

#### 4.3.2 Zeitverhalten prüfen (Timing-Modus)

Das *ADwin*-System ist so konzipiert, dass beim Auftreten eines Event-Signals für einen hochprioren Prozess (extern oder vom internen Zähler) sofort der entsprechende Prozesszyklus gestartet wird. Prozesse mit solch „gutem“

Zeitverhalten sind deterministisch und erledigen ihre Aufgaben genau zur vorbestimmten Zeit.

Das Prüfen des Zeitverhaltens von Prozessen erfordert einen gewissen Aufwand, zumal anschließend noch die Änderungen anstehen, um ein gutes Zeitverhalten zu erreichen. Dieser Aufwand lohnt sich dann, wenn höhere Frequenzen oder zusätzliche Aufgaben erforderlich sind, die Prozessor-Auslastung aber an die Grenzen stößt. Ein anderes Beispiel wäre, dass Prozesszyklen nicht genau genug zum vorbestimmten Zeitpunkt starten, die Messung dies aber erfordert.

Im Timing-Modus werden Informationen erzeugt, anhand derer Sie ausgewählte, hochpriore Prozesse auf „gutes“ Zeitverhalten prüfen können. Für diese Prozesse werden bei jedem Prozesszyklus 7 Kennwerte berechnet, die im Fenster „Timing Information“ angezeigt werden können.

Ein gutes Zeitverhalten von Prozessen zeichnet sich dadurch aus, dass folgende Situationen *nicht* (oder nur selten) auftreten:

1. Ein Event-Signal startet einen Prozesszyklus nicht sofort, sondern erst um eine gewisse (nicht genau definierte) Zeit später.
2. Ein Event-Signal startet überhaupt keinen Prozesszyklus, sondern geht „verloren“. Auch mehrere verlorene Event-Signale sind möglich.

Im ersteren Fall versucht das Betriebssystem, die Verzögerung wieder auszugleichen, indem es vorhandene Auslastungslücken des Prozessors nutzt, bis wieder alle Prozesszyklen zum vorbestimmten Zeitpunkt starten. Im letzteren Fall kann das Betriebssystem keinen Ausgleich schaffen: Es gehen tatsächlich Event-Signale, und damit auch Prozesszyklen verloren (siehe auch Kapitel 5.2.5 „Verschiedene Betriebszustände im Betriebssystem“, Seite 92).

Sie erreichen ein insgesamt optimales Zeitverhalten in 2 Schritten:

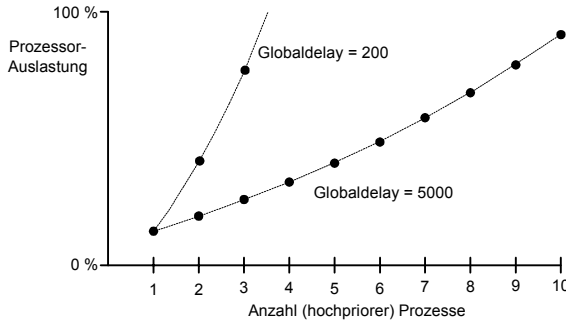
1. Anzahl und Priorität von Prozessen prüfen
2. Optimales Zeitverhalten eines Prozesses anstreben, insbesondere der hochprioren Prozesse.

### Anzahl und Priorität von Prozessen prüfen

In einem hochprioren Prozess sollten nur zeitkritische Aufgaben bearbeitet werden, alle anderen Aufgaben dagegen in einem oder mehreren niederprioren Prozessen (oder sogar nachträglich auf dem PC).

Verwenden Sie möglichst wenige, am besten nur einen einzigen hochprioren Prozess. Es ist oft möglich, mehrere Prozesse zu einem einzigen Prozess zusammenzufassen; bei gleichem Processdelay sollte man das auf jeden Fall

tun. Der Aufwand – insbesondere bei kürzerem Processdelay der Prozesse – ist lohnend, weil dann bei insgesamt gleicher Aufgabenstellung eine wesentlich geringere Prozessorauslastung erreichbar ist. Die unten stehende Grafik zeigt dies qualitativ.



Bei mehreren hochprioriten, zeitgesteuerten Prozessen ist es zudem nicht vermeidbar, dass immer wieder einzelne Prozesszyklen verzögert starten (außer die Processdelays sind ganzzahlige Vielfache voneinander).

### Optimales Zeitverhalten eines Prozesses

Das optimale Zeitverhalten hat ein hochprioriter Prozess, wenn er folgende Merkmale erfüllt:

- Alle Prozesszyklen des Prozesses haben etwa die gleiche Bearbeitungsdauer
- Die Bearbeitungsdauer für einen Prozesszyklus ist möglichst klein.
- Das Processdelay des Prozesses ist größer als die längste Bearbeitungsdauer aller Prozesszyklen.

Dabei muss die insgesamte Prozessor-Auslastung durch hochprioriere Prozesse den niederpriorien und dem Kommunikationsprozess genügend Prozessor-Zeit für ihre Arbeit lassen.

Um Informationen über das Zeitverhalten von interessanten Prozessen zu bekommen, gehen Sie vor wie folgt:

1. Aktivieren Sie die Timing-Option mit `Debug ▶ Enable timing analyzer.`
2. Kompilieren (und starten) Sie Ihren *ADbasic*-Quelltext.

Zu jedem Quelltext, den Sie bei aktiver Timing-Option kompilieren, werden automatisch Informationen über das Zeitverhalten erzeugt. Betrachten Sie wegen der zusätzlichen Berechnungszeiten nur möglichst wenige Prozesse auf einmal, um das Zeitverhalten insgesamt nicht zu stark zu verändern (s.u.).

3. Deaktivieren Sie die Option `Debug ▶ Enable timing analyzer` wieder, damit weitere Prozesse nicht unnötig Timing-Informationen erzeugen.
4. Öffnen Sie das Fenster „Timing Information“ mit dem Menüeintrag `Debug ▶ Show timing information`.

Beachten Sie bitte, dass das Zeitverhalten auf dem ADwin-System sehr von der Anzahl und auch von der Art der Prozesse abhängt, dass also auch entsprechend unterschiedliche Kennwerte entstehen. Dies liegt unter anderem an der Verwaltung der Prozesse durch das Betriebssystem (siehe Kapitel 5.2.5 „Verschiedene Betriebszustände im Betriebssystem“).



Die Berechnung der Informationen erfolgt zur Laufzeit und benötigt etwa 60 Taktzyklen (T9, T10 und T11) pro Prozesszyklus und Prozess zusätzlich. Die Kennwerte im Fenster werden laufend aktualisiert und beziehen sich auf den Zeitraum seit dem jeweils letzten Start der Prozesse. Eine kurze Erklärung der Werte finden Sie unter dem Menüeintrag „Show timing information“, Seite 26.



Die (geringfügige) Änderung des Zeitverhaltens durch den Timing-Modus ist leider nicht vermeidbar und entsteht auch, wenn die Kennwerte nicht angezeigt werden. Dies kann in Grenzfällen zu zusätzlichen Startverzögerungen (latency) führen, die dann auch in den entsprechenden Kennwerten wiedergegeben werden; bei kurzen Prozessen mit kleinem Processdelay kann manchmal auch eine Prozessor-Auslastung über 100% erreicht werden, d.h. die Kommunikation zum PC reisst ab.

Beachten Sie auch, dass das Kompilieren vieler hochpriorer Prozesse mit der Timing-Option einen niederprioreren Prozess erheblich verlangsamen kann..

### 4.3.3 Programmablauf verfolgen (Trace-Modus)

Der Trace-Modus ist ein Hilfsmittel zum Nachverfolgen des Programmablaufs. Dieser Modus erlaubt, während der Laufzeit im ADwin-System erzeugte Prozess-Informationen – vor allem Berechnungsergebnisse – *nachträglich* in einem Fenster der Entwicklungsumgebung zu betrachten. Dabei legen Sie selbst im Quelltext fest, auf welche Informationen Sie später zugreifen können.



Der Trace-Modus verändert das Zeitverhalten des gewählten Prozesses und benötigt zusätzlichen Speicherplatz, sowohl im Datenspeicher als auch im Programmspeicher. Dies gilt auch, wenn keine Prozess-Informationen definiert und abgefragt werden. Bei großem Datenvolumen kann der zusätzliche Zeit- und Speicherbedarf auch größer werden als der Bedarf des beobachteten Prozesses ohne Trace-Modus.

Zur Verwendung des Trace-Modus gehen Sie vor wie folgt:




1. Aktivieren Sie den Trace-Modus unter: **Debug ▶ Trace Setup** und aktivieren dort die Option **Enabled**.
2. Markieren Sie im Quelltext die fraglichen Zeilen und aktivieren diese im Kontextmenü (rechte Maustaste) mit „**Enable Trace**“ für den Trace-Modus. Aktive Zeilen werden mit einem Fragezeichen am Zeilenanfang gekennzeichnet.  
Aktive Zeilen können mit den Befehlen **TRACE\_MODE\_RESUME** und **TRACE\_MODE\_PAUSE** programmgesteuert deaktiviert und wieder aktiviert werden.
3. Kompilieren (und starten) Sie Ihren Quelltext. Beginnen Sie am besten mit einem einfachen Programm.  
Wichtig: Der Trace-Modus muss beim Kompilieren aktiviert sein.
4. Öffnen Sie das Trace-Fenster mit **Debug ▶ Show Trace**.

Im Trace-Fenster sehen Sie das vollständige Programm, darin sind inaktive Zeilen grau. Links neben den aktiven Quelltextzeilen stehen die Prozess-Informationen:

- Ein Variablenwert als Ergebnis einer Zuweisung mit dem Operator „**=**“ (die Operatoren **DEC** und **INC** werden nicht unterstützt).
- Der Wert einer Zählvariablen in einer Schleife, und davon abhängig die Variablenwerte innerhalb der Schleife (Einstellen der Zählvariablen in der Kopfzeile des Trace-Fensters rechts).
- Das Ergebnis einer Bedingung: **True** oder **False**.  
Wenn auf eine „**IF ... THEN**“-Bedingung eine Zuweisung an eine Variable folgt (einzeilige **IF**-Variante) und die Bedingung erfüllt ist, wird nur der Variablenwert angezeigt.
- Der Quelltext eines Makros:  
Klicken Sie hierzu mit der rechten Maustaste auf den Namen des Makros (**FUNCTION ...** oder **SUB ...**); der Makrotext wird dann über der Zeile mit dem Makroaufruf eingefügt. Auf dem gleichen Weg wird der Makrotext wieder verborgen.




Die angezeigten Informationen werden zur Laufzeit, also während Ihr Programm läuft, in ein globales Feld gespeichert (normalerweise `DATA_239`, siehe Menüeintrag „Trace Setup ...“). Die Entwicklungsumgebung kopiert den Feldinhalt anschließend zum PC und zeigt ihn an. Je nach Feldgröße können die Informationen mehr oder weniger Event-Durchläufe umfassen.

Sie aktualisieren die angezeigten Informationen mit der Schaltfläche New Values  in der Kopfzeile; wenn Sie die vorherigen Prozess-Informationen weiter benötigen, sollten Sie diese erst speichern  oder drucken .

Durch das Aktualisieren werden auch die Prozess-Daten zum Programmschnitt **INIT**: überschrieben.

Beachten Sie bitte, dass der Trace-Modus sich nur auf den aktiven Quelltext bezieht, d.h. importierte Bibliotheken (Libraries) und Include-Dateien werden nicht unterstützt. Außerdem können Sie in der Entwicklungsumgebung *ADbasic* nur einen einzigen Prozess beobachten; für jeden weiteren Prozess müssen Sie eine zusätzliche Instanz der Entwicklungsumgebung starten und den Trace-Modus einrichten. Achten Sie darauf, dass Sie in jeder *ADbasic*-Instanz unter Debug ▶ Trace Setup ein anderes globales Feld (`DATA_1` ... `DATA_200`) angeben.

Mit den Befehlen **TRACE\_MODE\_PAUSE** und **TRACE\_MODE\_RESUME** können Sie in einem *ADbasic*-Programm den Trace-Modus gezielt für bereits aktive Programmzeilen deaktivieren und aktivieren. So kann der Trace-Modus z.B. nur solange aktiviert werden, wie eine bestimmte Bedingung erfüllt ist. 



### 5 Prozesse im Betriebssystem

Ein *ADwin*-System stellt alle Möglichkeiten zur Verfügung, um komplexe Anlagen zu regeln, zu steuern und Messungen durchzuführen. Mit *ADbasic* programmieren Sie Prozesse, die diese Möglichkeiten nutzen: Sie legen in darin fest, wie und wann Ihr System analoge und digitale Daten verarbeitet und nach außen gibt.

Nach dem Start des Prozesses wird das Programm<sup>3</sup> im *ADwin*-System (typischerweise) in regelmäßigen Abständen neu aufgerufen und abgearbeitet. Ein solcher Aufruf eines Prozesszyklus wird durch eines der beiden folgenden Startsignale, sogenannte „Events“, ausgelöst:

1. Timer-Event: Ein Impuls des internen Zählers. Sie können für jeden Prozess separat festlegen, in welchem Zeitabstand (Processdelay) ein neuer Event ausgelöst wird.
2. Externer Event: Ein externes Signal, das am „Event“-Eingang Ihres *ADwin*-Systems eingeht. Dies könnte beispielsweise ein Impuls eines Inkrementalgebers sein.

Nur einer der 10 möglichen Prozesse kann von einem externen Event gesteuert werden, alle anderen Prozesse müssen zeitgesteuert ablaufen.

Die exakte Funktion eines Prozesses definieren Sie im *ADbasic*-Quelltext:

- Die Initialisierung in den Abschnitten **LOWINIT**: und/oder **INIT**: .
- Die eigentliche Funktion des Prozesszyklus im zentralen Abschnitt **EVENT**: .
- Die Schlussbearbeitung im Abschnitt **FINISH**: .

Vom PC aus können Sie die Prozesse eines Systems steuern, d.h. Sie können die Prozesse starten, stoppen oder deren Processdelay ändern. Dies können Sie sowohl aus *ADbasic* als auch aus Entwicklungsumgebungen wie C++ oder Visual Basic tun.

---

3. genauer: der Programmabschnitt **EVENT**: .

## 5.1 Prozessverwaltung

### 5.1.1 Prozessarten

Auf einem *ADwin*-System können mehrere Prozesse gleichzeitig ablaufen. Das Betriebssystem sorgt dafür, dass die Prozesszyklen nach bestimmten Regeln aufgerufen und vom Prozessor abgearbeitet werden, ohne sich gegenseitig zu blockieren (siehe auch Kapitel 5.2.5).

Wenn wir im folgenden von einem „Prozess“ sprechen, ist damit einer der von Ihnen programmierten Prozesse 1...10 gemeint.

Sie können jedem Prozess eine bestimmte Priorität zuweisen, und dadurch das zeitliche Zusammenspiel der Prozess-Abarbeitung bestimmen. Es gibt:

- Prozesse mit der Priorität „Hoch“
- Prozesse mit der Priorität „Niedrig“

Niederpriorie Prozesse werden weiter in die Stufen -10 (niedrig) bis +10 (hoch) unterteilt.

Weisen Sie einem Prozess seine Priorität über das Menü „Options \ Process Options“ zu.

Prozess	Funktion	Priorität <sup>a</sup>
1...10	Benutzerdefinierte Prozesse mit frei definierbarer Funktion und Priorität	Niedrig Stufe <i>n</i> / Hoch
11, 12	Vordefinierte Ein-Ausgabe-Prozesse	Hoch
15	Prozess zur Steuerung der Blink-LED bei <i>ADwin-Pro</i> - und <i>ADwin-Gold</i> -Systemen	Niedrig, Stufe 1
Kommunikation	Kommunikation zwischen <i>ADwin</i> -System und PC: Befehls- und Datenübertragung	Mittel

a. Die Bedeutung der Prioritäten wird in den folgenden Abschnitten erklärt.

Abb. 20 – Übersicht aller Prozesse

Die Standardprozesse 11 und 12 werden nur in Verbindung mit den Treibern für die Bedienoberflächen Labview und Testpoint benötigt. Für diesen Fall können Sie diese Prozesse beim Booten mit dem Betriebssystem in das *ADwin*-System übertragen, entweder aus der Bedienoberfläche (Näheres finden Sie in der Treiber-Dokumentation) oder aus *ADbasic*. In *ADbasic* stellen Sie hierzu im Menü „Options / Compiler“ die Option „Load Standard

processes“ auf „Yes“.

Für alle anderen Anwendungen können Sie die Übertragung der Standardprozesse beim Booten unterbinden (Einstellung „No“).

Der Kommunikationsprozess (siehe Seite 88) ist Teil des Betriebssystems. Er empfängt Kommandos des PC und tauscht Daten zwischen dem ADwin-System und dem PC aus, allerdings nur auf Anforderung des PC.

Wenn Sie mehr als einen Prozess mit derselben Prozess-Nummer auf das System übertragen, wird nur der zuletzt übertragene ausgeführt, weil die zuerst übertragenen Prozesse überschrieben werden.



### 5.1.2 Prozesse mit der Priorität „Hoch“

Prozesse mit der Priorität „Hoch“ werden vom Betriebssystem bevorzugt behandelt:

- Vom Aufruf des Prozesszyklus durch einen Event bis zu dessen Bearbeitungsbeginn vergehen maximal 300ns.
- Ein hochpriorer Prozesszyklus ist nicht unterbrechbar und wird immer vollständig abgearbeitet. Während dieser Zeit werden alle Prozesszyklen mit geringerer Priorität unterbrochen.

Weder ein anderer hochpriorer Prozesszyklus noch ein Stopp-Befehl kann einen laufenden, hochprioren Prozesszyklus unterbrechen. In beiden Fällen muss auf das Ende der Bearbeitung gewartet werden.

Bei zeitgesteuerten hochprioren Prozessen kann die Zykluszeit (Processdelay) in Schritten zu 25ns eingestellt werden.

Lassen Sie einen zeitkritischen Messprozess mit hoher Priorität laufen und alle anderen mit niedriger Priorität, damit der Prozessor die zeitkritischen Prozesszyklen ungestört bearbeiten kann.



Die Abschnitte **LOWINIT**: und **FINISH**: eines Prozesses werden – falls vorhanden – immer mit niedriger Priorität und Prioritätsstufe 1 ausgeführt, auch wenn Sie dem Prozess die Priorität „Hoch“ gegeben haben.



### 5.1.3 Prozesse mit der Priorität „Niedrig“

Prozesszyklen mit der Priorität „Niedrig“ werden sofort unterbrochen, wenn ein Prozesszyklus mit höherer Priorität aufgerufen wird und zwar so lange, wie dieser bearbeitet wird.

Niederpriorre Prozesse werden in die Prioritätsstufen -10 (niedrig) bis +10 (hoch) unterteilt. Prozesszyklen mit niedrigerer Stufe können jederzeit von

solchen mit höherer Stufe unterbrochen werden. Der Prozessor T11 beachtet die Prioritätsstufen bei der Prozessverwaltung sehr strikt (siehe Kapitel 5.2.3). Niederpriore Prozesszyklen mit gleicher Prioritätsstufe nehmen an einem Zeitscheibenverfahren teil. Dabei teilt das Betriebssystem den Prozesszyklen die Rechenzeit abwechselnd und in gleichen Zeitscheiben zu. Eine Zeitscheibe hat im Mittel eine Dauer von 2ms (Prozessor T9) oder 1ms (Prozessoren T10, T11).

Niederpriore Prozesse müssen immer zeitgesteuert sein. Die Zykluszeit (Processdelay) kann in diskreten Schritten eingestellt werden; Schrittweite siehe Abb. 21 auf Seite 89.

Prozesse mit niedriger Priorität beeinflussen also das Zeitverhalten eines hochprioren Prozesses prinzipiell nicht, wohl aber umgekehrt.

#### 5.1.4 Kommunikationsprozess

Der Kommunikationsprozess hat eine Prioritätsstufe zwischen den Prioritäten „Hoch“ und „Niedrig“. Er kann daher niederpriore Prozesszyklen jederzeit unterbrechen und wird selbst von hochprioren Prozesszyklen unterbrochen. Wenn der PC mit dem ADwin-System kommuniziert, muss der Kommunikationsprozess spätestens nach 250ms antworten (time out), sonst kann die Kommunikation zwischen PC und ADwin-System abreißen. Sie erhalten dann die Meldung „ADwin-System meldet sich nicht“ und müssen das System durch Booten initialisieren. Dieses „time out“ ist unabhängig von der Schnittstelle USB oder Ethernet.

Der Grund für ein Abreißen der Kommunikation ist, dass dem Kommunikationsprozess nicht genügend Rechenleistung zur Verfügung steht. Dies kann verursacht werden durch

- ein zu kleines Processdelay der hochprioren Prozesse oder
- eine zu lange Bearbeitungszeit eines hochprioren Prozesszyklus.

Weiteres zum Thema Kommunikation finden Sie in Kapitel 5.3.2.

## 5.2 Zeitverhalten von Prozessen

### 5.2.1 Processdelay

Der Zeitabstand, in dem zeitgesteuerte Prozesszyklen vom Zähler aufgerufen werden, ist die Zykluszeit. Sie wird üblicherweise in Taktzyklen des Zählers gemessen und dann als *Processdelay* bezeichnet (in früheren *ADbasic*-Ver-

sionen: Globaldelay). Sie können das Processdelay jedes Prozesses über den Wert der Systemvariablen **PROCESSDELAY** festlegen.

Die Dauer eines Zähler-Taktzyklus ist abhängig von der Prozesspriorität und dem Prozessortyp:

Prozessor	Priorität	
	Hoch	Niedrig
T9	25ns	100µs
T10	25ns	50µs
T11	3,3ns	3,3ns = 0,003µs

Abb. 21 – Dauer eines Zähler-Taktzyklus (Einheit des Processdelay)

Ein Processdelay mit dem Wert 1000 beispielsweise bedeutet in einem hochprioren Prozess mit dem Prozessor T9 einen regelmäßigen Aufruf im Zeitabstand von  $1000 \times 25\text{ns} = 25000\text{ns} = 25\mu\text{s}$ , bei einem niedriprioren Prozess dagegen von  $1000 \times 100\mu\text{s} = 100000\mu\text{s} = 100\text{ms}$ . Dies kann z.B. eingestellt werden mit der Programmzeile:

**PROCESSDELAY** = 1000

Damit jeder Prozesszyklus zu der (mit **PROCESSDELAY**) festgelegten Zeit aufgerufen wird, darf die Bearbeitungszeit eines Prozesszyklus die Zykluszeit auch im ungünstigsten Fall nicht überschreiten. Unterschiede bei der Berechnungszeit ergeben sich beispielsweise bei fallweisen Unterscheidungen (If, Case).

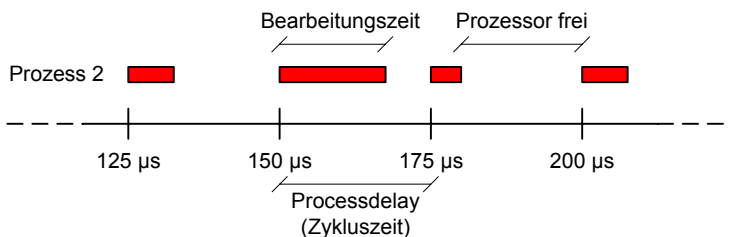


Abb. 22 – Processdelay und Bearbeitungszeit bei hochprioren Prozesszyklen

### Beispiel



Wenn eine umfangreiche Berechnung nur alle 1000 Messungen auftritt, dann muss auch die lange Bearbeitungszeit dieses Prozesszyklus

kürzer sein als die Zykluszeit. Um dennoch kurze Prozesszyklen zu erreichen, ist es eine gute Alternative, die Berechnung in kleine Schritte aufzuteilen und in jedem Prozesszyklus jeweils einen Schritt zu bearbeiten. Die Prozesszyklen erhalten dadurch eine durchweg gleichmäßige und kurze Bearbeitungszeit.

### 5.2.2 Zeitlich exakter Aufruf von Prozesszyklen

Wenn Sie (wie in Abb. 22) genau einen hochpriorien Prozess haben, so wird dieser exakt im Zeitraster aufgerufen und abgearbeitet.

Stellen Sie sicher, dass die Bearbeitungszeit eines hochpriorien Prozesszyklus seine Zykluszeit (im Beispiel unten: 25  $\mu$ s) nie überschreitet. Anderenfalls können – weil dieser Prozesszyklus nicht unterbrechbar ist – andere Prozesszyklen nur unvollständig oder gar nicht mehr bearbeitet werden, z. B. der wichtige Kommunikations-Prozess.

Gibt es mehrere hochpriorie Prozesse, kann der jeweils aktive Prozesszyklus das Zeitraster der übrigen Prozesszyklen beeinflussen. In Abb. 23 kann z. B. der geplante Aufruf im Prozess 1 erst verzögert geschehen, wenn die Bearbeitung des aktiven Prozesses 2 beendet ist.

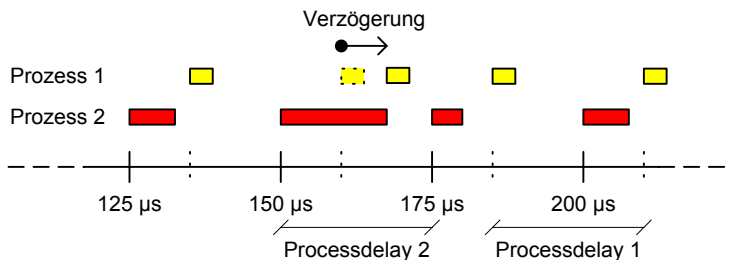


Abb. 23 – Verzögerung eines hochpriorien Prozesszyklus

- ⚠ Halten Sie die Ausführungszeit von hochpriorien Prozesszyklen so gering wie möglich. Lassen Sie zeitintensive Schleifen oder Berechnungen, deren Ergebnis nicht sofort weiter verarbeitet wird, immer in Prozesszyklen mit niedriger Priorität ablaufen.
- ⚠ Ein niederpriorier Prozess ist abhängig vom Zeitverhalten aller anderen Prozesszyklen mit höherer oder gleicher Priorität. Jede Unterbrechung verringert das Zeitfenster, in dem der niederpriorie Prozesszyklus Rechenleistung nutzen kann; im Extremfall wird er überhaupt nicht aufgerufen.

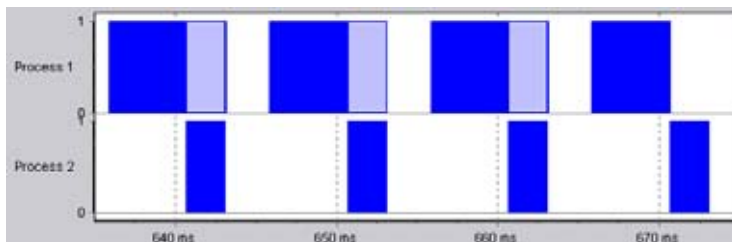


### 5.2.3 Niederpriorie Prozesse beim T11

Der Prozessor T11 verwaltet niederpriorie Prozesse strikt nach deren Prioritätsstufe. Im Vergleich dazu haben Prioritätsstufen bei T9 und T10 nur geringe Bedeutung. Der Vorrang von Kommunikationsprozess und hochpriorien Prozessen vor allen niederpriorien Prozessen bleibt davon unberührt.

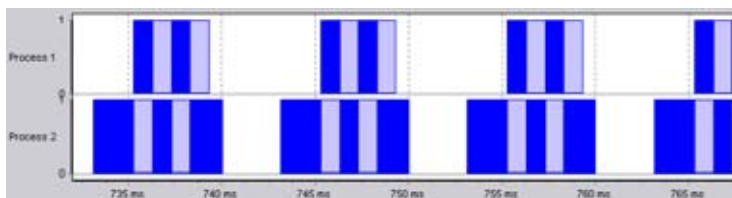
Bei niederpriorien Prozessen ist zu unterscheiden zwischen:

- Prozesse mit unterschiedlicher Prioritätsstufe: Alle Prozesse mit niedrigerer Prioritätsstufe werden unterbrochen, sobald und solange ein Prozess mit höherer Prioritätsstufe bearbeitet wird.



Hier hat Prozess 2 die höhere Prioritätsstufe und unterbricht deswegen immer wieder Prozess 1.

- Prozesse mit gleicher Prioritätsstufe: Die Prozesse nehmen am Zeitscheibenverfahren teil. Das Betriebssystem teilt den Prozesszyklen die Rechenzeit innerhalb der Prioritätsstufe abwechselnd und in gleichen Zeitscheiben (1 ms) zu.



Im Beispiel ist der Wechsel zwischen den beiden Prozessen gut zu sehen. Es zeigt auch die Regel, dass ein Prozess – hier Prozess 1 – bei jedem Aufruf sofort eine Zeitscheibe Rechenzeit erhält.

In einem seltenen Sonderfall wird das Zeitscheibenverfahren ausgehebelt: Ein Prozess erhält sehr viel Rechenzeit, wenn er sehr häufig aufgerufen wird und sein Prozesszyklus zugleich kürzer als eine Zeitscheibe dauert. Bei jedem Aufruf unterbricht dieser Prozess andere

Prozesse mit gleicher Prioritätsstufe und „stiehlt“ ihnen damit die Rechenzeit.

### 5.2.4 Auslastung des ADwin-Systems

Die Auslastung des Prozessors auf dem ADwin-System ist das Verhältnis von genutzter Rechenzeit zur insgesamt verfügbaren Rechenzeit, angegeben in Prozent.

Sie können die Auslastung des Prozessors an der Anzeige „Busy“ in der Statusleiste der Entwicklungsumgebung beobachten. Dieser Wert gibt Ihnen einen Anhaltspunkt, ob der Prozessor noch genügend Rechenzeit frei hat, um alle Prozesszyklen abarbeiten zu können.

Die Auslastung des Prozessors sollte 90% nur in Ausnahmefällen überschreiten, den Wert 100% jedoch niemals.

Bitte beachten Sie beim Prozessor T11: Trotz einer angezeigten Auslastung unter 90% kann eine Überlastung vorliegen, so dass z. B. manche Prozesszyklen verspätet bearbeitet werden. Die Überlastung liegt dann nicht beim Prozessor vor, sondern auf dem Pro I- oder Pro II-Bus, deren Auslastung nicht angezeigt werden kann.

### 5.2.5 Verschiedene Betriebszustände im Betriebssystem

Das Betriebssystem unterscheidet beim Zeitverhalten für hochpriorie Prozesse zunächst 2 Betriebszustände, je nachdem, ob nur 1 oder ob mehrere zeitgesteuerte (hochpriorie) Prozesse aktiv sind. Ob zusätzlich ein extern gesteuerter Prozess läuft, ist hierbei nicht von Belang. Der extern gesteuerte Prozess wird vom Betriebssystem separat organisiert und kann daher als 3. Betriebszustand verstanden werden.

#### Einzelner zeitgesteuerter Prozess

Das Betriebssystem verwendet bei einem einzelnen, zeitgesteuerten Prozess (u. a.) Hardware-Bausteine, um die Event-Signale des internen Zählers zu verarbeiten. Dadurch kann das Betriebssystem ein eintreffendes Event-Signal dieses Prozesses sehr schnell bearbeiten.

Die Hardware-Bausteine können zwischenspeichern, ob ein Event-Signal eingetroffen ist, allerdings nicht wieviele Event-Signale es waren. Ist ein Event-Signal eingetroffen, aktiviert das Betriebssystem den nächsten Prozesszyklus zum festgelegten Zeitpunkt (siehe Processdelay, Kapitel 5.2.1), wenn nicht gerade ein hochpriorer Prozesszyklus bearbeitet wird. In diesem Fall aktiviert das Betriebssystem den nächsten Prozesszyklus sofort nach dem aktuell bearbeiteten Prozesszyklus.

Wenn mehrere Event-Signale eintreffen, während gerade ein hochpriorer Prozesszyklus bearbeitet wird, werden nicht entsprechend viele Prozesszyklen aufgerufen, sondern nur ein einziger; es gehen also Event-Signale verloren. Achten Sie deshalb streng darauf, dass die Prozesszyklen kürzer sind als die Zykluszeit (Processdelay) des Prozesses.



### Mehrere zeitgesteuerte Prozesse

Bei mehreren zeitgesteuerten Prozessen werden eintreffende Event-Signale vom Betriebssystem selbst verwaltet. Dieser Betriebsmodus ist wegen des erforderlichen Verwaltungsaufwands zwar langsamer, dafür wird aber die Anzahl aller eintreffenden Event-Signale für jeden Prozess zwischengespeichert. Damit ist gewährleistet, dass zu jedem Event-Signal ein Prozesszyklus gestartet wird, wenn auch ggf. später als zum eigentlich festgelegten Zeitpunkt.

Meistens sind die Zeitraster für den Start der Prozesszyklen derart, dass immer wieder Event-Signale während der Bearbeitung eines anderen Prozesszyklus auftreten. Anders gesagt, die Processdelay-Werte sind keine ganzzahligen Vielfachen voneinander. Wir empfehlen daher, mit möglichst wenigen Prozessen auszukommen; meistens ist es möglich (und erzielt außerdem eine geringere Prozessorauslastung), mehrere Prozesse zu einem einzigen Prozess zusammenzufassen.

Beachten Sie immer, dass die Prozessorauslastung stark von der Zahl der laufenden Prozesse abhängt. So benötigt eine Aufgabe, die von 2 (oder sogar mehr) Prozessen ausgeführt wird, in jedem Fall mehr Prozessorzeit als die gleiche Aufgabe mit einem einzelnen Prozess. Dies fällt umso stärker ins Gewicht je kürzer das Processdelay der Prozesse ist (siehe auch Kapitel 4.3.2).



Beispiel: Prozess 1 und 2 mit sehr kleinem Processdelay erzeugen als Einzelprozess je 10% Auslastung; beide Prozesse gemeinsam erzeugen 55% Auslastung.

### Extern gesteuerter Prozess

Der Betriebsmodus für den extern gesteuerten Prozess ist unabhängig von den zeitgesteuerten Prozessen immer der gleiche. Der externe Prozess wird vom Betriebssystem verwaltet wie ein einzelner zeitgesteuerter Prozess (s.o.), d.h. eintreffende Event-Signale werden sehr schnell bearbeitet, jedoch können auch Event-Signale verloren gehen.

Ein externes Event-Signal ist eine besonders wichtige Information – schon weil sie vom ADwin-System nicht vorherbestimmbar ist – und darf auf keinen Fall verloren gehen (verlorene Events finden, siehe Seite 26). Achten Sie des-



halb in diesem Prozess ganz besonders auf kurze Prozesszyklen (im Abschnitt **EVENT** : ).

## 5.3 Kommunikation

### 5.3.1 Datenaustausch zwischen Prozessen

Sie können Daten zwischen verschiedenen Prozessen über die globalen Variablen (`PAR_1 ... PAR_80`, `FPAR_1 ... FPAR_80`) oder über globale Felder (`DATA_n`) austauschen. Auf diesem Weg ist auch der Datenaustausch mit Programmen im PC möglich.



Wenn Sie globale Felder in mehreren Prozessen verwenden, müssen Sie diese in jedem Prozess in absolut gleicher Weise deklarieren. In diesem Fall ist es praktisch, wenn Sie die Deklaration der globalen Felder in einer Include-Datei speichern und diese in allen Prozessen einbinden (siehe auch Kapitel 3.5.2 „Include-Dateien“).

Je nach Programmierung können Sie (jeweils gleiche) globale Variablen verwenden, um aus einem Prozess heraus einen anderen, gleichzeitig laufenden Prozess zu steuern.



#### Beispiel

Prozess 1 arbeitet als Funktionsgenerator und Prozess 2 als Regler. Der Funktionsgenerator schreibt regelmäßig den jeweils erzeugten Wert in die globale Variable `PAR_10`. Der Regler liest bei jedem Prozesszyklus diese globale Variable `PAR_10` aus und verwendet deren Inhalt als Sollwert des Regelkreises.

Damit steuert der Funktionsgenerator auf einfache Weise den Sollwertverlauf des Reglers. Alle *lokalen* Variablen und Felder des Prozesses 1 bleiben hierbei dem Prozess 2 verborgen (und umgekehrt). Beachten Sie bitte, dass bei der Zusammenarbeit der beiden Prozesse auch deren Zeitverhalten von Bedeutung ist.

### 5.3.2 Kommunikation zwischen PC und ADwin-System

Vom PC aus können Sie aus Anwendungen und Entwicklungsumgebungen Prozesse im ADwin-System steuern, sowie Daten von dort anfordern oder dorthin senden. Grundsätzlich kann ein ADwin-System nie selbstständig mit dem PC kommunizieren, sondern reagiert nur auf eine Anfrage des PC.

Jeder Datenaustausch (auch zwischen Prozessen) erfolgt über globale Variablen (`PAR_n`, `FPAR_n`) oder globale Felder (`DATA_n`).

Jede Kommunikation zum *ADwin*-System läuft unter Windows über die sogenannte „ADwin32.dll“ (dynamic-link library), im System übernimmt diese Aufgabe der Kommunikationsprozess (Seite 88).

Wenn Sie mit der ActiveX-Schnittstelle arbeiten, übernimmt diese – auf den ersten Blick ähnlich zentral – jede Kommunikation mit dem *ADwin*-System. Intern gibt die ActiveX-Schnittstelle die kommunizierten Daten weiter an die „ADwin32.dll“ oder erhält sie von dieser.

Die „ADwin32.dll“ übernimmt folgende Aufgaben:

- Kommunikation mit dem angesprochenen *ADwin*-System über die verbindende Schnittstelle: USB, Ethernet (TCP/IP).
- Erkennen und Behandeln von Fehlern.
- Verriegeln mehrerer PC-Anwendungen gegeneinander, wenn diese gleichzeitig auf dasselbe System zugreifen möchten.

Durch die Verriegelung können mehrere Anwendungen unabhängig voneinander und quasi gleichzeitig auf ein oder mehrere *ADwin*-Systeme zugreifen.

Wenn eine PC-Anwendung die Kommunikation zu einem bestimmten System aufnimmt, übergibt es neben der gewünschten Anweisung auch eine Gerätemummer, die sogenannte „Device No“. Anhand dieser „Device No“ unterscheidet die „ADwin32.dll“ die verschiedenen *ADwin*-Systeme und ordnet die entsprechenden Einstellungen zu.

### 5.3.3 Die Device No.

Jedes *ADwin*-System, das an einen PC angeschlossen ist, wird über eine (in diesem PC) eindeutige Gerätemummer, die „Device No“ angesprochen.

Sie stellen die Device-No. mit dem Programm *ADconfig* ein: .

In *ADconfig* verknüpfen Sie eine „Device No“ mit den Verbindungsparametern, mit denen ein System erreichbar ist (USB, TCP/IP). Auf diese Informationen greift die „ADwin32.dll“ zurück, um mit dem System kommunizieren zu können.

### 5.3.4 Kommunikation mit Entwicklungsumgebungen

Vom PC aus greifen Sie aus einer Bedienoberfläche auf das *ADwin*-System zu. Sie können diese mit einer der gängigen Entwicklungsumgebungen selbst gestalten, wie z. B. ActiveX, Visual Basic, C++, Delphi oder C#.NET, oder eine fertige Bedienoberfläche wie etwa TestPoint, DIAdem oder MATLAB verwenden.

Für jede dieser Möglichkeiten erhalten Sie bei uns eine passende Treiber-Software, mit der Sie auf das *ADwin*-System zugreifen können. Wenn Sie spezielle Wünsche haben, fragen Sie bei uns an. Sie erhalten bei uns auch maßgeschneiderte Messdaten-Auswertungsprogramme.

Unter Windows ermöglicht eine DLL- oder ActiveX-Schnittstelle die Kommunikation mit dem System auch aus mehreren Programmen gleichzeitig (siehe auch „Kommunikation zwischen PC und *ADwin*-System“ auf Seite 94).



Die speziellen Befehle für Ihre Bedienoberfläche werden in den Unterlagen zu der jeweiligen Treiber-Software genau erläutert.

Sie können aus Ihrer Bedienoberfläche:

- kompilierte Programme (Binärdateien) in das *ADwin*-System übertragen. Sie kompilieren ein Programm in *ADbasic* mit „Build/Make Bin File“ (siehe Kapitel 2.3.4).
- Prozesse im *ADwin*-System starten, steuern und anhalten.
- Daten vom *ADwin*-System anfordern oder dorthin senden.

Obwohl das *ADwin*-System autark arbeitet, können Sie aus der Bedienoberfläche jederzeit auf globale Variablen und Felder zugreifen, ohne zeitkritische Prozesse zu verzögern. Auf dem beschriebenen Weg können alle Prozesse mit dem PC (und auch untereinander) schnell Daten miteinander austauschen.

## 6 Befehlsreferenz

Im folgenden sind die in *ADbasic* verfügbaren Befehle aufgeführt:

- Kapitel 6.2 „Befehle L16, Gold, Pro“, Seite 98-234  
Alle Hardware-bezogenen Befehle für das *ADwin-Pro*-System finden Sie (aus Platzgründen) in der separaten Dokumentation „*ADwin-Pro* Software“.
- Kapitel 6.3 „ADwin-Gold und ADwin-light-16“, Seite 235-Seite 267
- Kapitel 6.4 „ADwin-light-16 DIO1/2 / ADwin-Gold CO1“, Seite 267-Seite 325
- Kapitel 6.5 „ADwin-Gold-CAN“, Seite 327-366
- Kapitel 6.6 „ADwin-L16 Rev. B“, Seite 369-374

Innerhalb dieser Kapitel finden Sie die Befehle in alphabetischer Reihenfolge. Im Anhang finden Sie je eine vollständige Befehlsliste nach Alphabet oder Befehlsgruppen sortiert.

### 6.1 Befehlssyntax

Beachten Sie bitte:

- Als Argument ist ein beliebiger Berechnungsausdruck möglich.
- Bei einigen Argumenten ist die Datenstruktur vorgeschrieben, die wie folgt gekennzeichnet ist:

**CONST** Konstante Zahlen wie 35 oder 3.14159 sowie Berechnungsausdrücke ohne Variablen.

Konstante Zeichenketten werden in doppelten Hochkommas angegeben wie "dieser Text".

**VAR** Variable oder Feldelement.

**ARRAY** Feld, in der Syntaxzeile auch erkennbar an den Klammern [ ] nach dem Feldnamen.

**FIFO** Fifo-Feld (als Fifo deklariertes *DATA\_n*-Feld).

- Neben einem Argument oder dem Rückgabewert einer Funktion ist der erwartete Datentyp angeben:

**LONG** ganze Zahl

- `FLOAT` Fließkomma-Zahl
- `STRING` Zeichenkette
- `LOGIC` logischer Ausdruck in einer Bedingung

Falls ein Argument nicht den erwarteten Datentyp hat, wird der Datentyp des Arguments gewandelt (siehe „Typkonvertierung“ auf Seite 65).

- Manche Befehle können nur benutzt werden, wenn eine bestimmte Library- oder Include-Datei eingebunden wird. Unter **Syntax** ist der jeweilige Befehl zum Einbinden angegeben (setzen Sie diese Befehlszeile bitte an den Anfang des Quelltextes).

Es wird davon ausgegangen, dass die erforderliche Library- oder Include-Datei in dem Verzeichnis liegt, das unter „Options ▶ Settings“ unter dem Reiter „Directory“ eingestellt ist (siehe auch die Befehle **#INCLUDE** oder **IMPORT**).

## 6.2 Befehle L16, Gold, Pro



## + (Addition)

Der Operator „+“ addiert je zwei Werte (siehe auch „+ (String-Addition)“).

### Syntax

```
val = val_1 + val_2
```

### Parameter

<code>val_1</code>	Summand 1	<div>FLOAT</div> <div>LONG</div>
<code>val_2</code>	Summand 2	<div>FLOAT</div> <div>LONG</div>

### Bemerkungen

Beachten Sie, dass die Verknüpfung von verschiedenen Variablentypen mit dem „+“-Operator zu einer Typ-Konvertierung führt. Bei der Wandlung vom Typ 

LONG

 in den Typ 

FLOAT

 kann es zu Rundungsabweichungen kommen, die das Berechnungsergebnis beeinflussen.

### Siehe auch

- (Subtraktion), \* (Multiplikation), / (Division), ^ (Potenz)

### Beispiel

```
PAR_1 = 9 + 4                    'PAR_1 = 13
```

## **+ (String-Addition)**

Der Operator „+“ verknüpft je zwei Strings zu einem neuen String (siehe auch „+ (Addition)“).

### **Syntax**

```
IMPORT STRING.LI*          '*.LI9 für T9, *.LIA für T10,  
                           '*.LIB für T11  
  
val = val_1 + val_2
```

### **Parameter**

val_1	Zeichenkette 1	STRING
val_2	Zeichenkette 2	STRING

### **Bemerkungen**

Wenn Sie Strings „addieren“ und einem weiteren String zuweisen, muss dieser Ziel-String größer oder gleich der Summe aller ASCII-Zeichen der Teil-Strings deklariert sein.

### **Siehe auch**

"" String, ASC, CHR, FLOTOSTR, FLO40TOSTR, LNGTOSTR, STRCOMP, STRLEFT, STRLEN, STRMID, STRRIGHT, VALF, VALI

### **Beispiel**

```
IMPORT string.li9  
  
REM 3 Strings dimensionieren: 10, 5, 4 Zeichen  
DIM res_str[10] AS STRING  
DIM str_1[5] AS STRING  
DIM str_2[4] AS STRING  
  
INIT:  
  str_1 = "ADwin"          '5 Zeichen  
  str_2 = "Gold"           '4 Zeichen  
  
EVENT:  
  res_str = str_1 + "-" + str_2 'Strings "addieren"  
  PAR_1 = STRLEN(res_str) 'PAR_1 = 10 (Anzahl der Zeichen)
```

## - (Subtraktion)

Der Operator „-“ subtrahiert je zwei Werte.

### Syntax

```
val = val_1 - val_2
```

### Parameter

val\_1            Minuend

FLOAT

LONG

val\_2            Subtrahend

FLOAT

LONG

### Bemerkungen

Beachten Sie, dass die Verknüpfung von verschiedenen Variablentypen mit dem „-“-Operator zu einer Typ-Konvertierung führt. Bei der Wandlung vom Typ `LONG` in den Typ `FLOAT` kann es zu Rundungsabweichungen kommen, die das Berechnungsergebnis beeinflussen.

Wenn Sie „-“ als Vorzeichen einer Variablen verwenden (unärer Operator), kann es in manchen Fällen zu unerwarteten Ergebnissen kommen, die Sie durch Klammersetzung vermeiden können (siehe auch Kapitel 3.4.1 auf Seite 64).

### Siehe auch

+ (Addition), \* (Multiplikation), / (Division), ^ (Potenz)

### Beispiel

```
PAR_1 = 9 - 4                    'PAR_1 = 5
```

## \* (Multiplikation)

Der Operator „**\***“ multipliziert je zwei Werte.

### Syntax

```
val = val_1 * val_2
```

### Parameter

val\_1                      Multiplikator 1

FLOAT

LONG

val\_2                      Multiplikator 2

FLOAT

LONG

### Bemerkungen

Beachten Sie, dass die Verknüpfung von verschiedenen Variablentypen mit dem „**\***“-Operator zu einer Typ-Konvertierung führt. Bei der Wandlung vom Typ `LONG` in den Typ `FLOAT` kann es zu Rundungsabweichungen kommen, die das Berechnungsergebnis beeinflussen.

### Siehe auch

+ (Addition), - (Subtraktion), / (Division), ^ (Potenz)

### Beispiel

```
PAR_1 = 9 * 4                      'PAR_1 = 36
```

## / (Division)

Der Operator „/“ dividiert je zwei Werte.

### Syntax

```
val = val_1 / val_2
```

### Parameter

val\_1            Dividend

FLOAT

LONG

val\_2            Divisor

FLOAT

LONG

### Bemerkungen

Beachten Sie, dass die Verknüpfung von verschiedenen Variablentypen mit dem „/“-Operator zu einer Typ-Konvertierung führt (siehe Kapitel 3.4.2 auf Seite 65). Bei der Wandlung vom Typ `LONG` in den Typ `FLOAT` kann es zu Rundungsabweichungen kommen, die das Berechnungsergebnis beeinflussen.

Wenn Sie durch eine Variable mit negativem Vorzeichen teilen, müssen Sie diese in Klammern setzen, damit das erwartete Ergebnis berechnet wird (siehe auch Kapitel 3.4.1 „Auswertung von Operatoren“ auf Seite 64).

### Siehe auch

+ (Addition), - (Subtraktion), \* (Multiplikation), ^ (Potenz)

### Beispiel

```
PAR_1 = 36 / 4            'PAR_1 = 9
PAR_2 = 2 / 4            'PAR_2 = 0 -> ganzzahlige Rechnung
PAR_3 = 27 / (-PAR_1)    'PAR_3 = -3
REM Beachten Sie die Kammersetzung in der letzten Zeile
```

## ^ (Potenz)

Der Operator „^“ berechnet eine beliebige Potenz eines Wertes.

### Syntax

```
val = val_1 ^ val_2
```

### Parameter

val_1	Basis	<div>FLOAT</div> <div>LONG</div>
val_2	Exponent	<div>FLOAT</div> <div>LONG</div>

### Bemerkungen

Beachten Sie, dass die Verknüpfung von verschiedenen Variablentypen mit dem Potenz-Operator zu einer Typ-Konvertierung führt. Bei der Wandlung vom Typ 

LONG

 in den Typ 

FLOAT

 kann es zu Rundungsabweichungen kommen, die das Berechnungsergebnis beeinflussen.

Wenn Basis und Exponent ganzzahlige Variablen (nicht aber Konstanten) sind, wird die Potenzfunktion intern dennoch mit FLOAT-Arithmetik ausgeführt. Bei großen Ergebniswerten führt dies zu den bei Float üblicherweise zu erwartenden Rechenungenauigkeiten bei großen Zahlen.

Beispiel :

```
PAR_2 = 31           ' variable
PAR_1 = 2^PAR_2      ' = 7FFFFFFE2h
```



Wenn die Basis und/oder der Exponent eine Variable mit negativem Vorzeichen ist, müssen Sie diese in Klammern setzen, damit das Vorzeichen bei der Potenzierung berücksichtigt wird (siehe auch Kapitel 3.4.1 „Auswertung von Operatoren“ auf Seite 64). Bei Konstanten ist dies nicht der Fall.



```
var1 = -2^2          'var1 = 4
var2 = -var1^2        'var2 = -16
var3 = (-var1)^2      'var3 = 16
```

Polynome werden schneller berechnet, wenn Sie die Potenzen mittels Ausklammerung durch wenige Multiplikationen ersetzen:



`y = a + b*x + c*x^2 + d*x^3 + e*x^4` 'langsame Variante

`y = a + x*(b + x*(c + x*(d + x*e)))` 'schnelle Variante

## Siehe auch

+ (Addition), - (Subtraktion), \* (Multiplikation), / (Division), EXP, LN, LOG

## Beispiel

`PAR_1 = 9 ^ 4`      `'PAR_1 = 6561`

## #... (Präprozessor-Anweisung)

Ein *ADbasic*-Befehl, der mit dem Zeichen „#“ beginnt, ist eine Anweisung für den sogenannten „Präprozessor“, den Quelltext auf eine bestimmte Weise zu bearbeiten. Das Ergebnis der Bearbeitung wird vom Compiler verarbeitet.

Folgende Präprozessor-Anweisungen stehen Ihnen zur Verfügung:

- |                     |   |
|---------------------|---|
| <b>#DEFINE</b>      | Definition symbolischer Konstanten: Zeichenfolgen im Quelltext werden durch andere Zeichenfolgen ersetzt.                 |
| <b>#INCLUDE</b>     | Datei einfügen: Eine Datei (mit Quelltext) wird in den Quelltext eingefügt.   |
| <b>#IF...#ENDIF</b> | Bedingte Kompilierung: Bei erfüllter Bedingung werden die entsprechenden Quelltextzeilen kompiliert, andernfalls gelöscht |



## : (Doppelpunkt)

Das Zeichen „:“ trennt Programmschritte innerhalb einer einzelnen Programmzeile.

### Syntax

```
[Schritt_1] : [Schritt_2] {: [Schritt_3] ...}
```

### Bemerkungen

[Schritt\_n] bezeichnet einen beliebigen Programmschritt, wie er sonst in einer einzelnen Programmzeile angegeben wird.

Eine Programmzeile darf nicht mehr als 255 Zeichen beinhalten (Ausnahme siehe **#INCLUDE** auf Seite 160).

Verwenden Sie den Befehl nur, wenn dadurch der Quelltext übersichtlicher wird.

### Beispiel

```
INC PAR_1 : INC PAR_2  
REM PAR_1 und PAR_2 erhöhen in *einer* Zeile
```

## = (Zuweisung)

Der Operator „=“ weist der Variablen oder dem Feldelement links vom Operator das Ergebnis des Ausdrucks rechts vom Operator zu.

### Syntax

```
var = expr
```

### Parameter

var                      Variable oder Feld

VAR	FLOAT
LONG	STRING

expr                    Berechnungsausdruck

FLOAT	LONG
STRING	

### Bemerkungen

Wenn das Datenformat des Berechnungsausdrucks nicht mit dem der Variablen oder des Felds übereinstimmt, wird es in das passende Datenformat gewandelt oder die Zuweisung wird als ungültig zurückgewiesen. Bei der Umwandlung kann es zu Rundungsabweichungen kommen, die das Berechnungsergebnis beeinflussen.

### Beispiel

```
DIM val_1, val_2 AS LONG'Deklaration
```

INIT:

```
val_1 = 69                      'Zuweisung einer Konstanten
```

EVENT:

```
val_2 = val_1 * 2              'Zuweisung eines Ausdrucks
```

## < = > (Vergleich)

Die Operatoren „<“, „=“ und „>“ dienen zum Vergleich zweier Werte. In *ADbasic* kommen diese Operatoren nur in Bedingungen vor.

### Syntax

```
IF (val_1 > val_2) THEN
```

### Parameter

val\_1            Operand

FLOAT

LONG

val\_2            Operand

FLOAT

LONG

### Bemerkungen

Folgende Vergleiche sind möglich:

Operator	Bedeutung
<	kleiner
<=	kleiner oder gleich
>	größer
>=	größer oder gleich
=	gleich
<>	ungleich

### Siehe auch

IF ... THEN ... {ELSE ...} ENDIF, #IF ... THEN ... {#ELSE ...} #ENDIF

### Beispiel

```
DIM value AS LONG
EVENT:
  value = -5
  IF (value < 0) THEN value = 0
  REM Ergebnis: value = 0
```

## ABSF

**ABSF** liefert den Betrag einer Float-Variablen.

### Syntax

```
ret_val = ABSF(value)
```

### Parameter

<code>value</code>	Argument	<code>FLOAT</code>
<code>ret_val</code>	Betrag des Arguments	<code>FLOAT</code>

### Bemerkungen

Die Ausführungszeit des Befehls beträgt beim T9 bis zu 150ns, beim T10 bis zu 75ns, beim T11 bis zu 17ns.

### Siehe auch

ABSI

### Beispiel

```
DIM val_1, val_2 AS FLOAT
EVENT:
    val_1 = -5.3
    val_2 = ABSF(val_1)    'Ergebnis: val_2 = 5.3
```

## ABSI

**ABSI** liefert den Betrag einer Long-Variablen.

### Syntax

```
ret_val = ABSI(value)
```

### Parameter

value	Argument	LONG
ret_val	Betrag des Arguments	LONG

### Bemerkungen

Die Ausführungszeit des Befehls beträgt beim T9 bis zu 75ns, beim T10 bis zu 50ns, beim T11 bis zu 17ns.

### Siehe auch

ABSF

### Beispiel

```
DIM val_1, val_2 AS LONG

EVENT:
  val_1 = -5
  val_2 = ABSI(val_1) 'Ergebnis: val_2 = 5
```

## AND

Der Operator **AND** verknüpft zwei ganzzahlige Werte bitweise oder zwei Boolesche Ausdrücke als Boolescher Operator.

### Syntax

```
ret_val = val_1 AND val_2      'Bitweiser Operator
IF ((expr1) AND (expr2)) THEN 'Boolescher Operator
```

### Parameter

val_1, val_2	Ganzzahliger Wert	<span style="border: 1px solid black; padding: 2px;">LONG</span>
expr1, expr2	Boolescher Ausdruck mit dem Wert „wahr“ oder „falsch“	<span style="border: 1px solid black; padding: 2px;">LOGIC</span>

### Bemerkungen

Sie können mit **AND** nur gleichartige Ausdrücke verknüpfen (ganzzahlige *oder* Boolesche), ein Mischen ist nicht möglich.

Sie können Boolesche Ausdrücke nur mit den Anweisungen **IF ... THEN ... ELSE** oder **DO ... UNTIL** verwenden (Variablen können keine Booleschen Werte annehmen).

Wenn Sie in einer Zeile mehrere Boolesche Operatoren verwenden, müssen Sie jede Verknüpfung separat in Klammern setzen. Bei der Verknüpfung ganzzahliger Werte ist dies nicht erforderlich.

### Siehe auch

NOT, OR, XOR

### Beispiel

```
REM Bitweise Verknüpfung von LONG-Variablen
DIM val_1, val_2, val3 AS LONG
val_1 = 0100b      '= 4
val_2 = 0110b      '= 5
val3 = val_1 AND val_2 'Bitweise Verknüpfung
REM Ergebnis: val3 = 0100b = 4
```

Oder:

REM Boolesche Verknüpfung von Booleschen Ausdrücken

**DIM** fval\_1 **AS** **FLOAT**

**DIM** val4 **AS** **LONG**

fval\_1 = 3.14

REM Boolesche Verknüpfung: (wahr) AND (wahr) = wahr

**IF** ((fval\_1 < 9.1) **AND** (fval\_1 > 3.1)) **THEN**

    val4 = 1

**ELSE**

    val4 = 0

**ENDIF**                                'Ergebnis: val4 = 1

## ARCCOS

**ARCCOS** liefert den Arcus-Cosinus eines Arguments.

### Syntax

```
ret_val = ARCCOS(val)
```

### Parameter

<code>val</code>	Argument (-1 ... +1)	<code>FLOAT</code>
<code>ret_val</code>	Arcus-Cosinus des Arguments im Bogenmaß (0... $\pi$ )	<code>FLOAT</code>

### Bemerkungen

Für `val` < -1 wird der Wert  $\pi$  (3,14159...) zurückgegeben, für `val` > 1 der Wert 0 (Null).

Die Ausführungszeit des Befehls beträgt beim T9 bis zu 2,9 $\mu$ s, beim T10 bis zu 1,45 $\mu$ s, beim T11 bis zu 0,68 $\mu$ s.

### Siehe auch

SIN, COS, TAN, ARCSIN, ARCTAN

### Beispiel

```
DIM val_1, val_2 AS FLOAT

EVENT:
  val_1 = 0.5
  val_2 = ARCCOS(val_1)
  REM Ergebnis: val_2 = 1.0472
```



## ARCSIN

**ARCSIN** liefert den Arcus-Sinus eines Arguments.

### Syntax

```
ret_val = ARCSIN(val)
```

### Parameter

val	Argument (-1 ... +1)	<span style="border: 1px solid black; padding: 2px;">FLOAT</span>
ret_val	Arcus-Sinus des Arguments im Bogenmaß ( $-\pi/2 \dots +\pi/2$ )	<span style="border: 1px solid black; padding: 2px;">FLOAT</span>

### Bemerkungen

Die Ausführungszeit des Befehls beträgt beim T9 bis zu 2,8µs, beim T10 bis zu 1,4µs, beim T11 bis zu 0,67µs.

### Siehe auch

SIN, COS, TAN, ARCCOS, ARCTAN

### Beispiel

```
DIM val_1, val_2 AS FLOAT

EVENT:
    val_1 = 0.5
    val_2 = ARCSIN(val_1)
    REM Ergebnis: val_2 = 0.5236
```

## ARCTAN

**ARCTAN** liefert den Arcus-Tangens eines Arguments.

### Syntax

```
ret_val = ARCTAN(val)
```

### Parameter

<code>val</code>	Argument (gesamter Wertebereich, siehe <code>FLOAT</code> „Zahlenwerte eingeben“ auf Seite 49).
<code>ret_val</code>	Arcus-Tangens des Arguments im Bogenmaß <code>FLOAT</code> $(-\pi/2 \dots \pi/2)$

### Bemerkungen

Die Ausführungszeit des Befehls beträgt beim T9 bis zu 1,8µs, beim T10 bis zu 0,9µs, beim T11 bis zu 0,42µs.

### Siehe auch

SIN, COS, TAN, ARCSIN, ARCCOS

### Beispiel

```
DIM val_1, val_2 AS FLOAT

EVENT:
  val_1 = 0.5
  val_2 = ARCTAN(val_1)
  REM Ergebnis: val_2 = 0.4636
```

## ASC

**ASC** ermittelt die zugehörige ASCII-Nummer zu einem einzelnen Zeichen oder zum ersten Zeichen einer Zeichenfolge.

### Syntax

```
ret_val = ASC(string)
```

### Parameter

<code>string</code>	Zeichenfolge	STRING
<code>ret_val</code>	ASCII-Nummer (0...255) des (ersten) Zeichens	LONG

### Siehe auch

"" String, + (String-Addition), CHR, FLOTOSTR, FLO40TOSTR, LNGTOSTR, STRCOMP, STRLEFT, STRLEN, STRMID, STRRIGHT, VALF, VALI

### Beispiel

```
DIM text[10] AS STRING

INIT:
  text="Hallo"

EVENT:
  PAR_1=ASC(text)      'PAR_1 = 48h = 72
  PAR_2=ASC("?")      'PAR_1 = 3Fh = 63
```

## CAST\_FLOATTOLONG

**CAST\_FLOATTOLONG** wandelt den Datentyp eines Arguments von Float (Fließkomma) in Long (ganze Zahl).

### Syntax

```
ret_val = CAST_FLOATTOLONG(var)
```

### Parameter

<code>var</code>	Bitmuster mit Datentyp Float (Fließkomma)	<span style="border: 1px solid black; padding: 2px;">FLOAT</span>
<code>ret_val</code>	Identisches Bitmuster mit Datentyp Long (ganze Zahl)	<span style="border: 1px solid black; padding: 2px;">LONG</span>

### Bemerkungen

Es handelt sich bei dieser Funktion **nicht** um die übliche Typkonvertierung eines Zahlenwerts (siehe Kapitel 3.4.2 „Typkonvertierung“, Seite 65). Für die Zuweisung einer Fließkomma-Zahl an eine ganzzahlige Variable genügt der Operator „=“.

Die Funktion ist im Zusammenhang mit der Umkehrfunktion **CAST\_LONGTOFLOAT** sinnvoll einsetzbar, wenn ein Bitmuster eine Fließkomma-Zahl repräsentiert, jedoch mit dem Datentyp Long vorliegt. Die Wandlung lässt das Bitmuster unverändert und ändert nur den Datentyp, so dass die Zahl wieder korrekt als Fließkomma-Zahl interpretiert wird (siehe auch Kapitel 3.2.3 auf Seite 48).

Ein Anwendungsbeispiel tritt bei der Datenübertragung auf: CAN- oder RSxxx-Busse übertragen nur 8 Bit-Datenpakete mit ganzzahligem Datentyp. Der Datentyp eines 32-Bit Fließkomma-Werts muss deshalb mit **CAST\_FLOATTOLONG** von Float in Long gewandelt und der Wert anschließend in 4 separate 8 Bit-Pakete aufgeteilt werden. Beim Empfänger werden die Datenpakete wieder zusammengesetzt und mit **CAST\_LONGTOFLOAT** wieder vom Datentyp Long in Float gewandelt.

### Siehe auch

CAST\_LONGTOFLOAT

## CAST\_LONGTOFLOAT

**CAST\_LONGTOFLOAT** wandelt den Datentyp eines Bitmusters von Long (ganze Zahl) in Float (Fließkomma).

### Syntax

```
ret_val = CAST_LONGTOFLOAT(val)
```

### Parameter

<code>val</code>	Bitmuster mit Datentyp Long (ganze Zahl)	<code>LONG</code>
<code>ret_val</code>	Identisches Bitmuster mit Datentyp Float (Fließkomma)	<code>FLOAT</code>

### Bemerkungen

Es handelt sich bei dieser Funktion **nicht** um die übliche Typkonvertierung eines Zahlenwerts (siehe Kapitel 3.4.2 „Typkonvertierung“, Seite 65). Für die Zuweisung einer Fließkomma-Zahl an eine ganzzahlige Variable genügt der Operator „=“.

Die Funktion ist sinnvoll einsetzbar, wenn ein Bitmuster eine Fließkomma-Zahl repräsentiert, jedoch mit dem Datentyp Long vorliegt. Die Wandlung lässt das Bitmuster unverändert und ändert nur den Datentyp, so dass die Zahl wieder korrekt als Fließkomma-Zahl interpretiert wird (siehe auch Kapitel 3.2.3 auf Seite 48).

Ein Anwendungsbeispiel tritt bei der Datenübertragung auf: CAN- oder RSxxx-Busse übertragen nur 8 Bit-Datenpakete mit ganzzahligem Datentyp. Der Datentyp eines 32-Bit Fließkomma-Werts muss deshalb mit **CAST\_FLOATTOLONG** von Float in Long gewandelt und der Wert anschließend in 4 separate 8 Bit-Pakete aufgeteilt werden. Beim Empfänger werden die Datenpakete wieder zusammengesetzt und mit **CAST\_LONGTOFLOAT** wieder vom Datentyp Long in Float gewandelt.

### Siehe auch

CAST\_FLOATTOLONG

## CHR

**CHR** weist einer String-Variablen ein Zeichen mit einer bestimmten ASCII-Nummer zu.

### Syntax

```

IMPORT STRING.LI*          '*.LI9 für T9, *.LIA für T10,
                             '*.LIB für T11

CHR(vascii, dest_text)

```

### Parameter

<code>vascii</code>	ASCII-Nummer (0...255) des zugewiesenen Zeichens	<code>LONG</code>
<code>dest_text</code>	String-Variable, der das Zeichen zugewiesen wird	<code>STRING</code>

### Bemerkungen

Wenn eine String-Variable mehr als ein Zeichen (oder Element) hat, weist **CHR** die ASCII-Nummer nur dem ersten Zeichen zu.

### Siehe auch

"" String, + (String-Addition), ASC, FLOTOSTR, FLO40TOSTR, LNG-TOSTR, STRCOMP, STRLEFT, STRLEN, STRMID, STRRIGHT, VALF, VALI

### Beispiel

```

IMPORT STRING.LI9

DIM text_a[1], text_b[1] AS STRING

EVENT:
  CHR(13, text_a)      'Carriage Return
  CHR(10, text_b)     'Line Feed

```

## COS

**COS** liefert den Cosinus eines Winkels.

### Syntax

```
ret_val = COS(angle)
```

### Parameter

<code>angle</code>	Winkel im Bogenmaß ( $-\pi \dots \pi$ )	<code>FLOAT</code>
<code>ret_val</code>	Cosinus des Winkels ( $-1 \dots 1$ )	<code>FLOAT</code>

### Bemerkungen

Wenn Sie für den Winkel Werte außerhalb von  $-\pi \dots +\pi$  verwenden, nimmt der Berechnungsfehler mit wachsendem Wert zu.

Die Ausführungszeit des Befehls beträgt beim T9 bis zu 1,3µs, beim T10 bis zu 0,7µs, beim T11 bis zu 0,31µs.

### Siehe auch

SIN, TAN, ARCCOS, ARCSIN, ARCTAN

### Beispiel

```
DIM val_1, val_2 AS FLOAT
EVENT:
    val_1 = -5.3
    val_2 = COS(val_1)    'Ergebnis: val_2 = 0.55...
```

## CPU\_SLEEP

Nur für Prozessor T11: **CPU\_SLEEP** lässt den Prozessor für eine bestimmte Zeit warten.

### Syntax

**CPU\_SLEEP** (*val*)

### Parameter

*val*                      Anzahl ( $9 \dots 715827879 \approx 2^{30} / 1,5$ ) der zu wartenden Zeiteinheiten in 10ns. LONG

### Bemerkungen

Alternativ gibt es die Anweisungen **P1\_SLEEP** und **P2\_SLEEP** (siehe auch Kapitel 4.2.4 „Wartezeit genau einstellen“). Verwenden Sie bei Prozessoren bis T10 die Anweisung **SLEEP**.

Die Wartezeit sollte grundsätzlich kleiner sein als die mit **PROCESSDELAY** eingestellte Zykluszeit.



Die Anweisung **CPU\_SLEEP** kann bei einem hochprioren Prozess nicht unterbrochen werden. Bitte beachten Sie, dass sehr große Werte bei hochprioren Prozessen zu einem Abbruch der Kommunikation führen können.

Verwenden Sie nach Möglichkeit eine Konstante als Argument. Wenn das Argument *val* eine Berechnung erfordert, benötigt dies eine bestimmte Zeitspanne zusätzlich; diese ist jeweils konstant und beträgt einige wenige Taktzyklen.

Folgende Fälle erfordern eine Berechnung:

- Das Argument ist ein Berechnungsausdruck mit Variablen oder Feldelementen.
- Die Variable im Argument ist für den Speicherbereich **DRAM\_EXTERN** deklariert. Die Zeitspanne kann hier schwanken, weil sie von mehreren Voraussetzungen abhängt.
- Das Argument ist ein Feld.
- Das Argument ist ein Fließkomma-Wert.

### Siehe auch

NOP, P1\_SLEEP, P2\_SLEEP, SLEEP



### Beispiel

#### EVENT:

REM Warten, um eine nachfolgende Messung genau 100 µs  
REM nach dem externen Event-Signal zu starten.

**CPU\_SLEEP**(10000)

...

## DATA\_n

Mit **DIM DATA\_n[...]** **AS** ... wird ein globales **DATA**-Feld dimensioniert.  
Weitere Informationen zur Dimensionierung mit **DIM** siehe Seite 129.

### Syntax

```
DIM DATA_n[dim1] { , DATA_n[dim2] } AS <ARR_TYPE> { AT
<MEM_TYPE> }

DIM DATA_n[dim1] { [dim2] } AS <ARR_TYPE> { AT <MEM_TY-
PE> }
```

### Parameter

<b>DATA_n</b>	Name des deklarierten <b>DATA</b> -Felds (n: 1...200).	
<ARR_TYPE>	Datentyp: <b>FLOAT</b> , <b>LONG</b> , <b>STRING</b>	
dim1, dim2	Feldgröße: Anzahl der Elemente vom Typ <b>ARR_TYPE</b> im Feld.	<b>CONST</b> LONG
<MEM_TYPE>	Speicher, in dem die Variablen abgelegt werden: <b>DRAM_EXTERN</b> : externer Datenspeicher (Default) <b>DM_LOCAL</b> : interner Datenspeicher nur ab T11 verfügbar: <b>EM_LOCAL</b> : Zusätzlicher interner Programm- oder Datenspeicher	

### Bemerkungen

Bei einem Feld können Sie auf die Elemente 1...dim zugreifen. Das Feldelement [0] dürfen Sie nicht verwenden, weil es für interne Zwecke benutzt wird.

Die maximale Feldgröße ist abhängig vom verfügbaren physikalischen Speicher auf dem *ADwin*-System.

Ein globales Feld kann auch 2-dimensional deklariert werden.

## Siehe auch

DIM, FIFO, „2-dimensionale Felder“ auf Seite 57,  
„Variablen und Felder im Datenspeicher“ auf Seite 54

## Beispiel

```
REM Dimensioniere das globale Feld DATA_20 mit  
REM 1000 LONG-Elementen  
DIM DATA_20[1000] AS LONG
```

```
REM Dimensioniere das globale Feld DATA_5 mit  
REM 20 x 75 FLOAT-Elementen  
DIM DATA_5[20][75] AS FLOAT
```

## DEC

**DEC** verringert den Wert einer Long-Variablen um 1.

### Syntax

```
DEC (var)
```

### Parameter

var

Name einer lokalen oder globalen Long-Variablen

**VAR**

**CONST**

LONG

### Bemerkungen

Die Anweisung **DEC** (var) führt zum gleichen Ergebnis wie die Programmzeile: `var=val-1`. Außerdem kann die Anweisung **DEC** eine geringere Ausführungszeit haben.

### Siehe auch

INC, - (Subtraktion)

### Beispiel

```
DIM index AS LONG
```

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

```
INIT:
```

```
index=1000
```

```
EVENT:
```

```
  DAC(1, DATA_1[index]) 'Wert auf DAC1 ausgeben
```

```
  DEC(index) 'index um 1 verringern
```

```
  IF (index<1) THEN
```

```
    index=1000 'Nach 1000 Ausgaben von
```

```
  ENDIF 'vorne beginnen
```

## #DEFINE

**#DEFINE** ersetzt im Quelltext einen symbolischen Namen durch einen frei definierbaren Ausdruck, z. B. eine Konstante.

### Syntax

```
#DEFINE name expression
```

### Parameter

<code>name</code>	Symbolischer Name, <i>ohne</i> Hochkommata. Sonderzeichen sind nicht erlaubt, nur alphanumerische Zeichen (a...z, A...Z, 0...9) und der Unterstrich (_).	<b>CONST</b> <b>STRING</b>
<code>expression</code>	Ausdruck, für den der symbolische Name steht; <i>ohne</i> Hochkommata. Alle Zeichen sind erlaubt.	<b>CONST</b> <b>STRING</b>

### Bemerkungen

Stellen Sie diese Anweisung an den Beginn eines Quelltextes.



Die Funktion **#DEFINE** ist ein Präprozessor-Befehl, d. h. die Ersetzung findet statt, wenn Sie den Quelltext kompilieren lassen (noch bevor der Compiler das lauffähige Programm erzeugt). Verwenden Sie die Funktion, um im Quelltext aussagekräftige Namen anstelle von Konstanten, Parametern oder Berechnungsausdrücken zu verwenden.

Die erste Zeichenfolge bis zu einem Leerzeichen wird als symbolischer Name interpretiert, der nachfolgende Zeileninhalt bis zum Zeilenumbruch als einzufügender Ausdruck<sup>1</sup>. Der Ausdruck wird exakt so eingefügt, wie Sie ihn definiert haben; Variablennamen im Ausdruck werden also nicht durch deren aktuellen Wert, sondern als Zeichenfolge ersetzt.

Groß- und Kleinschreibung wird beim Suchen und Ersetzen nicht unterschieden.

Wenn Sie für `expression` einen Rechenausdruck einsetzen, empfehlen wir, diesen mit Klammern zu umgeben. Sie vermeiden damit

---

1. Text hinter einem Kommentarzeichen „`/*`“ wird vom Compiler ignoriert.

eventuelle Fehler im Zusammenhang mit weiteren Rechenausdrücken.

### Siehe auch

#INCLUDE

### Beispiel

```
#DEFINE setpoint PAR_1'Kommentar, wird nicht ersetzt  
#DEFINE measured DATA_1  
#DEFINE pi 3.141592654
```

Mit diesen Anweisungen können Sie im Quelltext anstelle von `PAR_1`, `DATA_1` und der Ziffernfolge die Namen `setpoint`, `measured` und `pi` verwenden.

```
#DEFINE Sollwert (13 + 4^3)  
PAR_1 = 2 * Sollwert      '= 2 * (13 + 4^3)
```

Ohne die Klammern im **#DEFINE**-Ausdruck würden Sie statt des erwarteten Ergebnisses „154“ den Wert „90“ erhalten.

## DIM

**DIM** deklariert ein oder mehrere

- *lokale* Variablen
- *lokale* eindimensionale Felder (auch Strings)
- *globale* eindimensionale Felder `DATA_n[n]` (auch FIFO-Felder)
- *globale* zweidimensionale Felder `DATA_n[n][m]`

Grundlagen zu Variablen und Datentypen finden Sie in Kapitel 3.2.3 sowie Informationen zu FIFO-Feldern unter dem Stichwort FIFO auf Seite 138.

## Syntax

```

DIM var1 {, var2, ...} AS <VAR_TYPE>

DIM array1[dim1] {, array2[dim2]} AS <VAR_TYPE> {AT
<MEM_TYPE>}

DIM DATA_n[dim1] {, DATA_n[dim2]} AS <VAR_TYPE> {AS
FIFO} {AT <MEM_TYPE>}

DIM DATA_n[dim1][dim2] AS <VAR_TYPE> {AT <MEM_TYPE>}

```

## Parameter

var1, var2	Namen der deklarierten Variablen	
array1, array2, DATA_n	Namen der deklarierten Felder. Für DATA_n kann n aus 1...200 gewählt werden.	
<VAR_TYPE>	Datentyp: <b>FLOAT</b> , <b>LONG</b> für Felder zusätzlich: <b>STRING</b>	
dim1, dim2	Feldgröße: Anzahl ( $\geq 1$ ) der Feldelemente vom Typ <VAR_TYPE>.	<b>CONST</b> <b>LONG</b>
<MEM_TYPE>	Speicher, in dem die Variablen abgelegt werden: <b>DRAM_EXTERN</b> : externer Datenspeicher (Default für Felder) <b>DM_LOCAL</b> : interner Datenspeicher (Default für Variablen) nur ab T11 verfügbar: <b>EM_LOCAL</b> : Zusätzlicher interner Programm- oder Datenspeicher	



## Bemerkungen

Die globalen Variablen `PAR_n` und `FPAR_n` dürfen nicht deklariert werden, weil sie vordefiniert sind.

Wenn Sie vom PC oder aus mehreren Prozessen auf Daten zugreifen wollen, ist dies nur über *globale* Variablen und Felder möglich.

Bei einem Feld können Sie auf die Elemente `1...dim` zugreifen. Das Feldelement `[0]` dürfen Sie nicht verwenden, weil es für interne Zwecke benutzt wird.

Die maximale Feldgröße ist abhängig vom verfügbaren physikalischen Speicher auf dem ADwin-System.

String-Variablen sind *lokale* Felder vom Typ `STRING` (siehe „Strings“ auf Seite 61). Sie können nicht als FIFO deklariert werden.

## Siehe auch

`DATA_n`, `EVENT`:, `FIFO`, `FINISH`:, `INIT`:, `LOWINIT`:, "" String, „2-dimensionale Felder“ auf Seite 57, „Variablen und Felder im Datenspeicher“ auf Seite 54

## Beispiel

```
REM Dimensioniere var1 als LONG-Variable
DIM var1 AS LONG

REM Dimensioniere das Feld array1 mit 1000 LONG-Elementen
DIM array1[1000] AS LONG

REM Dimensioniere das globale Feld DATA_20 mit
REM 1000 LONG-Elementen als Ringspeicher
DIM DATA_20[1000] AS LONG AS FIFO

REM Dimensioniere das Feld TEXT mit
REM 50 Elementen als String-Variable
DIM text[50] AS STRING
```

## DO ... UNTIL

**DO...UNTIL** definiert eine Schleife, deren Anweisungsblock mindestens einmal durchlaufen werden. Die Schleife wird abgebrochen, wenn die Abbruchbedingung den Wert „Wahr“ hat.

### Syntax

```
DO
    ...
UNTIL (condition)
```

### Parameter

condition	Boolesche Abbruchbedingung mit den Operatoren <, >, =, <b>AND</b> und <b>OR</b> .	LOGIC
-----------	---	-------

### Siehe auch

<=> (Vergleich), AND, OR, FOR ... TO ... {STEP ...} NEXT, SELECT-CASE

### Bemerkungen

Sie können **DO...UNTIL**-Schleifen beliebig tief verschachteln; nur die Speichergröße kann Sie hierbei begrenzen.

Vermeiden Sie Schleifen mit langer Ausführungszeit in hochprioren Prozessen, weil diese nicht unterbrochen werden können.

### Beispiel

```
DIM count AS LONG
DIM DATA_1[100] AS LONG AS FIFO

INIT:
    count = 1

EVENT:
    DO
        DATA_1 = ADC(1,4)
        INC count
    UNTIL (count > 100)
```

'Schleife beginnen  
'Messwert auslesen  
'Zählvariable erhöhen  
'100 Messungen durchgeführt?

## END

**END** beendet einen Prozess im **EVENT** : -Abschnitt.

### Syntax

**END**

### Bemerkungen

Der Befehl **END** beendet die Ausführung des **EVENT** : -Abschnitts sofort und beginnt die Ausführung des Abschnitts **FINISH** : (sofern vorhanden). Wenn im Abschnitt **EVENT** : nach der Anweisung **END** weitere Programmzeilen folgen, werden sie nicht mehr ausgeführt.

In den anderen Programmabschnitten ist anstelle von **END** der Befehl **EXIT** anzuwenden.

### Siehe auch

EXIT, PROZESSn\_RUNNING, RESTART\_PROCESS, START\_PROCESS, START\_PROCESS\_DELAYED, STOP\_PROCESS

### Beispiel

```
EVENT:
  IF (ADC(1) > 3000) THEN 'Messen und Vergleichen
    END                  'Prozess beenden, aber FINISH:
  ENDIF                'noch ausführen

FINISH:
  SET_DIGOUT(1)        'Dig. Ausgang 1 setzen
```

## EVENT:

Das Kennwort **EVENT**: bezeichnet den Anfang des Haupt-Programmabschnitts, der bei jedem Event-Signal aufgerufen wird.

### Syntax

**EVENT:** { **AT** <MEM\_TYPE> }

### Parameter

<MEM\_TYPE> nur für T11: Speicherbereich, in dem der Programmabschnitt abgelegt wird.  
**PM\_LOCAL**: interner Programmspeicher (Default)  
**EM\_LOCAL**: Zusätzlicher interner Programm- oder  
 Datenspeicher **DRAM\_EXTERN**: externer Datenspeicher

### Bemerkungen

Zur Übersicht der Programmabschnitte siehe Kapitel 3.1.1 auf Seite 45.

Der Programmabschnitt **EVENT**: ist der zentrale Funktionsabschnitt, der im Prozess (typischerweise) in regelmäßigen Abständen aufgerufen wird, bis er gestoppt wird. Je nach Einstellung wird der Aufruf durch einen zyklischen Timer-Event oder durch einen externen Event ausgelöst. Näheres ist in Kapitel 5 „Prozesse im Betriebssystem“ beschrieben.

Bei dem Prozessortyp T11 kann jeder Programmabschnitt separat einem bestimmten Speicherbereich zugeordnet werden (siehe Kapitel 3.3.2 „Speicherbereiche“). Der große, aber langsamere Speicherbereich **DRAM\_EXTERN** sollte nur für zeitunkritische Programmabschnitte genutzt werden; meistens sind das die Abschnitte **LOWINIT**:, **INIT**:, **FINISH**..

### Siehe auch

**DIM**, **LOWINIT**:, **INIT**:, **FINISH**:

## Beispiel

```
DIM val_1 AS FLOAT
```

```
EVENT:
```

```
val_1 = -5.3
```

## EXIT

**EXIT** beendet einen Prozess in den Abschnitten **LOWINIT:**, **INIT:** oder **FINISH:**.

### Syntax

**EXIT**

### Bemerkungen

Der Prozess wird sofort beendet. Wenn im gleichen Abschnitt noch weitere Programmzeilen folgen, werden sie nicht mehr ausgeführt.

Verwenden Sie im Abschnitt **EVENT:** den Befehl **END**.

### Siehe auch

END, PROZESSn\_RUNNING, RESET\_EVENT, RESTART\_PROCESS, START\_PROCESS, START\_PROCESS\_DELAYED, STOP\_PROCESS

### Beispiel

```
INIT:
  IF (ADC(1) > 3000) THEN 'Messen und Vergleichen
    SET_DIGOUT(0)         'Dig. Ausgang setzen
    EXIT                  'Diesen Prozess beenden
  ENDIF
```

## EXP

**EXP** berechnet eine Potenz zur Basis e.

### Syntax

```
ret_val = EXP(val)
```

### Parameter

val	Argument	FLOAT
ret_val	Exponentialwert des Arguments zur Basis e.	FLOAT

### Bemerkungen

Die Ausführungszeit des Befehls beträgt beim T9 bis zu 1,3µs, beim T10 bis zu 0,7µs, beim T11 bis zu 0,31µs.

### Siehe auch

LN, LOG

### Beispiel

```
DIM val_1, val_2 AS FLOAT

EVENT:
  val_1 = 5
  val_2 = EXP(val_1)    'Ergebnis: val_2 = 148,41...
```

## FIFO

Mit **DIM DATA\_n AS FIFO** wird ein globales **DATA**-Feld als Ringspeicher dimensioniert.

### Syntax

```
DIM DATA_n[dim] AS <ARR_TYPE> AS FIFO
```

### Parameter

<b>DATA_n</b>	Name des deklarierten <b>DATA</b> -Felds (n: 1...200).	
<b>&lt;ARR_TYPE&gt;</b>	Festgelegter Datentyp: <b>FLOAT</b> , <b>LONG</b>	
<b>dim</b>	Feldgröße ( $\geq 1$ ): Anzahl der Elemente vom Typ <b>ARR_TYPE</b> im Feld. Beim Prozessortyp T11 ist der Wertebereich für <b>dim</b> nur in Schritten einstellbar: $\text{dim} = 4 \times a + 3; a \geq 0$	<div><b>CONST</b></div> <div><b>LONG</b></div>

### Bemerkungen

Sie können nur **DATA**-Felder als FIFO-Ringspeicher verwenden (siehe auch Kapitel 3.3.4 auf Seite 58). Dadurch kann ein FIFO-Feld nicht mehr gleichzeitig als „normales“ Feld verwendet werden.

FIFO-Felder (First in, first out) werden mit Datenzeigern verwaltet. Nach der Dimensionierung sollten Sie diese Datenzeiger mit dem Befehl **FIFO\_CLEAR** initialisieren, z.B. im Abschnitt **LOWINIT**: oder **INIT**:. Die im FIFO enthaltenen Daten werden weder beim Initialisieren noch beim Dimensionieren nicht verändert.

Wenn Sie schneller Daten in ein FIFO-Feld schreiben als auslesen, werden alte gespeicherte Daten überschrieben und gehen damit verloren. Zur Abhilfe können Sie die Befehle **FIFO\_EMPTY** und **FIFO\_FULL** verwenden.

Wenn Sie beim Prozessor T11 eine nicht erlaubte Feldgröße **dim** angeben, wird das FIFO-Feld automatisch mit der nächstgrößeren erlaubten Feldgröße dimensioniert. Beispielsweise ändert der Compiler die Feldgröße [1000] automatisch in [1003].



**Siehe auch**

DIM, DATA\_n, FIFO\_CLEAR, FIFO\_EMPTY, FIFO\_FULL

**Beispiel**

```
REM Dimensioniere das globale Feld DATA_20 mit  
REM 1000 LONG-Elementen als Ringspeicher  
DIM DATA_20[1000] AS LONG AS FIFO
```

## FIFO\_CLEAR

**FIFO\_CLEAR** initialisiert den Schreib- und Lese-Zeiger eines FIFO-Felds.

### Syntax

```
FIFO_CLEAR(arraynum)
```

### Parameter

`arraynum`      Nummer des DATA-FIFO-Felds (1...200)

LONG
------

### Bemerkungen

Das Initialisieren des Schreib- und des Lese-Zeigers verändert die im Feld enthaltenen Daten nicht.

Die FIFO-Zeiger werden bei der Dimensionierung nicht initialisiert. Sie sollten dies im Abschnitt **LOWINIT**: oder **INIT**: mit **FIFO\_CLEAR** tun.

Während des Programmlaufes ist das Initialisieren der FIFO-Zeiger sinnvoll, wenn Sie alle im Feld gesammelten Daten (z.B. wegen eines Messfehlers) verwerfen wollen.

### Siehe auch

FIFO, FIFO\_EMPTY, FIFO\_FULL

### Beispiel

```
DIM DATA_1[20000] AS LONG AS FIFO 'Deklaration
DIM reinit_fifo_flag AS LONG

INIT:
    FIFO_CLEAR(1)           'FIFO-Zeiger initialisieren

EVENT:
    REM Anzahl der freien Plätze im FIFO-Feld abfragen
    IF (FIFO_EMPTY(1) > 1) THEN
        'Analogen Eingang 1 messen und im FIFO speichern
        DATA_1 = ADC(1)
    ENDIF

    .
    .                       ' Programmtext
    .

    IF (reinit_fifo_flag) THEN 'z.B. Fehler geschehen
        FIFO_CLEAR(1)         'FIFO Zeiger initialisieren
    ENDIF
```

## FIFO\_EMPTY

**FIFO\_EMPTY** ermittelt die Anzahl der freien Elemente in einem FIFO-Feld.

### Syntax

```
ret_val = FIFO_EMPTY(arraynum)
```

### Parameter

arraynum	Nummer des DATA-FIFO-Felds (1...200)	LONG
ret_val	Anzahl der freien Feldelemente	LONG

### Bemerkungen

Wenn Sie Daten in ein FIFO-Feld schreiben wollen, sollten Sie vorher mit diesem Befehl überprüfen, ob noch genügend Platz im FIFO frei ist.

### Siehe auch

FIFO, FIFO\_CLEAR, FIFO\_FULL

### Beispiel

```
DIM DATA_1[20000] AS LONG AS FIFO'Deklaration

INIT:
    FIFO_CLEAR(1)           'FIFO-Zeiger initialisieren

EVENT:
    REM Anzahl der freien Plätze im FIFO-Feld abfragen
    IF (FIFO_EMPTY(1) > 1) THEN
        REM Analogen Eingang 1 messen und im FIFO speichern
        DATA_1 = ADC(1)
    ENDIF
```

## FIFO\_FULL

**FIFO\_FULL** ermittelt die Anzahl der belegten Elemente in einem FIFO-Feld.

### Syntax

```
ret_val = FIFO_FULL(arraynum)
```

### Parameter

arraynum	Nummer des DATA-FIFO-Felds (1...200)	LONG
ret_val	Anzahl der belegten Feldelemente (0...dim)	LONG

### Bemerkungen

Wenn Sie Daten aus einem FIFO-Feld lesen oder verwenden wollen, sollten Sie vorher mit diesem Befehl überprüfen, ob im FIFO noch Daten enthalten sind. Falls keine Daten mehr vorhanden sind, wird aus dem FIFO-Feld ein undefinierter Wert gelesen.

### Siehe auch

FIFO, FIFO\_CLEAR, FIFO\_EMPTY

### Beispiel

```
DIM DATA_1[20000] AS LONG AS FIFO 'Deklaration

INIT:
    FIFO_CLEAR(1)           'FIFO-Zeiger initialisieren

EVENT:
    REM Abfrage, ob Daten im FIFO 'enthalten sind
    IF (FIFO_FULL(1) > 0) THEN
        REM Einen FIFO-Wert auf dem analogen Ausgang 1 ausgeben
        DAC(1, DATA_1)
    ENDIF
```

## FINISH:

Das Kennwort **FINISH:** bezeichnet den Anfang des Programmabschnitts zur Schlussbearbeitung. Der Programmabschnitt hat in jedem Fall niedrige Priorität, Stufe 1.

### Syntax

```
FINISH: {AT MEM_TYPE}
```

### Parameter

**<MEM\_TYPE>** nur für T11: Speicherbereich, in dem der Programmabschnitt abgelegt wird.  
**PM\_LOCAL:** interner Programmspeicher (Default)  
**EM\_LOCAL:** Zusätzlicher interner Programm- oder Datenspeicher  
**DRAM\_EXTERN:** externer Datenspeicher

### Bemerkungen

Zur Übersicht der Programmabschnitte siehe Kapitel 3.1.1 auf Seite 45.

Der Programmabschnitt **FINISH:** wird einmalig durchlaufen, sobald der Prozess gestoppt wird.

Bei dem Prozessortyp T11 kann jeder Programmabschnitt separat einem bestimmten Speicherbereich zugeordnet werden (siehe Kapitel 3.3.2 „Speicherbereiche“). Der große, aber langsamere Speicherbereich **DRAM\_EXTERN** sollte nur für zeitunkritische Programmabschnitte genutzt werden; meistens sind das die Abschnitte **LOWINIT:**, **INIT:**, **FINISH:**.

### Siehe auch

**DIM, LOWINIT:, INIT:, EVENT:**

### Beispiel

```
DIM val_1 AS FLOAT
```

```
FINISH:  
    val_1 = -5.3
```

## FLOTOSTR

**FLOTOSTR** konvertiert eine Fließkomma-Zahl (float) in eine Zeichenfolge (String).

### Syntax

```
IMPORT STRING.LI*          '*.LI9 für T9, *.LIA für T10,  
                             '*.LIB für T11  
  
FLOTOSTR(val, string[])
```

### Parameter

<code>val</code>	Zu wandelnder Wert.	<b>FLOAT</b>
<code>string[]</code>	Erstellte Zeichenfolge im Format: {-}#.#####E{-}##	<b>ARRAY</b> <b>STRING</b>

### Bemerkungen

Die Länge der Zeichenfolge variiert von 11 bis 13 Zeichen, abhängig von den Vorzeichen der Mantisse und des Exponenten.

### Siehe auch

"" String, ASC, CHR, FLO40TOSTR, LNGTOSTR, STRCOMP, STRLEFT, STRLEN, STRMID, STRRIGHT, VALF, VALI

**Beispiel**

```
IMPORT STRING.LI9          'String-Library für T9

DIM text[13] AS STRING
DIM pi, number AS FLOAT

INIT:
  pi = 3.141592654
  FPAR_1 = -pi^-20

EVENT:
  REM Fließkomma-Zahl in einen String wandeln
  FLOTOSTR(FPAR_1, text)
  PAR_1 = text[1]          'String-Länge = 13
  PAR_2 = text[2]          'ASCII-Zeichen 2Dh = "-"
  PAR_3 = text[3]          'ASCII-Zeichen 31h = "1"
  PAR_4 = text[4]          'ASCII-Zeichen 2Eh = "."
  PAR_5 = text[5]          'ASCII-Zeichen 31h = "1"
  PAR_6 = text[6]          'ASCII-Zeichen 34h = "4"
  PAR_7 = text[7]          'ASCII-Zeichen 30h = "0"
  PAR_8 = text[8]          'ASCII-Zeichen 32h = "2"
  PAR_9 = text[9]          'ASCII-Zeichen 35h = "5"
  PAR_10 = text[10]         'ASCII-Zeichen 35h = "5"
  PAR_11 = text[11]         'ASCII-Zeichen 45h = "E"
  PAR_12 = text[12]         'ASCII-Zeichen 2Dh = "-"
  PAR_13 = text[13]         'ASCII-Zeichen 31h = "1"
  PAR_14 = text[14]         'ASCII-Zeichen 30h = "0"
  PAR_15 = text[15]         'String-Ende-Zeichen = 0
```



## FLO40TOSTR

Nur für Prozessor T11: **FLO40TOSTR** konvertiert eine Fließkomma-Zahl (float) von 40 Bit in eine Zeichenfolge (String).

### Syntax

```
IMPORT STRING.LIB          '*.LIB für T11
FLO40TOSTR(val, string[])
```

### Parameter

<code>val</code>	Zu wandelnder Wert	<b>FLOAT</b>
<code>string[]</code>	Erstellte Zeichenfolge im Format: {-}#.#####E{-}##	<b>ARRAY</b> <b>STRING</b>

### Bemerkungen

Die Länge der Zeichenfolge variiert von 13 bis 15 Zeichen, abhängig von den Vorzeichen der Mantisse und des Exponenten.

### Siehe auch

"" String, ASC, CHR, FLOTOSTR, LNGTOSTR, STRCOMP, STRLEFT, STRLEN, STRMID, STRRIGHT, VALF, VALI

**Beispiel**

```

IMPORT STRING.LIB          'String-Library für T11

DIM text[15] AS STRING
DIM pi, number AS FLOAT

INIT:
    pi = 3.141592654
    FPAR_1 = -pi^-20

EVENT:
    REM Fließkomma-Zahl in einen String wandeln
    FLO40TOSTR(FPAR_1, text)
    PAR_1 = text[1]          'String-Länge = 15
    PAR_2 = text[2]          'ASCII-Zeichen 2Dh = "-"
    PAR_3 = text[3]          'ASCII-Zeichen 31h = "1"
    PAR_4 = text[4]          'ASCII-Zeichen 2Eh = "."
    PAR_5 = text[5]          'ASCII-Zeichen 31h = "1"
    PAR_6 = text[6]          'ASCII-Zeichen 34h = "4"
    PAR_7 = text[7]          'ASCII-Zeichen 30h = "0"
    PAR_8 = text[8]          'ASCII-Zeichen 32h = "2"
    PAR_9 = text[9]          'ASCII-Zeichen 35h = "5"
    PAR_10 = text[10]         'ASCII-Zeichen 35h = "6"
    PAR_11 = text[11]         'ASCII-Zeichen 35h = "4"
    PAR_12 = text[12]         'ASCII-Zeichen 35h = "7"
    PAR_13 = text[13]         'ASCII-Zeichen 45h = "E"
    PAR_14 = text[14]         'ASCII-Zeichen 2Dh = "-"
    PAR_15 = text[15]         'ASCII-Zeichen 31h = "1"
    PAR_16 = text[16]         'ASCII-Zeichen 30h = "0"
    PAR_17 = text[17]         'String-Ende-Zeichen = 0

```

## FOR ... TO ... {STEP ... } NEXT

**FOR...NEXT** definiert eine Schleife, die eine bestimmte Zahl an Durchläufen haben sollen.

### Syntax

```
FOR i = X TO Y {STEP Z}
...
'Anweisungsblock
NEXT i
```

### Parameter

i	lokale Zählvariable	LONG
X	Startwert der Laufvariablen	LONG
Y	Endwert der Laufvariablen	LONG
Z	Schrittweite ( $\geq 1$ ) der Laufvariablen; Vorgabewert: 1	LONG

### Bemerkungen

Der Anweisungsblock wird in jedem Fall einmal ausgeführt, auch wenn der Startwert **X** größer als der Endwert **Y** ist.

Deklarieren Sie die Zählvariable als lokale Variable (Datentyp **LONG**).

Ein hochpriorer Prozess kann von keinem anderen Prozess unterbrochen werden, auch wenn gerade eine zeitaufwändige Schleife bearbeitet wird. Während dieser Zeit kann der Prozessor des *ADwin*-Systems nicht auf andere Events reagieren. Die Schleife darf deshalb in hochprioren Prozessen nur verwendet werden, wenn die Anzahl der Schleifendurchläufe niedrig gehalten wird.



### Siehe auch

DO ... UNTIL, IF ... THEN ... {ELSE ... } ENDIF, SELECTCASE

**Beispiel**

```
DIM index AS LONG
DIM sinus[360] AS FLOAT 'Feld für Sinus-Werte
DIM pi AS FLOAT

INIT:
  pi = 3.14159
  REM Sinuswerte in Grad-Schritten berechnen (0° bis 359°)
  FOR index = 1 TO 360
    sinus[index] = (2047*SIN((index - 1) * 2*pi/360))
  NEXT index
  index = 1           'Zählindex initialisieren

EVENT:
  DAC(1, sinus[index]) 'Amplitudenwert ausgeben
  INC index             'Zählindex erhöhen
  REM Ab 360 Grad wieder bei 0 beginnen
  IF (index > 360) THEN index = 1
```

## FUNCTION ... ENDFUNCTION

**FUNCTION...ENDFUNCTION** definiert ein Funktions-Makro mit Übergabeparametern und einem Rückgabewert.

### Syntax

```

FUNCTION macro_name({val_1, val_2, ...}) AS <VAR_TYPE>
    {DIM var AS <VAR_TYPE>}
    ...
    'Anweisungsblock
    macro_name = ... 'Rückgabewert zuweisen
ENDFUNCTION

```

### Parameter

<code>macro_name</code>	Name der Funktion und Rückgabewert, Datentyp <b>&lt;VAR_TYPE&gt;</b>	
<code>val_1, val_2</code>	Namen der Übergabeparameter; für Felder ist die Syntax mit Dimensionsklammern erforderlich: <code>array[]</code> oder <code>DATA_n[]</code> .	<div style="border: 1px solid black; padding: 2px; display: inline-block;">FLOAT</div> <div style="border: 1px solid black; padding: 2px; display: inline-block;">LONG</div> <div style="border: 1px solid black; padding: 2px; display: inline-block;">STRING</div>
<b>&lt;VAR_TYPE&gt;</b>	Datentyp der Funktion bzw. des Rückgabewerts: <b>FLOAT</b> , <b>LONG</b> ; nicht aber <b>STRING</b>	

### Bemerkungen

Allgemeine Informationen über Makros finden Sie in Kapitel 3.5.1 auf Seite 68.

Diese Anweisung definiert ein Funktions-Makro, d.h. der vollständige Anweisungsblock zwischen **FUNCTION** und **ENDFUNCTION** wird an der aufrufenden Stelle eingefügt.

Funktionen erhöhen die Übersichtlichkeit Ihres Quelltextes. Beachten Sie aber, dass jeder Funktionsaufruf die kompilierte Datei vergrößert.

Sie können Funktionen an 3 Stellen einfügen:

1. Vor dem Abschnitt **INIT**: / **LOWINIT**:
2. Nach dem Abschnitt **FINISH**:
3. In einer separaten Datei, die Sie mit **#INCLUDE** einbinden (aber nur an einer der Stellen, die unter 1. und 2. angegeben sind).

Beachten Sie bitte, dass Sie in Funktionen:

- keine Prozess-Abschnitte wie **LOWINIT**:, **INIT**:, **EVENT**:, oder **FINISH**: definieren.
- am Anfang lokale Variablen definieren können, die nur innerhalb der Funktion und für die Dauer der Abarbeitung verfügbar sind.  
Eine lokale Variable kann den gleichen Namen haben wie eine Variable, die außerhalb der Funktion definiert wurde.
- dem Funktionsnamen einen Wert zuweisen, damit dieser zum Rückgabewert an die aufrufende Stelle wird.

Eine Funktion wird mit ihrem Namen und allen definierten Argumenten aufgerufen; die Funktion muss in der aufrufenden Programmzeile als Argument verwendet werden, z. B. in einer Zuweisung (siehe Beispiel). Als Argument ist jeder Berechnungsausdruck (auch Felder) zulässig, solange er den passenden Datentyp hat.

Wenn Sie keine Argumente definieren, müssen Sie dennoch beim Aufruf der Funktion die Leerklammern verwenden: `name()`.

Wenn ein Feld als Übergabeparameter einer Funktion verwendet wird, ist die Syntax für Definition und Aufruf unterschiedlich:

- Funktions-Definition *mit* Dimensions-Klammern:  
**FUNCTION** `name`(`array`[])
- Funktions-Aufruf *ohne* Dimensions-Klammern:  
`ret_val=name`(`array`)



Wenn Sie in einer Funktion einem Übergabeparameter einen Wert zuweisen, darf beim Funktionsaufruf für diesen Übergabeparameter nur eine Variable oder ein einzelnes Feld-Element als Argument verwendet werden.

Bei Berechnungsausdrücken in einer Funktion sollten die Übergabeparameter in Klammern stehen. Auf diese Weise vermeiden Sie Probleme mit der Rangfolge von Operatoren (z. B. Punkt- vor Strich-Rechnung).

### Siehe auch

```
#INCLUDE, SUB ... ENDSUB, LIB_FUNCTION ... LIB_ENDFUNCTION, LIB_SUB ... LIB_ENDSUB
```

**Beispiel**

```
FUNCTION average(w1, w2, w3) AS FLOAT
REM Die Funktion berechnet den Mittelwert aus den Werten
REM w1, w2 und w3
    DIM sum AS FLOAT
    sum = w1 + w2 + w3
    average = sum/3
ENDFUNCTION
```

Ein Aufruf der Funktion erfolgt z.B. mit den Programmzeilen:

```
x = average(x1, x2, x3)
DAC(1,average(x1, x2, x3))
```

Die gleiche Funktion mit einem Feld als Übergabe-Parameter:

```
FUNCTION average_array(array[]) AS FLOAT
    average_array=(array[1] + array[2] + array[3])/3
ENDFUNCTION
```

Der Aufruf dieser Funktion erfolgt wieder in ähnlicher Weise (allerdings *ohne* die Dimensionsklammern):

```
x = average_array(array)
DAC(1,average_array(array))
```

Beim Aufruf können Sie für `array` ein globales oder ein lokales Feld angeben. Tragen Sie nur den Feldnamen ein ohne Elementnummer und eckige Klammern.

## IF ... THEN ... {ELSE ... } ENDIF

Die Kontrollstruktur bewirkt in Abhängigkeit von einer Bedingung die Ausführung einer Anweisung (**IF...THEN...**) oder eines Anweisungsblocks (**IF...THEN...ELSE...ENDIF**).

### Syntax

```

IF (condition) THEN
    ...                               'Anweisungsblock
{ ELSE                               'Der else-Teil ist optional
    ...                               'Anweisungsblock }
ENDIF
oder
IF (condition) THEN instr

```

### Parameter

condition	Boolesche Bedingung mit den Operatoren <b>&lt;</b> , <span style="border: 1px solid black; padding: 0 2px;">LOGIC</span> <b>&gt;</b> , <b>=</b> , <b>AND</b> und <b>OR</b> . Wenn die Bedingung „Wahr“ ist, werden die Anweisungen nach <b>THEN</b> ausgeführt.
instr	Anweisung (entspricht einer Befehlszeile).

### Bemerkungen

Sie können **IF**-Strukturen beliebig tief verschachteln; nur die Speichergröße begrenzt Sie hierbei.

Der Anweisungsblock nach **ELSE** wird (falls vorhanden) schneller ausgeführt als der nach **IF...THEN**. Dies beschleunigt die Gesamt-Ausführungszeit des **EVENT**:-Abschnitts, weil die Bedingung meistens den Wert „Falsch“ hat, z.B. bei der Prüfung auf Überschreiten von Grenzwerten.

In der einzeiligen Variante darf die Anweisung weder ein Unterprogramm-Makro (**SUB**) noch ein Funktion-Makro (**FUNCTION**) aufrufen.

### Siehe auch

**< = >** (Vergleich), **AND**, **OR**, **DO ... UNTIL**, **SELECTCASE**



### Beispiel

```
DIM val AS LONG           'Deklaration

EVENT:
    val = ADC(1)             'Messwert erfassen

    IF (val > 3000) THEN      'Grenzwert überschritten:
        CLEAR_DIGOUT(1)      'Rücksetzen DIGOUT 1
        SET_DIGOUT(0)        'Setzen DIGOUT 0
    ELSE                     'Grenzwert nicht überschritten:
        CLEAR_DIGOUT(0)      'Rücksetzen DIGOUT 0
        SET_DIGOUT(1)        'Setzen DIGOUT 1
    ENDIF                   'Kontrollstruktur Ende
```

## #IF ... THEN ... {#ELSE ... } #ENDIF

Die Präprozessor-Anweisung bewirkt in Abhängigkeit von einer Bedingung die Kompilierung eines Anweisungsblocks (**#IF...THEN...#ELSE...#ENDIF**).

### Syntax

```
#IF condition THEN
...
                                'Anweisungsblock
{ #ELSE                        'Der else-Teil ist optional
...
                                'Anweisungsblock}
#ENDIF
```

### Parameter

**condition**      Boolesche Bedingung (ohne Klammern und LOGIC Anführungszeichen) in der Form:

**<SYSPAR>** = value

Wenn die Bedingung erfüllt ist, werden die Anweisungen nach **THEN** ausgeführt.

Die Systemparameter **<SYSPAR>** und die zugehörigen Werte **value** sind in der Tabelle unten aufgeführt.

<b>&lt;SYSPAR&gt;</b>	value	Bedeutung
<b>ADWIN_SYSTEM</b>	ADWIN_CARD ADWIN_GOLD ADWIN_L16 ADWIN_PRO	Einstellung „System“ im Fenster „Compiler Options“.
<b>PROZESSOR</b>	T9 T10 T11	Einstellung „Processor“ im Fenster „Compiler Options“.

### Bemerkungen

In der Bedingung darf nur der Operator „=“ verwendet werden; weder Boolesche Verknüpfungen mit **AND** und **OR** noch Klammerungen sind erlaubt. Sie können **#IF**-Strukturen beliebig tief verschachteln; nur die Speichergröße begrenzt Sie hierbei.

Es gibt keine einzeilige Variante wie bei **IF...THEN**.

Bei einem Kommandozeilen-Aufruf des Compilers (siehe Seite A-12) beziehen sich die Systemparameter auf die beim Aufruf übergebenen Parameter /Sx und /Px.

### Siehe auch

< = > (Vergleich), IF ... THEN ... {ELSE ... } ENDIF

### Beispiel

```
REM niederprioreres Processdelay auf 800µs setzen
#IF PROZESSOR = T11 THEN
    REM T11: 800µs = 240000 x 3,3ns
    PROCESSDELAY = 240000
#ELSE
    #IF PROZESSOR = T10 THEN
        REM T10: 800µs = 16 x 50µs
        PROCESSDELAY = 16
    #ELSE
        REM T9: 800µs = 8 x 100µs (auch andere CPUs)
        PROCESSDELAY = 8
    #ENDIF
#ENDIF
```

## IMPORT

**IMPORT** bindet Funktionen und Unterprogramme aus der angegebenen Library-Datei bei der Kompilierung mit ein.

### Syntax

```
IMPORT {path}file
```

### Parameter

file	Dateiname der Library-Datei <i>ohne</i> Anführungszeichen. Die Dateiendung ist .LI9 für T9, .LIA für T10, .LIB für T11.	<b>CONST</b> <b>STRING</b>
path	Pfadname der Library-Datei (mit Laufwerk); <i>ohne</i> Anführungszeichen	<b>CONST</b> <b>STRING</b>

### Bemerkungen

Fügen Sie **IMPORT**-Anweisungen ganz am Anfang Ihres Quelltextes ein (vor der Variablendeklaration). Wenn Sie im Quelltext einer Library-Datei weitere Library-Dateien importieren, müssen Sie dies zusätzlich auch im aufrufenden Quelltext tun.

Es werden nur diejenigen Funktionen und Unterprogramme aus der Library-Datei eingebunden, die Sie in Ihrem Quelltext aufrufen.

Sie sollten immer den vollständigen Pfadnamen angeben, da ansonsten nur im Standard-Verzeichnis (siehe Das Menü „Options“ Blatt „Directory“, Seite 25) gesucht wird. Verwenden Sie bei der Pfadangabe den Backslash "\", um Verzeichnisnamen voneinander zu trennen.

### Siehe auch

#INCLUDE, LIB\_FUNCTION ... LIB\_ENDFUNCTION, LIB\_SUB ... LIB\_ENDSUB

### Beispiel

```
IMPORT STRING.LI9      'Importiert die String-Library
                        'für den Prozessor T9
IMPORT C:\MyFiles\ADwinLibs\dig2volt.LIA'Importiert eine
                        'benutzerdefinierte Library für T10
```

## INC

**INC** erhöht den Wert einer lokalen oder globalen ganzzahligen Variablen um Eins.

### Syntax

**INC** (*var*)

### Parameter

*var*

Name einer lokalen oder globalen Long-Variablen

**VAR**

**CONST**

LONG

### Bemerkungen

Die Anweisung **INC** (*val*) führt zum gleichen Ergebnis wie die Programmzeile: *val=val+1*. Außerdem kann diese Anweisung eine geringere Ausführungszeit haben.

### Siehe auch

DEC, + (Addition)

### Beispiel

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

INIT:
    index=1

EVENT:
    DATA_1[index] = ADC(1) 'Messwert im Feld ablegen
    INC(index)              'index um 1 erhöhen
    IF (index>1000) THEN END 'Nach 1000 Messungen das Programm
                              'beenden
```

## #INCLUDE

**#INCLUDE** bindet den vollständigen Inhalt einer Include-Datei in den Quelltext ein.

### Syntax

```
#INCLUDE {path}filename
```

### Parameter

<code>filename</code>	Name der einzubindenden Datei (mit Endung „.inc“) ohne Anführungszeichen	<b>CONST</b> <b>STRING</b>
<code>path</code>	vollständiger Pfad mit Laufwerk.	<b>CONST</b> <b>STRING</b>

### Bemerkungen

Allgemeine Informationen über Include-Dateien finden Sie in Kapitel 3.5.2 auf Seite 69.

Fügen Sie **#INCLUDE**-Anweisungen ganz am Anfang Ihres Quelltextes ein (vor der Variablendeklaration). Im Quelltext einer Include-Datei können Sie weitere Include-Dateien importieren.

Wenn die eingebundene Include-Datei Library-Funktionen verwendet, müssen Sie mit **IMPORT** auch die entsprechenden Library-Dateien einbinden.

Sie sollten immer den vollständigen Pfadnamen angeben, da ansonsten nur im Standard-Verzeichnis (siehe Das Menü „Options“ Blatt „Directory“, Seite 25) gesucht wird. Verwenden Sie bei der Pfadangabe den Backslash "\", um Verzeichnisnamen voneinander zu trennen.



Beachten Sie bitte: Eine Zeile mit **#INCLUDE**-Anweisung darf höchstens 136 Zeichen lang sein (Zeilenlänge für alle anderen Zeilen siehe Seite 107). Alle weiteren Zeichen der Zeile schneidet der Compiler ab.

### Siehe auch

**#DEFINE**, **IMPORT**, **FUNCTION ... ENDFUNCTION**, **SUB ... END-SUB**

### Beispiel

```
#INCLUDE C:\Test\demofunc.inc
```

```
#INCLUDE demofunc.inc 'Datei im Standard-Verzeichnis suchen'
```

## INIT:

Das Kennwort **INIT**: bezeichnet den Anfang eines Programmabschnitts zur Initialisierung.

### Syntax

**INIT:** {**AT MEM\_TYPE**}

### Parameter

**<MEM\_TYPE>** nur für T11: Speicherbereich, in dem der Programmabschnitt abgelegt wird.  
**PM\_LOCAL**: interner Programmspeicher (Default)  
**EM\_LOCAL**: Zusätzlicher interner Programm- oder Datenspeicher  
**DRAM\_EXTERN**: externer Datenspeicher

### Bemerkungen

Zur Übersicht der Programmabschnitte siehe Kapitel 3.1.1 auf Seite 45.

Der Programmabschnitt **INIT**: wird einmalig durchlaufen, sobald der Prozess gestartet wird und der Programmabschnitt **LOWINIT**: (falls vorhanden) beendet ist.

Der Programmabschnitt hat die für den Prozess eingestellte Priorität (Menüpunkt „Options / Process“). Bei hoher Priorität kann der Programmabschnitt nicht unterbrochen werden und sollte dann möglichst kurz sein.

Bei dem Prozessortyp T11 kann jeder Programmabschnitt separat einem bestimmten Speicherbereich zugeordnet werden (siehe Kapitel 3.3.2 „Speicherbereiche“). Der große, aber langsamere Speicherbereich **DRAM\_EXTERN** sollte nur für zeitunkritische Programmabschnitte genutzt werden; meistens sind das die Abschnitte **LOWINIT**:, **INIT**:, **FINISH**:.

### Siehe auch

**DIM**, **LOWINIT**:, **EVENT**:, **FINISH**:



## Beispiel

```
DIM val_1 AS FLOAT
INIT:
    val_1 = -5.3
```

## LIB\_FUNCTION ... LIB\_ENDFUNCTION

**LIB\_FUNCTION...LIB\_ENDFUNCTION** definiert in einer Library-Datei eine Funktion mit Übergabe- und Rückgabeparametern.

### Syntax

```
LIB_FUNCTION lib_name (<LIB_PAR1> {, <LIB_PAR2>, ...} )
  AS <FCT_TYPE>

  {DIM var AS <VAR_TYPE>}

  {#DEFINE name expression}

  ...                               'Anweisungsblock

  lib_name = ...

LIB_ENDFUNCTION
```

Syntax der Übergabeparameter <LIB\_PAR>:

```
<BY_TYPE> var_name AS <VAR_TYPE> {AT <MEM_TYPE>}
```

### Parameter

<code>lib_name</code>	Name der Library-Funktion und des Rückgabewerts; Datentyp <b>&lt;FCT_TYPE&gt;</b> .
<b>&lt;FCT_TYPE&gt;</b>	Datentyp: <b>FLOAT, LONG</b> .
<code>var_name</code>	Name eines Übergabeparameters innerhalb der Library-Funktion; für Felder ist die Syntax mit Dimensionsklammern erforderlich: <code>array[]</code> oder <code>DATA_n[]</code> .
<b>&lt;BY_TYPE&gt;</b>	Methode zur Übergabe der Parameter: <b>BYREF</b> : Zeiger auf Variable oder Feld übergeben. <b>BYVAL</b> : Wert übergeben.
<b>&lt;VAR_TYPE&gt;</b>	Datentyp: <b>FLOAT, LONG, STRING</b> .
<b>&lt;MEM_TYPE&gt;</b>	Sinnvoll nur für Prozessor T10: Speicher, in dem die Variablen abgelegt werden; nur gemeinsam mit Fel- dern verfügbar: <b>DRAM_EXTERN</b> : externer Datenspeicher. <b>DM_LOCAL</b> : interner Datenspeicher.

### Bemerkungen

Allgemeine Informationen über Library-Dateien finden Sie in Kapitel 3.5.3 auf Seite 69.

Erstellen Sie Library-Funktionen (und -Unterprogramme) in einer eigenen Quelltext-Datei. Nach der Kompilierung mit „Build / Make lib file“ können Sie mit dem Befehl **IMPORT** diejenigen Module einer Library in einen Prozess einbinden, die dort tatsächlich aufgerufen werden.

Innerhalb einer Library-Funktion können Sie

- lokale Variablen und Felder (nur eindimensional) deklarieren und verwenden.  
Deklarieren Sie Variablen immer am Anfang, keinesfalls außerhalb des Unterprogramms.
- globale Variablen und Felder verwenden, wenn sie als Parameter übergeben werden.
- nur eindimensionale Felder bearbeiten.  
Sie können zweidimensionale Felder als Parameter übergeben, diese werden aber innerhalb der Funktion wie eindimen-

sionale Felder behandelt (siehe auch Kapitel 3.3.3 auf Seite 57).

- dem Funktionsnamen einen Wert zuweisen, damit dieser zum Rückgabewert an die aufrufende Stelle wird.
- keine Prozess-Abschnitte wie **LOWINIT** : , **INIT** : , **EVENT** : , oder **FINISH** : definieren.
- keine Library-Funktion oder -Unterprogramm aus der gleichen Library-Datei aufrufen.  
Gegebenenfalls müssen Sie die aufzurufende Funktion in eine neue Library-Datei auslagern und von dort importieren.
- die Anweisung **SELECTCASE** nicht benutzen.

Die Methoden zur Übergabe von Parametern unterscheiden sich folgendermaßen:

- **BYREF**: Die Library-Funktion kann den Parameter verändern, so dass der geänderte Wert anschließend im aufrufenden Programm vorliegt (es wird die Adresse des Parameters übergeben).
- **BYVAL**: Die Library-Funktion kann nur auf den Wert des Parameters zugreifen, diesen aber selbst nicht ändern. Der Parameter bleibt also für das aufrufende Programm gleich.



Sie sollten alle Übergabeparameter immer **AT <MEM\_TYPE>** definieren, um damit kostbare Prozessorzeit zu sparen (wobei **<MEM\_TYPE>** zur Deklaration des Übergabeparameters im aufrufenden Programm passen muss, siehe **DIM**). Anderenfalls muss die Library-Funktion zur Laufzeit herausfinden, in welchem Speicher die Übergabeparameter abgelegt sind.

Wenn ein Feld als Übergabeparameter einer Library verwendet wird, ist die Syntax für Definition und Aufruf unterschiedlich:

- Definition des Funktionsparameters *mit* Klammern:  
**LIB\_FUNCTION** name(array[]) ...
- Funktionsaufruf mit dem Parameter *ohne* Klammern:  
ret\_val=name(array)

Definieren Sie Felder als Übergabeparameter immer **BYREF** und ohne Feldgröße. Sie können keine FIFO-Felder als Übergabeparameter verwenden.

### Siehe auch

LIB\_SUB ... LIB\_ENDSUB, IMPORT, FUNCTION ... ENDFUNCTION,  
SUB ... ENDSUB

### Beispiel

```

REM ----- Mittelwertbildung -----
LIB_FUNCTION average (BYREF array[] AS LONG, BYVAL ptr AS LONG,
BYVAL cnt AS LONG) AS LONG
    DIM i AS LONG
    average = 0
    IF (cnt > 0) THEN
        FOR i = ptr TO (ptr + cnt)
            average = average + array[i]
        NEXT i
        average = average / cnt
    ENDIF
LIB_ENDFUNCTION

```

Den Aufruf der Library-Funktion `average` sehen Sie im folgenden Beispiel, einem sogenannten „moving average filter“:

```

REM Library 'MEAN' importieren
IMPORT C:\MyFiles\ADwinLibs\MEAN.LI9
#DEFINE cnt 10                'Anzahl der Summanden (Samples)
#DEFINE samples DATA_1 'Anzahl Messwerte
#DEFINE filtered DATA_2 'Anzahl gefilterte Messwerte
#DEFINE length 1000          'Feldlänge
DIM samples[length] AS LONG 'Quell-Feld
DIM filtered[length] AS LONG 'Ziel-Feld
DIM i AS LONG                'Zählvariable

INIT:
    i = 1                    'Zählvariable initialisieren
    PROCESSDELAY = 40000     'Messung mit 1 kHz

EVENT:
    samples[i] = ADC(1)      'Analogwerte messen und speichern
    INC i                    'Zählvariable erhöhen
    IF (i > length) THEN END '1000 Messungen durchgeführt?
                                'Wenn ja: FINISH abarbeiten

FINISH:
    FOR i = 1 TO (length - cnt) 'Alle Messwerte aufrufen
        REM Library-Funktion "average" aufrufen
        filtered[i + cnt] = average(samples,i,cnt)
        REM Beachten Sie die Syntax beim Feld 'samples'
        REM als Übergabeparameter ohne eckige Klammern
    NEXT i

```

## LIB\_SUB ... LIB\_ENDSUB

**LIB\_SUB...LIB\_ENDSUB** definiert in einer Library-Datei ein Unterprogramm mit Übergabeparametern.

### Syntax

```
LIB_SUB lib_name(<LIB_PAR1> {, <LIB_PAR2>, ...})
    {DIM var AS <VAR_TYPE>}
    {#DEFINE name expression}
    ...                               'Anweisungsblock
LIB_ENDSUB
```

Syntax der Übergabeparameter **<LIB\_PAR>**:

**<BY\_TYPE>** var\_name AS **<VAR\_TYPE>** {AT **<MEM\_TYPE>**}

### Parameter

lib_name	Name des Library-Unterprogramms.
var_name	Name eines Übergabeparameters innerhalb des Library-Unterprogramms; für Felder ist die Syntax mit Dimensionsklammern erforderlich: array[] oder DATA_n[].
<b>&lt;BY_TYPE&gt;</b>	Methode zur Übergabe eines Parameters: <b>BYREF</b> : Zeiger auf Variable oder Feld übergeben. <b>BYVAL</b> : Wert übergeben.
<b>&lt;VAR_TYPE&gt;</b>	Datentyp: <b>FLOAT</b> , <b>LONG</b> , <b>STRING</b> .
<b>&lt;MEM_TYPE&gt;</b>	Sinnvoll nur für Prozessor T10: Speicher, in dem die Variablen abgelegt werden; nur gemeinsam mit Feldern verfügbar: <b>DRAM_EXTERN</b> : externer Datenspeicher. <b>DM_LOCAL</b> : interner Datenspeicher.

### Bemerkungen

Allgemeine Informationen über Bibliotheken (Libraries) finden Sie in Kapitel 3.5.3 auf Seite 69.

Erstellen Sie Library-Unterprogramme (und -Funktionen) in einer eigenen Quelltext-Datei. Mit „Build / Make lib file“ kompilieren sie diese und erzeugen die Library-Datei. Der Befehl **IMPORT** bindet diejenigen Module einer Library in einen Prozess ein, die dort tatsächlich aufgerufen werden.

Innerhalb eines Library-Unterprogramms können Sie

- lokale Variablen und Felder (nur eindimensional) deklarieren und verwenden.  
Deklarieren Sie Variablen immer am Anfang, keinesfalls außerhalb des Unterprogramms.
- globale Variablen und Felder verwenden, wenn sie als Parameter übergeben werden.
- nur eindimensionale Felder bearbeiten.  
Sie können zweidimensionale Felder als Parameter übergeben, diese werden aber innerhalb der Funktion wie eindimensionale Felder behandelt (siehe auch Kapitel 3.3.3 auf Seite 57).
- keine Prozess-Abschnitte wie **LOWINIT** :, **INIT** :, **EVENT** :, oder **FINISH** : definieren.
- keine Library-Funktion oder -Unterprogramm aus der gleichen Library-Datei aufrufen.  
Gegebenenfalls müssen Sie die aufzurufende Funktion in eine neue Library-Datei auslagern und von dort importieren.
- die Anweisung **SELECTCASE** nicht benutzen.

Die Methoden zur Übergabe von Parametern unterscheiden sich folgendermaßen:

- **BYREF**: Das Library-Unterprogramm kann den Parameter verändern, so dass der geänderte Wert anschließend im aufrufenden Programm vorliegt (es wird die Adresse des Parameters übergeben).
- **BYVAL**: Das Library-Unterprogramm kann nur auf den Wert des Parameters zugreifen, diesen aber selbst nicht ändern. Der Parameter bleibt also für das aufrufende Programm gleich.

Betrifft nur Prozessor T10: Sie sollten alle Übergabeparameter immer **AT <MEM\_TYPE>** definieren, um damit kostbare Prozessorzeit zu sparen (wobei **<MEM\_TYPE>** zur Deklaration des Übergabeparameters im aufrufenden Programm passen muss, siehe **DIM**). Anderenfalls muss das Library-Unterprogramm zur Laufzeit herausfinden, in welchem Speicher die Übergabeparameter abgelegt sind.



Wenn ein Feld als Übergabeparameter einer Library verwendet wird, ist die Syntax für Definition und Aufruf unterschiedlich:

- Definition des Unterprogrammparameters *mit* Klammern: **LIB\_SUB** subname(array[]) ...
- Aufruf mit dem Parameter *ohne* Klammern: subname(array)

Definieren Sie Felder als Übergabeparameter immer **BYREF** und ohne Feldgröße. Sie können keine FIFO-Felder als Übergabeparameter verwenden.

### Siehe auch

LIB\_FUNCTION ... LIB\_ENDFUNCTION, IMPORT, FUNCTION ...  
ENDFUNCTION, SUB ... ENDSUB

### Beispiel:

```
REM Messwertumrechnung von Digit(0...65535) nach Volt (±10V)
LIB_SUB dig2volt(BYREF digit[] AS LONG, BYVAL ptr AS LONG,
BYVAL cnt AS LONG, BYVAL gain AS LONG, BYREF volt[] AS FLOAT)
  DIM i AS LONG
  FOR i = ptr TO (ptr + cnt)
    volt[i] = ((digit[i] * 20 / 65536) - 10) / gain
  NEXT i
LIB_ENDSUB
```



Den Aufruf der Library-Funktion `dig2volt` sehen Sie im folgenden Beispiel, einer Messwert-Umwandlung:

```
REM Die Library 'DIG2VOLT' wird importiert
IMPORT C:\MyFiles\ADwinLibs\DIG2VOLT.LI9
#DEFINE cnt 1000           'Anzahl der Samples
#DEFINE ptr 1              'Erstes zu konvertierendes Sample
#DEFINE gain 1             'Verstärkungsfaktor des PGA
#DEFINE samples DATA_1    'Speicher für Messwerte
#DEFINE scaled DATA_2     'Speicher für konvertierte Messwerte
#DEFINE length 1000       'Feldlänge
DIM samples[length] AS LONG 'Quell-Feld
DIM i AS LONG             'Zählvariable

INIT:
    i = 1                   'Zählvariable initialisieren
    PROCESSDELAY = 40000    'Messung mit 1 kHz

EVENT:
    samples[i] = ADC(1)     'Analogwerte messen und speichern
    INC i                   'Zählvariable erhöhen
    IF (i > length) THEN END '1000 Messungen durchgeführt?
                                'Wenn ja: FINISH abarbeiten

FINISH:
    REM Die gewünschten Messwerte konvertieren durch
    REM Aufruf des Library-Unterprogramms 'dig2volt'
    dig2volt(samples,ptr,cnt,gain,scaled)
    REM Beachten Sie die Syntax beim Feld 'samples'
    REM als Übergabeparameter ohne eckige Klammern []
```

## LN

**LN** liefert den natürlichen Logarithmus (zur Basis e) eines Arguments.

### Syntax

```
ret_val = LN(val)
```

### Parameter

<code>val</code>	Argument	<code>FLOAT</code>
<code>ret_val</code>	natürlicher Logarithmus des Arguments	<code>FLOAT</code>

### Bemerkungen

Die Ausführungszeit des Befehls beträgt beim T9 bis zu 1,45µs, beim T10 bis zu 0,7µs, beim T11 bis zu 0,37µs.

### Siehe auch

LOG, EXP

### Beispiel

```
DIM val_1, val_2 AS FLOAT
```

```
EVENT:
```

```
val_1 = 5.3
```

```
val_2 = LN(val_1)      'Ergebnis: val_2 = 1.667...
```

## LNGTOSTR

**LNGTOSTR** konvertiert einen ganzzahligen Wert in einen String.

### Syntax

```
IMPORT STRING.LI*          '*.LI9 für T9, *.LIA für T10,  
                             '*.LIB für T11  
  
LNGTOSTR(value, string[])
```

### Parameter

<code>value</code>	Wert, der gewandelt werden soll	<div>LONG</div>
<code>string[]</code>	Ergebnis-Zeichenfolge	<div>ARRAY</div> <div>STRING</div>

### Bemerkungen

Die erzeugte String-Länge ist nicht konstant, sondern richtet sich nach der zu konvertierenden Zahl und dem Vorzeichen. Es sind String-Längen von 1 bis 11 Zeichen möglich.

Informationen zum String-Aufbau finden Sie in Kapitel 3.3.5 auf Seite 61.

### Siehe auch

"" String, + (String-Addition), ASC, CHR, FLOTOSTR, FLO40TOSTR, STRCOMP, STRLEFT, STRLEN, STRMID, STRRIGHT, VALF, VALI

**Beispiel**

```
IMPORT STRING.LI9
DIM digits[11] AS STRING'Ergebnis-String
DIM a AS LONG

INIT:
  a = -1234567890

EVENT:
  LNGTOSTR(a,digits)      'zum String konvertieren
  PAR_1=digits[1]         'String-Länge = 11
  PAR_2=digits[2]         'ASCII-Zeichen 45 = "-"
  PAR_3=digits[3]         'ASCII-Zeichen 49 = "1"
  PAR_4=digits[4]         'ASCII-Zeichen 50 = "2"
  PAR_5=digits[5]         'ASCII-Zeichen 51 = "3"
  PAR_6=digits[6]         'ASCII-Zeichen 52 = "4"
  PAR_7=digits[7]         'ASCII-Zeichen 53 = "5"
  PAR_8=digits[8]         'ASCII-Zeichen 54 = "6"
  PAR_9=digits[9]         'ASCII-Zeichen 55 = "7"
  PAR_10=digits[10]       'ASCII-Zeichen 56 = "8"
  PAR_11=digits[11]       'ASCII-Zeichen 57 = "9"
  PAR_12=digits[12]       'ASCII-Zeichen 48 = "0"
  PAR_13=digits[13]       'String-Ende-Zeichen = 0
```

## LOG

**LOG** liefert den dekadischen Logarithmus (zur Basis 10) eines Arguments.

### Syntax

```
ret_val = LOG(val)
```

### Parameter

val	Argument	FLOAT
ret_val	dekadischer Logarithmus des Arguments	FLOAT

### Bemerkungen

Die Ausführungszeit des Befehls beträgt beim T9 bis zu 1,5µs, beim T10 bis zu 0,75µs, beim T11 bis zu 0,38µs.

### Siehe auch

LN, EXP

### Beispiel

```
DIM val_1, val_2 AS FLOAT
```

```
EVENT:
```

```
val_1 = 5.3
```

```
val_2 = LOG(val_1) 'Ergebnis: val_2 = 0.724...
```

## LOWINIT:

Das Kennwort **LOWINIT** : bezeichnet den Anfang eines Programmabschnitts zur Initialisierung. Der Programmabschnitt hat in jedem Fall niedrige Priorität, Stufe 1.

### Syntax

**LOWINIT** : { **AT MEM\_TYPE** }

### Parameter

**<MEM\_TYPE>** nur für T11: Speicherbereich, in dem der Programmabschnitt abgelegt wird.

**PM\_LOCAL**: interner Programmspeicher (Default)

**EM\_LOCAL**: Zusätzlicher interner Programm- oder Datenspeicher

**DRAM\_EXTERN**: externer Datenspeicher

### Bemerkungen

Zur Übersicht der Programmabschnitte siehe Kapitel 3.1.1 auf Seite 45.

Der Abschnitt wird bei jedem Start des Prozesses einmalig durchlaufen und dient zur Initialisierung, z.B. von Variablen oder Datenleitungen. **LOWINIT** : wird immer vor dem Abschnitt **INIT** : ausgeführt (falls vorhanden).



Der Abschnitt **LOWINIT** : ist für umfangreiche Initialisierungs-Sequenzen geeignet, da er (wegen seiner niedriger Priorität) unterbrochen werden kann.

Bei dem Prozessortyp T11 kann jeder Programmabschnitt separat einem bestimmten Speicherbereich zugeordnet werden (siehe Kapitel 3.3.2 „Speicherbereiche“). Der große, aber langsamere Speicherbereich **DRAM\_EXTERN** sollte nur für zeitunkritische Programmabschnitte genutzt werden; meistens sind das die Abschnitte **LOWINIT** : , **INIT** : , **FINISH** : .

### Siehe auch

**DIM**, **INIT** : , **EVENT** : , **FINISH** :

## Beispiel

```
DIM val_1 AS FLOAT
LOWINIT:
  val_1 = -5.3
```

## MEMCPY

Nur für Prozessor T11: **MEMCPY** kopiert eine bestimmte Anzahl von Feldelementen aus einem Quellfeld in ein Zielfeld.

### Syntax

```
MEMCPY(source[i1], dest[i2], count)
```

### Parameter

<code>source[]</code>	Name des Quellfelds.	<div>ARRAY</div> <div>LONG</div> <div>FLOAT</div> <div>STRING</div>
<code>i1</code>	Index ( $\geq 1$ ) des ersten kopierten Feldelements.	LONG
<code>dest[]</code>	Name des Zielfelds.	<div>ARRAY</div> <div>LONG</div> <div>FLOAT</div> <div>STRING</div>
<code>i2</code>	Index ( $\geq 1$ ) des ersten Feldelements, das beschrieben wird.	LONG
<code>count</code>	Anzahl ( $\geq 1$ ) der zu kopierenden Feldelemente.	LONG

### Bemerkungen

**MEMCPY** ist die einfache und deutlich schnellere Alternative zum Kopieren von Daten in einer **FOR...NEXT**-Schleife.

Die Anweisung darf nicht zum Kopieren von und in FIFO-Felder oder lokale Variablen benutzt werden.



Beachten Sie bitte: Die Datentypen von Quellfeld und Zielfeld müssen übereinstimmen und das Zielfeld muss groß genug dimensioniert sein, um alle kopierten Daten aufnehmen zu können. Der Zugriff auf zu große oder zu kleine Elementindexe des Zielfelds kann mit dem Debug-Modus überwacht werden (siehe Option „Debug mode“ auf Seite 31). Für das Quellfeld ist die Überwachung nicht möglich.



## Siehe auch

DIM

## Beispiel

```
DIM DATA_1[75], DATA_2[100] AS FLOAT
```

**EVENT:**

```
REM 70 Feldelemente aus DATA_1 nach DATA_2 kopieren
```

```
MEMCPY (DATA_1[5], DATA_2[30], 70)
```

## NOP

**NOP** (No OPeration) lässt den Prozessor für die Zeit von einem Taktzyklus warten.

### Syntax

**NOP**

### Bemerkungen

Diese Anweisung benötigt in der Regel einen Prozessor-Taktzyklus:

T9	25,0ns
T10	12,5ns
T11	3,3ns

Sie können mit dieser Anweisung erforderliche Wartezeiten überbrücken (beispielsweise nach **SET\_MUX**), wenn es keine sinnvolle andere Verwendung der Prozessorzeit gibt.

### Siehe auch

CPU\_SLEEP, P1\_SLEEP, P2\_SLEEP, SLEEP

## NOT

Der Operator **NOT** invertiert die Bits eines Werts.

### Syntax

```
ret_val = NOT(val)
```

### Parameter

<code>val</code>	zu invertierender Wert (kein logischer Ausdruck)	LONG
<code>ret_val</code>	invertiertes Argument	LONG

### Bemerkungen

Verwenden Sie diesen Operator möglichst nur mit ganzzahligen Werten (Typ **LONG**).

Fließkomma-Werte (Typ **FLOAT**) werden vor der Invertierung in ganzzahlige konvertiert: Die Nachkommastellen werden abgeschnitten und ggf. der Wertebereich angepasst, so dass das Berechnungsergebnis beeinflusst wird.

**NOT** ist ein bitweiser, kein Boolescher Operator. Daher können Sie damit keine logischen Ausdrücke (True / False) negieren. Falsch wäre also: **NOT**(`PAR_2 > 2`).

### Siehe auch

AND, IF ... THEN ... {ELSE ... } ENDIF, OR, XOR

### Beispiel

```
DIM val_1 AS LONG
DIM val_2 AS LONG

val_1 = -3           '-3 =
                    ' 11111111111111111111111111111101b
val_2 = NOT(val_1)   'Ergebnis: val_2=010b=2
```

## OR

Der Operator **OR** verknüpft zwei ganzzahlige Werte bitweise oder zwei Boolesche Ausdrücke als Boolescher Operator.

### Syntax

```
ret_val = val_1 OR val_2      'Bitweiser Operator
IF ((expr1) OR (expr2)) THEN 'Boolescher Operator
```

### Parameter

val_1, val_2	Ganzzahliger Wert	<span style="border: 1px solid black; padding: 2px;">LONG</span>
expr1, expr2	Boolescher Ausdruck mit dem Wert „wahr“ oder „falsch“	<span style="border: 1px solid black; padding: 2px;">LOGIC</span>

### Bemerkungen

Sie können mit **OR** nur gleichartige Ausdrücke verknüpfen (ganzzahlige oder Boolesche), ein Mischen ist nicht möglich.

Sie können Boolesche Ausdrücke nur mit den Anweisungen **IF ... THEN ... ELSE** oder **DO ... UNTIL** verwenden (Variablen können keine Booleschen Werte annehmen).

Wenn Sie in einer Zeile mehrere Boolesche Operatoren verwenden, müssen Sie jede Verknüpfung separat in Klammern setzen. Bei der Verknüpfung ganzzahliger Werte ist dies nicht erforderlich.

### Siehe auch

AND, IF ... THEN ... {ELSE ... } ENDIF, NOT, XOR

### Beispiel

Bitweise Operator:

```
DIM val_1, val_2, val3 AS LONG

val_1 = 0100b
val_2 = 0110b
val3 = val_1 OR val_2  'Ergebnis: val3 = 0110b
```

Boolescher Operator:

```
DIM x AS LONG
```

```
DIM val4 AS LONG
```

```
INIT:
```

```
  x = 15
```

```
EVENT:
```

```
  IF ((x < 9) OR (x > 3)) THEN
```

```
    val4 = 1
```

```
  ELSE
```

```
    val4 = 0
```

```
  ENDIF
```

```
'Ergebnis: val4 = 1
```

## P1\_SLEEP

Nur für Prozessor T11: **P1\_SLEEP** lässt den Bus des Pro I-Systems für eine bestimmte Zeit warten.

### Syntax

**P1\_SLEEP**(val)

### Parameter

<b>val</b>	Anzahl der zu wartenden Zeiteinheiten in 10ns: bei Konstanten: 7...715827879 ( $\approx 2^{30} / 1,5$ ) bei Variablen: 9...715827879	LONG
------------	--	------

### Bemerkungen

Alternativ gibt es die Anweisungen **CPU\_SLEEP** und **P2\_SLEEP** (siehe auch Kapitel 4.2.4 „Wartezeit genau einstellen“). Verwenden Sie bei Prozessoren bis T10 die Anweisung **SLEEP**.

Die Anweisung **P1\_SLEEP** wird eingesetzt, wenn zwischen 2 Zugriffen auf Module am Pro I-Bus eine definierte Zeit gewartet werden muss.

Die Wartezeit sollte grundsätzlich kleiner sein als die mit **PROCESSDELAY** eingestellte Zykluszeit.



Die Anweisung **P1\_SLEEP** kann bei einem hochprioren Prozess nicht unterbrochen werden. Bitte beachten Sie, dass sehr große Werte bei hochprioren Prozessen zu einem Abbruch der Kommunikation führen können.

Verwenden Sie nach Möglichkeit eine Konstante als Argument. Wenn das Argument **val** eine Berechnung erfordert, benötigt dies eine bestimmte Zeitspanne zusätzlich; diese ist jeweils konstant und beträgt einige wenige Taktzyklen.

Folgende Fälle erfordern eine Berechnung:

- Das Argument ist ein Berechnungsausdruck mit Variablen oder Feldelementen.
- Die Variable im Argument ist für den Speicherbereich **DRAM\_EXTERN** deklariert. Die Zeitspanne kann hier schwanken, weil sie von mehreren Voraussetzungen abhängt.
- Das Argument ist ein Feld.
- Das Argument ist ein Fließkomma-Wert.

### Siehe auch

CPU\_SLEEP, NOP, P2\_SLEEP, SLEEP

### Beispiel

#### EVENT:

<b>SET_MUX</b> (0)	'Multiplexer setzen
<b>P1_SLEEP</b> (250)	'2.5 $\mu$ s (=250*10ns) warten
	'= Einschwingzeit des MUX
<b>START_CONV</b> (1)	'Konvertierung starten
...	

## P2\_SLEEP

Nur für Prozessor T11: **P2\_SLEEP** lässt lässt den Bus des Pro II-Systems für eine bestimmte Zeit warten.

### Syntax

**P2\_SLEEP**(val)

### Parameter

val      Gerade Anzahl ( $14 \dots 715827878 \approx 2^{30} / 1,5$ ) LONG  
 der zu wartenden Zeiteinheiten in 10ns.  
 Eine ungerade Anzahl ist nicht erlaubt.

### Bemerkungen

Alternativ gibt es die Anweisungen **CPU\_SLEEP** und **P1\_SLEEP** (siehe auch Kapitel 4.2.4 „Wartezeit genau einstellen“). Verwenden Sie bei Prozessoren bis T10 die Anweisung **SLEEP**.

Die Anweisung **P2\_SLEEP** wird eingesetzt, wenn zwischen 2 Zugriffen auf Module am Pro II-Bus eine definierte Zeit gewartet werden muss.

Die Wartezeit sollte grundsätzlich kleiner sein als die mit **PROCESSDE-LAY** eingestellte Zykluszeit.



Die Anweisung **P2\_SLEEP** kann bei einem hochprioren Prozess nicht unterbrochen werden. Bitte beachten Sie, dass sehr große Werte bei hochprioren Prozessen zu einem Abbruch der Kommunikation führen können.

Verwenden Sie nach Möglichkeit eine Konstante als Argument. Wenn das Argument **val** eine Berechnung erfordert, benötigt dies eine bestimmte Zeitspanne zusätzlich; diese ist jeweils konstant und beträgt einige wenige Taktzyklen.

Folgende Fälle erfordern eine Berechnung:

- Das Argument ist ein Berechnungsausdruck mit Variablen oder Feldelementen.
- Die Variable im Argument ist für den Speicherbereich **DRAM\_EXTERN** deklariert.
- Das Argument ist ein Feld.
- Das Argument ist ein Fließkomma-Wert.



### Siehe auch

CPU\_SLEEP, NOP, P1\_SLEEP, SLEEP

### Beispiel

**EVENT:**

<b>P2_SET_MUX</b> (0)	'Multiplexer setzen
<b>P2_SLEEP</b> (250)	'2.1 $\mu$ s (=210*10ns) warten
	'= Einschwingzeit des MUX
<b>P2_START_CONV</b> (1)	'Konvertierung starten
...	

## PEEK

**PEEK** liest den Inhalt einer bestimmten Speicherstelle auf dem *ADwin*-System.

### Syntax

```
ret_val = PEEK(addr)
```

### Parameter

<code>addr</code>	Adresse der auszulesenden Speicherstelle	LONG
<code>ret_val</code>	Inhalt der Speicherstelle	LONG

### Beschreibung

Sie finden eine Übersicht der Registeradressen (*Gold* und *Light-16*) in Ihrer Hardware-Dokumentation.

### Siehe auch

POKE, READ\_TIMER

### Beispiel

Die unten stehende Anweisung liest den Wert der Speicheradresse 30h, das ist bei *ADwin-Gold* das Datenregister des ADC1 und enthält den gewandelten Analogwert.

```
REM Speicherstellen eines ADwin-Gold auslesen  
val = PEEK(30h)
```

## POKE

**POKE** schreibt einen Wert in eine bestimmte Speicherstelle auf dem ADwin-System.

### Syntax

```
POKE(addr, value)
```

### Parameter

addr	Adresse der beschreibenden Speicherstelle	LONG
value	Zu schreibender Wert	LONG

### Bemerkungen

Sie überschreiben mit **POKE** die angegebene Speicheradresse. Informationen, die zuvor dort gespeichert waren, gehen unwiderruflich verloren.

Überschreiben Sie in keinem Fall Speicheradressen, deren Funktion Ihnen unbekannt ist. Es können dadurch für den Betrieb wichtige Daten, Prozesse oder gar das Betriebssystem zerstört werden. Falls dies dennoch geschieht, gehen vorhandene Messdaten verloren. Booten Sie das ADwin-System und laden die Prozesse erneut.



Sie finden eine Übersicht der Registeradressen (*Gold* und *Light-16*) in Ihrer Hardware-Dokumentation.

### Siehe auch

PEEK, READ\_TIMER

### Beispiel

```
REM Speicherstellen eines ADwin-Gold verändern
REM DAC-Register beschreiben: 3072 (+=+5V im ±10V Bereich)
POKE(50h, 3072)
POKE(50h, 011b)      'Ausgabe auf allen DACs starten
POKE(0C0h, 111100b)  'Setze DIGOUT Bits 2 bis 5
```

## PROCESSDELAY

Die Systemvariable **PROCESSDELAY** definiert das Processdelay (die Zykluszeit) eines Prozesses.

**PROCESSDELAY** ersetzt die Systemvariable **GLOBALDELAY**, die aus Kompatibilitätsgründen weiterhin gültig ist.

### Syntax

```
ret_val = PROCESSDELAY
```

oder

```
PROCESSDELAY = expr
```

### Parameter

ret_val	Aktuell eingestellte Zykluszeit in Taktzyklen.	LONG
expr	Einzustellende Zykluszeit: Anzahl ( $\geq 1$ ) der Taktzyklen.	LONG

### Bemerkungen

Bei einem zeitgesteuerten Prozess wird der Abschnitt **EVENT**: vom internen Zähler zyklisch und in festen Zeitabständen aufgerufen. Der Zeitabstand zwischen zwei Aufrufen, Zykluszeit oder Processdelay genannt, wird in Zähler-Taktzyklen gezählt.

Die Zeiteinheit des Processdelay ist abhängig von der Priorität des Prozesses und vom Prozessortyp:

Prozessor	Priorität	
	Hoch	Niedrig
T9	25ns	100µs
T10	25ns	50µs
T11	3,3ns	3,3ns = 0,003µs

Wählen Sie bei hochprioren Prozessen eine ausreichend großes Processdelay, um eine Überlastung des ADwin-Systems zu vermeiden (siehe auch Kapitel 5.1.4 auf Seite 88). Als Faustregel sollte die Auslastung des Prozessors (Anzeige: „Busy x%“ in der Statusleiste) möglichst unter 90% bleiben und darf keinesfalls 100% übersteigen. Wenn die Bearbeitungszeit des Abschnitts **EVENT**: größer ist als das

Processdelay, kommen der nächste Zähler-Aufruf und die folgenden verspätet. Sollte diese Verzögerung nicht innerhalb von 250ms aufgeholt werden, kann die Kommunikation zwischen *ADwin*-System und PC zusammenbrechen.

Ein konstantes Processdelay können Sie festlegen, indem Sie der Variablen **PROCESSDELAY** im Abschnitt **INIT** : / **LOWINIT** : einen Wert zuweisen. Sie überschreiben damit ggf. die Voreinstellung, die Sie in der Entwicklungsumgebung *ADbasic* im Dialogfenster „Options / Process“ unter „Initial Processdelay“ eingegeben haben.

Sie können die Systemvariable innerhalb eines Abschnitts nur einmal beschreiben.

Wenn der Parameter **PROCESSDELAY** innerhalb eines Prozesszyklus, d.h. im Abschnitt **EVENT** : geändert wird, wird die Zykluszeit dadurch sofort verändert. Dies kann insbesondere bei einer Verkürzung der Zykluszeit kritisch sein: Achten Sie immer darauf, dass die Bearbeitungszeit des Prozesszyklus kürzer bleibt als die neu eingestellte Zykluszeit.

### Siehe auch

READ\_TIMER

### Beispiel

```
INIT:
  REM Zykluszeit einstellen
  PROCESSDELAY = 40000
  REM Für T9 und T10, hohe Priorität: 1ms
  REM Für T11: 0,133ms
  ...
```

Wenn Sie eine längere Zykluszeit benötigen als mit **PROCESSDELAY** einstellbar ist, können Sie eine Hilfsvariable verwenden:

**INIT:**

```
REM Max. Zykluszeit einstellen
PROCESSDELAY = 2147483647
REM Für T9 und T10, hohe Priorität: 53,7s
REM Für T11, hohe+niedrige Priorität: 7,2s
REM Hilfsvariable initialisieren
PAR_1 = 0
```

**EVENT:**

```
INC PAR_1
REM 100fache Zykluszeit verwenden
IF (PAR_1 = 100) THEN
  PAR_1 = 0
  REM Programm ausführen
...
ENDIF
```

## PROZESSn\_RUNNING

Die Systemvariable **PROZESSn\_RUNNING** gibt den aktuellen Status eines bestimmten Prozesses zurück.

### Syntax

```
ret_val = PROZESSn_RUNNING
```

### Parameter

n                      Prozessnummer (0...12, 15)

**CONST****LONG**

ret\_val                Prozess-Status:  
1    Prozess läuft  
0    Prozess ist gestoppt  
-1   Prozess wird gestoppt

**LONG**

### Bemerkungen

Diese Systemvariable kann nur gelesen werden.

### Siehe auch

END, EXIT, RESTART\_PROCESS, START\_PROCESS, START\_PROCESS\_DELAYED, STOP\_PROCESS

### Beispiel

```
EVENT:  
REM Status von Prozess 2 ermitteln  
PAR_2 = PROZESS2_RUNNING
```

## READ\_TIMER

**READ\_TIMER** gibt den aktuellen Zählerstand des *ADwin*-Systems zurück.

### Syntax

```
ret_val = READ_TIMER()
```

### Parameter

`ret_val`      Aktueller Zählerstand.

LONG

### Bemerkungen

Die Systemvariable kann nur gelesen werden.

Es gibt im *ADwin*-System 2 Zähler (32 Bit), die in unterschiedlichen Zeiteinheiten zählen:

Prozesspriorität	T9	T10	T11
hoch	25ns	25ns	3,3ns
niedrig	100µs	50µs	3,3ns

Sie können eine Zeitdifferenz aus der Differenz von 2 Zählerständen ermitteln. Beachten Sie dabei bitte, dass ein gelesener Zählerstand nach einer bestimmten Zeit wieder erneut erreicht wird. Dieser Zählerüberlauf hängt von den oben angegebenen Zeiteinheiten ab:

Prozesspriorität	T9	T10	T11
hoch	107,4s	107,4s	14,3s
niedrig	119,3h	59,7h	14,3s

### Siehe auch

PROCESSDELAY

### Beispiel

```
DIM timervalue AS LONG
```

```
EVENT:
```

```
timervalue = READ_TIMER()
```



## REM, '

Die Compiler-Anweisungen *REM* oder „'“ ermöglichen das Einfügen von Kommentaren im Quelltext. Sie bewirken, dass der in einer Programmzeile folgende Text vom Compiler ignoriert wird.

### Syntax

```
REM comment  
instr : REM comment  
instr 'comment
```

### Parameter

comment	Beliebige Zeichenfolge
instr	<i>ADbasic</i> -Anweisung

### Bemerkungen

Die Anweisung gilt nur für die Zeile, in der sie benutzt wird. Für einen mehrzeiligen Kommentar müssen Sie jede Zeile einzeln mit der Anweisung *REM* oder „'“ beginnen.

Wenn Sie eine *REM*-Anweisung hinter einer anderen Anweisung in einer Befehlszeile einfügen, dann trennen Sie beide durch einen Doppelpunkt `:`. Bei dem Kommentarzeichen `'` ist dies nicht erforderlich.

### Beispiel

```
REM Dies ist ein Kommentar, der mehr als eine  
REM Textzeile benötigt  
'Dies ist auch eine Kommentarzeile  
DIM min AS LONG : REM Kommentar nach einer Anweisung  
DIM max AS LONG      'Kommentar nach einer Anweisung
```

## RESET\_EVENT

**RESET\_EVENT** löscht alle externen Event-Signale, die zur Ausführung anstehen.

### Syntax

**RESET\_EVENT**

### Bemerkungen

Die Anweisung ist nur bei dem extern gesteuerten Prozess zulässig, nicht für zeitgesteuerte Prozesse.

Wir empfehlen, die Anweisung am Ende des Abschnitts **INIT**: auszuführen; beim Prozessor T11 ist dies sogar zwingend erforderlich. Damit stellen Sie sicher, dass ein zu frühes – während der Initialisierung aufgetretenes – Event-Signal nicht sofort den Hauptteil des Programms (Abschnitt **EVENT**:) startet.

Näheres zum Betriebsmodus des Betriebssystems bei einem extern gesteuerten Prozess siehe Abschnitt „Extern gesteuerter Prozess“ auf Seite 93.

### Siehe auch

END, EXIT, PROZESSn\_RUNNING, START\_PROCESS, STOP\_PROCESS

### Beispiel

```
INIT:
    REM Initialisierung
    ...
    RESET_EVENT           'Bisherige EVENT-Signale löschen

EVENT:
    REM Hauptprogramm startet, sobald ein Event-Signal auftritt
    ...
```

## RESTART\_PROCESS

Nur für Prozessor T11: **RESTART\_PROCESS** startet den gleichen Prozess erneut.

### Syntax

**RESTART\_PROCESS**

### Bemerkungen

Die Anweisung ist nur im Abschnitt **FINISH**: zulässig.

Alle Befehlszeilen des Abschnitts nach **RESTART\_PROCESS** werden noch ausgeführt, bevor der Prozess neu gestartet wird. Zur besseren Lesbarkeit empfehlen wir daher, die Anweisung ans Ende des Abschnitts zu stellen.

Die Anweisung kann zu einer Endlos-Schleife führen. Verwenden Sie **RESTART\_PROCESS** deshalb auf jeden Fall innerhalb einer Bedingung.



### Siehe auch

END, EXIT, IF ... THEN ... {ELSE ... } ENDIF, START\_PROCESS, START\_PROCESS\_DELAYED, STOP\_PROCESS

### Beispiel

**EVENT:**

...

**FINISH:**

...

**IF** (cond = 2) **THEN**

REM Wenn Bedingung erfüllt, Prozess erneut starten

**RESTART\_PROCESS**

**ENDIF**

## SELECTCASE

Die Kontrollstruktur bewirkt in Abhängigkeit von einem Wert die Ausführung eines von mehreren Anweisungsblöcken.

### Syntax

```

SELECTCASE var
CASE const1a{,const1b, ...}
    ...                               'Anweisungsblock'
CCASE const2a{,const2b, ...}
    ...                               'Anweisungsblock'
CASEELSE
    ...                               'Anweisungsblock'
ENDSELECT

```

### Parameter

var	Auszuwertendes Argument (kein Berechnungsausdruck).	LONG
const1a, const1b, const2a, const2b	Wert von var (0...255), bei dem der nachfolgende Anweisungsblock ausgeführt wird.	CONST LONG

### Bemerkungen

In einer Library-Funktion oder einem Library-Unterprogramm kann die Kontrollstruktur nicht verwendet werden.

Sie können mehrere **SELECTCASE**-Strukturen beliebig tief verschachteln; nur die Speichergröße begrenzt Sie hierbei.

Je nach Argument können Sie mit **SELECTCASE** verschachtelte **IF**-Strukturen ersetzen und damit übersichtlicher gestalten; vor allem aber wird diese Struktur schneller ausgeführt als mehrere aufeinander folgende **IF**-Strukturen.

Wenn das auszuwertende Argument mit keiner der **CASE**-Konstanten übereinstimmt, wird – falls vorhanden – nur der **CASEELSE**-Anwei-

sungsblock ausgeführt. Dies geschieht ebenfalls, wenn das auszuwertende Argument außerhalb des Wertebereichs der Konstanten liegt.

**CCASE** steht für „Continue Case“: Wenn ein **CASE**- oder **CCASE**-Anweisungsblock abgearbeitet wurde, dann wird ein direkt folgender **CCASE**-Anweisungsblock ebenfalls abgearbeitet.

Im Beispiel unten wird deshalb nicht nur die Anweisung **ADC** (5), sondern auch **ADC** (7) ausgeführt. Wäre dagegen **PAR\_1**=3, dann würde nur **ADC** (7) ausgeführt.

Wenn Sie in den Anweisungsblöcken Variablen so verändern, dass sich der Wert des Arguments ändert, wird dies erst bei der nächsten **SELECTCASE**-Abfrage berücksichtigt.

Die Struktur verwendet intern eine Sprungtabelle, deren Speicherbedarf sich nach der größten angegebenen **CASE**-/**CCASE**-Konstanten richtet. Um den Speicherplatz-Bedarf zu beschränken, ist der Wertebereich der Konstanten auf 0...255 eingeschränkt. Es gilt:

Speicherbedarf in Bytes = [ (größter Konstantenwert)+1 ] × 4

Beispielsweise wäre der Speicherbedarf bei **CASE** 200:

$(200 + 1) \times 4 = 804$  Bytes; der max. Bedarf beträgt genau 1 kB.

### Siehe auch

DO ... UNTIL, FOR ... TO ... {STEP ...} NEXT, IF ... THEN ... {ELSE ...} ENDIF

**Beispiel**

```

EVENT:
  PAR_1=2
SELECTCASE PAR_1      'PAR_1 auswerten
  CASE 0                'Ist PAR_1 = 0?
    PAR_10 = ADC(1)    'ADC(1) auslesen
  CASE 1                'Ist PAR_1 = 1?
    PAR_10 = ADC(3)    'ADC(3) auslesen
  CASE 2                'Ist PAR_1 = 2?
    PAR_10 = ADC(5)    'ADC(5) und auch (durch CCASE)
                        ' ADC(7) auslesen
  CCASE 3               'Ist PAR_1 = 3?
    PAR_11 = ADC(7)    'ADC(7) auslesen
  CASE 4,5,6,7,16      'Ist PAR_1 = 4, 5, 6, 7 oder 16?
    PAR_2 = DIGIN_WORD() 'digitale Eingänge einlesen
  CASEELSE             'PAR_1: sonstige Werte
    DIGOUT_WORD(PAR_10) 'Wert von PAR_10 an digitale
                        'Ausgänge ausgeben
ENDSELECT             'Abschluss der Auswahl

```

## SHIFT\_LEFT

**SHIFT\_LEFT** verschiebt alle Bits eines Werts um eine bestimmte Stellenzahl nach links. Rechts frei werdende Bits werden zu 0 gesetzt.

### Syntax

```
ret_val = SHIFT_LEFT(val, num)
```

### Parameter

val	Argument	LONG
num	Anzahl der Stellen, um die das Argument geschoben wird (0...31).	LONG
ret_val	Argument mit verschobenen Bits oder 0 für (num<0) und für (num>31)	LONG

### Bemerkungen

Verwenden Sie als Argument möglichst nur ganzzahlige Werte. Fließkomma-Werte (Typ **FLOAT**) werden vor dem Schieben in ganzzahlige konvertiert: Die Nachkommastellen werden abgeschnitten und ggf. der Wertebereich angepasst, so dass das Berechnungsergebnis beeinflusst wird.

Das Verschieben um  $n$  Stellen nach links entspricht einer Multiplikation mit  $2^n$ . Ein eventuell vorkommender Überlauf wird nicht berücksichtigt, d.h. ein gesetztes Bit ist verloren, wenn es aus dem Argument nach links „heraus geschoben“ wird.

Die Ausführungszeit ist gleich derjenigen bei einem vergleichbaren Multiplikations-Operator.

### Siehe auch

SHIFT\_RIGHT

### Beispiel

```
DIM val_1, val_2 AS LONG
```

```
EVENT:
```

```
val_1 = 1024
```

```
val_2 = SHIFT_LEFT(val_1, 2) 'Ergebnis: val_2=4096
```

## SHIFT\_RIGHT

**SHIFT\_RIGHT** verschiebt alle Bits eines Werts um eine bestimmte Stellenzahl nach rechts. Links frei werdende Bits werden zu 0 gesetzt.

### Syntax

```
ret_val = SHIFT_RIGHT(val, num)
```

### Parameter

val	Argument	LONG
num	Anzahl der Stellen, um die das Argument geschoben wird (0...31).	LONG
ret_val	Argument mit verschobenen Bits oder 0 für (num<0) und für (num>31)	LONG

### Bemerkungen

Verwenden Sie als Argument möglichst nur ganzzahlige Werte. Fließkomma-Werte (Typ **FLOAT**) werden vor dem Schieben in ganzzahlige konvertiert: Die Nachkommastellen werden abgeschnitten und ggf. der Wertebereich angepasst, so dass das Berechnungsergebnis beeinflusst wird.

Falls das Argument eine positive Zahl ist, entspricht das Verschieben um  $n$  Stellen nach rechts einer Division durch  $2^n$ . Ein eventuell vorkommender Divisions-Rest wird nicht berücksichtigt, d.h. ein gesetztes Bit, das aus dem Argument nach rechts „heraus geschoben“ wird, ist verloren.

Die Ausführungszeit ist geringer als bei einem vergleichbaren Divisions-Operator, d.h. `val_2 = SHIFT_RIGHT(val_1, 3)` ist schneller als `val_2 = val_1 / 8`.

### Siehe auch

SHIFT\_LEFT



### Beispiel

```
DIM val_1, val_2 AS LONG
```

```
EVENT:
```

```
val_1 = 1024
```

```
val_2 = SHIFT_RIGHT(val_1, 3) 'Ergebnis: val_2=128
```

## SIN

**SIN** liefert den Sinus eines Arguments.

### Syntax

```
ret_val = SIN(angle)
```

### Parameter

angle	Winkel im Bogenmaß ( $-\pi \dots \pi$ )	Float
ret_val	Sinus des Winkels ( $-1 \dots 1$ )	Float

### Bemerkungen

Wenn Sie für den Winkel Werte außerhalb von  $-\pi \dots \pi$  verwenden, nimmt der Berechnungsfehler mit wachsendem Wert zu.

Die Ausführungszeit des Befehls beträgt beim T9 bis zu 1,25µs, beim T10 bis zu 0,63µs, beim T11 bis zu 0,28µs.

### Siehe auch

COS, TAN, ARCSIN, ARCCOS, ARCTAN

### Beispiel

```
DIM val_1, val_2 AS FLOAT
```

```
EVENT:
```

```
val_1 = -5.3
```

```
val_2 = SIN(val_1) 'Ergebnis: val_2=0.83...
```

## SLEEP

Nur bis Prozessor T10: **SLEEP** lässt den Prozessor für eine bestimmte Zeit warten.

### Syntax

**SLEEP**(*val*)

### Parameter

*val*

Anzahl der zu wartenden Zeiteinheiten in  
100ns ( $\geq 1$ )

LONG

### Bemerkungen

Bei dem Prozessor T11 muss **SLEEP** ersetzt werden durch einen der Befehle **CPU\_SLEEP**, **P1\_SLEEP** oder **P2\_SLEEP** (siehe auch Kapitel 4.2.4 „Wartezeit genau einstellen“).

Da der Befehl **SLEEP** als Zählschleife ausgeführt wird, kann er bei einem hochprioren Prozess nicht unterbrochen werden.

Stellen Sie unbedingt (insbesondere bei Variablen) sicher, dass das Argument in keinem Fall einen Wert kleiner 1 hat, weil sonst das ADwin-System in einen instabilen Zustand geraten kann. Beachten Sie bitte auch, dass sehr große Werte bei hochprioren Prozessen zu einem Abbruch der Kommunikation führen können.



Verwenden Sie nach Möglichkeit eine Konstante als Argument. Wenn das Argument *val* eine Berechnung erfordert, benötigt dies eine bestimmte Zeitspanne zusätzlich; diese ist jeweils konstant und beträgt einige wenige Taktzyklen.

Folgende Fälle erfordern eine Berechnung:

- Das Argument ist ein Berechnungsausdruck mit Variablen oder Feldelementen.
- Die Variable im Argument ist für den Speicherbereich **DRAM\_EXTERN** deklariert.
- Das Argument ist ein Feld.
- Das Argument ist ein Fließkomma-Wert.

### Siehe auch

CPU\_SLEEP, NOP, P1\_SLEEP, P2\_SLEEP

**Beispiel**

```
EVENT:
  SET_MUX(0)           'Multiplexer setzen
  SLEEP(25)            '2.5 µs (=25*100ns) warten
                       '= Einschwingzeit des MUX
  START_CONV(1)        'Konvertierung starten
  ...
```

## SQRT

**SQRT** gibt die quadratische Wurzel eines Werts zurück.

### Syntax

```
ret_val = SQRT(val)
```

### Parameter

<code>val</code>	Argument	<code>FLOAT</code>
<code>ret_val</code>	Quadratwurzel des Arguments oder 0 für ( <code>val</code> <0)	<code>FLOAT</code>

### Bemerkungen

Die Ausführungszeit des Befehls beträgt beim T9 bis zu 0,9µs, beim T10 bis zu 0,45µs, beim T11 bis zu 0,26µs.

### Beispiel

```
DIM val_1, val_2 AS FLOAT
```

```
EVENT:
```

```
val_1 = 16
```

```
val_2 = SQRT(val_1) 'Ergebnis: val_2 = 4
```

## START\_PROCESS

**START\_PROCESS** startet einen bestimmten Prozess.

### Syntax

**START\_PROCESS** (*processnum*)

### Parameter

*processnum*    Nummer des zu startenden Prozesses  
(1...12, 15)

LONG

### Bemerkungen



Stellen Sie auf jeden Fall sicher, dass der zu startende Prozess bereits auf das ADwin-System übertragen wurde, bevor Sie ihn starten.

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

- bereits läuft oder
- gleich der Nummer des aufrufenden Prozesses ist.

Sie können einen Prozess mit **START\_PROCESS** nur aus einem anderen Prozess heraus starten (anders aber mit **RESTART\_PROCESS**). Es ist daher nicht möglich, dass ein Prozess sich selbst startet, beispielsweise im Abschnitt **FINISH**:

### Siehe auch

END, EXIT, RESTART\_PROCESS, START\_PROCESS\_DELAYED, STOP\_PROCESS

### Beispiel

```
EVENT:
IF (ADC(1) > 3072) THEN'Schwellwert überschritten?
  START_PROCESS(2)      'Messprozess 2 starten
END
ENDIF
```

## START\_PROCESS\_DELAYED

Nur für Prozessor T11: **START\_PROCESS\_DELAYED** startet einen bestimmten Prozess (Abschnitt **EVENT** :) mit einer definierten Verzögerung.

### Syntax

**START\_PROCESS\_DELAYED**(processnum, delay)

### Parameter

processnum	Nummer des zu startenden Prozesses (1...10)	LONG
delay	Verzögerungszeit (>30) in Taktzyklen des Zählers. Beim T11 dauert 1 Taktzyklus 3,3 ns.	LONG

### Bemerkungen

Stellen Sie auf jeden Fall sicher, dass der zu startende Prozess bereits auf das ADwin-System übertragen wurde, bevor Sie ihn starten.



Die Anweisung kann nur einen zeitgesteuerten Prozess mit hoher Priorität starten; sie bleibt ohne Auswirkung, wenn Sie die Nummer eines Prozesses angeben, für den eine der folgenden Bedingungen gilt:

- Der Prozess wird vom externen Event-Signal gesteuert.
- Der Prozess hat niedrige Priorität.
- Der Prozess läuft bereits.
- Der Prozess hat die gleiche Nummer wie der aufrufende Prozess.

Sie können einen Prozess mit **START\_PROCESS\_DELAYED** nur aus einem anderen Prozess heraus starten (anders mit **RESTART\_PROCESS**).

Der gestartete Prozess beginnt immer mit dem Abschnitt **EVENT** :, die Programmabschnitte **INIT** : und **LOWINIT** : werden nicht ausgeführt.

Für den gewünschten Startzeitpunkt gilt:

- Die Verzögerung beginnt mit der Verarbeitung der Anweisung **START\_PROCESS\_DELAYED**; die Verarbeitung der Anweisung dauert 30 Taktzyklen.
- Aus einem hochprioriten Programmabschnitt heraus kann der Startzeitpunkt nur eingehalten werden, wenn die Verzöge-

rungszeit `delay` größer ist als die restliche Bearbeitungszeit des Abschnitts.  
In dem Programmabschnitt werden alle noch folgenden Zeilen verarbeitet, bevor der gewählte Prozess starten kann. Der Startzeitpunkt wird also durch eine zu lange Restbearbeitungszeit verzögert.

**Siehe auch**

RESTART\_PROCESS, START\_PROCESS, STOP\_PROCESS

**Beispiel**

**EVENT:**

```
...
IF (cond = 2) THEN
    REM Wenn Bedingung erfüllt, Prozess 2 um 100 Zyklen = 333 ns
    REM verzögert starten.
    START_PROCESS_DELAYED(2,100)
ENDIF
REM Es folgen keine weiteren Programmzeilen, damit der
REM Startzeitpunkt sicher eingehalten wird.
```



## STOP\_PROCESS

**STOP\_PROCESS** stoppt aus einem Prozess heraus einen bestimmten anderen Prozess.

### Syntax

**STOP\_PROCESS** (processnum)

### Parameter

processnum Nummer des zu stoppenden Prozesses  
(1...12, 15)

LONG

### Bemerkungen

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

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

Das Unterbinden des Aufrufs von **EVENT**: läuft ab wie folgt:

- Der Prozess bekommt zunächst den Status "Prozess wird gestoppt" (siehe **PROZESSn\_RUNNING**), bei niederprioriten Prozessen erst nach einer gewissen Zeitspanne (time-out).
- Wenn der Abschnitt **EVENT**: gerade bearbeitet wird, während das Stopp-Signal eintrifft, wird diese Bearbeitung in jedem Fall zu Ende geführt.
- In der Regel wird der Abschnitt **EVENT**: noch ein weiteres Mal aufgerufen und bearbeitet.
- Falls vorhanden, wird der Abschnitt **FINISH**: abgearbeitet (immer mit niedriger Priorität).
- Der Prozess ruht nun, kann aber jederzeit wieder gestartet werden.

Wenn der Prozess sich selbst stoppen soll, verwenden Sie den Befehl **END** oder **EXIT**.



### Siehe auch

END, EXIT, PROZESSn\_RUNNING, RESTART\_PROCESS, START\_PROCESS, START\_PROCESS\_DELAYED

**Beispiel**

```
EVENT:
  IF (ADC(1) > 3072) THEN 'Schwellwert überschritten?
    STOP_PROCESS(2)      'Messprozess 2 stoppen
  END
ENDIF
```

## """ String

Zeichenfolgen (Strings) werden in Anführungszeichen " " angegeben.

### Syntax

```

IMPORT STRING.LI*      '*.LI9 für T9, *.LIA für T10,
                        '*.LIB für T11

DIM text[length] AS STRING

text = "ADwin"

```

### Parameter

text[]	Name der Text-Variablen	<b>ARRAY</b> <b>STRING</b>
length	Länge der Text-Variablen	<b>CONST</b> <b>LONG</b>

### Bemerkungen

Dimensionieren Sie Text-Variablen mit **DIM ... AS STRING** (siehe Seite 129). Setzen Sie eine Zeichenfolge, die sie der Variablen zuweisen möchten, in Anführungszeichen.

Näheres zu Textvariablen und dem Aufbau von Strings finden Sie unter „Strings“ auf Seite 61.

Sie können Zeichenfolgen mit den unten aufgeführten Anweisungen bearbeiten. Außerdem können Sie Zeichenfolgen mit dem „+“-Operator aneinander hängen.

### Siehe auch

+ (String-Addition), DIM, ASC, CHR, FLOTOSTR, FLO40TOSTR, LNGTOSTR, STRCOMP, STRLEFT, STRLEN, STRMID, STRRIGHT, VALF, VALI

## Beispiel

```

IMPORT STRING.LI9

REM Strings mit 3 und 1 Zeichen dimensionieren
DIM chars[3] AS STRING
DIM char[1] AS STRING

INIT:
    REM Den Strings eine Zeichenfolge übergeben
    chars = "ABC"
    char = "z"

EVENT:
    PAR_1 = chars[1]      'PAR_1 = 3 Anzahl der Zeichen
    PAR_2 = chars[2]      'PAR_2 = 65 (= "A")
    PAR_3 = chars[3]      'PAR_3 = 66 (= "B")
    PAR_4 = chars[4]      'PAR_4 = 67 (= "C")
    PAR_5 = chars[5]      'PAR_5 = 0 String-Terminierung

    REM Wandlung in Großbuchstaben:
    REM Kleinbuchstabe: a, b, c, ..., x, y, z?
    PAR_6 = ASC(char)
    IF (PAR_6>96 AND PAR_6<133) THEN
        REM 32 subtrahieren um in Großbuchstaben zu wandeln
        CHR(PAR_6-32,char)
    ENDIF

```

## STRCOMP

**STRCOMP** überprüft zwei Zeichenketten auf Gleichheit.

### Syntax

```

IMPORT STRING.LI*          '*.LI9 für T9, *.LIA für T10,
                             '*.LIB für T11

ret_val = STRCOMP(string1[], string2[])

```

### Parameter

string1[], Zeichenkette  
string2[]

**ARRAY**

**STRING**

**CONST**

ret\_val      0: Zeichenketten sind gleich  
             -1: Zeichenketten sind ungleich

**LONG**

### Bemerkungen

Wenn die Zeichenketten von unterschiedlicher Länge sind, wird immer ein negativer Wert zurückgegeben, auch wenn die kürzere Zeichenkette in der längeren enthalten ist.

### Siehe auch

"" String, + (String-Addition), ASC, CHR, FLOTOSTR, FLO40TOSTR, LNGTOSTR, STRLEFT, STRLEN, STRMID, STRRIGHT, VALF, VALI

### Beispiel

```

IMPORT STRING.LI9

DIM text1[7], text2[7], text3[8] AS STRING

INIT:
text1 = "ADbasic"      'ADbasic richtig geschrieben
text2 = "ADbasci"      'ADbasic falsch geschrieben
text3 = "ADbasica"     'ADbasic falsch geschrieben

EVENT:
PAR_1 = STRCOMP(text1,text2) 'PAR_1=-1
PAR_2 = STRCOMP(text1,text3) 'PAR_2=-1

```

## STRLEFT

**STRLEFT** kopiert aus einer Zeichenkette von links her eine bestimmte Anzahl von Zeichen in eine zweite Zeichenkette.

### Syntax

```
IMPORT STRING.LI*          '*.LI9 für T9, *.LIA für T10,  
                           '*.LIB für T11  
  
STRLEFT(string1[], length, string2[])
```

### Parameter

<code>string1[]</code>	Zeichenkette, aus der kopiert wird	<b>ARRAY</b> STRING
<code>length</code>	Anzahl der zu kopierenden Zeichen	LONG
<code>string2[]</code>	Zeichenkette, in die kopiert wird	<b>ARRAY</b> STRING

### Siehe auch

"" String, + (String-Addition), ASC, CHR, FLOTOSTR, FLO40TOSTR, LNGTOSTR, STRCOMP, STRLEN, STRMID, STRRIGHT, VALF, VALI

**Beispiel**

```
IMPORT STRING.LI9

REM Quell- und Ziel-String dimensionieren:
DIM text1[32], text2[13] AS STRING

INIT:
  REM Quell-String definieren
  text1 = "MEGA-Echtzeit mit ADwin-Systemen"

EVENT:
  REM Hole 13 Zeichen von links aus dem String text1
  STRLEFT(text1,13,text2)
  PAR_1 = text2[1]      'String-Länge = 13 Zeichen
  PAR_2 = text2[2]      'ASCII-Zeichen 4Dh = "M"
  PAR_3 = text2[3]      'ASCII-Zeichen 45h = "E"
  PAR_4 = text2[4]      'ASCII-Zeichen 47h = "G"
  PAR_5 = text2[5]      'ASCII-Zeichen 41h = "A"
  PAR_6 = text2[6]      'ASCII-Zeichen 2Dh = "-"
  PAR_7 = text2[7]      'ASCII-Zeichen 45h = "E"
  PAR_8 = text2[8]      'ASCII-Zeichen 63h = "c"
  PAR_9 = text2[9]      'ASCII-Zeichen 68h = "h"
  PAR_10 = text2[10]     'ASCII-Zeichen 74h = "t"
  PAR_11 = text2[11]     'ASCII-Zeichen 7Ah = "z"
  PAR_12 = text2[12]     'ASCII-Zeichen 65h = "e"
  PAR_13 = text2[13]     'ASCII-Zeichen 69h = "i"
  PAR_14 = text2[14]     'ASCII-Zeichen 74h = "t"
  PAR_15 = text2[15]     'String-Ende-Zeichen = 0
```

## STRLEN

**STRLEN** gibt die Anzahl der Zeichen in einer Zeichenfolge zurück.

### Syntax

```

IMPORT STRING.LI*          '*.LI9 für T9, *.LIA für T10,
                             '*.LIB für T11

ret_val = STRLEN(string[])

```

### Parameter

<code>string[]</code>	Zeichenkette, deren Länge ermittelt wird	<b>ARRAY</b> <b>STRING</b>
<code>ret_val</code>	Anzahl der Zeichen in der Zeichenfolge	<b>LONG</b>

### Siehe auch

"" String, + (String-Addition), ASC, CHR, FLOTOSTR, FLO40TOSTR, LNGTOSTR, STRCOMP, STRLEFT, STRMID, STRRIGHT, VALF, VALI

### Beispiel

```

IMPORT STRING.LI9
DIM text[50] AS STRING

INIT:
    text = "MEGA-Echtzeit mit ADwin-Systemen"

EVENT:
    PAR_1 = STRLEN(text) 'String-Länge: PAR_1 = 32

```



## STRMID

**STRMID** kopiert aus einer Zeichenkette ab einer anzugebenden Position eine bestimmte Anzahl von Zeichen in eine zweite Zeichenkette.

### Syntax

```
IMPORT STRING.LI*      '*.LI9 für T9, *.LIA für T10,  
                        '*.LIB für T11  
  
STRMID(string1[], start, length, string2[])
```

### Parameter

<code>string1[]</code>	Zeichenkette, aus der kopiert wird	<b>ARRAY</b> <b>STRING</b>
<code>start</code>	Position des ersten Zeichens, das kopiert wird	<b>LONG</b>
<code>length</code>	Anzahl der zu kopierenden Zeichen	<b>LONG</b>
<code>string2[]</code>	Zeichenkette, in die kopiert wird	<b>ARRAY</b> <b>STRING</b>

### Siehe auch

"" String, + (String-Addition), ASC, CHR, FLOTOSTR, FLO40TOSTR, LNGTOSTR, STRCOMP, STRLEFT, STRLEN, STRRIGHT, VALF, VALI

**Beispiel**

```
IMPORT STRING.LI9

REM Quell- und Ziel-String dimensionieren:
DIM text1[32], text2[18] AS STRING

INIT:
    REM Quell-String definieren
    text1 = "MEGA-Echtzeit mit ADwin-Systemen"

EVENT:
    REM Kopiere 18 Zeichen ab dem 6. Zeichen aus dem String text1
    STRMID(text1,6,18,text2)
    PAR_1 = text2[1]      'String-Länge = 18 Zeichen
    PAR_2 = text2[2]      'ASCII-Zeichen 45h = "E"
    PAR_3 = text2[3]      'ASCII-Zeichen 63h = "c"
    PAR_4 = text2[4]      'ASCII-Zeichen 68h = "h"
    PAR_5 = text2[5]      'ASCII-Zeichen 74h = "t"
    PAR_6 = text2[6]      'ASCII-Zeichen 7Ah = "z"
    PAR_7 = text2[7]      'ASCII-Zeichen 65h = "e"
    PAR_8 = text2[8]      'ASCII-Zeichen 69h = "i"
    PAR_9 = text2[9]      'ASCII-Zeichen 74h = "t"
    PAR_10 = text2[10]    'ASCII-Zeichen 20h = " "
    PAR_11 = text2[11]    'ASCII-Zeichen 6Dh = "m"
    PAR_12 = text2[12]    'ASCII-Zeichen 69h = "i"
    PAR_13 = text2[13]    'ASCII-Zeichen 74h = "t"
    PAR_14 = text2[14]    'ASCII-Zeichen 20h = " "
    PAR_15 = text2[15]    'ASCII-Zeichen 41h = "A"
    PAR_16 = text2[16]    'ASCII-Zeichen 44h = "D"
    PAR_17 = text2[17]    'ASCII-Zeichen 77h = "w"
    PAR_18 = text2[18]    'ASCII-Zeichen 69h = "i"
    PAR_19 = text2[19]    'ASCII-Zeichen 6Eh = "n"
    PAR_20 = text2[20]    'String-Ende-Zeichen = 0
```

## STRRIGHT

**STRRIGHT** kopiert aus einer Zeichenkette von rechts her eine bestimmte Anzahl von Zeichen in eine zweite Zeichenkette.

### Syntax

```
IMPORT STRING.LI*      '*.LI9 für T9, *.LIA für T10,  
                        '*.LIB für T11  
  
STRRIGHT(string1[], length, string2[])
```

### Parameter

<code>string1[]</code>	Zeichenkette, aus der kopiert wird	<b>ARRAY</b> STRING
<code>length</code>	Anzahl der zu kopierenden Zeichen	LONG
<code>string2[]</code>	Zeichenkette, in die kopiert wird	<b>ARRAY</b> STRING

### Siehe auch

"" String, + (String-Addition), ASC, CHR, FLOTOSTR, FLO40TOSTR, LNGTOSTR, STRCOMP, STRLEFT, STRLEN, STRMID, VALF, VALI

**Beispiel**

```

IMPORT STRING.LI9

REM Quell- und Ziel-String dimensionieren:
DIM text1[32], text2[13] AS STRING

INIT:
  REM Quell-String definieren
  text1 = "MEGA-Echtzeit und ADwin-Systeme"

EVENT:
  REM Hole 13 Zeichen von rechts aus dem String text1
  STRRIGHT(text1,13,text2)
  PAR_1 = text2[1]      'String-Länge = 13 Zeichen
  PAR_2 = text2[7]      'ASCII-Zeichen 41h = "A"
  PAR_3 = text2[8]      'ASCII-Zeichen 44h = "D"
  PAR_4 = text2[9]      'ASCII-Zeichen 77h = "w"
  PAR_5 = text2[10]     'ASCII-Zeichen 69h = "i"
  PAR_6 = text2[11]     'ASCII-Zeichen 6Eh = "n"
  PAR_7 = text2[12]     'ASCII-Zeichen 2Dh = "-"
  PAR_8 = text2[13]     'ASCII-Zeichen 53h = "S"
  PAR_9 = text2[14]     'ASCII-Zeichen 79h = "y"
  PAR_10 = text2[13]    'ASCII-Zeichen 73h = "s"
  PAR_11 = text2[13]    'ASCII-Zeichen 74h = "t"
  PAR_12 = text2[13]    'ASCII-Zeichen 65h = "e"
  PAR_13 = text2[14]    'ASCII-Zeichen 6Dh = "m"
  PAR_14 = text2[13]    'ASCII-Zeichen 65h = "e"
  PAR_15 = text2[15]    'String-Ende-Zeichen = 0

```

## SUB ... ENDSUB

**SUB...ENDSUB** definiert ein Unterprogramm-Makro mit Übergabeparametern.

### Syntax

```
SUB macro_name({val_1, val_2, ...})
    {DIM var AS <VAR_TYPE>}
    ...
    'Anweisungsblock
ENDSUB
```

### Parameter

**macro\_name** Name des Unterprogramms

**val\_1, val\_2** Namen der Übergabeparameter;  
für Felder ist die Syntax mit Dimensionsklammern erforderlich: `array[]` oder `DATA_n[]`.

FLOAT
LONG

### Bemerkungen

Allgemeine Informationen über Makros finden Sie in Kapitel 3.5.1 auf Seite 68.

Diese Anweisung definiert ein Unterprogramm-Makro, d.h. der vollständige Anweisungsblock zwischen **SUB** und **ENDSUB** wird an der aufrufenden Stelle eingefügt.

Unterprogramm-Makros erhöhen die Übersichtlichkeit Ihres Quelltextes. Beachten Sie aber, dass aber auch jeder Aufruf des Unterprogramms die kompilierte Datei vergrößert.

Sie können Unterprogramme an 3 Stellen einfügen:

1. Vor dem Abschnitt **INIT**: / **LOWINIT**:
2. Nach dem Abschnitt **FINISH**:
3. In einer separaten Datei, die Sie mit **#INCLUDE** einbinden (aber nur an einer der Stellen, die unter 1. und 2. angegeben sind).

Beachten Sie bitte, dass Sie in Unterprogrammen:

- keine Prozess-Abschnitte wie **LOWINIT**:, **INIT**:, **EVENT**:, oder **FINISH**: definieren dürfen.
- am Anfang lokale Variablen definieren können, die nur innerhalb des Unterprogramms und für die Dauer der Abarbeitung

verfügbar sind.

Dies gilt auch, wenn die Variablen den gleichen Namen haben wie Variablen außerhalb des Unterprogramms.

Bei Berechnungsausdrücken in einem Unterprogramm sollten die Übergabeparameter in Klammern stehen. Auf diese Weise vermeiden Sie Probleme mit der Rangfolge von Operatoren (z.B. Punkt- vor Strich-Rechnung).

Ein Unterprogramm wird mit seinem Namen und allen definierten Argumenten aufgerufen. Als Argument ist jeder Berechnungsausdruck (auch Felder) zulässig, solange er den passenden Datentyp hat. Wenn Sie keine Argumente definieren, müssen Sie dennoch beim Aufruf des Unterprogramms die Leerklammern verwenden: `name()`.



Wenn Sie in einem Unterprogramm einem Übergabeparameter einen Wert zuweisen, darf beim Unterprogramm-Aufruf für diesen Übergabeparameter nur eine Variable oder ein einzelnes Feld-Element als Argument verwendet werden.

Wenn ein Feld (kein Feldelement) als Übergabeparameter eines Unterprogramms verwendet wird, ist die Syntax für Definition und Aufruf unterschiedlich:

- Unterprogramm-Definition *mit* Dimensions-Klammern:  
`SUB subname(array[]) ...`
- Unterprogramm-Aufruf *ohne* Dimensions-Klammern:  
`subname(array)`

### Siehe auch

#INCLUDE, FUNCTION ... ENDFUNCTION, LIB\_SUB ... LIB\_ENDSUB, LIB\_FUNCTION ... LIB\_ENDFUNCTION

### Beispiel

```
SUB Fast_Dac1(val_1)
REM Gibt val_1 auf dem analogen Ausgang 1 eines ADwin-Gold aus
POKE(20400050h, (val_1)) 'Wert ins Ausgangsregister
POKE(20400010h, 11011b) 'Wandlung starten
ENDSUB
```

Der Aufruf des Unterprogramms `Fast_Dac1` erfolgt mit der Programmzeile:

```
Fast_Dac1(NeuerWert)
```

Das gleiche Unterprogramm mit einem Feld als Übergabe-Parameter:

```
SUB Fast_Dac1(array[]) AS FLOAT
    REM Gibt Element 3 des Felds array auf dem
    REM analogen Ausgang eines ADwin-Gold aus
    POKE(20400050h, (array[3])) 'Wert ins Ausgangsregister
    POKE(20400010h, 11011b) 'Wandlung starten
ENDFUNCTION
```

Der Aufruf dieses Unterprogramms erfolgt wieder in ähnlicher Weise (allerdings *ohne* die Dimensionsklammern):

```
Fast_Dac1(array)
```

Beim Aufruf können Sie für `array` ein globales oder ein lokales Feld angeben. Tragen Sie nur den Feldnamen ein ohne Elementnummer und eckige Klammern.

## TAN

**TAN** liefert den Tangens eines Arguments.

### Syntax

```
ret_val = TAN(angle)
```

### Parameter

<code>angle</code>	Winkel im Bogenmaß ( $-\pi/2 \dots \pi/2$ )	Float
<code>ret_val</code>	Tangens des Winkels ( $-1 \dots 1$ )	Float

### Bemerkungen

Wenn Sie für den Winkel Werte außerhalb von  $-\pi/2 \dots \pi/2$  verwenden, nimmt der Berechnungsfehler mit wachsendem Wert zu.

Die Ausführungszeit des Befehls beträgt beim T9 bis zu 1,33  $\mu$ s, beim T10 bis zu 0,67  $\mu$ s, beim T11 bis zu 0,31  $\mu$ s.

### Siehe auch

SIN, COS, ARCSIN, ARCCOS, ARCTAN

### Beispiel

```
DIM val_1, val_2 AS FLOAT
```

```
EVENT:
```

```
val_1 = 5.3
```

```
val_2 = TAN(val_1)      'Ergebnis: val_2 = -1.50...
```



## TRACE\_MODE\_PAUSE

**TRACE\_MODE\_PAUSE** deaktiviert den Trace-Modus.

### Syntax

**TRACE\_MODE\_PAUSE**

### Bemerkungen

Mit dem Befehl **TRACE\_MODE\_PAUSE** kann in einem *ADbasic*-Programm der Trace-Modus gezielt deaktiviert werden. Mit dem Befehl **TRACE\_MODE\_RESUME** kann der Trace-Modus wieder aktiviert werden. Die Deaktivierung/Aktivierung betrifft nur Trace-aktive Programmzeilen, die mit ? (Fragezeichen) markiert wurden.

Beide Befehle gemeinsam erlauben, den Trace-Modus gezielt für bestimmte Programmzeilen oder -abschnitte einzuschalten. So kann der Trace-Modus z.B. nur solange aktiviert werden, wie eine bestimmte Bedingung erfüllt ist.

Die Aktivierung des Trace-Modus und dessen Optionen sind auf Seite 30 unter Menüeintrag „Trace Setup ...“ erklärt; Informationen zur Anwendung finden Sie in Kapitel 4.3.3 auf Seite 81.

### Siehe auch

**TRACE\_MODE\_RESUME**

### Beispiel

```
EVENT:
  PAR_1 = ADC(1,4)
  IF (PAR_1 > 32768) THEN
    TRACE_MODE_RESUME 'Trace-Modus ein

    ...                'Der Trace-Modus ist für diesen
    ...                'Programmabschnitt durchgehend aktiv

    TRACE_MODE_PAUSE  'Trace-Modus aus
  ENDIF
```

## TRACE\_MODE\_RESUME

**TRACE\_MODE\_RESUME** aktiviert den Trace-Modus ab der nächsten Programmzeile.

### Syntax

**TRACE\_MODE\_RESUME**

### Bemerkungen

Mit dem Befehl **TRACE\_MODE\_RESUME** kann in einem *ADbasic*-Programm der Trace-Modus wieder aktiviert werden, wenn er zuvor mit **TRACE\_MODE\_PAUSE** deaktiviert wurde. Die Deaktivierung/Aktivierung betrifft nur Trace-aktive Programmzeilen, die mit ? (Fragezeichen) markiert wurden.

Beide Befehle gemeinsam erlauben, den Trace-Modus gezielt für bestimmte Programmzeilen oder -abschnitte einzuschalten. So kann der Trace-Modus z.B. nur solange aktiviert werden, wie eine bestimmte Bedingung erfüllt ist.

Die Aktivierung des Trace-Modus und dessen Optionen sind auf Seite 30 unter Menüeintrag „Trace Setup ...“ erklärt; Informationen zur Anwendung finden Sie in Kapitel 4.3.3 auf Seite 81.

### Siehe auch

**TRACE\_MODE\_PAUSE**

### Beispiel

```
EVENT:
  PAR_1 = ADC(1,4)
  IF (PAR_1 > 32768) THEN
    TRACE_MODE_RESUME    'Trace-Modus ein

    ...                  'Der Trace-Modus ist für diesen
    ...                  'Programmabschnitt durchgehend aktiv

    TRACE_MODE_PAUSE    'Trace-Modus aus
  ENDIF
```

## VALF

**VALF** konvertiert eine Zeichenfolge (String) in eine Fließkomma-Zahl (float).

### Syntax

```

IMPORT STRING.LI*      '*.LI9 für T9, *.LIA für T10,
                        '*.LIB für T11

ret_val = VALF(string[])

```

### Parameter

`string[]`      Zu wandelnde Zeichenfolge im Format:

**ARRAY**  
**STRING**

Mantisse (max. 10 Zeichen)				Exponent (0...99)	
{+}	vvvvv	.	nnnnn	e	{+} nn
-		,		E	-

`ret_val`      Erzeugter Fließkomma-Wert

**FLOAT**

### Bemerkungen

Wenn Sie kein Vorzeichen angeben, wird ein positives Vorzeichen angenommen.

Das Zeichen "E" trennt Mantisse und Exponent. Von der Mantisse werden bei T9 und T10 bis zu 7 Zeichen (Vor- und Nachkommastellen) ausgewertet, bei T11 bis zu 10 Zeichen. Wenn Sie mehr Ziffern angeben, gehen die jeweils letzten verloren. Als Dezimalzeichen sind sowohl der Punkt als auch das Komma zulässig.

Beachten Sie den Wertebereich für Float-Werte in Kapitel 3.2.3 auf Seite 48. Werte außerhalb des Wertebereichs werden als „Unendlich“ oder Null interpretiert

Wenn Sie unzulässige Zeichen verwenden (= andere als im Format angegeben), wird die Zeichenfolge nur bis zum ersten unzulässigen Zeichen ausgewertet.

**Siehe auch**

"" String, ASC, CHR, FLOTOSTR, FLO40TOSTR, LNGTOSTR, STRCOMP, STRLEFT, STRLEN, STRMID, STRRIGHT, VALI

**Beispiel**

```
IMPORT STRING.LI9

DIM text[20] AS STRING

INIT:
  text="-271.8282E-02" 'Zu konvertierender String
  PAR_1 = text[1]      'String-Länge
  PAR_2 = text[2]      'ASCII-Zeichen 2Dh = "-"
  PAR_3 = text[3]      'ASCII-Zeichen 32h = "2"
  PAR_4 = text[4]      'ASCII-Zeichen 37h = "7"
  PAR_5 = text[5]      'ASCII-Zeichen 2Eh = "."
  PAR_6 = text[6]      'ASCII-Zeichen 31h = "1"
  PAR_7 = text[7]      'ASCII-Zeichen 34h = "4"
  PAR_8 = text[8]      'ASCII-Zeichen 31h = "1"
  PAR_9 = text[9]      'ASCII-Zeichen 35h = "5"
  PAR_10 = text[10]     'ASCII-Zeichen 39h = "9"
  PAR_11 = text[11]     'ASCII-Zeichen 45h = "E"
  PAR_12 = text[12]     'ASCII-Zeichen 2Dh = "-"
  PAR_13 = text[13]     'ASCII-Zeichen 31h = "1"
  PAR_14 = text[14]     'ASCII-Zeichen 30h = "0"
  PAR_15 = text[15]     'String-Ende-Zeichen

EVENT:
  FPAR_1 = VALF(text)   'String zu Float wandeln
```

## VALI

**VALI** konvertiert eine Zeichenfolge (String) in eine ganze Zahl (Long).

### Syntax

```

IMPORT STRING.LI*      '*.LI9 für T9, *.LIA für T10,
                        '*.LIB für T11

ret_val = VALI(string)

```

### Parameter

<code>string[]</code>	Zu wandelnde Zeichenfolge im Format : Vorzeichen: + (optional) oder – Vorkommastellen: max. 10 Ziffern	<b>ARRAY</b> <b>STRING</b>
	<hr/> <div style="display: flex; justify-content: space-between;"> <span>{+}</span> <span>vvvvvvvvvv</span> </div> <hr/> <div style="display: flex; justify-content: space-between;"> <span>–</span> <span></span> </div> <hr/>	
<code>ret_val</code>	Erzeugter Long-Wert	<b>LONG</b>

### Bemerkungen

Wenn Sie kein Vorzeichen angeben, wird ein positives Vorzeichen angenommen.

Beachten Sie den Wertebereich für Long-Werte:

–2.147.483.648 bis +2.147.483.647

Werte außerhalb dieses Bereichs werden als Null interpretiert.

Wenn Sie unzulässige Zeichen verwenden (= andere als im Format angegeben), wird nur die Zeichenfolg bis zum ersten unzulässigen Zeichen ausgewertet.

### Siehe auch

"" String, ASC, CHR, FLOTOSTR, FLO40TOSTR, LNGTOSTR, STRCOMP, STRLEFT, STRLEN, STRMID, STRRIGHT, VALF

**Beispiel**

```
IMPORT STRING.LI9

DIM text[20] AS STRING

INIT:
  text="-1234567890"      'Zu konvertierender String
  PAR_1 = text[1]         'String-Länge = 11
  PAR_2 = text[2]         'ASCII-Zeichen 2Dh = "-"
  PAR_3 = text[3]         'ASCII-Zeichen 31h = "1"
  PAR_4 = text[4]         'ASCII-Zeichen 32h = "2"
  PAR_5 = text[5]         'ASCII-Zeichen 33h = "3"
  PAR_6 = text[6]         'ASCII-Zeichen 34h = "4"
  PAR_7 = text[7]         'ASCII-Zeichen 35h = "5"
  PAR_8 = text[8]         'ASCII-Zeichen 36h = "6"
  PAR_9 = text[9]         'ASCII-Zeichen 37h = "7"
  PAR_10 = text[10]        'ASCII-Zeichen 38h = "8"
  PAR_11 = text[11]        'ASCII-Zeichen 39h = "9"
  PAR_12 = text[12]        'ASCII-Zeichen 30h = "0"
  PAR_13 = text[13]        'String-Ende-Zeichen

EVENT:
  PAR_20 = VALI(text)    'String zu Long wandeln
```

## XOR

Der Operator **XOR** (Exklusiv-Oder) verknüpft zwei ganzzahlige Werte bitweise.

### Syntax

```
... val_1 XOR val_2 ...
```

### Parameter

val\_1, val\_2 Ganzzahliger Wert

LONG
------

### Siehe auch

AND, NOT, OR

### Beispiel

```
DIM value AS LONG
```

```
EVENT:
```

```
value = 0100b XOR 0110b 'Ergebnis: value = 4 XOR 5 = 0010b = 2
```





### 6.3 ADwin-Gold und ADwin-light-16

Verwenden Sie die folgenden Befehle nur mit den Systemen *ADwin-Gold* und *ADwin-light-16*, auch wenn manche Befehle für das *ADwin-Pro*-System gleich oder ähnlich lauten.

Für die Befehle dieses Abschnitts benötigen Sie keine Include-Datei.

Für *ADwin-light-16* (Grundversion) und die Erweiterung *ADwin-light-16-CO1* sind zusätzlich folgende Zähler-Befehle aus Kapitel 6.4 gültig:

- CNT\_CLEAR, Seite 267
- CNT\_ENABLE, Seite 271
- CNT\_LATCH, Seite 278
- CNT\_READ, Seite 282
- CNT\_READLATCH, Seite 284

## ADC

**ADC** misst die Spannung an einem analogen Eingang über den 16 Bit-Wandler und gibt eine (dem Messergebnis entsprechende) ganze Zahl zurück, falls angegeben mit einem Verstärkungsfaktor.

Für den 12 Bit-/14 Bit-Wandler des *ADwin-Gold*-Systems verwenden Sie bitte die Anweisung **ADC12**.

### Syntax

```
ret_val = ADC(input_ch{,gain})
```

### Parameter

<code>input_ch</code>	Nummer des analogen Eingangskanals Gold: 1...16; L16: 1, 3, 5, ..., 15.	LONG
<code>gain</code>	Verstärkungsfaktor (1, 2, 4, 8).	LONG CONST
<code>ret_val</code>	Messergebnis in Digits (0...65535).	LONG

### Bemerkungen

**ADC** ist eine Zusammenstellung aufeinander folgender Funktionen:

- Multiplexer auf den angegebenen Eingangskanal stellen (**SET\_MUX**).
- Einschwingen des Multiplexers abwarten.
- Messung starten: Das angelegte Analogsignal am 16 Bit-Wandler (unter Berücksichtigung des Verstärkungsfaktors) in einen Digitalwert konvertieren (**START\_CONV**).
- Das Ende der Konvertierung abwarten (**WAIT\_EOC**).
- Den Digitalwert aus einem Register lesen und zurückgeben (**READADC**).

Die Befehls-Ausführungszeit ist abhängig vom verwendeten System. Sie finden Angaben zur Multiplexer-Einschwingzeit und der Wandlungszeit in der Hardware-Dokumentation Ihres Systems.

Wenn Sie die Zykluszeit des Prozesses (**PROCESSDELAY**) auf einen Wert unter 20µs einstellen, beträgt die Ausführungszeit des Befehls nur noch etwa die Hälfte. Dies ist möglich, indem der Compiler für diesen Fall die Wartezeit für das Einschwingen des Multiplexers überspringt. Es wird also angenommen, dass Sie eine Messung ohne

Umstellung des Multiplexers beabsichtigen.

Wenn (bei solch kurzen Zykluszeiten) auch die erste Messung korrekt sein soll, müssen Sie eine bestimmte Zeit vor der ersten Benutzung der Anweisung **ADC** mit **SET\_MUX** den Multiplexer auf den gewünschten Eingangskanal setzen. Diese Zeit muss mindestens so groß sein wie die Multiplexer-Einschwingzeit.

In folgenden Fällen sollten Sie anstelle der Anweisung **ADC** die Anweisungen **SET\_MUX**, **START\_CONV**, **WAIT\_EOC** und **READADC** verwenden:

- Sehr kurze Zykluszeiten: **PROCESSDELAY** < 240 (s.o.).
- Hoher Innenwiderstand (>3kΩ) der Spannungsquelle des Messsignals: Dies verlängert die Einschwingzeit des Multiplexers.
- Sie möchten unvermeidliche Wartezeiten für zusätzliche Programmschritte nutzen.

Wenn Sie einen nicht vorhandenen Eingangskanal angeben, ist das Messergebnis undefiniert.

Der Messbereich ist abhängig vom Verstärkungsfaktor:

Verstärkung	Eingangs-Spannungsbereich	Messbereich
1	-10V ... 10V	20V
2	-5V ... 5V	10V
4	-2,5V ... 2,5V	5V
8	-1,25V ... 1,25V	2,5V

Mit der folgenden Formel berechnen Sie aus dem zurückgegebenen Digitalwert die gemessene Spannung:

$$\text{Spannung} = (\text{Digits} - 32768_{\text{bipolar}}) \cdot \frac{\text{Messbereich}}{65536}$$

Für den Fall, dass die Verstärkung gleich 1 gewählt wurde (Messbereich von 20 Volt), gelten die in der Tabelle angegebenen Werte:

Messbereich	Rückgabewert von <b>ADC</b>			
	0	32768	65535	1 Digit
20V	-10V	0V	+9,999695V	305,175µV

**Siehe auch**

ADC12, READADC, SET\_MUX, START\_CONV, WAIT\_EOC

**Gültig für**

Gold, L16

**Beispiel**

```
DIM iw AS LONG           'Deklaration

EVENT:
  REM Analogen Eingang 1 mit Verstärkung 4 messen
  iw = ADC(1,4)
  REM Messwert in globale Variable schreiben, damit er
  REM vom PC gelesen werden kann
  PAR_1 = iw
```

## ADC12

Nur *ADwin-Gold*: **ADC12** misst die Spannung eines analogen Eingangs über den 12 Bit- (Rev. A) oder 14 Bit-Wandler (Rev. B).

Das Messergebnis wird in Digits zurückgegeben, falls angegeben mit einem Verstärkungsfaktor.

Für den 16 Bit-Wandler verwenden Sie bitte die Anweisung **ADC**.

### Syntax

```
ret_val = ADC12(input_ch{,gain})
```

### Parameter

<code>input_ch</code>	Nummer des analogen Eingangskanals (1...16)	LONG
<code>gain</code>	Verstärkungsfaktor (1, 2, 4, 8)	LONG
<code>ret_val</code>	Messergebnis in Digits: 12 Bit: 0, 16, 32, ..., 65520 14 Bit: 0, 4, 8, ..., 65532	LONG

### Beschreibung

Die Anweisung **ADC12** ist eine Zusammenstellung von aufeinander folgenden Funktionen:

- Multiplexer auf den angegebenen Eingangskanal stellen (**SET\_MUX**).
- Einschwingen des Multiplexers abwarten.
- Messung starten: Den angelegten Analogwert am 12 Bit- oder 14 Bit-Wandler (unter Berücksichtigung des Verstärkungsfaktors) in einen Digitalwert konvertieren (**START\_CONV**).
- Das Ende der Konvertierung abwarten (**WAIT\_EOC**).
- Den Digitalwert aus einem Register lesen und zurückgeben (**READADC12**).

Die Befehls-Ausführungszeit ist abhängig vom verwendeten System. Sie finden Angaben zur Multiplexer-Einschwingzeit und der Wandlungszeit in der Hardware-Dokumentation Ihres Systems.

Die Schrittweiten 16 und 4 der zurückgegebenen Messwerte ergeben sich daraus, dass das 12 Bit- und 14 Bit-Wandlungsergebnis jeweils

als 16 Bit-Wert übergeben wird: Bei 12 Bit-Wandlern sind die Bits 0 bis 3 immer 0 (Null), bei 14 Bit-Wandlern die Bits 0 und 1.

In folgenden Fällen sollten Sie anstelle der Anweisung **ADC** die Anweisungen **SET\_MUX**, **START\_CONV**, **WAIT\_EOC** und **READADC12** verwenden:

- Sehr kurze Zykluszeiten: **PROCESSDELAY** < 200: Die Anweisung **ADC12** kann nicht mehr innerhalb der Zykluszeit durchgeführt werden.
- Hoher Innenwiderstand (>3kΩ) der Spannungsquelle des Messsignals: Dies verlängert die Einschwingzeit des Multiplexers.
- Sie möchten unvermeidliche Wartezeiten für zusätzliche Programmschritte nutzen.

Wenn Sie einen nicht vorhandenen Eingangskanal angeben, ist das Messergebnis undefiniert.

Der Messbereich ist abhängig vom Verstärkungsfaktor:

Verstärkung	Messbereich	Messbereich
1	-10V ... 10V	20V
2	-5V ... 5V	10V
4	-2,5V ... 2,5V	5V
8	-1,25V ... 1,25V	2,5V

Für die Umrechnung des Wandlungsergebnisses auf die Eingangsspannung gilt folgende Formel:

$$\text{Spannung} = (\text{Digits} - 32768_{\text{bipolar}}) \cdot \frac{\text{Messbereich}}{65536}$$

Für den Fall, dass die Verstärkung gleich 1 gewählt wurde (Messbereich von 20 Volt), gelten die in der Tabelle angegebenen Werte:

Messbereich	Rückgabewert von <b>ADC12</b>			
	0	32768	65520	16 Digits
20V	-10V	0V	+9,99512V	4,88mV

### Siehe auch

ADC, SET\_MUX, START\_CONV, WAIT\_EOC, READADC12

## Gültig für

Gold

## Beispiel

```
DIM iw AS LONG           'Deklaration

EVENT:
  REM Analogen Eingang 1 mit Verstärkung 4 messen
  iw = ADC12(1,4)
  REM Messwert in globale Variable schreiben, damit er
  REM vom PC gelesen werden kann
  PAR_1 = iw
```

## CLEAR\_DIGOUT

**CLEAR\_DIGOUT** setzt einen der digitalen Ausgänge auf 0 (TTL-Pegel low).

### Syntax

**CLEAR\_DIGOUT** (*output\_no*)

### Parameter

*output\_no*      Nummer, die den Ausgang festlegt:

**CONST**

LONG

<i>bit_n</i>	15	...	5	...	1	0
<i>ADwin-Gold</i>	DIO31	...	DIO21	...	DIO17	DIO16
<i>ADwin-light-16</i>	–	–	5	...	1	0

### Beschreibung

**CLEAR\_DIGOUT** akzeptiert als Parameter nur eine Konstante. Wenn Sie den Ausgang durch eine Variable festlegen wollen, verwenden Sie den Befehl **DIGOUT\_WORD**.

Sie müssen den zu löschenden Kanal vorher als Ausgang konfiguriert haben, anderenfalls hat **CLEAR\_DIGOUT** keine Funktion.

Sie können mit der Anweisung **CONF\_DIO** die digitalen Kanäle in Gruppen zu je 8 als Ein- oder Ausgang konfigurieren. Wir empfehlen die Einstellung mit **CONF\_DIO** (1100b): Kanäle 0...15 als Eingänge, Kanäle 16...31 als Ausgänge.

**CLEAR\_DIGOUT** löscht ein Bit in dem Ausgangs-Register der Kanäle DIO16...DIO31. Dadurch wird am zugehörigen Kanal – falls dieser als Ausgang geschaltet ist – der TTL-Pegel low angelegt.

Wenn Sie einen der Kanäle 0...15 auf 0 setzen möchten, löschen Sie das entsprechende Bit im Ausgangs-Register der Kanäle DIO00...DIO15; auch hier gilt: konfigurieren Sie zuerst den Kanal als Ausgang. Gehen Sie vor wie folgt (siehe Beispiel unten):

- Lesen Sie das Register mit **PEEK** aus. Die Registernummer entnehmen Sie bitte Ihrem Hardware-Handbuch.
- Lesen Sie das Register mit **PEEK** aus. Die Registernummer entnehmen Sie bitte Ihrem Hardware-Handbuch.
- Löschen Sie das zum Kanal gehörige Bit (**AND**-Maskierung).
- Schreiben Sie den Wert mit **POKE** in das Register zurück.



### Siehe auch

CONF\_DIO, DIGOUT\_WORD, SET\_DIGOUT

### Gültig für

Gold, L16, L16-CO1, L16-DIO1, L16-DIO2

### Beispiel

```
DIM val AS LONG           'Deklaration

INIT:
    SET_DIGOUT(0)          'Dig. Ausgang DIO16 auf 0 setzen

EVENT:
    val = ADC(1)           'Messwerterfassung
    IF (val > 3000) THEN
        CLEAR_DIGOUT(0)    'Dig. Ausgang DIO16/0 zurücksetzen
    ENDIF
```

Nur *ADwin-Gold*: Ein Unterprogramm, das ein einzelnes Bit der DIO-Leitungen 0...15 auf 0 setzt, könnte wie folgt aussehen:

```
SUB CLEAR_DIGOUT_CONN1(bitno)
    POKE(204001C0h,
    PEEK(204001C0h) AND NOT(SHIFT_LEFT(1,bitno)) )
ENDSUB
```

## CONF\_DIO

Nur *ADwin-Gold*: **CONF\_DIO** konfiguriert die 32 digitalen Kanäle in Gruppen zu je 8 als Ein- oder Ausgänge.

### Syntax

**CONF\_DIO**(*val*)

### Parameter

*val* Bitmuster, das die digitalen Kanäle als Ein- oder Ausgang konfiguriert: **CONST**  
 Bit=0: Kanäle als Eingänge **LONG**  
 Bit=1: Kanäle als Ausgänge

Bitnr. in <i>val</i>	15...4	3	2	1	0
Kanäle	–	DIO31	DIO23	DIO15	DIO07
		...	...	...	...
		DIO24	DIO16	DIO08	DIO00

### Beschreibung

Die digitalen Kanäle des *ADwin-Gold*-Systems sind nach dem Einschalten zunächst als Eingänge konfiguriert (und können also auch noch nicht als Ausgänge angesprochen werden). Sie können nur in Gruppen zu je 8 als Ein- oder Ausgänge konfiguriert werden.

Wir empfehlen Ihnen die Konfiguration **CONF\_DIO**(1100b), d.h. DIO00...DIO15 sind Eingänge und DIO16...DIO31 sind Ausgänge. Auf diese Einstellung sind die Anweisungen **CLEAR\_DIGOUT**, **SET\_DIGOUT**, **DIGIN\_WORD**, **DIGOUT\_WORD**, **DIGIN** abgestimmt; eine andere Konfiguration kann deren Funktion einschränken oder sie ganz aufheben.

Wenn Sie eine andere als die empfohlene Konfiguration verwenden, können Sie die digitalen Kanäle nur verarbeiten und setzen, indem Sie die entsprechenden Hardware-Register (siehe *ADwin-Gold* Hardware-Handbuch) mit **PEEK** und **POKE** auslesen und beschreiben.

Verwenden Sie für die Angabe des Bitmusters vorzugsweise die binäre Schreibweise (Suffix „b“). Sie stellt die Zuordnung zwischen Bits und Kanalgruppen besser dar als die (gleichermaßen zulässige) dezimale oder hexadezimale Schreibweise.

### Siehe auch

CLEAR\_DIGOUT, DIGIN, DIGIN\_WORD, DIGOUT\_WORD,  
SET\_DIGOUT

### Gültig für

Gold, L16

### Beispiel

```
REM Konfiguriere DIO00...DIO15 als Eingänge  
REM und DIO16...DIO31 als Ausgänge  
CONF_DIO (1100b)
```

## DAC

**DAC** gibt eine definierte Spannung auf einem bestimmten analogen Ausgang aus.

### Syntax

```
DAC (num, val)
```

### Parameter

num	Nummer des analogen Ausgangs: 1...8	LONG
val	Wert in Digits, der die auszugebende Spannung definiert: 0...65535	LONG

### Beschreibung

Wenn Sie einen Digit-Wert außerhalb des zulässigen Wertebereichs angeben, wird er automatisch auf den systemspezifischen Minimal- oder Maximalwert korrigiert.

### Gültig für

Gold, Gold-DA, L16

### Beispiel

```
REM Digitaler P-Regler
DIM set_to, gain, diff, out AS LONG'Deklaration

EVENT:
    set_to = PAR_1          'Sollwert
    gain = PAR_2            'Dimensionieren
    diff = set_to - ADC(1) 'Regelabweichung berechnen
    out = diff * gain       'Stellgröße berechnen
    DAC(1, out)            'Ausgabe der Stellgröße
```

## DIGIN

**DIGIN** gibt den Wert eines der digitalen Eingänge DIO00...DIO15 zurück.

### Syntax

```
ret_val = DIGIN(channel_no)
```

### Parameter

**channel\_no** Nummer, die den abzufragenden Eingang festlegt:

LONG

CONST

*ADwin-Gold:*

<b>channel_no</b>	0	1	...	14	15
Eingang Nr.	DIO00	DIO01	...	DIO14	DIO15

*ADwin-light-16:*

<b>channel_no</b>	0	1	...	5
Eingang Nr.	0	1	...	5

**ret\_val** 1: TTL-Pegel high liegt an  
0: TTL-Pegel low liegt an

LONG

### Bemerkungen

Diese Anweisung ist für das Auslesen weniger Bits geeignet. Wenn mehrere Bits (z.B. auch in Schleifen) auszulesen sind, ist die Verwendung der Anweisung **DIGIN\_WORD** deutlich schneller. Beachten Sie dies insbesondere bei zeitkritischen Anwendungen.

Die folgenden Anmerkungen beziehen sich nur auf *ADwin-Gold*:

**DIGIN** erfordert, dass Sie den entsprechenden Kanal vorher als Eingang konfiguriert haben. Bei einem Ausgang wird kein sinnvoller Wert zurückgegeben.

Sie können mit der Anweisung **CONF\_DIO** die digitalen Kanäle in Gruppen zu je 8 als Ein- oder Ausgang konfigurieren. Wir empfehlen die Einstellung mit **CONF\_DIO** (1100b): Kanäle 0...15 als Eingänge, Kanäle 16...31 als Ausgänge.

Wenn Sie den Wert eines der Kanäle DIO16...DIO31 benötigen, lesen Sie das entsprechende Bit aus dem Eingangs-Register dieser Kanäle

(auch hier: Konfigurieren Sie zuerst den Kanal als Eingang). Gehen Sie vor wie folgt (siehe 2. Beispiel `DIGIN_CONN2`):

- Lesen Sie das Register mit **PEEK** aus. Die Registernummer entnehmen Sie bitte Ihrem Hardware-Handbuch.
- Löschen Sie alle Bits außer dem zum Kanal gehörigen Bit (**AND**-Maskierung).

### Siehe auch

`CONF_DIO` (nur *ADwin-Gold*), `DIGIN_WORD`, `DIGOUT_WORD`

### Gültig für

Gold, L16

### Beispiel

`REM Beispiel für Gold und L16`

`DIM DATA_1[10000] AS LONG AS FIFO`

**EVENT:**

`REM Ist der digitale Eingang 0 gesetzt?`

`IF (DIGIN(0) = 1) THEN`

`DATA_1 = ADC(1) 'Messwerterfassung`

`ENDIF`

Nur *ADwin-Gold*: Eine Funktion, die den Wert eines der Kanäle DIO16...DIO31 zurückgibt, könnte wie folgt aussehen:

**FUNCTION** `DIGIN_CONN2`(bitno) **AS LONG**

`DIGIN_CONN2=SHIFT_RIGHT(PEEK(204001B0h), bitno) AND 1`

**ENDFUNCTION**

## DIGIN\_WORD

**DIGIN\_WORD** gibt die Werte aller digitalen Eingänge auf einmal zurück.

### Syntax

```
ret_val = DIGIN_WORD()
```

### Parameter

**ret\_val** Bitmuster, das den TTL-Pegeln an den digitalen Eingängen entspricht (Zuordnung s.u.).  
 1: TTL-Pegel high liegt an  
 0: TTL-Pegel low liegt an

LONG

*ADwin-Gold:*

Bitnummer in <code>ret_val</code>	31 ... 16	15	14	...	1	0
Eingang Nr.	–	DIO15	DIO14	...	DIO01	DIO00

*ADwin-light-16:*

Bitnummer in <code>ret_val</code>	31 ... 6	5	...	0
Eingang Nr.	–	5	...	0

### Bemerkungen (nur ADwin-Gold)

**DIGIN\_WORD** erfordert, dass Sie die Kanäle DIO00... DIO15 vorher als Eingänge konfiguriert haben. Für Ausgangskanäle wird kein sinnvoller Wert zurückgegeben.

Sie können mit der Anweisung **CONF\_DIO** die digitalen Kanäle in Gruppen zu je 8 als Ein- oder Ausgang konfigurieren. Wir empfehlen die Einstellung mit **CONF\_DIO** (1100b): Kanäle 0...15 als Eingänge, Kanäle 16...31 als Ausgänge.

Wenn Sie die Werte der Kanäle DIO16...DIO31 benötigen, lesen Sie das Eingangs-Register dieser Kanäle aus (auch hier: Konfigurieren Sie zuerst die Kanäle als Eingänge); siehe auch 2. Beispiel [DIGIN\\_WORD\\_CONN2](#). Die Registernummer entnehmen Sie bitte Ihrem Hardware-Handbuch. Die Bits in diesem Rückgabewert sind den Kanälen wie folgt zugeordnet:

Bitnummer	31...16	15	...	1	0
Eingang Nr.	–	DIO31	...	DIO17	DIO16

**Siehe auch**

CONF\_DIO (nur *ADwin-Gold*), DIGOUT\_WORD

**Gültig für**

Gold, L16

**Beispiel**

```
REM Beispiel für Gold und L16
DIM DATA_1[10000] AS LONG AS FIFO

EVENT:
  REM Abfrage, ob die Eingänge 0 und 1 gesetzt sind
  IF ((DIGIN_WORD() AND 11b) = 11b) THEN
    DATA_1 = ADC(1)      'Messwerterfassung
  ENDIF
```

Nur *ADwin-Gold*: Eine Funktion, die den Wert der Kanäle DIO16...DIO31 zurückgibt, könnte wie folgt aussehen:

```
FUNCTION DIGIN_WORD_CONN2() AS LONG
  DIGIN_WORD_CONN2=PEEK(204001B0h)
ENDFUNCTION
```



## DIGOUT\_WORD

**DIGOUT\_WORD** setzt mit einem Bitmuster gleichzeitig die digitalen Ausgänge auf definierte TTL-Pegel.

### Syntax

**DIGOUT\_WORD** (*val*)

### Parameter

*val* Bitmuster, das den TTL-Pegeln an den digitalen Ausgängen entspricht (s. Tabelle).  
 1: Setzen auf TTL-Pegel high  
 0: Setzen auf TTL-Pegel low

LONG

*ADwin-Gold:*

Bit-Nr. in <i>val</i>	31 ... 16	15	14	...	1	0
Ausgang Nr.	–	DIO31	DIO30	...	DIO17	DIO16

*ADwin-light-16:*

Bit-Nr. in <i>val</i>	31 ... 6	5	...	0
Ausgang Nr.	–	5	...	0

### Bemerkungen (nur ADwin-Gold)

Diese Anweisung erfordert, dass Sie die Kanäle DIO16... DIO31 vorher als Ausgänge konfiguriert haben. Anderenfalls hat sie keine Funktion.

Sie können mit der Anweisung **CONF\_DIO** die digitalen Kanäle in Gruppen zu je 8 als Ein- oder Ausgang konfigurieren. Wir empfehlen die Einstellung mit **CONF\_DIO** (1100b) : Kanäle 0...15 als Eingänge, Kanäle 16...31 als Ausgänge.

Wenn Sie die Pegel der Kanäle DIO16...DIO31 setzen wollen, geben Sie das entsprechende Bitmuster auf das Ausgangs-Register dieser Kanäle aus (auch hier: Konfigurieren Sie zuerst die Kanäle als Ausgänge); siehe auch 2. Beispiel **DIGIN\_WORD\_CONN1**. Die Registernummer entnehmen Sie bitte Ihrem Hardware-Handbuch.

**Siehe auch**

CONF\_DIO (nur *ADwin-Gold*), DIGIN\_WORD, CLEAR\_DIGOUT, SET\_DIGOUT

**Gültig für**

Gold, L16

**Beispiel**

```
REM Beispiel für Gold und L16
DIM value AS LONG

INIT:
  REM Ein- und Ausgänge konfigurieren (nur für ADwin-Gold)
  CONF_DIO(1100b)

EVENT:
  value = ADC(1)           'Meßwerterfassung
  IF (value > 3000) THEN 'Grenzwert überschritten?
    DIGOUT_WORD(101b)      'Ausgänge 0 und 2 setzen, alle anderen
                           'Ausgänge werden gelöscht!
  ENDIF
```

Nur *ADwin-Gold*: Ein Unterprogramm, das die TTL-Pegel der Kanäle DIO00...DIO15 setzt, könnte wie folgt aussehen:

```
SUB DIGOUT_WORD_CONN1(value)
  POKE(204001C0h,value)
ENDSUB
```

## READADC

**READADC** gibt einen gewandelten Wert von einem 16 Bit A/D-Wandler zurück.

### Syntax

```
ret_val = READADC (num)
```

### Parameter

num	Nummer des zu lesenden 16 Bit-Wandlers (1, 2)	LONG
ret_val	Messwert in Digits, der der anliegenden Spannung am Wandler entspricht.	LONG

### Bemerkungen

Beim *ADwin-Gold*-System lesen Sie gewandelte Werte des 12 Bit / 14 Bit A/D-Wandlers mit der Anweisung **READADC12** aus.

### Siehe auch

ADC, READADC12, SET\_MUX, START\_CONV, WAIT\_EOC

### Gültig für

Gold, L16

### Beispiel

```
EVENT:
REM Multiplexer setzen: ADC1 auf Kanal 3, ADC2
REM auf Kanal 4 (ohne Verstärkung)
SET_MUX (1001b)
...
START_CONV (1b)      'MUX-Einschwingzeit überbrücken
WAIT_EOC (11b)       'Wandlung für beide ADC starten
PAR_1 = READADC (1)  'Ende der Wandlungen abwarten
                       'Wert von ADC1 einlesen
PAR_2 = READADC (2)  'Wert von ADC2 einlesen
```

## READADC12

Nur *ADwin-Gold*: **READADC12** gibt einen gewandelten Wert von einem der beiden 12 Bit / 14 Bit A/D-Wandler zurück.

### Syntax

```
ret_val = READADC12 (num)
```

### Parameter

num	Nummer des zu lesenden 12 Bit / 14 Bit -Wandlers (1, 2)	LONG
ret_val	Messwert in Digits, der der anliegenden Spannung am Wandler entspricht: 12 Bit: 0, 16, 32, ..., 65520 14 Bit: 0, 4, 8, 16, ..., 65532	LONG

### Bemerkungen

Lesen Sie gewandelte Werte des 16 Bit A/D-Wandlers mit der Anweisung **READADC** aus.

Die A/D-Wandler (ADC) teilen den Messbereich von 20 Volt in gleich große Bereiche (Digits) ein, bei 12 Bit-Wandlern in 4096 Digits, bei 14 Bit-Wandlern sind es 16384 Digits.

Um den Vergleich mit Messwerten der 16 Bit-Wandler zu vereinfachen, gibt die Anweisung **READADC12** das Ergebnis „linksbündig“ zurück, also von Bit 31 absteigend; die Bits 3...0 (12 Bit-Wandler) oder 1 und 0 (14 Bit-Wandler) haben stets den Wert 0.

Bei gleicher anliegender Spannung liefern daher die Anweisungen **READADC** und **READADC12** in den oberen 12 / 14 Bits das gleiche Ergebnis.

### Siehe auch

ADC12, SET\_MUX, START\_CONV, WAIT\_EOC

### Gültig für

Gold

### Beispiel

```
DIM val1, val2 AS LONG
```

#### EVENT:

```
REM Multiplexer setzen: ADC12-1 auf Kanal 3, ADC12-2  
REM auf Kanal 4 (ohne Verstärkung)  
SET_MUX(1001b)  
... 'MUX-Einschwingzeit überbrücken  
START_CONV(11000b) 'Wandlung für beide ADC starten  
WAIT_EOC(11000b) 'Ende der Wandlungen abwarten  
val1 = READADC12(1) 'Wert von ADC12-1 einlesen  
val2 = READADC12(2) 'Wert von ADC12-2 einlesen
```

## SET\_DIGOUT

**SET\_DIGOUT** setzt einen der digitalen Ausgänge auf 1 (TTL-Pegel high).

### Syntax

**SET\_DIGOUT** (*channelno*)

### Parameter

*channelno*      Nummer, die den zu setzenden Ausgang festlegt: **CONST**  
LONG

<i>channelno</i>	0	1	...	5	...	15
<i>ADwin-Gold</i>	DIO16	DIO17	...	DIO21	...	DIO31
<i>ADwin-light-16</i>	0	1	...	5	–	–

### Beschreibung

Diese Anweisung ist für das Setzen weniger Bits geeignet. Wenn mehrere Bits (z.B. auch in Schleifen) zu setzen sind, ist die Verwendung der Anweisung **DIGOUT\_WORD** deutlich schneller. Beachten Sie dies insbesondere bei zeitkritischen Anwendungen.

Die folgenden Anmerkungen beziehen sich nur auf *ADwin-Gold*:

Wenn Sie den zu setzenden Ausgang durch den Wert einer Variablen bestimmen wollen, verwenden Sie den Befehl **DIGOUT\_WORD**.

erfordert, dass Sie den entsprechenden Kanal vorher als Ausgang konfiguriert haben. Anderenfalls hat sie keine Funktion.

Sie können mit der Anweisung **CONF\_DIO** die digitalen Kanäle in Gruppen zu je 8 als Ein- oder Ausgang konfigurieren. Wir empfehlen die Einstellung mit **CONF\_DIO** (1100b): Kanäle 0...15 als Eingänge, Kanäle 16...31 als Ausgänge.

Die Anweisung setzt ein Bit in dem Ausgangs-Register der Kanäle DIO16...DIO31. Dadurch wird am zugehörigen Kanal - falls dieser als Ausgang geschaltet ist - der TTL-Pegel high angelegt.

Wenn Sie einen der Kanäle 0...15 auf 1 setzen möchten, setzen Sie das entsprechende Bit im Ausgangs-Register der Kanäle DIO0 ...

DIO15 (auch hier: Konfigurieren Sie zuerst den Kanal als Ausgang).  
Gehen Sie vor wie folgt (siehe 2. Beispiel `SET_DIGOUT_CONN1`):

- Lesen Sie das Register mit **PEEK** aus. Die Registernummer entnehmen Sie bitte Ihrem Hardware-Handbuch.
- Setzen Sie das zum Kanal gehörige Bit (**OR**-Maskierung).
- Schreiben Sie den Wert mit **POKE** in das Register zurück.

### Siehe auch

`CONF_DIO` (nur *ADwin-Gold*), `CLEAR_DIGOUT`, `DIGOUT_WORD`

### Gültig für

Gold, L16

### Beispiel

```
REM Beispiel für Gold und L16
DIM val AS LONG

INIT:
  REM Dig. Ein-/Ausgänge konfigurieren (nur ADwin-Gold)
  CONF_DIO(1100b)

EVENT:
  val = ADC(1)           'Messwerterfassung
  IF (val > 3000) THEN
    SET_DIGOUT(0)        'Dig. Ausgang DIO16 / 0 setzen
  ENDIF
```

Nur *ADwin-Gold*: Ein Unterprogramm, das ein einzelnes Bit der DIO-Leitungen 0...15 auf 1 setzt, könnte wie folgt aussehen:

```
SUB SET_DIGOUT_CONN1(bitno)
  POKE(204001C0h, PEEK(204001C0h) OR SHIFT_LEFT(1,bitno) )
ENDSUB
```

## SET\_MUX

**SET\_MUX** setzt einen oder mehrere Multiplexer auf den gewählten Messkanal und stellt (nur bei *ADwin-Gold*) die Verstärkung ein.

### Syntax

**SET\_MUX**(*pattern*)

### Parameter

*pattern* Bitmuster zur Zuordnung von Messkanälen und Verstärkung LONG

Bitnr.	9	8	7	6	5	4	3	2	1	0
	PGA 2		PGA 1		MUX 2			MUX 1		

**PGA 1 / 2** Nur *ADwin-Gold*: Mit jeweils 2 Bits (6...7 / 8...9) bestimmen Sie den Verstärkungsfaktor des Multiplexers:

2 Bits PGA 1 / PGA 2

00: Faktor 1

01: Faktor 2

10: Faktor 4

11: Faktor 8

**MUX 1 / 2** Mit jeweils 3 Bits (0...2 / 3...5) bestimmen Sie den Kanal, auf den der Multiplexer eingestellt wird:

3 Bits MUX 2 MUX 1

000: Kanal 2 Kanal 1

001: Kanal 4 Kanal 3

010: Kanal 6 Kanal 5

011: Kanal 8 Kanal 7

100: Kanal 10 Kanal 9

101: Kanal 12 Kanal 11

110: Kanal 14 Kanal 13

111: Kanal 16 Kanal 15

### Beschreibung

Beachten Sie bitte, dass die Umstellung des Multiplexers auf einen anderen Kanal eine definierte Einschwingzeit benötigt. Erst danach dürfen Sie die Wandlung der anliegenden Spannung mit **START\_CONV**



starten. Bitte entnehmen Sie die erforderliche Einschwingzeit (und auch die Wandlungszeit) der Hardware-Dokumentation Ihres Systems.

Wir empfehlen für die Angabe des Bitmusters die binäre Schreibweise (Suffix „b“). Sie können damit die erforderlichen Bitmuster besser darstellen als mit der (gleichfalls zulässigen) dezimalen oder hexadezimalen Schreibweise.

### Siehe auch

ADC, ADC12, START\_CONV, WAIT\_EOC, READADC, READADC12

### Gültig für

Gold, L16

### Beispiel

Sie möchten den Multiplexer des ADC1 auf einem *ADwin-Gold* auf den Kanal 5 und die Verstärkung 8 einstellen, und gleichzeitig den Multiplexer des ADC2 auf den Kanal 10 und die Verstärkung 2.

Hierzu benötigen Sie das Bitmuster: 0111100010b (dezimal: 482).

Bei *ADwin-light-16* kann keine Verstärkung eingestellt werden; verwenden Sie hier das verkürzte Bitmuster: 011010b (dezimal: 26).

```
DIM val AS LONG
```

```
EVENT:
```

```
SET_MUX(0111100010b) 'Multiplexer setzen (s.o.)
REM Nutzen Sie hier die Einschwingzeit des Multi-
REM plexers durch einige Befehlszeilen.
START_CONV(1)         'Start AD-Wandlung ADC1
WAIT_EOC(1)           'Wandlungsende des ADC1 abwarten
val = READADC(1)      'Wert von ADC1 einlesen
```

## START\_CONV

**START\_CONV** kann die Wandlung an einem oder mehreren A/D-Wandlern sowie an allen D/A-Wandler starten.

### Syntax

**START\_CONV**(pattern)

### Parameter

pattern

Bitmuster, das die zu startenden Wandler festlegt (nur Bits 0...4 verwendbar).

**CONST**

LONG

Bit-Nr.	4	3	2	1	0	Systeme
ADC1, 16 Bit	–	–	–	–	x	Gold, L16
ADC2, 16 Bit	–	–	–	x	–	Gold
alle DAC	–	–	x	–	–	Gold, L16
ADC1, 12 Bit / ADC1, 14 Bit	–	x	–	–	–	Gold
ADC2, 12 Bit / ADC2, 14 Bit	x	–	–	–	–	Gold

### Beschreibung

Beachten Sie bitte, dass es sich bei ADC1 und ADC2 um einen Analog-Digital-Wandler mit entweder 12 Bit/14 Bit oder 16 Bit handeln kann. Entnehmen Sie weitere Informationen hierzu bitte Ihrem Hardware-Handbuch.

Sie können als Parameter nur Konstanten einsetzen, keine Variablen.

Wir empfehlen für die Angabe des Bitmusters die binäre Schreibweise (Suffix „b“). Sie können damit die erforderlichen Bitmuster besser darstellen als mit der (gleichfalls zulässigen) dezimalen oder hexadezimalen Schreibweise.

### Siehe auch

ADC, ADC12, SET\_MUX, WAIT\_EOC, READADC, READADC12

### Gültig für

Gold, L16

### Beispiel

```
DIM val1 AS LONG
```

#### EVENT:

```
SET_MUX(0)           'Multiplexer auf Kanal 1 setzen
REM Überbrücken Sie hier die Einschwingzeit des
REM Multiplexers mit Befehlszeilen
START_CONV(1)         'Start ADC1 A/D-Wandlung
WAIT_EOC(1)           'Ende der Wandlung abwarten
val1 = READADC(1)     'Wert auslesen
```

## WAIT\_EOC

**WAIT\_EOC** wartet auf das Ende der Wandlung an einem bestimmten A/D-Wandler.

### Syntax

**WAIT\_EOC** (*pattern*)

### Parameter

*pattern*

Bitmuster, das die zu startenden Wandler festlegt (nur Bits 0...4 verwendbar).

**CONST**

LONG

Bit-Nr.	4	3	2	1	0	Systeme
ADC1, 16 Bit	–	–	–	–	x	Gold, L16
ADC2, 16 Bit	–	–	–	x	–	Gold
ADC1, 12/14 Bit	–	x	–	–	–	Gold
ADC2, 12/14 Bit	x	–	–	–	–	Gold

### Beschreibung

Wenn Sie mehr als eines der Bits setzen, wird so lange gewartet, bis die Wandlung an allen entsprechenden ADC beendet ist.

Setzen Sie immer nur die Bits vorhandener ADC. Anderenfalls führt dies in einem hochprioren Prozess zur Unterbrechung der Kommunikation zwischen ADwin-System und PC.

### Siehe auch

ADC, ADC12, READADC, SET\_MUX, START\_CONV

### Gültig für

Gold, L16

### Beispiel

```
DIM val AS LONG
```

```
EVENT:
```

```
SET_MUX(001000b)      'MUX des ADC2 auf Kanal 4 setzen  
REM Überbrücken Sie hier die Einschwingzeit des  
REM Multiplexers mit Befehlszeilen  
START_CONV(2)          'Start A/D-Wandlung ADC2  
WAIT_EOC(2)            'Wandlungsende an ADC2 abwarten  
val = READADC(2)       'Wert auslesen
```



### 6.4 ADwin-light-16 DIO1/2 / ADwin-Gold CO1

Die Befehle dieses Abschnitts sind aufgeteilt in folgende Gruppen:

- **Zähler-Befehle** (**CNT** ...; Seite 267 ff)  
für *ADwin-light-16* (Basis, CO1, DIO1, DIO2) und *ADwin-Gold* (CO1).

Die Zähler sind von 1 aufsteigend nummeriert. Viele Zähler-Befehle verwenden ein Bitmuster, dessen Bits den Zählern wie folgt zugeordnet sind:

Bit-Nr.	31...4	3	2	1	0
Zähler-Nr.	–	4 <sup>a</sup>	3 <sup>a</sup>	2 <sup>b</sup>	1

a. nur für *ADwin-Gold* CO1

b. bei *ADwin-light-16*: nicht für Erweiterung CO1

Wir empfehlen das Bitmuster in binärer Schreibweise (Suffix „b“) anzugeben. Die Angabe „10b“ oder „1100b“ stellt besser dar, welcher Zähler angesprochen wird und welcher nicht, als es mit der (gleichfalls zulässigen) dezimalen oder hexadezimalen Schreibweise möglich ist.

- **Digitalkanal-Befehle** (**DIG**...; Seite 294 ff);  
anwendbar nur für *ADwin-light-16* DIO1 und DIO2.
- **CAN-Bus-Befehle** (Seite 312 ff);  
anwendbar nur für *ADwin-light-16* DIO1.

Innerhalb der Gruppen sind die Befehle alphabetisch sortiert.

Beachten Sie, dass Sie für die Systeme die jeweils passende Include-Datei einbinden müssen (*ADWGCNT.INC* oder *ADWL16.INC*).

Für *ADwin-light-16* DIO2 sind zusätzlich einige Befehle aus Kapitel 6.5 gültig:

SSI_MODE	Seite 356	SSI_SET_CLOCK	Seite 362
SSI_READ	Seite 358	SSI_START	Seite 364
SSI_SET_BITS	Seite 360	SSI_STATUS	Seite 366

**Befehle in diesem Abschnitt**

Die Befehle in diesem Abschnitt sind für folgende ADwin-Systeme gültig:

Befehl	Gold CO1	L16			
		Basis	CO1	DIO1	DIO2
CNT_CLEAR (Seite 267)	x	x	x	x	x
CNT_CLEARENABLE (Seite 269)	–	–	–	x	x
CNT_ENABLE (Seite 271)	x	x	x	x	x
CNT_GETSTATUS (Seite 273)	x	–	–	x	x
CNT_INPUTMODE (Seite 276)	x	–	–	x	x
CNT_LATCH (Seite 278)	x	x	x	x	x
CNT_MODE (Seite 280)	x	–	–	x	x
CNT_READ (Seite 282)	x	x	x	x	x
CNT_READLATCH (Seite 284)	x	x	x	x	x
CNT_READFLATCH (Seite 286)	x	–	–	x	x
CNT_RESETSTATUS (Seite 288)	x	–	–	–	–
CNT_SE_DIFF (Seite 290)	x	–	–	–	–
CNT_SET (Seite 292)	x	–	–	x	x
CONF_DIO_E (Seite 294 ff)	–	–	–	x	x
DIGIN_WORD1_E, DIGIN_WORD2_E DIGOUT_SET1_E, DIGOUT_SET2_E, DIGOUT_RESET1_E, DIGOUT_RESET2_E DIGOUT_WORD1_E, DIGOUT_WORD2_E DIGIN_LONG_E, DIGOUT_LONG_E	–	–	–	–	x
INIT_CAN (Seite 312 ff) EN_INTERRUPT, EN_RECEIVE, EN_TRANSMIT CAN_MSG, READ_MSG, TRANSMIT SET_CAN_BAUDRATE, GET_CAN_REG, SET_CAN_REG	–	–	–	x	–



## CNT\_CLEAR

**CNT\_CLEAR** setzt einen oder mehrere Zähler auf Null, gemäß dem Bitmuster in `pattern`.

### Syntax

```
#INCLUDE ADWGCNT.INC      'nur für ADwin-Gold
#include ADWL16.INC       'nur für ADwin-light-16

CNT_CLEAR(pattern)
```

### Parameter

`pattern`      Bitmuster LONG  
 Bit = 0: Kein Einfluss  
 Bit = 1: Zähler auf Null setzen

Bit-Nr.	31...4	3	2	1	0
Zähler-Nr.	–	4 <sup>a</sup>	3 <sup>a</sup>	2 <sup>b</sup>	1

a. nur für *ADwin-Gold* CO1

b. bei *ADwin-light-16*: nicht für Erweiterung CO1

### Bemerkungen

Nach Ausführung der Anweisung wird das Bitmuster automatisch auf 0 (Null) zurückgesetzt, d.h. die zurückgesetzten Zähler beginnen zu zählen.

### Siehe auch

CNT\_CLEARENABLE, CNT\_ENABLE, CNT\_GETSTATUS, CNT\_INPUTMODE, CNT\_LATCH, CNT\_MODE, CNT\_READ, CNT\_READLATCH, CNT\_READFLATCH, CNT\_RESETSTATUS, CNT\_SET, CNT\_SE\_DIFF

### Gültig für

Gold-CO1, L16, L16-CO1, L16-DIO1, L16-DIO2

**Beispiel**

```

#INCLUDE ADWGCNT.INC      'Nur für ADwin-Gold
#INCLUDE ADWL16.INC      'Nur für ADwin-light-16

DIM old_1, new_1 AS LONG 'Variablen...
DIM old_2, new_2 AS LONG 'Dimensionieren

INIT:
    old_1 = 0              'Variablen...
    old_2 = 0              'initialisieren
    CNT_SE_DIFF (11b)      'Nur für ADwin-Gold:
                            'Alle Zähleringänge differentiell
    CNT_MODE (0)           'Alle Zähler auf externen Takteingang
    CNT_SET (11b)          'Zähler 1+2 mit Takt (CLK)- und
                            'Richtungs (DIR)-Eingang
    CNT_INPUTMODE (0)      'Funktionalität CLR/LATCH festlegen:
                            'Alle als CLR-Eingang
    CNT_CLEARENABLE (11b) 'Schaltet die CLR-Funktion beider
                            'Zähler frei (nur light-16)
    CNT_CLEAR (11b)        'Zähler 1+2 auf 0 zurücksetzen
    CNT_ENABLE (11b)      'Zähler 1+2 starten

EVENT:
    CNT_LATCH (11b)        'Zähler 1+2 gleichzeitig latches
    new_1 = CNT_READLATCH (1) 'Latch A Zähler 1 und...
    new_2 = CNT_READLATCH (2) 'Latch A Zähler 2 auslesen.
    PAR_1 = new_1 - old_1 'Differenz bilden (f = Impulse / Zeit)
    PAR_2 = new_2 - old_2 '-"-
    old_1 = new_1          'Neuen Zählerstand als alten speichern
    old_2 = new_2          '-"-

```

## CNT\_CLEARENABLE

Nur L16-DIO1: **CNT\_CLEARENABLE** sperrt den CLR-Eingang eines oder mehrerer Zähler oder gibt ihn frei, gemäß dem Bitmuster in **pattern**.

### Syntax

```
#INCLUDE ADWL16.INC          'nur für ADwin-light-16
CNT_CLEARENABLE(pattern)
```

### Parameter

**pattern**

Bitmuster

LONG

Bit = 0: CLR-Eingang am Zähler sperren

Bit = 1: CLR-Eingang am Zähler freigeben

Bit-Nr.	31...2	1	0
Zähler-Nr.	–	2	1

### Bemerkungen

Der Befehl beeinflusst alle Zähler gleichzeitig. Er ist nur sinnvoll einsetzbar, wenn mit **CNT\_INPUTMODE** der CLR-Modus eingestellt ist.

Verwenden Sie diesen Befehl möglichst nur bei gesperrtem Zähler.

### Siehe auch

CNT\_CLEAR, CNT\_ENABLE, CNT\_GETSTATUS,  
CNT\_INPUTMODE, CNT\_LATCH, CNT\_MODE, CNT\_READ,  
CNT\_READLATCH, CNT\_READFLATCH, CNT\_SET

### Gültig für

L16-DIO1, L16-DIO2

**Beispiel**

```

#INCLUDE ADWL16.INC

DIM old_1, new_1 AS LONG 'Variablen...
DIM old_2, new_2 AS LONG ' dimensionieren

INIT:
  old_1 = 0                'Variablen...
  old_2 = 0                ' initialisieren
  CNT_SE_DIFF(11b)        'Nur für ADwin-Gold:
                           'Alle Zähleringänge differentiell
  CNT_MODE(0)             'Alle Zähler auf externen Takteingang
  CNT_SET(11b)            'Zähler 1+2 mit Takt(CLK)- und
                           'Richtungs(DIR)-Eingang
  CNT_INPUTMODE(0)        'Funktionalität CLR/LATCH festlegen:
                           'Alle als CLR-Eingang
  CNT_CLEARENABLE(11b)    'Schaltet die CLR-Funktion beider
                           'Zähler frei
  CNT_CLEAR(11b)         'Zähler 1+2 auf 0 zurücksetzen
  CNT_ENABLE(11b)        'Zähler 1+2 starten

EVENT:
  CNT_LATCH(11b)          'Zähler 1+2 gleichzeitig latchen
  new_1 = CNT_READLATCH(1) 'Latch A Zähler 1 und...
  new_2 = CNT_READLATCH(2) 'Latch A Zähler 2 auslesen.
  PAR_1 = new_1 - old_1     'Differenz bilden (f = Impulse / Zeit)
  PAR_2 = new_2 - old_2    '-"-
  old_1 = new_1            'Neuen Zählerstand als alten speichern
  old_2 = new_2            '-"-

```

## CNT\_ENABLE

**CNT\_ENABLE** hält die mittels `pattern` gewählten Zähler an oder gibt sie frei, um eingehende Impulse zu zählen.

### Syntax

```
#INCLUDE ADWGCNT.INC      'nur für ADwin-Gold
#include ADWL16.INC       'nur für ADwin-light-16

CNT_ENABLE(pattern)
```

### Parameter

`pattern`

Bitmuster

LONG

Bit = 0: Zähler anhalten

Bit = 1: Zähler freigeben

Bit-Nr.	31...4	3	2	1	0
Zähler-Nr.	–	4 <sup>a</sup>	3 <sup>a</sup>	2 <sup>b</sup>	1

a. nur für *ADwin-Gold* CO1

b. bei *ADwin-light-16*: nicht für Erweiterung CO1

### Siehe auch

CNT\_CLEAR, CNT\_CLEARENABLE, CNT\_GETSTATUS,  
CNT\_INPUTMODE, CNT\_LATCH, CNT\_MODE, CNT\_READ,  
CNT\_READLATCH, CNT\_READFLATCH, CNT\_RESETSTATUS,  
CNT\_SET, CNT\_SE\_DIFF

### Gültig für

Gold-CO1, L16, L16-CO1, L16-DIO1, L16-DIO2

**Beispiel**

```

#INCLUDE ADWGCNT.INC      'Nur für ADwin-Gold
#INCLUDE ADWL16.INC      'Nur für ADwin-light-16

DIM old_1, new_1 AS LONG 'Variablen...
DIM old_2, new_2 AS LONG 'Dimensionieren

INIT:
    old_1 = 0              'Variablen...
    old_2 = 0              'initialisieren
    CNT_SE_DIFF (11b)      'Nur für ADwin-Gold:
                            'Alle Zähleringänge differentiell
    CNT_MODE (0)           'Alle Zähler auf externen Takteingang
    CNT_SET (11b)          'Zähler 1+2 mit Takt (CLK)- und
                            'Richtungs (DIR)-Eingang
    CNT_INPUTMODE (0)     'Funktionalität CLR/LATCH festlegen:
                            'Alle als CLR-Eingang
    CNT_CLEARENABLE (11b) 'Schaltet die CLR-Funktion beider
                            'Zähler frei (nur light-16)
    CNT_CLEAR (11b)       'Zähler 1+2 auf 0 zurücksetzen
    CNT_ENABLE (11b)      'Zähler 1+2 starten

EVENT:
    CNT_LATCH (11b)       'Zähler 1+2 gleichzeitig latches
    new_1 = CNT_READLATCH (1) 'Latch A Zähler 1 und...
    new_2 = CNT_READLATCH (2) 'Latch A Zähler 2 auslesen.
    PAR_1 = new_1 - old_1 'Differenz bilden (f = Impulse / Zeit)
    PAR_2 = new_2 - old_2 '-"-
    old_1 = new_1         'Neuen Zählerstand als alten speichern
    old_2 = new_2         '-"-

```

## CNT\_GETSTATUS

**CNT\_GETSTATUS** gibt den Inhalt des Zähler-Statusregisters zurück.

### Syntax

```
#INCLUDE ADWGCNT.INC      'nur für ADwin-Gold
#include ADWL16.INC       'nur für ADwin-light-16

ret_val = CNT_GETSTATUS()
```

### Parameter

**ret\_val**      Inhalt des Statusregisters:  
Hinweise auf mögliche Fehlerquellen;  
Bedeutung der Bits siehe Tabelle.

LONG

### Tabelle ADwin-Gold CO1

Bit Nr.	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
Signal	-	-	-	-	-	-	-	-	N4	N3	N2	N1	-	-	-	-
Bit Nr.	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Signal	L4	C4	L3	C3	L2	C2	L1	C1	B4	A4	B3	A3	B2	A2	B1	A1

- :don't care (Signalzustände undefiniert, mit 0F 0F 00 33h ausmaskieren)

Ax:Signal A (statisch)

Bx: Signal B (statisch)

Cx:Korrelationsfehler (Signal A und B sind identisch, d. h. nicht um ca. 90° phasenverschoben)

Lx: Leitungsfehler (Kabel abgezogen oder Leitung unterbrochen)

Nx:CLR-/LATCH-Eingang (statisch)

x:Zählernummer (1, 2, 3 oder 4)

Tabelle ADwin-light-16 DIO1

Bit Nr.	31...28	27	26	25	24	23...20	19	18	17	16	15...06	05	04	03...02	01	00
Signal	-	L2	C2	L1	C1	-	B2	A2	B1	A1	-	N2	N1	-	R2	R1

- :don't care (Signalzustände sind nicht definiert (mit 0F0F0033h ausmaskieren))

Ax:Signal A (statisch)

Bx: Signal B (statisch)

Cx:Korrelationsfehler\* (Signal A und B sind identisch, d.h. nicht um ca. 90° phasenverschoben)

Lx: Leitungsfehler\* (Kabel abgezogen oder Leitung unterbrochen)

Nx:CLR-/LATCH-Eingang (statisch)

Rx:Reset-Enable (Wert, der durch CNT\_CLEARENABLE gesetzt wurde)

x:Zählernummer (1 oder 2)

\* Auto-Reset (wird beim Auslesen zurückgesetzt)

### Bemerkungen

Ein Leitungsfehler (Lx) kann nur bei differentiellen Eingängen detektiert werden! Bei TTL-Eingängen sind diese Bits stets 0.

Nur ADwin-Gold CO1: Das Statusregister wird durch das Auslesen nicht zurückgesetzt; dies wird mit dem Befehl CNT\_RESETSTATUS erreicht.

### Siehe auch

CNT\_CLEAR, CNT\_CLEARENABLE, CNT\_ENABLE, CNT\_INPUTMODE, CNT\_LATCH, CNT\_MODE, CNT\_READ, CNT\_READLATCH, CNT\_READFLATCH, CNT\_RESETSTATUS, CNT\_SET, CNT\_SE\_DIFF

### Gültig für

Gold-CO1, L16-DIO1, L16-DIO2



### Beispiel (nur ADwin-light-16 DIO1)

```
#INCLUDE ADWL16.INC
DIM error AS LONG

INIT:
  CNT_MODE(0)           'Alle Zähler auf externen Takteingang
  CNT_SET(0)            'Alle Zähler mit A/B-Eingang (z.B.
                        'für Inkremental-Encoder)
  CNT_INPUTMODE(0)      'Funktionalität CLR/LATCH festlegen:
                        'Alle als CLR-Eingang
  CNT_CLEARENABLE(11b) 'Schaltet die CLR-Funktion beider
                        'Zähler frei
  CNT_CLEAR(11b)        'Zähler 1+2 auf 0 zurücksetzen
  CNT_ENABLE(1)         'Zähler 1 starten
  error = 0             'Fehlerindikator zurücksetzen

EVENT:
  PAR_1 = CNT_READ(1)    'Zähler 1 auslesen
  PAR_2 = CNT_GETSTATUS(1) AND 0F0F0033h 'Statusregister
                        'Zähler 1 auslesen
  IF (PAR_2 AND 2000000h = 2000000h) THEN 'Leitungs- bzw.
                        'Kabelfehler Zähler 1?
    INC PAR_3            'Anzahl Leitungs- bzw. Kabelfehler bis
                        'jetzt...
    error = 1            'Fehlerindikator setzen
  ENDIF
  IF (PAR_2 AND 1000000h = 1000000h) THEN 'Korrelationsfehler
                        'Zähler 1?
    INC PAR_4            'Anzahl Korrelationsfehler bis jetzt...
    error = 1            'Fehlerindikator setzen
  ENDIF
  PAR_5 = SHIFT_RIGHT(PAR_2 AND 10h,4)
                        'akt. Zustand CLR-Eingang
  PAR_6 = SHIFT_RIGHT(PAR_2 AND 10000h,16)
                        'akt. Zustand A-Eingang
  PAR_7 = SHIFT_RIGHT(PAR_2 AND 20000h,17)
                        'akt. Zustand B-Eingang
```

## CNT\_INPUTMODE

**CNT\_INPUTMODE** stellt die Funktion des CLR/LATCH-Eingangs eines oder mehrerer Zähler ein.

### Syntax

```
#INCLUDE ADWGCNT.INC      'nur für ADwin-Gold
#include ADWL16.INC       'nur für ADwin-light-16

CNT_INPUTMODE (pattern)
```

### Parameter

**pattern**      Bitmuster LONG  
                  Bit = 0: CLR-Modus einstellen  
                  Bit = 1: LATCH-Modus einstellen

Bit-Nr.	31...4	3	2	1	0
Zähler-Nr.	–	4 <sup>a</sup>	3 <sup>a</sup>	2	1

a. nur für *ADwin-Gold* CO1

### Bemerkungen

Verwenden Sie diesen Befehl möglichst nur bei gesperrtem Zähler.

### Siehe auch

CNT\_CLEAR, CNT\_CLEARENABLE, CNT\_ENABLE,  
 CNT\_GETSTATUS, CNT\_LATCH, CNT\_MODE, CNT\_READ,  
 CNT\_READLATCH, CNT\_READFLATCH, CNT\_RESETSTATUS,  
 CNT\_SET, CNT\_SE\_DIFF

### Gültig für

Gold-CO1, L16-DIO1, L16-DIO2

### Beispiel

```
#INCLUDE ADWGCNT.INC      'Nur für ADwin-Gold
#INCLUDE ADWL16.INC       'Nur für ADwin-light-16

DIM old_1, new_1 AS LONG 'Variablen...
DIM old_2, new_2 AS LONG 'Dimensionieren

INIT:
  old_1 = 0                'Variablen ...
  old_2 = 0                'initialisieren
  CNT_SE_DIFF(11b)        'Nur für ADwin-Gold:
                           'Alle Zähleringänge differentiell
  CNT_MODE(0)              'Alle Zähler auf externen Takteingang
  CNT_SET(11b)             'Zähler 1+2 mit Takt(CLK)- und
                           'Richtungs(DIR)-Eingang
  CNT_INPUTMODE(0)        'Funktionalität CLR/LATCH festlegen:
                           'Alle als CLR-Eingang
  CNT_CLEARENABLE(11b)    'Schaltet die CLR-Funktion beider
                           'Zähler frei (nur light-16)
  CNT_CLEAR(11b)          'Zähler 1+2 auf 0 zurücksetzen
  CNT_ENABLE(11b)         'Zähler 1+2 starten

EVENT:
  CNT_LATCH(11b)          'Zähler 1+2 gleichzeitig latchen
  new_1 = CNT_READLATCH(1) 'Latch A Zähler 1 und...
  new_2 = CNT_READLATCH(2) 'Latch A Zähler 2 auslesen.
  PAR_1 = new_1 - old_1    'Differenz bilden (f = Impulse / Zeit)
  PAR_2 = new_2 - old_2   '-"-
  old_1 = new_1           'Neuen Zählerstand als alten speichern
  old_2 = new_2           '-"-
```

## CNT\_LATCH

**CNT\_LATCH** überträgt den aktuellen Zählerstand eines oder mehrerer Zähler in das zugehörige Latch A, je nach Bitmuster in **pattern**.

### Syntax

```
#INCLUDE ADWGCNT.INC      'nur für ADwin-Gold
#include ADWL16.INC       'nur für ADwin-light-16

CNT_LATCH(pattern)
```

### Parameter

**pattern**      Bitmuster LONG  
 Bit = 0: keine Funktion  
 Bit = 1: Zählerstand in Latch A übertragen

Bit-Nr.	31...4	3	2	1	0
Zähler-Nr.	–	4 <sup>a</sup>	3 <sup>a</sup>	2 <sup>b</sup>	1

a. nur für *ADwin-Gold* CO1

b. bei *ADwin-light-16*: nicht für Erweiterung CO1

### Bemerkungen

Nach Ausführung des Befehls wird das Bitmuster automatisch auf 0 (Null) zurückgesetzt.

Das Latch A wird mit **CNT\_READLATCH** in eine Variable ausgelesen.

### Siehe auch

CNT\_CLEAR, CNT\_CLEARENABLE, CNT\_ENABLE,  
 CNT\_GETSTATUS, CNT\_INPUTMODE, CNT\_MODE, CNT\_READ,  
 CNT\_READLATCH, CNT\_READFLATCH, CNT\_RESETSTATUS,  
 CNT\_SET, CNT\_SE\_DIFF

### Gültig für

Gold-CO1, L16, L16-CO1, L16-DIO1, L16-DIO2

### Beispiel

```
#INCLUDE ADWGCNT.INC      'Nur für ADwin-Gold
#include ADWL16.INC       'Nur für ADwin-light-16

DIM old_1, new_1 AS LONG 'Variablen...
DIM old_2, new_2 AS LONG 'Dimensionieren

INIT:
    old_1 = 0              'Variablen...
    old_2 = 0              'initialisieren
    CNT_SE_DIFF(11b)      'Nur für ADwin-Gold:
                           'Alle Zähleringänge differentiell
    CNT_MODE(0)            'Alle Zähler auf externen Takteingang
    CNT_SET(11b)           'Zähler 1+2 mit Takt(CLK)- und
                           'Richtungs(DIR)-Eingang
    CNT_INPUTMODE(0)      'Funktionalität CLR/LATCH festlegen:
                           'Alle als CLR-Eingang
    CNT_CLEARENABLE(11b)  'Schaltet die CLR-Funktion beider
                           'Zähler frei (nur light-16)
    CNT_CLEAR(11b)        'Zähler 1+2 auf 0 zurücksetzen
    CNT_ENABLE(11b)       'Zähler 1+2 starten

EVENT:
    CNT_LATCH(11b)        'Zähler 1+2 gleichzeitig latchen
    new_1 = CNT_READLATCH(1) 'Latch A Zähler 1 und...
    new_2 = CNT_READLATCH(2) 'Latch A Zähler 2 auslesen.
    PAR_1 = new_1 - old_1 'Differenz bilden (f = Impulse / Zeit)
    PAR_2 = new_2 - old_2 '-'-
    old_1 = new_1          'Neuen Zählerstand als alten speichern
    old_2 = new_2          '-'-
```

## CNT\_MODE

**CNT\_MODE** definiert die Betriebsart aller Zähler, d.h. welchen Takteingang diese nutzen, gemäß dem Bitmuster in **pattern**.

### Syntax

```
#INCLUDE ADWGCNT.INC      'nur für ADwin-Gold
#include ADWL16.INC       'nur für ADwin-light-16

CNT_MODE(pattern)
```

### Parameter

**pattern**      Bitmuster LONG

Bit = 0: externer Takteingang für Ereigniszählung (CLK/DIR oder A/B)

Bit = 1: interner Takteingang für Pulsbreitenmessung (5MHz oder 20MHz)

Bit-Nr.	31...4	3	2	1	0
Zähler-Nr.	–	4 <sup>a</sup>	3 <sup>a</sup>	2	1

a. nur für *ADwin-Gold* CO1

### Bemerkungen

Bestimmen Sie mit der Anweisung **CNT\_SET** den Modus des gewählten Takteingangs.

Verwenden Sie diesen Befehl möglichst nur bei gesperrtem Zähler.

### Siehe auch

CNT\_CLEAR, CNT\_CLEARENABLE, CNT\_ENABLE, CNT\_GETSTATUS, CNT\_INPUTMODE, CNT\_LATCH, CNT\_READ, CNT\_READLATCH, CNT\_READFLATCH, CNT\_RESETSTATUS, CNT\_SET, CNT\_SE\_DIFF

### Gültig für

Gold-CO1, L16-DIO1, L16-DIO2

### Beispiel

```
#INCLUDE ADWGCNT.INC      'Nur für ADwin-Gold
#include ADWL16.INC       'Nur für ADwin-light-16

DIM old_1, new_1 AS LONG 'Variablen...
DIM old_2, new_2 AS LONG 'dimensionieren

INIT:
    old_1 = 0              'Variablen...
    old_2 = 0              'initialisieren
    CNT_SE_DIFF(11b)      'Nur für ADwin-Gold:
                           'Alle Zähleringänge differentiell
    CNT_MODE(0)            'Alle Zähler auf externen Takteingang
    CNT_SET(1)             'Zähler 1 mit 20MHz
    CNT_INPUTMODE(0)       'Funktionalität CLR/LATCH festlegen:
                           'Alle als CLR-Eingang
    CNT_CLEARENABLE(11b)  'Schaltet die CLR-Funktion beider
                           'Zähler frei (nur light-16)
    CNT_CLEAR(11b)        'Zähler 1+2 auf 0 zurücksetzen
    CNT_ENABLE(11b)       'Zähler 1+2 starten

EVENT:
    CNT_LATCH(11b)        'Zähler 1+2 gleichzeitig latches
    new_1 = CNT_READLATCH(1) 'Latch A Zähler 1 und...
    new_2 = CNT_READLATCH(2) 'Latch A Zähler 2 auslesen.
    PAR_1 = new_1 - old_1 'Differenz bilden (f = Impulse / Zeit)
    PAR_2 = new_2 - old_2 '-
    old_1 = new_1         'Neuen Zählerstand als alten speichern
    old_2 = new_2         '-'
```

## CNT\_READ

**CNT\_READ** überträgt einen aktuellen Zählerstand in das zugehörige Latch A und gibt ihn als Rückgabewert zurück.

### Syntax

```
#INCLUDE ADWGCNT.INC      'nur für ADwin-Gold
#include ADWL16.INC       'nur für ADwin-light-16

ret_val = CNT_READ (CounterNo)
```

### Parameter

CounterNo	Zählernummer (L16, L16-DIO1: 1...2, L16-CO1: 1, Gold-CO1: 1...4)	LONG
ret_val	Zählerstand	LONG

### Bemerkungen

Verwenden Sie den Rückgabewert in Berechnungen (z. B. Differenzen oder Zählrichtung) nur mit Variablen vom Typ **LONG**.

### Siehe auch

CNT\_CLEAR, CNT\_CLEARENABLE, CNT\_ENABLE,  
CNT\_GETSTATUS, CNT\_INPUTMODE, CNT\_LATCH, CNT\_MODE,  
CNT\_READLATCH, CNT\_READFLATCH, CNT\_RESETSTATUS,  
CNT\_SET, CNT\_SE\_DIFF

### Gültig für

Gold-CO1, L16, L16-CO1, L16-DIO1, L16-DIO2



### Beispiel

```
#INCLUDE ADWGCNT.INC      'Nur für ADwin-Gold
#include ADWL16.INC       'Nur für ADwin-light-16

DIM old_1, new_1 AS LONG 'Variablen...
DIM old_2, new_2 AS LONG 'Dimensionieren

INIT:
    old_1 = 0              'Variablen...
    old_2 = 0              'initialisieren
    CNT_SE_DIFF(11b)      'Nur für ADwin-Gold:
                           'Alle Zähleringänge differentiell
    CNT_MODE(0)            'Alle Zähler auf externen Takteingang
    CNT_SET(11b)           'Zähler 1+2 mit Takt(CLK)- und
                           'Richtungs(DIR)-Eingang
    CNT_INPUTMODE(0)       'Funktionalität CLR/LATCH festlegen:
                           'Alle als CLR-Eingang
    CNT_CLEARENABLE(11b)   'Schaltet die CLR-Funktion beider
                           'Zähler frei (nur light-16)
    CNT_CLEAR(11b)         'Zähler 1+2 auf 0 zurücksetzen
    CNT_ENABLE(11b)        'Zähler 1+2 starten

EVENT:
    new_1 = CNT_READ(1)    'Zähler 1 latchen und anschl. Latch A
                           'auslesen
    new_2 = CNT_READ(2)    'Zähler 2 latchen und anschl. Latch A
                           'auslesen
    PAR_1 = new_1 - old_1  'Differenz bilden (f = Impulse / Zeit)
    PAR_2 = new_2 - old_2 ' "-"
    old_1 = new_1          'Neuen Zählerstand als alten speichern
    old_2 = new_2          ' "-"
```

## CNT\_READLATCH

**CNT\_READLATCH** gibt den Wert aus dem Latch A eines Zählers als Rückgabewert zurück.

### Syntax

```
#INCLUDE ADWGCNT.INC      'nur für ADwin-Gold
#include ADWL16.INC       'nur für ADwin-light-16

ret_val = CNT_READLATCH(CounterNo)
```

### Parameter

CounterNo	Zählernummer (L16, L16-DIO1: 1...2, L16-CO1: 1, Gold-CO1: 1...4)	LONG
ret_val	Inhalt des Latch A des Zählers	LONG

### Bemerkungen

Verwenden Sie den Rückgabewert in Berechnungen (z. B. Differenzen oder Zählrichtung) nur mit Variablen vom Typ **LONG**.

Der Zeitpunkt, zu dem der Zählerstand ins Latch übertragen wurde, hängt von der Einstellung mit **CNT\_MODE** ab:

- Externer Takteingang (**CNT\_MODE**-Bit = 0): Nur der Befehl **CNT\_LATCH** überträgt den Zählerstand (in Latch A).
- Interner Takteingang (**CNT\_MODE**-Bit = 1): Jede Flanke des extern angelegten Messsignals löst einen Latch-Vorgang aus.

In Latch A wird der Zählerstand bei der positiven Flanke des Eingangssignals zwischengespeichert, während in Latch B (siehe **CNT\_READFLATCH**) der Zählerstand bei der negativen Flanke des Eingangssignals gespeichert wird.

### Siehe auch

CNT\_CLEAR, CNT\_CLEARENABLE, CNT\_ENABLE,  
CNT\_GETSTATUS, CNT\_INPUTMODE, CNT\_LATCH, CNT\_MODE,  
CNT\_READ, CNT\_READFLATCH, CNT\_RESETSTATUS, CNT\_SET,  
CNT\_SE\_DIFF

### Gültig für

Gold-CO1, L16, L16-CO1, L16-DIO1, L16-DIO2

### Beispiel

```
#INCLUDE ADWGCNT.INC      'Nur für ADwin-Gold
#include ADWL16.INC       'Nur für ADwin-light-16
DIM rise, rise_old, fall, fall_old AS LONG
#define high PAR_1
#define low PAR_2
#define T PAR_9
#define f PAR_10

INIT:
    rise_old = 0          'Variablen...
    fall_old = 0          'initialisieren
    CNT_SE_DIFF(11b)      'Nur für ADwin-Gold:
                           'Alle Zählerringänge differentiell
    CNT_MODE(11b)          'Zähler 1+2 auf internen Takteingang
    CNT_SET(0)             'Alle Zähler mit 20 MHz internem
                           'Referenztakt
    CNT_INPUTMODE(11b)     'Funktionalität CLR/LATCH festlegen:
                           'Zähler 1+2 als LATCH-Eingang
    CNT_CLEARENABLE(0)     'Sperrt die CLR-Funktion beider Zähler
                           '(nur light-16)
    CNT_CLEAR(11b)         'Zähler 1+2 auf 0 zurücksetzen
    CNT_ENABLE(1)          'Zähler 1 starten

EVENT:
    rise = CNT_READLATCH(1) 'Latch A Zähler 1 auslesen
    fall = CNT_READFLATCH(1) 'Latch B Zähler 1 auslesen
    IF (rise <> rise_old) THEN 'Steigende Flanke detektiert?
        T = rise - rise_old 'Periodendauer in Nanosekunden
        f = 1E9 / T         'Frequenz in Hertz
        IF (fall <> fall_old) THEN 'Fallende Flanke detektiert?
            high = (fall - rise) * 25 'Impulsdauer in Nanosekunden
            low = (rise - fall_old) * 25 'Pausendauer in Nanosekunden
        ELSE
            'Keine fallende Flanke detektiert
            high = (fall - rise_old) * 25 'Impulsdauer in Nanosekunden
            low = (rise - fall) * 25 'Pausendauer in Nanosekunden
        ENDIF
    ENDIF
    rise_old = rise        'Latch-Inhalt speichern
    fall_old = fall        'Latch-Inhalt speichern
```

## CNT\_READFLATCH

**CNT\_READFLATCH** gibt den Wert aus dem Latch B eines Zählers als Rückgabewert zurück (nur im Modus Pulsbreitenmessung einsetzbar).

### Syntax

```
#INCLUDE ADWGCNT.INC      'nur für ADwin-Gold
#include ADWL16.INC       'nur für ADwin-light-16

ret_val = CNT_READFLATCH(CounterNo)
```

### Parameter

CounterNo	Zählernummer (L16-DIO1: 1...2, Gold-CO1: 1...4)	LONG
ret_val	Inhalt des Latch B des Zählers	LONG

### Bemerkungen

Verwenden Sie den Rückgabewert in Berechnungen (z. B. Differenzen oder Zählrichtung) nur mit Variablen vom Typ **LONG**.

Der Zeitpunkt, zu dem der Zählerstand ins Latch übertragen wurde, hängt von der Einstellung mit **CNT\_MODE** ab:

- Externer Takteingang (**CNT\_MODE**-Bit = 0): Nur der Befehl **CNT\_LATCH** überträgt den Zählerstand (in Latch A).
- Interner Takteingang (**CNT\_MODE**-Bit = 1): Jede Flanke des extern angelegten Messsignals löst einen Latch-Vorgang aus.

In Latch A wird der Zählerstand bei der positiven Flanke des Eingangssignals zwischengespeichert, während in Latch B (siehe **CNT\_READFLATCH**) der Zählerstand bei der negativen Flanke des Eingangssignals gespeichert wird.

### Siehe auch

CNT\_CLEAR, CNT\_CLEARENABLE, CNT\_ENABLE, CNT\_GETSTATUS, CNT\_INPUTMODE, CNT\_LATCH, CNT\_MODE, CNT\_READ, CNT\_READLATCH, CNT\_RESETSTATUS, CNT\_SET, CNT\_SE\_DIFF

### Gültig für

Gold-CO1, L16-DIO1, L16-DIO2

### Beispiel

```
#INCLUDE ADWGCNT.INC      'Nur für ADwin-Gold
#INCLUDE ADWL16.INC      'Nur für ADwin-light-16
DIM rise, rise_old, fall, fall_old AS LONG
#DEFINE high PAR_1
#DEFINE low PAR_2
#DEFINE T PAR_9
#DEFINE f PAR_10

INIT:
    rise_old = 0          'Variablen...
    fall_old = 0          'initialisieren
    CNT_SE_DIFF(11b)      'Nur für ADwin-Gold:
                           'Alle Zählerringänge differentiell
    CNT_MODE(11b)         'Zähler 1+2 auf internen Takteingang
    CNT_SET(0)            'Alle Zähler mit 20 MHz internem
                           'Referenztakt
    CNT_INPUTMODE(11b)    'Funktionalität CLR/LATCH festlegen:
                           'Zähler 1+2 als LATCH-Eingang
    CNT_CLEARENABLE(0)    'Sperrt die CLR-Funktion beider Zähler
                           '(nur light-16)
    CNT_CLEAR(11b)        'Zähler 1+2 auf 0 zurücksetzen
    CNT_ENABLE(1)         'Zähler 1 starten

EVENT:
    rise = CNT_READLATCH(1) 'Latch A Zähler 1 auslesen
    fall = CNT_READFLATCH(1) 'Latch B Zähler 1 auslesen
    IF (rise <> rise_old) THEN 'Steigende Flanke detektiert?
        T = rise - rise_old 'Periodendauer in Nanosekunden
        f = 1E9 / T         'Frequenz in Hertz
        IF (fall <> fall_old) THEN 'Fallende Flanke detektiert?
            high = (fall - rise) * 25 'Impulsdauer in Nanosekunden
            low = (rise - fall_old) * 25 'Pausendauer in Nanosekunden
        ELSE
            'Keine fallende Flanke detektiert
            high = (fall - rise_old) * 25 'Impulsdauer in Nanosekunden
            low = (rise - fall) * 25 'Pausendauer in Nanosekunden
        ENDIF
    ENDIF
    rise_old = rise        'Latch-Inhalt speichern
    fall_old = fall        'Latch-Inhalt speichern
```

## CNT\_RESETSTATUS

Nur ADwin-Gold: **CNT\_RESETSTATUS** löscht das Statusregister aller vier 32 Bit-Zähler.

### Syntax

```
#INCLUDE ADWGCNT.INC      'nur für ADwin-Gold
CNT_RESETSTATUS()
```

### Bemerkungen

Das Statusregister wird mit der Anweisung **CNT\_GETSTATUS** gelesen.

### Siehe auch

CNT\_CLEAR, CNT\_ENABLE, CNT\_GETSTATUS,  
CNT\_INPUTMODE, CNT\_LATCH, CNT\_MODE, CNT\_READ,  
CNT\_READLATCH, CNT\_READFLATCH, CNT\_SET, CNT\_SE\_DIFF

### Gültig für

Gold-CO1

### Beispiel

```
#INCLUDE ADWGCNT.INC      'Nur ADwin-Gold

DIM error AS LONG

DIM old_1, new_1 AS LONG 'Variablen...
DIM old_2, new_2 AS LONG 'dimensionieren

INIT:
  CNT_ENABLE(0)           'Alle Zähler stoppen
  CNT_SE_DIFF(11b)        'Alle Zähleringänge differentiell
  CNT_CLEAR(1111b)        'Alle Zähler löschen
  CNT_SE_DIFF(11b)        'Alle Zähler auf diff. Eingänge
  CNT_MODE(0)             'Externer Takteingang
  CNT_SET(0)              'Vier-Flanken-Auswertung
  CNT_INPUTMODE(0)        'CLR-Eingang freischalten
  CNT_ENABLE(1111b)       'Alle Zähler starten
  old_1 = 0               'Variablen...
  old_2 = 0               'initialisieren

  error = 0              'Fehlerindikator zurücksetzen

EVENT:
  PAR_1 = CNT_READ(1)     'Zähler 1 auslesen
  PAR_2 = CNT_GETSTATUS(1) AND 0FFFF00F0h 'Statusregister
                                'auslesen und maskieren
  IF (PAR_2 AND 2000000h = 2000000h) THEN 'Leitungs- bzw.
                                'Kabelfehler Zähler 1?
    INC PAR_3              'Anzahl Leitungs- bzw. Kabelfehler bis
                                'jetzt...
    error = 1              'Fehlerindikator setzen
  ENDIF
  IF (PAR_2 AND 1000000h = 1000000h) THEN 'Korrelationsfehler
                                'am Zähler 1?
    INC PAR_4              'Anzahl Korrelationsfehler bis jetzt
    error = 1              'Fehlerindikator setzen
  ENDIF
  CNT_RESETSTATUS()       'Leitungs- und Korrelationsfehler-
                                'Bits löschen
  PAR_5 = SHIFT_RIGHT(PAR_2 AND 10h,4) 'Zustand CLR-Eingg
  PAR_6 = SHIFT_RIGHT(PAR_2 AND 10000h,16) 'Zustand Eingang A
  PAR_7 = SHIFT_RIGHT(PAR_2 AND 20000h,17) 'Zustand Eingang B
```

## CNT\_SE\_DIFF

Nur *ADwin-Gold*: **CNT\_SE\_DIFF** stellt die Eingänge von jeweils 2 Zählern auf den Betriebsmodus single-ended oder differentiell ein.

### Syntax

```
#INCLUDE ADWGCNT.INC      'nur für ADwin-Gold
CNT_SE_DIFF(pattern)
```

### Parameter

**pattern**      Bitmuster zur Auswahl der Zählerpaare (siehe Tabelle) und des Betriebsmodus der Eingänge: LONG

Bit = 0: Betriebsmodus single-ended  
 Bit = 1: Betriebsmodus differentiell

Bit-Nr. in <b>pattern</b>	31 ... 2	1	0
Eingang zum Zähler Nr.	–	3 + 4	1 + 2

### Bemerkungen

Nach dem Start des Systems *ADwin-Gold* ist der Betriebsmodus undefiniert; stellen Sie also in jedem Fall an allen Zählereingängen den gewünschten Betriebsmodus ein.

### Siehe auch

CNT\_CLEAR, CNT\_ENABLE, CNT\_GETSTATUS,  
 CNT\_INPUTMODE, CNT\_LATCH, CNT\_MODE, CNT\_READ,  
 CNT\_READLATCH, CNT\_READFLATCH, CNT\_RESETSTATUS,  
 CNT\_SET

### Gültig für

Gold-CO1



### Beispiel

```
#INCLUDE ADWGCNT.INC      'nur ADwin-Gold

DIM error AS LONG          'Variablen...
DIM old_1, new_1 AS LONG 'dimensionieren
DIM old_2, new_2 AS LONG

INIT:
  CNT_ENABLE(0)             'Alle Zähler stoppen
  CNT_SE_DIFF(11b)          'Alle Zähleringänge differentiell
  CNT_CLEAR(1111b)          'Alle Zähler löschen
  CNT_SE_DIFF(11b)          'Alle Zähler auf diff. Eingänge
  CNT_MODE(0)               'Externer Takteingang
  CNT_SET(0)                'Vier-Flanken-Auswertung
  CNT_INPUTMODE(0)          'CLR-Eingang freischalten
  CNT_ENABLE(1111b)         'Alle Zähler starten
  old_1 = 0                 'Variablen...
  old_2 = 0                 'initialisieren

  error = 0                 'Fehlerindikator zurücksetzen

EVENT:
  PAR_1 = CNT_READ(1)        'Zähler 1 auslesen
  PAR_2 = CNT_GETSTATUS(1) AND 0FFFF00F0h 'Statusregister
                                     'auslesen und maskieren
  IF (PAR_2 AND 2000000h = 2000000h) THEN 'Leitungs- bzw.
                                     'Kabelfehler Zähler 1?
    INC PAR_3                 'Anzahl Leitungs- bzw. Kabelfehler bis
                                     'jetzt...
    error = 1                 'Fehlerindikator setzen
  ENDIF
  IF (PAR_2 AND 1000000h = 1000000h) THEN 'Korrelationsfehler
                                     'am Zähler 1?
    INC PAR_4                 'Anzahl Korrelationsfehler bis jetzt
    error = 1                 'Fehlerindikator setzen
  ENDIF
  CNT_RESETSTATUS()          'Leitungs- und Korrelationsfehler-
                                     'Bits löschen
  PAR_5 = SHIFT_RIGHT(PAR_2 AND 10h,4) 'Zustand CLR-Eingg
  PAR_6 = SHIFT_RIGHT(PAR_2 AND 10000h,16) 'Zustand Eingang A
  PAR_7 = SHIFT_RIGHT(PAR_2 AND 20000h,17) 'Zustand Eingang B
```

## CNT\_SET

**CNT\_SET** definiert den Betriebsmodus für alle Zähler (in Abhängigkeit von **CNT\_MODE**) gemäß dem Bitmuster in **pattern**.

### Syntax

```
#INCLUDE ADWGCNT.INC      'nur für ADwin-Gold
#include ADWL16.INC       'nur für ADwin-light-16

CNT_SET(pattern)
```

### Parameter

**pattern**      Bitmuster, Bit-Bedeutung siehe Tabelle.      LONG

Bit-Wert in <b>pattern</b>	externer Takteingang Bit = 0 in <b>CNT_MODE</b>	interner Takteingang Bit = 1 in <b>CNT_MODE</b>
Bit = 0	4-Flankenauswertung	Referenztakt 20 MHz
Bit = 1	Takt- und Richtungseingang	Referenztakt 5 MHz

Bit-Nr.	31...4	3	2	1	0
Zähler-Nr.	–	4 <sup>a</sup>	3 <sup>a</sup>	2	1

a. nur für *ADwin-Gold* CO1

### Bemerkungen

Verwenden Sie diesen Befehl möglichst nur bei gesperrtem Zähler.

### Siehe auch

CNT\_CLEAR, CNT\_CLEARENABLE, CNT\_ENABLE,  
CNT\_GETSTATUS, CNT\_INPUTMODE, CNT\_LATCH, CNT\_MODE,  
CNT\_READ, CNT\_READLATCH, CNT\_READFLATCH,  
CNT\_RESETSTATUS, CNT\_SE\_DIFF

### Gültig für

Gold-CO1, L16-DIO1, L16-DIO2

### Beispiel

```
#INCLUDE ADWGCNT.INC      'Nur für ADwin-Gold
#INCLUDE ADWL16.INC       'Nur für ADwin-light-16

INIT:
  CNT_SE_DIFF(11b)        'Nur für ADwin-Gold:
                           'Alle Zähleringänge differentiell
  CNT_MODE(0)             'Alle Zähler auf externen Takteingang
  CNT_SET(1,1100b)        'Zähler 3+4 (nur Gold) auf Takt-
                           '/Richtungsauswertung, Zähler 1+2 mit
                           'Vierflankenauswertung
  CNT_CLEAR(1,1100b)      'Zähler 3+4 (nur Gold) löschen
  CNT_ENABLE(1,1100b)     'Zähler 3+4 (nur Gold) aktivieren,
                           '1+2 deaktivieren
```

## CONF\_DIO\_E

**CONF\_DIO\_E** konfiguriert die digitalen Kanäle in Gruppen zu je 8 als Ein- oder Ausgänge.

### Syntax

```
#INCLUDE ADWL16.INC
```

```
CONF_DIO_E(pattern)
```

### Parameter

**pattern** Bitmuster, das die digitalen Kanäle als LONG  
 Ein- oder Ausgang konfiguriert:  
 Bit=0: Kanäle als Eingänge  
 Bit=1: Kanäle als Ausgänge

Bitnr. in <b>val</b>	15...4	3	2	1	0
Kanäle	–	DIO31	DIO23	DIO15	DIO07
		...	...	...	...
		DIO24	DIO16	DIO08	DIO00

### Bemerkungen

Nach dem Einschalten sind alle digitalen I/O-Leitungen als Eingänge konfiguriert.

### Siehe auch

DIGIN\_WORD1\_E, DIGIN\_WORD2\_E, DIGOUT\_RESET1\_E,  
 DIGOUT\_RESET2\_E, DIGOUT\_SET1\_E, DIGOUT\_SET2\_E,  
 DIGOUT\_WORD1\_E, DIGOUT\_WORD2\_E

### Gültig für

L16-DIO1, L16-DIO2

### Beispiel

```
#INCLUDE ADWL16.INC
```

```
INIT:
CONF_DIO_E(1100b) 'Konfiguriert DIOs 15:00 als Eingänge
                  'und DIOs 31:16 als Ausgänge
```

## DIGIN\_WORD1\_E

**DIGIN\_WORD1\_E** gibt die Werte der digitalen Eingänge 0...15 auf einmal zurück.

### Syntax

```
#INCLUDE ADWL16.INC

ret_val = DIGIN_WORD1_E()
```

### Parameter

**ret\_val** Bitmuster, das den TTL-Pegeln an den digitalen Eingängen entspricht (Zuordnung s.u.).  
 1: TTL-Pegel high liegt an  
 0: TTL-Pegel low liegt an

LONG

Bitnummer in <b>ret_val</b>	31 ... 16	15	14	...	1	0
Eingang Nr.	–	DIO15	DIO14	...	DIO01	DIO00

### Bemerkungen

Wenn Sie Kanäle als Ausgang konfiguriert haben, so wird der Inhalt des Ausgangsregisters an diesen Bits zurückgelesen.

### Siehe auch

CONF\_DIO\_E, DIGIN\_WORD2\_E, DIGOUT\_RESET1\_E, DIGOUT\_RESET2\_E, DIGOUT\_SET1\_E, DIGOUT\_SET2\_E, DIGOUT\_WORD1\_E, DIGOUT\_WORD2\_E

### Gültig für

L16-DIO1, L16-DIO2

**Beispiel**

```
#INCLUDE ADWL16.INC
```

```
INIT:
```

```
    CONF_DIO_E(1100b)      'Konfiguriert DIOs 15:00 als Eingänge  
                           'und DIOs 31:16 als Ausgänge
```

```
EVENT:
```

```
    PAR_1 = DIGIN_WORD1_E() 'Unteres Wort (Bits 15:00) einlesen
```

## DIGIN\_WORD2\_E

**DIGIN\_WORD2\_E** gibt die Werte der digitalen Eingänge 16...31 auf einmal zurück.

### Syntax

```
#INCLUDE ADWL16.INC

ret_val = DIGIN_WORD2_E()
```

### Parameter

**ret\_val** Bitmuster, das den TTL-Pegeln an den digitalen Eingängen entspricht (Zuordnung s.u.).  
 1: TTL-Pegel high liegt an  
 0: TTL-Pegel low liegt an

LONG

Bitnummer in <b>ret_val</b>	31 ... 16	15	14	...	1	0
Eingang Nr.	–	DIO31	DIO30	...	DIO17	DIO16

### Bemerkungen

Wenn Sie Kanäle als Ausgang konfiguriert haben, so wird der Inhalt des Ausgangsregisters an diesen Bits zurückgelesen.

### Siehe auch

CONF\_DIO\_E, DIGIN\_WORD1\_E, DIGOUT\_RESET1\_E, DIGOUT\_RESET2\_E, DIGOUT\_SET1\_E, DIGOUT\_SET2\_E, DIGOUT\_WORD1\_E, DIGOUT\_WORD2\_E

### Gültig für

L16-DIO1, L16-DIO2

**Beispiel**

```
#INCLUDE ADWL16.INC
```

**INIT:**

```
CONF_DIO_E(0)           'Konfiguriert DIOs 31:00 als Eingänge  
                        ' (auch power-up-Zustand!)
```

**EVENT:**

```
PAR_1 = DIGIN_WORD1_E() 'Unteres Wort (Bits 15:00) einlesen  
PAR_2 = DIGIN_WORD2_E() 'Oberes Wort (Bits 31:16) einlesen
```



## DIGIN\_LONG\_E

**DIGIN\_LONG\_E** gibt die Werte der digitalen Eingänge 0...31 auf einmal zurück.

### Syntax

```
#INCLUDE ADWL16.INC

ret_val = DIGIN_LONG_E()
```

### Parameter

**ret\_val** Bitmuster, das den TTL-Pegeln an den digitalen Eingängen entspricht (Zuordnung s.u.).  
 1: TTL-Pegel high liegt an  
 0: TTL-Pegel low liegt an

LONG

Bitnummer in <code>ret_val</code>	31	30	...	1	0
Eingang Nr.	DIO31	DIO30	...	DIO01	DIO00

### Bemerkungen

Wenn Sie Kanäle als Ausgang konfiguriert haben, so wird der Inhalt des Ausgangsregisters an diesen Bits zurückgelesen.

### Siehe auch

CONF\_DIO\_E, DIGIN\_WORD1\_E, DIGIN\_WORD2\_E,  
 DIGOUT\_RESET1\_E, DIGOUT\_RESET2\_E, DIGOUT\_SET1\_E,  
 DIGOUT\_SET2\_E, DIGOUT\_WORD1\_E, DIGOUT\_WORD2\_E,  
 DIGOUT\_LONG\_E

### Gültig für

L16-DIO2

**Beispiel**

```
#INCLUDE ADWL16.INC
```

**INIT:**

```
CONF_DIO_E(0)           'Konfiguriert DIOs 31:00 als Eingänge  
                        '(auch power-up-Zustand!)
```

**EVENT:**

```
PAR_1 = DIGIN_LONG_E() 'Bits 31:00 einlesen
```

## DIGOUT\_RESET1\_E

**DIGOUT\_RESET1\_E** setzt bestimmte der digitalen Ausgänge 0...15 auf TTL-Pegel low.

### Syntax

```
#INCLUDE ADWL16.INC
DIGOUT_RESET1_E(pattern)
```

### Parameter

**pattern** Bitmuster, um bestimmte Ausgänge zu setzen: LONG  
 Bit = 1: Setzen auf TTL-Pegel low  
 Bit = 0: Kein Einfluss

Bitnummer in <b>pattern</b>	31 ... 16	15	14	...	1	0
Eingang Nr.	–	DIO15	DIO14	...	DIO01	DIO00

### Siehe auch

CONF\_DIO\_E, DIGIN\_WORD1\_E, DIGIN\_WORD2\_E,  
 DIGOUT\_RESET2\_E, DIGOUT\_SET1\_E, DIGOUT\_SET2\_E,  
 DIGOUT\_WORD1\_E, DIGOUT\_WORD2\_E

### Gültig für

L16-DIO1, L16-DIO2

**Beispiel**

```
#INCLUDE ADWL16.INC
```

**INIT:**

```
CONF_DIO_E(11b)      'Konfiguriert DIOs 15:00 als Ausgänge  
                     'und DIOs 31:16 als Eingänge
```

**INIT:**

```
PAR_1 = 5555h         'Alle ungeraden Bits des unteren Worts  
                     'bei der Ausgabe löschen  
DIGOUT_WORD1_E(0FFFFh) 'DIO-Bits 15:00 ausgeben
```

**EVENT:**

```
DIGOUT_RESET1_E(PAR_1) 'DIO-Bits entsprechend PAR_1 löschen  
PAR_1 = PAR_1 XOR 0FFFFh 'Output-Word invertieren  
DIGOUT_WORD1_E(PAR_1) 'DIO-Bits 15:00 ausgeben
```

## DIGOUT\_RESET2\_E

**DIGOUT\_RESET2\_E** setzt bestimmte der digitalen Ausgänge 16...31 auf TTL-Pegel low.

### Syntax

```
#INCLUDE ADWL16.INC
DIGOUT_RESET2_E(pattern)
```

### Parameter

**pattern** Bitmuster, um bestimmte Ausgänge zu setzen: LONG  
 Bit = 1: Setzen auf TTL-Pegel low  
 Bit = 0: Kein Einfluss

Bitnummer in <b>pattern</b>	31 ... 16	15	14	...	1	0
Eingang Nr.	–	DIO31	DIO30	...	DIO17	DIO16

### Siehe auch

CONF\_DIO\_E, DIGIN\_WORD1\_E, DIGIN\_WORD2\_E,  
 DIGOUT\_RESET1\_E, DIGOUT\_SET1\_E, DIGOUT\_SET2\_E,  
 DIGOUT\_WORD1\_E, DIGOUT\_WORD2\_E

### Gültig für

L16-DIO1, L16-DIO2

**Beispiel**

```
#INCLUDE ADWL16.INC
```

**INIT:**

```
CONF_DIO_E(1100b)      'Konfiguriert DIOs 15:00 als Eingänge  
                        'und DIOs 31:16 als Ausgänge
```

**INIT:**

```
PAR_2 = 5555h           'Alle ungeraden Bits des High-Words  
                        'bei der Ausgabe löschen
```

```
DIGOUT_WORD1_E(0FFFFh) 'DIO-Bits 15:00 ausgeben
```

**EVENT:**

```
DIGOUT_RESET2_E(PAR_2) 'DIO-Bits entsprechend PAR_2 löschen
```

```
PAR_2 = PAR_2 XOR 0FFFFh 'Output-Word invertieren
```

```
DIGOUT_WORD2_E(PAR_2) 'DIO-Bits 31:16 ausgeben
```

## DIGOUT\_SET1\_E

**DIGOUT\_SET1\_E** setzt bestimmte der digitalen Ausgänge 0...15 auf TTL-Pegel high.

### Syntax

```
#INCLUDE ADWL16.INC
DIGOUT_SET1_E(pattern)
```

### Parameter

**pattern** Bitmuster, um bestimmte Ausgänge zu setzen:  
 Bit = 1: Setzen auf TTL-Pegel high  
 Bit = 0: Kein Einfluss

LONG

Bitnummer in <b>pattern</b>	31 ... 16	15	14	...	1	0
Eingang Nr.	–	DIO15	DIO14	...	DIO01	DIO00

### Siehe auch

CONF\_DIO\_E, DIGIN\_WORD1\_E, DIGIN\_WORD2\_E,  
 DIGOUT\_RESET1\_E, DIGOUT\_RESET2\_E, DIGOUT\_SET2\_E,  
 DIGOUT\_WORD1\_E, DIGOUT\_WORD2\_E

### Gültig für

L16-DIO1, L16-DIO2

**Beispiel**

```
#INCLUDE ADWL16.INC
```

**INIT:**

```
CONF_DIO_E(1100b)    'Konfiguriert DIOs 15:00 als Ausgänge  
                     'und DIOs 31:16 als Eingänge  
PAR_1 = 0AAAAh       'Alle geraden Bits des unteren Worts  
                     'bei der Ausgabe setzen  
DIGOUT_WORD1_E(0)    'DIO-Bits 15:00 ausgeben
```

**EVENT:**

```
DIGOUT_SET1_E(PAR_1) 'DIO-Bits entsprechend PAR_1 setzen  
PAR_1 = PAR_1 XOR 0FFFFh'Output-Word invertieren  
DIGOUT_WORD1_E(PAR_1)'DIO-Bits 15:00 ausgeben
```



## DIGOUT\_SET2\_E

**DIGOUT\_SET2\_E** setzt bestimmte der digitalen Ausgänge 16...31 auf TTL-Pegel high.

### Syntax

```
#INCLUDE ADWL16.INC
DIGOUT_SET2_E(pattern)
```

### Parameter

**pattern** Bitmuster, um bestimmte Ausgänge zu setzen: LONG  
 Bit = 1: Setzen auf TTL-Pegel high  
 Bit = 0: Kein Einfluss

Bitnummer in <b>pattern</b>	31 ... 16	15	14	...	1	0
Eingang Nr.	–	DIO31	DIO30	...	DIO17	DIO16

### Siehe auch

CONF\_DIO\_E, DIGIN\_WORD1\_E, DIGIN\_WORD2\_E,  
 DIGOUT\_RESET1\_E, DIGOUT\_RESET2\_E, DIGOUT\_SET1\_E,  
 DIGOUT\_WORD1\_E, DIGOUT\_WORD2\_E

### Gültig für

L16-DIO1, L16-DIO2

**Beispiel**

```
#INCLUDE ADWL16.INC
```

**INIT:**

```
CONF_DIO_E(1100b)    'Konfiguriert DIOs 15:00 als Ausgänge  
                     'und DIOs 31:16 als Eingänge  
PAR_1 = 0AAAAh       'Alle geraden Bits des unteren Worts  
                     'bei der Ausgabe setzen  
DIGOUT_WORD2_E(0)    'DIO-Bits 15:00 ausgeben
```

**EVENT:**

```
DIGOUT_SET2_E(PAR_2) 'DIO-Bits entsprechend PAR_1 setzen  
PAR_2 = PAR_2 XOR 0FFFFh'Output-Word invertieren  
DIGOUT_WORD2_E(PAR_2)'DIO-Bits 15:00 ausgeben
```

## DIGOUT\_WORD1\_E

**DIGOUT\_WORD1\_E** setzt mit einem Bitmuster gleichzeitig die digitalen Ausgänge 0...15 auf definierte TTL-Pegel.

### Syntax

```
#INCLUDE ADWL16.INC

DIGOUT_WORD1_E(pattern)
```

### Parameter

<b>pattern</b>	Bitmuster, das den TTL-Pegeln an den digitalen Ausgängen entspricht (Zuordnung s.u.). Bit = 1: Setzen auf TTL-Pegel high Bit = 0: Setzen auf TTL-Pegel low	LONG
----------------	--	------

Bitnummer in <b>pattern</b>	31 ... 16	15	14	...	1	0
Eingang Nr.	–	DIO15	DIO14	...	DIO01	DIO00

### Siehe auch

CONF\_DIO\_E, DIGIN\_WORD1\_E, DIGIN\_WORD2\_E, DIGOUT\_RESET1\_E, DIGOUT\_RESET2\_E, DIGOUT\_SET1\_E, DIGOUT\_SET2\_E, DIGOUT\_WORD2\_E

### Gültig für

L16-DIO1, L16-DIO2

### Beispiel

```
#INCLUDE ADWL16.INC

INIT:
    CONF_DIO_E(0011b)    'Konfiguriert DIOs15:00 als Ausgänge
                        'und DIOs 31:16 als Eingänge
    PAR_1 = 5555h        'Alle ungeraden Bits des unteren Worts
                        'setzen

EVENT:
    DIGOUT_WORD1_E(PAR_1) 'DIO-Bits 15:00 ausgeben
```

## DIGOUT\_WORD2\_E

**DIGOUT\_WORD2\_E** setzt mit einem Bitmuster gleichzeitig die digitalen Ausgänge 16...31 auf definierte TTL-Pegel.

### Syntax

```
#INCLUDE ADWL16.INC
DIGOUT_WORD2_E(pattern)
```

### Parameter

**pattern** Bitmuster, das den TTL-Pegeln an den digitalen Ausgängen entspricht (Zuordnung s.u.). LONG

Bit = 1: Setzen auf TTL-Pegel high  
Bit = 0: Setzen auf TTL-Pegel low

Bitnummer	31 ... 16	15	14	...	1	0
in <b>pattern</b>						
Eingang Nr.	–	DIO31	DIO30	...	DIO17	DIO16

### Siehe auch

CONF\_DIO\_E, DIGIN\_WORD1\_E, DIGIN\_WORD2\_E,  
DIGOUT\_RESET1\_E, DIGOUT\_RESET2\_E, DIGOUT\_SET1\_E,  
DIGOUT\_SET2\_E, DIGOUT\_WORD1\_E

### Gültig für

L16-DIO1, L16-DIO2

### Beispiel

```
#INCLUDE ADWL16.INC

INIT:
  CONF_DIO_E(12)      'Konfiguriert DIOs 15:00 als Eingänge
                      'und DIOs 31:16 als Ausgänge
  PAR_2 = 0AAAAh      'Alle geraden Bits des unteren Worts
                      'setzen

EVENT:
  DIGOUT_WORD2_E(PAR_2) 'DIO-Bits 31:16 ausgeben
```

## DIGOUT\_LONG\_E

**DIGOUT\_LONG\_E** setzt mit einem Bitmuster gleichzeitig die digitalen Ausgänge 0...31 auf definierte TTL-Pegel.

### Syntax

```
#INCLUDE ADWL16.INC
DIGOUT_LONG_E(pattern)
```

### Parameter

**pattern** Bitmuster, das den TTL-Pegeln an den digitalen Ausgängen entspricht:  
 Bit = 1: Setzen auf TTL-Pegel high  
 Bit = 0: Setzen auf TTL-Pegel low

LONG

Bitnummer in <b>pattern</b>	31	30	...	1	0
Eingang Nr.	DIO31	DIO30	...	DIO01	DIO00

### Siehe auch

CONF\_DIO\_E, DIGIN\_WORD1\_E, DIGIN\_WORD2\_E,  
 DIGIN\_LONG\_E, DIGOUT\_RESET1\_E, DIGOUT\_RESET2\_E,  
 DIGOUT\_SET1\_E, DIGOUT\_SET2\_E, DIGOUT\_WORD1\_E,  
 DIGOUT\_WORD2\_E

### Gültig für

L16-DIO2

### Beispiel

```
#INCLUDE ADWL16.INC

INIT:
  CONF_DIO_E(1111b)    'Konfiguriert DIOs 31:00 als Ausgänge
  PAR_2 = 0AAAAAAAh    'Alle geraden Bits setzen

EVENT:
  DIGOUT_LONG_E(PAR_2) 'DIO-Bits 31:00 ausgeben
```

## CAN\_MSG

CAN-Bus: `CAN_MSG[]` ist ein eindimensionales Feld mit 9 Elementen, in dem die Message-Objekte gespeichert sind oder werden.

### Syntax

```
#INCLUDE ADWL16.INC
```

```
CAN_MSG[n] = value
```

oder

```
value = CAN_MSG[n]
```

### Parameter

<code>n</code>	Elementnummer im Feld <code>CAN_MSG</code> (1...9)	LONG
<code>value</code>	Wert (8 Bit), der in das Message-Objekt geschrieben oder daraus gelesen wird.	LONG

### Bemerkungen

Die Elemente des Felds `CAN_MSG[]` haben folgende Funktion:

Elementnr. in <code>CAN_MSG</code>	1...8	9
Inhalt	Message-Objekt(e) = Datenbyte(s)	Anzahl (0...8) belegter Datenbytes

Tragen Sie die zu übertragenden Werte in das Feld `CAN_MSG[]` ein, bevor Sie diese mit **TRANSMIT** übertragen.

### Siehe auch

EN\_INTERRUPT, EN\_RECEIVE, EN\_TRANSMIT, GET\_CAN\_REG, INIT\_CAN, READ\_MSG, SET\_CAN\_BAUDRATE, SET\_CAN\_REG, TRANSMIT

### Gültig für

L16-DIO1

**Beispiel**

REM Sendet eine 32 Bit-FLOAT-Zahl (hier: Pi) als Folge von  
REM 4 Bytes in einem Message-Objekt  
REM (Empfangen einer Fließkomma-Zahl siehe Bsp. bei READ\_MSG)

```
#INCLUDE ADWL16.INC
#DEFINE pi 3.14159265
DIM i AS LONG

INIT:
  INIT_CAN()                'CAN-Controller initialisieren

  REM Message-Objekt 6 der Schnittstelle initialisieren
  REM zum Senden von CAN-Nachrichten mit dem Identifier 40
  EN_TRANSMIT(6,40,0)

  REM Bitmuster von Pi mit Datenformat Long erzeugen
  PAR_1 = CAST_FLOATTOLONG(pi)

  REM Bitmuster (32 Bit) in 4 Bytes aufteilen
  CAN_MSG[4] = PAR_1 AND 0FFh 'LSB zuweisen
  FOR i = 1 TO 3
    CAN_MSG[4-i] = SHIFT_RIGHT(PAR_1,8*i) AND 0FFh
  NEXT i
  CAN_MSG[9] = 4            'Länge der Nachricht in Bytes

EVENT:
  TRANSMIT(6)              'Message-Objekt 6 senden
```

## EN\_INTERRUPT

CAN-Bus: **EN\_INTERRUPT** konfiguriert ein bestimmtes Message-Objekt so, dass bei Eintreffen einer Nachricht ein externer Event erzeugt wird.

### Syntax

```
#INCLUDE ADWL16.INC  
  
EN_INTERRUPT(objectno)
```

### Parameter

`objectno`      Nummer (1...15) des Message-Objektes      LONG

### Siehe auch

CAN\_MSG, EN\_RECEIVE, GET\_CAN\_REG

### Gültig für

L16-DIO1

### Beispiel

```
#INCLUDE ADWL16.INC  
  
INIT:  
  INIT_CAN()                    'CAN-Controller initialisieren  
  EN_RECEIVE(1,200,0)        'Initialisiere das Message-Objekt 1  
                                'zum Empfangen von CAN-Nachrichten mit  
                                'dem Identifier 200  
  EN_INTERRUPT(1)            'Gibt das Auslösen von Interrupts  
                                '(ext. EVENT) beim Empfang des  
                                'Message-Objektes 1 frei
```



## EN\_RECEIVE

CAN-Bus: **EN\_RECEIVE** gibt ein bestimmtes Message-Objekt zum Nachrichten-Empfang frei.

### Syntax

```
#INCLUDE ADWL16.INC  
EN_RECEIVE(objectno, id, extend)
```

### Parameter

objectno	Nummer (1...15) des Message-Objektes	LONG
id	Identifizier ( $0 \dots 2^{11}$ oder $0 \dots 2^{29}$ ) der Nachrichten, die in diesem Message-Objekt empfangen werden können.	LONG
extend	Länge des Identifiers: 0: 11 Bit 1: 29 Bit	LONG

### Bemerkungen

Ein Message-Objekt kann nur Nachrichten vom CAN-Bus empfangen, wenn Sie es zuvor mit **EN\_RECEIVE** zum Empfang freigegeben haben.

Das Message-Objekt empfängt nur Nachrichten mit dem von Ihnen angegebenen Identifizier.

### Siehe auch

CAN\_MSG, EN\_TRANSMIT, GET\_CAN\_REG

### Gültig für

L16-DIO1

**Beispiel**

```
#INCLUDE ADWL16.INC
```

```
INIT:
```

```
    INIT_CAN()
```

```
    EN_RECEIVE(1,200,0)
```

```
'CAN-Controller initialisieren
```

```
'Initialisiere das Message-Objekt 1
```

```
'zum Empfangen von CAN-Nachrichten mit
```

```
'dem Identifier 200
```

## EN\_TRANSMIT

CAN-Bus: **EN\_TRANSMIT** gibt ein bestimmtes Message-Objekt zum Nachrichten-Senden frei.

### Syntax

```
#INCLUDE ADWL16.INC  
  
EN_TRANSMIT(objectno, id, extend)
```

### Parameter

objectno	Nummer (1...14) des Message-Objektes	LONG
id	Identifizier, der mit den Nachrichten dieses Message-Objekts gesendet wird.	LONG
extend	Länge des Identifiers: 0: 11 Bit 1: 29 Bit	LONG

### Bemerkungen

Ein Message-Objekt kann nur Nachrichten auf dem CAN-Bus senden, wenn Sie es zuvor mit **EN\_TRANSMIT** zum Senden freigegeben haben.

### Siehe auch

CAN\_MSG, EN\_RECEIVE, GET\_CAN\_REG

### Gültig für

L16-DIO1

### Beispiel

```
#INCLUDE ADWL16.INC  
  
INIT:  
  INIT_CAN()           'CAN-Controller initialisieren  
  EN_TRANSMIT(6,40,0)  'Initialisiere das Message-Objekt 6  
                        'zum Senden von CAN-Nachrichten mit  
                        'dem Identifizier 40
```

## GET\_CAN\_REG

CAN-Bus: **GET\_CAN\_REG** liest den Wert eines bestimmten Registers im CAN-Controller.

### Syntax

```
#INCLUDE ADWL16.INC

ret_val = GET_CAN_REG(regno)
```

### Parameter

regno	Register-Nummer (0...255) im CAN-Controller	LONG
ret_val	Inhalt des Registers (übergeben in den unteren 8 Bit)	LONG

### Bemerkungen

Sie finden die Registernummern des CAN-Controllers AN82527 im Intel®-Datenblatt.

### Siehe auch

SET\_CAN\_BAUDRATE, SET\_CAN\_REG

### Gültig für

L16-DIO1

### Beispiel

```
#INCLUDE ADWL16.INC

INIT:
    INIT_CAN() 'CAN-Controller initialisieren
    PAR_1 = GET_CAN_REG(0) 'Control-Register auslesen
```

## INIT\_CAN

CAN-Bus: **INIT\_CAN** initialisiert den CAN-Controller.

### Syntax

```
#INCLUDE ADWL16.INC  
  
INIT_CAN()
```

### Bemerkungen

führt folgende Aktionen aus:

- Reset (Hardware-Reset des CAN-Controllers)
- Alle Filter werden auf "must match" gesetzt.
- Clockout-Register wird auf 0 gesetzt (= externe Frequenz wird nicht geteilt).
- Register „Bus-Configuration“ wird auf 0 gesetzt.
- Die Übertragungsrate für den CAN-Bus wird auf 1 MBit/s gesetzt.
- Alle Message-Objekte werden gesperrt.

Sie müssen diese Anweisung ausführen, bevor Sie mit anderen Befehlen auf den CAN-Controller zugreifen. Wir empfehlen die Angabe im Prozessabschnitt **LOWINIT**: oder **INIT**:

### Siehe auch

CAN\_MSG, EN\_RECEIVE, EN\_TRANSMIT, GET\_CAN\_REG

### Gültig für

L16-DIO1

### Beispiel

```
#INCLUDE ADWL16.INC  
  
INIT:  
    INIT_CAN()                'Initialisiere den CAN-Controller 1  
                                'auf CAN-Modul 1
```

## READ\_MSG

CAN-Bus: **READ\_MSG** prüft, ob neue Nachrichten in einem bestimmten Message-Objekt empfangen wurden.

Falls ja, wird die Nachricht in **CAN\_MSG** gespeichert und der Identifier der Nachricht zurückgegeben.

### Syntax

```
#INCLUDE ADWL16.INC  
  
ret_val = READ_MSG(msgno)
```

### Parameter

msgno	Nummer (1...15) des Message-Objektes	LONG
ret_val	-1: keine neue Nachricht >0: Neue Nachricht; Wert = Identifier der Nachricht	LONG

### Bemerkungen

Sie können eine empfangene Nachricht nur einmal auslesen.

Das Message-Objekt, das Sie auslesen möchten, müssen Sie zuvor mit **EN\_RECEIVE** zum Empfang freigegeben haben.

### Siehe auch

CAN\_MSG, EN\_RECEIVE, EN\_TRANSMIT, GET\_CAN\_REG

### Gültig für

L16-DIO1

### Beispiel

```
REM Wenn eine neue Nachricht mit dem passenden Identifier  
REM empfangen wurde, werden die Daten gelesen. Die  
REM ersten 4 Bytes der Nachricht werden zu einer Fließkomma-  
REM Zahl mit 32 Bit Länge zusammengesetzt (Senden einer  
REM Fließkomma-Zahl siehe Bsp. bei TRANSMIT).
```

```
#INCLUDE ADWL16.INC
```

```
DIM n AS LONG
```

#### INIT:

```
PAR_1 = 0
```

```
INIT_CAN() 'CAN-Controller initialisieren
```

```
EN_RECEIVE(1,40,0) 'Message-Objekt 1 initialisieren  
'zum Empfangen von CAN-Nachrichten mit  
'dem Identifier 40
```

#### EVENT:

```
REM Wenn das Message-Objekt geändert wurde, werden die  
REM empfangenen Daten aus Objekt 1 gelesen und der  
REM Identifier an PAR_9 übergeben.
```

```
REM Die Daten stehen im Feld CAN_MSG[] bereit.
```

```
PAR_9 = READ_MSG(1)
```

```
IF (PAR_9 = 40) THEN
```

```
REM Für das Message-Objekt ist eine neue Nachricht mit dem  
REM Identifier 40 eingetroffen
```

```
PAR_1 = CAN_MSG[1] 'High-Byte auslesen
```

```
FOR n = 2 TO 4 'Mit restlichen 3 Bytes zu 32 Bit-Zahl
```

```
PAR_1 = SHIFT_LEFT(PAR_1,8) + CAN_MSG[n] 'zusammenfügen
```

```
NEXT n
```

```
REM Das Bitmuster in PAR_1 in den Datentyp FLOAT wandeln und  
REM der Variablen FPAR_1 zuweisen.
```

```
FPAR_1 = CAST_LONGTOFLOAT(PAR_1)
```

```
ENDIF
```

## SET\_CAN\_BAUDRATE

CAN-Bus: **SET\_CAN\_BAUDRATE** stellt die Baudrate des CAN-Controllers ein.

### Syntax

```
#INCLUDE ADWL16.INC

ret_val = SET_CAN_BAUDRATE(rate)
```

### Parameter

<code>rate</code>	Baudrate in Bits/Sekunde	LONG
<code>ret_val</code>	0: Baudrate wurde eingestellt 1: Baudrate unzulässig	LONG

### Bemerkungen

Die möglichen Baudraten (= Busfrequenzen) entnehmen Sie bitte der Tabelle im Anhang, Seite A-3. Übernehmen Sie bitte die genaue Schreibweise, d.h. nicht ganzzahlige Baudraten mit 4 Nachkommastellen; Werte mit abweichender Schreibweise werden als nicht zulässig zurückgewiesen.

führt folgende Aktionen aus:

- Prüfen, ob die übergebene Baudrate zulässig ist. Falls nicht, dann den Rückgabewert auf 1 setzen und die Bearbeitung beenden.
- Die Register des CAN-Controllers für die Baudrate setzen.
- Sampling Mode auf 0 setzen: Ein Sample pro Bit.
- Die Einstellungen so wählen, dass der Sample-Punkt immer zwischen 60% und 72% der Gesamt-Bitlänge liegt.
- Die Sprungweite zur Synchronisation auf 1 setzen.

In Sonderfällen kann es vorteilhaft sein, die Einstellungen anders zu wählen als oben beschrieben. Sie finden hierzu eine Erläuterung im Hardware-Handbuch.



sollte in den Programm-Abschnitten **LOWINIT** oder **INIT** aufgerufen werden, und zwar erst nach der Anweisung **INIT\_CAN**, weil sonst die eingestellte Baudrate wieder mit der Standardeinstellung (1 MBit/s) überschrieben wird.

### Siehe auch

GET\_CAN\_REG, SET\_CAN\_REG



### Gültig für

L16-DIO1

### Beispiel

```
#INCLUDE ADWL16.INC
```

```
INIT:
```

```
    INIT_CAN() 'CAN-Controller initialisieren
```

```
    SET_CAN_BAUDRATE(125000) 'Baudrate 125 kBit/s setzen
```

## SET\_CAN\_REG

CAN-Bus: **SET\_CAN\_REG** schreibt einen Wert in ein bestimmtes Register des CAN-Controllers.

### Syntax

```
#INCLUDE ADWL16.INC  
  
SET_CAN_REG(regno, value)
```

### Parameter

regno	Register-Nummer (0...255) im CAN-Controller	LONG
value	Wert (8 Bit), der ins Register geschrieben wird.	LONG

### Bemerkungen

Sie finden die Registernummern des CAN-Controllers AN82527 im Intel®-Datenblatt.

### Siehe auch

SET\_CAN\_BAUDRATE, GET\_CAN\_REG

### Gültig für

L16-DIO1

### Beispiel

```
#INCLUDE ADWL16.INC  
  
INIT:  
    INIT_CAN()                'CAN-Controller initialisieren  
    SET_CAN_REG(0,1)          'Control-Register auf den Wert 1  
                               'setzen
```

## TRANSMIT

CAN-Bus: **TRANSMIT** sendet die Nachricht in `CAN_MSG` über ein bestimmtes Message-Objekt.

### Syntax

```
#INCLUDE ADWL16.INC
```

```
TRANSMIT (msgno)
```

### Parameter

`msgno`

Nummer (1...14) des Message-Objektes

**LONG**

### Bemerkungen

Geben Sie die Nachricht – Datenbytes und Anzahl der Datenbytes – in das Feld `CAN_MSG` ein, bevor Sie die Übertragung starten.

Sie müssen das Message-Objekt vorher mit **EN\_TRANSMIT** zum Senden freigeben.

sendet die Nachricht, sobald das Message-Objekt Zugriffsrecht auf den CAN-Bus hat.

### Siehe auch

INIT\_CAN, READ\_MSG, EN\_TRANSMIT

### Gültig für

L16-DIO1

**Beispiel**

REM Sendet eine 32 Bit-FLOAT-Zahl (hier: Pi) als Folge von  
REM 4 Bytes in einem Message-Objekt  
REM (Empfangen einer Fließkomma-Zahl siehe Bsp. bei READ\_MSG)

```
#INCLUDE ADWL16.INC
#DEFINE pi 3.14159265
DIM i AS LONG

INIT:
  INIT_CAN()                'CAN-Controller initialisieren

  REM Message-Objekt 6 der Schnittstelle initialisieren
  REM zum Senden von CAN-Nachrichten mit dem Identifier 40
  EN_TRANSMIT(6,40,0)

  REM Bitmuster von Pi mit Datenformat Long erzeugen
  PAR_1 = CAST_FLOATTOLONG(pi)

  REM Bitmuster (32 Bit) in 4 Bytes aufteilen
  CAN_MSG[4] = PAR_1 AND 0FFh 'LSB zuweisen
  FOR i = 1 TO 3
    CAN_MSG[4-i] = SHIFT_RIGHT(PAR_1,8*i) AND 0FFh
  NEXT i
  CAN_MSG[9] = 4             'Länge der Nachricht in Bytes

EVENT:
  TRANSMIT(6)                'Message-Objekt 6 senden
```

## 6.5 ADwin-Gold-CAN

Dieser Abschnitt beschreibt die Befehle für die CAN-Erweiterung von *ADwin-Gold*; die Befehle für den SSI-Decoder sind auch für die L16-Erweiterung L16-DIO2 gültig. Für die Anwendung der Gold-Befehle benötigen Sie die Include-Datei `ADWGCAN.inc`, für die L16-Erweiterung die Datei `ADWL16.inc`.

Die Gold-CAN-Erweiterung enthält 4 SSI-Decoder, 2 CAN-Schnittstellen und 2 RSxxx-Schnittstellen. Folgende Befehle stehen zur Verfügung:

### CAN-Schnittstellen

CAN_MSG	Seite 328	INIT_CAN	Seite 336
EN_CAN_INTERRUPT	Seite 330	READ_MSG	Seite 337
EN_RECEIVE	Seite 331	SET_CAN_BAUDRATE	Seite 339
EN_TRANSMIT	Seite 333	SET_CAN_REG	Seite 341
GET_CAN_REG	Seite 335	TRANSMIT	Seite 342

### RSxxx-Schnittstellen

CHECK_SHIFT_REG	Seite 344	RS_RESET	Seite 350
GET_RS	Seite 346	RS485_SEND	Seite 351
READ_FIFO	Seite 347	SET_RS	Seite 353
RS_INIT	Seite 348	WRITE_FIFO	Seite 354

### SSI-Decoder

SSI_MODE	Seite 356	SSI_SET_CLOCK	Seite 362
SSI_READ	Seite 358	SSI_START	Seite 364
SSI_SET_BITS	Seite 360	SSI_STATUS	Seite 366

## CAN\_MSG

CAN-Bus: `CAN_MSG[]` ist ein eindimensionales Feld mit 9 Elementen, in dem die Message-Objekte gespeichert sind oder werden.

### Syntax

```
#INCLUDE ADWGCAN.INC
```

```
CAN_MSG[n] = value
```

oder

```
value = CAN_MSG[n]
```

### Parameter

<code>n</code>	Elementnummer im Feld <code>CAN_MSG</code> (1...9)	LONG
<code>value</code>	Wert (8 Bit), der in das Message-Objekt geschrieben oder daraus gelesen wird.	LONG

### Bemerkungen

Die Elemente des Felds `CAN_MSG[]` haben folgende Funktion:

Elementnr.	1...8	9
Inhalt	Datenbytes im Message-Objekt	Anzahl (0...8) belegter Datenbytes

Tragen Sie die zu übertragenden Werte in das Feld `CAN_MSG[]` ein, bevor Sie diese mit **TRANSMIT** übertragen.

### Siehe auch

EN\_RECEIVE, EN\_TRANSMIT, INIT\_CAN, READ\_MSG, TRANSMIT

### Gültig für

Gold-CAN

**Beispiel**

REM Sendet eine 32 Bit-FLOAT-Zahl (hier: Pi) als Folge von  
REM 4 Bytes in einem Message-Objekt  
REM (Empfangen einer Fließkomma-Zahl siehe Bsp. bei READ\_MSG)

```
#INCLUDE ADWGCAN.INC
#DEFINE pi 3.14159265
DIM i AS LONG

INIT:
  INIT_CAN(2)          'CAN-Controller 2 initialisieren

  REM Initialisiere das Message-Objekt 6 der Schnittstelle 2
  REM zum Senden von CAN-Nachrichten mit dem Identifier 40
  EN_TRANSMIT(2, 6,40,0)

  REM Bitmuster von Pi mit Datenformat Long erzeugen
  PAR_1 = CAST_FLOATTOLONG(pi)

  REM Bitmuster (32 Bit) in 4 Bytes aufteilen
  CAN_MSG[4] = PAR_1 AND 0FFh 'LSB zuweisen
  FOR i = 1 TO 3
    CAN_MSG[4-i] = SHIFT_RIGHT(PAR_1,8*i) AND 0FFh
  NEXT i
  CAN_MSG[9] = 4          'Länge der Nachricht in Bytes

EVENT:
  TRANSMIT(2,6)          'Message-Objekt 6 senden
```

## EN\_CAN\_INTERRUPT

CAN-Bus: **EN\_CAN\_INTERRUPT** konfiguriert ein bestimmtes Message-Objekt einer CAN-Schnittstelle so, dass bei Eintreffen einer Nachricht ein externer Event erzeugt wird.

### Syntax

```
#INCLUDE ADWGCAN.INC

EN_CAN_INTERRUPT (Can_No, objectno)
```

### Parameter

Can_No	Nummer (1, 2) der CAN-Schnittstelle	LONG
objectno	Nummer (1...15) des Message-Objektes	LONG

### Siehe auch

CAN\_MSG, EN\_RECEIVE, EN\_TRANSMIT

### Gültig für

Gold-CAN

### Beispiel

```
#INCLUDE ADWGCAN.INC

INIT:
  INIT_CAN(1)          'CAN-Controller 1 initialisieren
  EN_RECEIVE(1,1,200,0) 'Initialisiere das Message-Objekt 1
                        'der CAN-Schnittstelle 1 zum Empfangen
                        'von CAN-Nachrichten mit dem
                        'Identifizier 200
  EN_CAN_INTERRUPT(1,1) 'Gibt das Auslösen von Interrupts
                        '(ext. EVENT) beim Empfang des
                        'Message-Objektes 1 frei
```



## EN\_RECEIVE

CAN-Bus: **EN\_RECEIVE** gibt ein bestimmtes Message-Objekt einer CAN-Schnittstelle zum Nachrichten-Empfang frei.

### Syntax

```
#INCLUDE ADWGCAN.INC
```

```
EN_RECEIVE (Can_No, objectno, id, extend)
```

### Parameter

Can_No	Nummer (1, 2) der CAN-Schnittstelle	LONG
objectno	Nummer (1...15) des Message-Objektes	LONG
id	Identifizier (0...2 <sup>11</sup> oder 0...2 <sup>29</sup> ) der Nachrichten, die in diesem Message-Objekt empfangen werden können.	LONG
extend	Länge des Identifiers: 0: 11 Bit 1: 29 Bit	LONG

### Siehe auch

CAN\_MSG, EN\_CAN\_INTERRUPT, EN\_TRANSMIT,  
GET\_CAN\_REG

### Bemerkungen

Ein Message-Objekt kann nur Nachrichten vom CAN-Bus empfangen, wenn Sie es zuvor mit **EN\_RECEIVE** zum Empfang freigegeben haben.

Das Message-Objekt empfängt nur Nachrichten mit dem von Ihnen angegebenen Identifizier.

### Gültig für

Gold-CAN

**Beispiel**

```
#INCLUDE ADWGCAN.INC
```

```
INIT:
```

```
  INIT_CAN(1)           'CAN-Controller 1 initialisieren  
  EN_RECEIVE(1,1,200,0) 'Initialisiere Message-Objekt 1 der  
                        'Schnittstelle 1 zum Empfangen von  
                        'Nachrichten mit dem Identifier 200
```

## EN\_TRANSMIT

CAN-Bus: **EN\_TRANSMIT** gibt ein bestimmtes Message-Objekt einer CAN-Schnittstelle zum Nachrichten-Senden frei.

### Syntax

```
#INCLUDE ADWGCAN.INC  
EN_TRANSMIT(Can_No, objectno, id, extend)
```

### Parameter

Can_No	Nummer (1, 2) der CAN-Schnittstelle	LONG
objectno	Nummer (1...14) des Message-Objektes	LONG
id	Identifizier, der mit den Nachrichten dieses Message-Objekts gesendet wird.	LONG
extend	Länge des Identifiers: 0: 11 Bit 1: 29 Bit	LONG

### Siehe auch

CAN\_MSG, EN\_CAN\_INTERRUPT, EN\_RECEIVE,  
GET\_CAN\_REG, TRANSMIT

### Bemerkungen

Erst wenn ein Message-Objekt mit **EN\_TRANSMIT** zum Senden freigegeben ist, kann das Objekt Nachrichten auf dem CAN-Bus senden.

### Gültig für

Gold-CAN

**Beispiel**

```
#INCLUDE ADWGCAN.INC
```

```
INIT:
```

```
  INIT_CAN(1)           'CAN-Controller 1 initialisieren  
  REM Initialisiere Message-Objekte der Schnittstelle 1:  
  REM Objekt 2 zum Empfangen mit Identifier 200,  
  REM Objekt 6 zum Senden mit Identifier 40  
  EN_RECEIVE(1,1,200,0)  
  EN_TRANSMIT(1,6,40,0)
```

## GET\_CAN\_REG

CAN-Bus: **GET\_CAN\_REG** liest den Wert eines bestimmten Registers im Controller einer CAN-Schnittstelle.

### Syntax

```
#INCLUDE ADWGCAN.INC  
  
ret_val = GET_CAN_REG(Can_No, regno)
```

### Parameter

Can_No	Nummer (1, 2) der CAN-Schnittstelle	LONG
regno	Register-Nummer (0...255) im CAN-Controller	LONG
ret_val	Inhalt des Registers (übergeben in den unteren 8 Bit)	LONG

### Siehe auch

INIT\_CAN, SET\_CAN\_BAUDRATE, SET\_CAN\_REG

### Bemerkungen

Sie finden die Registernummern des CAN-Controllers AN82527 im Intel®-Datenblatt.

### Gültig für

Gold-CAN

### Beispiel

```
#INCLUDE ADWGCAN.INC  
INIT:  
    INIT_CAN(1)           'CAN-Controller 1 initialisieren  
    PAR_1 = GET_CAN_REG(1,0) 'Control-Register auslesen
```

## INIT\_CAN

CAN-Bus: **INIT\_CAN** initialisiert den Controller einer CAN-Schnittstelle.

### Syntax

```
#INCLUDE ADWGCAN.INC  
  
INIT_CAN (Can_No)
```

### Parameter

Can\_No

Nummer (1, 2) der CAN-Schnittstelle

LONG

### Bemerkungen

Die Anweisung führt folgende Aktionen aus:

- Reset (Hardware-Reset des CAN-Controllers)
- Alle Filter werden auf "must match" gesetzt.
- Clockout-Register wird auf 0 gesetzt (= externe Frequenz wird nicht geteilt).
- Register „Bus-Configuration“ wird auf 0 gesetzt.
- Die Übertragungsrate für den CAN-Bus wird auf 1 MBit/s gesetzt.
- Alle Message-Objekte werden gesperrt.

Sie müssen diese Anweisung ausführen, bevor Sie mit anderen Befehlen auf den CAN-Controller zugreifen. Wir empfehlen die Angabe im Prozessabschnitt **LOWINIT**: oder **INIT**:

### Siehe auch

CAN\_MSG, EN\_CAN\_INTERRUPT, EN\_RECEIVE, EN\_TRANSMIT,  
GET\_CAN\_REG, READ\_MSG

### Gültig für

Gold-CAN

### Beispiel

```
#INCLUDE ADWGCAN.INC  
  
INIT:  
    INIT_CAN (1)                'Initialisiere den CAN-Controller 1
```

## READ\_MSG

CAN-Bus: **READ\_MSG** prüft, ob neue Nachrichten in einem bestimmten Message-Objekt einer CAN-Schnittstelle empfangen wurden.

Falls ja, wird die Nachricht in **CAN\_MSG** gespeichert und der Identifier der Nachricht zurückgegeben.

### Syntax

```
#INCLUDE ADWGCAN.INC  
  
ret_val = READ_MSG(Can_No, msgno)
```

### Parameter

Can_No	Nummer (1, 2) der CAN-Schnittstelle	LONG
msgno	Nummer (1...15) des Message-Objektes	LONG
ret_val	-1: keine neue Nachricht >0: Neue Nachricht; Wert = Identifier der Nachricht	LONG

### Bemerkungen

Um eine Nachricht zu empfangen, müssen Sie folgende Reihenfolge einhalten:

- Geben Sie das Message-Objekt mit **EN\_RECEIVE** zum Empfangen frei.
- Prüfen Sie auf eine neue Nachricht und – falls vorhanden – speichern die Nachricht in **CAN\_MSG** mit **READ\_MSG**.

Sie können eine empfangene Nachricht nur einmal auslesen.

### Siehe auch

CAN\_MSG, EN\_RECEIVE, EN\_TRANSMIT, GET\_CAN\_REG, TRANSMIT

### Gültig für

Gold-CAN

## Beispiel

```

REM Wenn eine neue Nachricht mit dem passenden Identifier
REM empfangen wurde, werden die Daten gelesen. Die
REM ersten 4 Bytes der Nachricht werden zu einer Fließkomma-
REM Zahl mit 32 Bit Länge zusammengesetzt (Senden einer
REM Fließkomma-Zahl siehe Bsp. bei TRANSMIT).
#INCLUDE ADWGCAN.INC
DIM n AS LONG

INIT:
    PAR_1 = 0
    INIT_CAN(1)           'CAN-Controller 1 initialisieren
    EN_RECEIVE(1,1,40,0)  'Message-Objekt 1 initialisieren
                           'zum Empfangen von CAN-Nachrichten mit
                           'dem Identifier 40

EVENT:
    REM Wenn das Message-Objekt geändert wurde, werden die
    REM empfangenen Daten aus Objekt 1 gelesen und der
    REM Identifier an PAR_9 übergeben.
    REM Die Daten stehen im Feld CAN_MSG[] bereit.
    PAR_9 = READ_MSG(1,1)

    IF (PAR_9 = 40) THEN
        REM Für das Message-Objekt ist eine neue Nachricht mit dem
        REM Identifier 40 eingetroffen
        PAR_1 = CAN_MSG[1]  'High-Byte auslesen
        FOR n = 2 TO 4      'Mit restlichen 3 Bytes zu 32 Bit-Zahl
            PAR_1 = SHIFT_LEFT(PAR_1,8) + CAN_MSG[n] 'zusammenfügen
        NEXT n
        REM Das Bitmuster in PAR_1 in den Datentyp FLOAT wandeln und
        REM der Variablen FPAR_1 zuweisen.
        FPAR_1 = CAST_LONGTOFLOAT(PAR_1)
    ENDIF

```



## SET\_CAN\_BAUDRATE

CAN-Bus: **SET\_CAN\_BAUDRATE** stellt die Baudrate des Controllers einer CAN-Schnittstelle ein.

### Syntax

```
#INCLUDE ADWGCAN.INC  
  
ret_val = SET_CAN_BAUDRATE (Can_No, rate)
```

### Parameter

Can_No	Nummer (1, 2) der CAN-Schnittstelle	LONG
rate	Baudrate in Bits/Sekunde	LONG
ret_val	0: Baudrate wurde eingestellt 1: Baudrate unzulässig	LONG

### Bemerkungen

Die möglichen Baudraten (= Busfrequenzen) entnehmen Sie bitte der Tabelle (Anhang Seite A-3). Übernehmen Sie bitte die genaue Schreibweise, d. h. nicht ganzzahlige Baudraten mit 4 Nachkommastellen; Werte mit abweichender Schreibweise werden als nicht zulässig zurückgewiesen.

Die Anweisung führt folgende Aktionen aus:

- Prüfen, ob die übergebene Baudrate zulässig ist. Falls nicht, dann den Rückgabewert auf 1 setzen und die Bearbeitung beenden.
- Die Register des CAN-Controllers für die Baudrate setzen.
- Sampling Mode auf 0 setzen: Ein Sample pro Bit.
- Die Einstellungen so wählen, dass der Sample-Punkt immer zwischen 60% und 72% der Gesamt-Bitlänge liegt.
- Die Sprungweite zur Synchronisation auf 1 setzen.

In Sonderfällen kann es vorteilhaft sein, die Einstellungen anders zu wählen als oben beschrieben. Sie finden hierzu eine Erläuterung im Hardware-Handbuch.

Die Anweisung sollte in den Programm-Abschnitten **LOWINIT**: oder **INIT**: aufgerufen werden, und zwar erst nach der Anweisung **INIT\_CAN**, weil sonst die eingestellte Baudrate wieder mit der Standardeinstellung (1MBit/s) überschrieben wird.



**Siehe auch**

INIT\_CAN, GET\_CAN\_REG, SET\_CAN\_REG

**Gültig für**

Gold-CAN

**Beispiel**

```
#INCLUDE ADWGCAN.INC
```

```
INIT:
```

```
    INIT_CAN(1)                'CAN-Controller 1 initialisieren
```

```
    SET_CAN_BAUDRATE(1,125000) 'Baudrate 125 kBit/s setzen
```

## SET\_CAN\_REG

CAN-Bus: **SET\_CAN\_REG** schreibt einen Wert in ein bestimmtes Register des Controllers einer CAN-Schnittstelle.

### Syntax

```
#INCLUDE ADWGCAN.INC  
  
SET_CAN_REG(Can_No, regno, value)
```

### Parameter

Can_No	Nummer (1, 2) der CAN-Schnittstelle	LONG
regno	Register-Nummer (0...255) im CAN-Controller	LONG
value	Wert (8 Bit), der ins Register geschrieben wird.	LONG

### Bemerkungen

Sie finden die Registernummern des CAN-Controllers AN82527 im Intel®-Datenblatt.

### Siehe auch

INIT\_CAN, SET\_CAN\_BAUDRATE, GET\_CAN\_REG

### Gültig für

Gold-CAN

### Beispiel

```
#INCLUDE ADWGCAN.INC  
  
INIT:  
    INIT_CAN(1)           'CAN-Controller 1 initialisieren  
    SET_CAN_REG(1,0,1)    'Control-Register auf den Wert 1  
                           'setzen
```

## TRANSMIT

CAN-Bus: **TRANSMIT** sendet die Nachricht in `CAN_MSG` über ein bestimmtes Message-Objekt einer CAN-Schnittstelle.

### Syntax

```
#INCLUDE ADWGCAN.INC  
TRANSMIT (Can_No, msgno)
```

### Parameter

<code>Can_No</code>	Nummer (1, 2) der CAN-Schnittstelle	LONG
<code>msgno</code>	Nummer (1...14) des Message-Objektes	LONG

### Bemerkungen

Um eine Nachricht zu senden, müssen Sie folgende Reihenfolge einhalten:

- Geben Sie das Message-Objekt mit **EN\_TRANSMIT** zum Senden frei.
- Geben Sie die Nachricht in das Feld `CAN_MSG` ein: Die Datenbytes und die Anzahl der Datenbytes.
- Senden Sie die Nachricht mit **TRANSMIT**.

Die CAN-Schnittstelle sendet die Nachricht, sobald das Message-Objekt Zugriffsrecht auf den CAN-Bus hat.

### Siehe auch

`CAN_MSG`, `INIT_CAN`, `READ_MSG`, `EN_TRANSMIT`

### Gültig für

Gold-CAN

**Beispiel**

REM Sendet eine 32 Bit-FLOAT-Zahl (hier: Pi) als Folge von  
REM 4 Bytes in einem Message-Objekt  
REM (Empfangen einer Fließkomma-Zahl siehe Bsp. bei READ\_MSG)

```
#INCLUDE ADWGCAN.INC
#DEFINE pi 3.14159265
DIM i AS LONG

INIT:
    INIT_CAN(2)                'CAN-Controller 2 initialisieren

    REM Initialisiere das Message-Objekt 6 der Schnittstelle 2
    REM zum Senden von CAN-Nachrichten mit dem Identifier 40
    EN_TRANSMIT(2, 6,40,0)

    REM Bitmuster von Pi mit Datenformat Long erzeugen
    PAR_1 = CAST_FLOATTOLONG(pi)

    REM Bitmuster (32 Bit) in 4 Bytes aufteilen
    CAN_MSG[4] = PAR_1 AND 0FFh 'LSB zuweisen
    FOR i = 1 TO 3
        CAN_MSG[4-i] = SHIFT_RIGHT(PAR_1,8*i) AND 0FFh
    NEXT i
    CAN_MSG[9] = 4              'Länge der Nachricht in Bytes

EVENT:
    TRANSMIT(2,6)              'Message-Objekt 6 senden
```

## CHECK\_SHIFT\_REG

RSxxx: **CHECK\_SHIFT\_REG** gibt zurück, ob alle Daten gesendet sind, die in den Sende-FIFO der RSxxx-Schnittstelle geschrieben wurden.

### Syntax

```
#INCLUDE ADWGCAN.INC  
  
ret_val = CHECK_SHIFT_REG(interface)
```

### Parameter

<code>interface</code>	Nummer (1, 2) der Schnittstelle, deren Sende-Status geprüft wird.	LONG
<code>ret_val</code>	Sende-Status: 0: Daten sind gesendet (= keine Daten im Sende-FIFO vorhanden). 1: Noch nicht alle Daten gesendet (= im Sende-FIFO sind noch Daten vorhanden).	LONG

### Bemerkungen

Bei dem Rückgabewert 0 ist sowohl das Sende-FIFO als auch das Ausgangs-Shiftregister leer. Bei dem Rückgabewert 1 ist mindestens ein Bit noch nicht gesendet.

Benutzen Sie diesen Befehl nur, wenn Sie sich bereits eingehend mit dem eingesetzten Controller vertraut gemacht haben (Datenblatt des Herstellers Texas Instruments). Für allgemeine Anwendungen stehen Ihnen komfortablere Befehle aus der Include-Datei zur Verfügung.

### Siehe auch

GET\_RS, RS\_INIT, RS\_RESET, WRITE\_FIFO

### Gültig für

Gold-CAN

### Beispiel

```
#INCLUDE ADWGCAN.INC
```

```
EVENT:
```

```
...
```

```
PAR_1 = CHECK_SHIFT_REG(1) 'Prüft, ob Schnittstelle 1 noch  
                                'Daten zu senden hat
```

```
...
```

## GET\_RS

RSxxx: **GET\_RS** liest den Inhalt eines bestimmten Controller-Registers aus.

### Syntax

```
#INCLUDE ADWGCAN.INC  
ret_val = GET_RS(reg_addr)
```

### Parameter

reg_addr	Adresse des zu lesenden Controller-Registers	LONG
ret_val	Inhalt des Controller-Registers	LONG

### Bemerkungen

Benutzen Sie diesen Befehl nur, wenn Sie sich bereits eingehend mit dem eingesetzten Controller vertraut gemacht haben (Datenblatt des Herstellers Texas Instruments). Für allgemeine Anwendungen stehen Ihnen komfortablere Befehle aus der Include-Datei zur Verfügung.

### Siehe auch

CHECK\_SHIFT\_REG, RS\_INIT, RS\_RESET, SET\_RS

### Gültig für

Gold-CAN

### Beispiel

-/-



## READ\_FIFO

RSxxx: **READ\_FIFO** liest einen Wert aus dem Eingangs-FIFO einer bestimmten Schnittstelle.

### Syntax

```
#INCLUDE ADWGCAN.INC  
  
ret_val = READ_FIFO(interface)
```

### Parameter

interface	Nummer (1, 2) der auszulesenden Schnittstelle	LONG
ret_val	Inhalt des Eingangs-FIFO: -1: FIFO ist leer ≥0: Übertragener Datenwert	LONG

### Bemerkungen

-/-

### Siehe auch

RS\_INIT, RS\_RESET, RS485\_SEND, WRITE\_FIFO

### Gültig für

Gold-CAN

### Beispiel

```
#INCLUDE ADWGCAN.INC  
  
INIT:  
  RS_RESET()  
  RS_INIT(1,9600,0,8,0,1)'Initialisierung von Schnittstelle 1  
                           'mit 9600 Baud, ohne Parität,  
                           '8 Datenbits, 1 Stopbit und  
                           'Hardwarehandshake.  
  
EVENT:  
  PAR_1 = READ_FIFO(1) 'Einen Wert aus dem FIFO holen. Wenn  
                       'der FIFO leer ist, wird -1  
                       'zurückgeliefert.
```

## RS\_INIT

RSxxx: **RS\_INIT** initialisiert die angegebene Schnittstelle.

Folgende Kennwerte werden gesetzt:

- Übertragungsgeschwindigkeit in Baud
- Anwendung von Prüf-Bits
- Datenlänge
- Anzahl der Stopp-Bits
- Übertragungs-Protokoll (Handshake)

### Syntax

```
#INCLUDE ADWGCAN.INC
```

```
RS_INIT(interface,baud,parity,bits,stop,  
        handshake)
```

### Parameter

<code>interface</code>	Nummer (1, 2) der Schnittstelle, die initialisiert werden soll	LONG
<code>baud</code>	Übertragungsgeschwindigkeit in Baud	LONG
<code>parity</code>	Anwendung von Prüf-Bits: 0: ohne Paritäts-Bit 1: gerade Parität (even) 2: ungerade Parität (odd)	LONG
<code>bits</code>	Anzahl der Daten-Bits (5, 6, 7 oder 8)	LONG
<code>stop</code>	Anzahl der Stopp-Bits 0: 1 Stopp-Bit 1: 1½ Stopp-Bits bei 5 Daten-Bits; 2 Stopp-Bits bei 6, 7 oder 8 Daten-Bits	LONG
<code>handshake</code>	Übertragungs-Protokoll: 0: RS232, kein Handshake 1: RS232, Hardware Handshake (RTS/CTS) 2: RS232, Software-Handshake (Xon/Xoff) 3: RS485 (Voreinstellung)	LONG

### Bemerkungen

Diese Anweisung ist vor dem ersten Arbeiten mit der gewählten Schnittstelle notwendig, um deren Parameter einzustellen. Sie müssen für eine korrekte Übertragung mit der Gegenstelle identisch sein.



Die Initialisierung ist auch dann erforderlich, nachdem Sie mit **RS\_RESET** einen Hardware-Reset ausgeführt haben.

Wenn das Übertragungs-Protokoll RS485 eingestellt wird, muss auch die Übertragungsrichtung festgelegt werden (mit **RS485\_SEND**).

### Siehe auch

CHECK\_SHIFT\_REG, GET\_RS, RS485\_SEND, RS\_RESET, SET\_RS

### Gültig für

Gold-CAN

### Beispiel

```
#INCLUDE ADWGCAN.INC
```

```
INIT:
```

```
    RS_RESET()           'RS-Controller zurücksetzen  
    RS_INIT(1,9600,0,8,0,1)'Initialisierung von Schnittstelle 1  
                           'mit 9600 Baud, ohne Parität,  
                           '8 Datenbits, 1 Stoppbit und  
                           'Hardware-Handshake.
```

## RS\_RESET

RSxxx: **RS\_RESET** führt einen Hardware-Reset des RSxxx-Controllers durch.

### Syntax

```
#INCLUDE ADWGCAN.INC  
  
RS_RESET ()
```

### Bemerkungen

sendet einen Reset-Impuls auf den entsprechenden Eingang des Controllers TL16C754. Sie können dem Datenblatt des Controllers 16C754 von Texas Instruments entnehmen, auf welche Werte die Register durch den Hardware-Reset gesetzt werden.

Nach einem Hardware-Reset muss eine Initialisierung mit **RS\_INIT** folgen, um den Controller zu initialisieren und die gewünschten Schnittstellen-Parameter einzustellen.

### Siehe auch

CHECK\_SHIFT\_REG, GET\_RS, RS\_INIT, SET\_RS

### Gültig für

Gold-CAN

### Beispiel

```
#INCLUDE ADWGCAN.INC  
  
INIT:  
  RS_RESET()                'RSxxx Controller zurücksetzen  
  RS_INIT(1,9600,0,8,0,1) 'Initialisierung von Schnittstelle 1  
                           'mit 9600 Baud, ohne Parität,  
                           '8 Datenbits, 1 Stoppbit und  
                           'Hardware-Handshake.
```

## RS485\_SEND

RSxxx: **RS485\_SEND** legt die Übertragungsrichtung für eine bestimmte Schnittstelle fest.

### Syntax

```
#INCLUDE ADWGCAN.INC  
RS485_SEND(interface, dir)
```

### Parameter

interface	Einzustellende Schnittstelle (1, 2)	LONG
dir	Übertragungsrichtung der Schnittstelle: 0: Schnittstelle als Empfänger einstellen. 1: Schnittstelle als Sender einstellen. 2: Schnittstelle als Sender einstellen, der gleichzeitig die gesendeten Daten empfängt. 3: Schnittstelle stumm schalten, d.h. die Schnittstelle arbeitet als Empfänger, nimmt aber keine Daten in den Eingangs-FIFO auf.	LONG

### Bemerkungen

Die Einstellung der Übertragungsrichtung bedeutet:

- Empfänger: Der Controller kann Daten auf dem Bus ausschließlich lesen, auch wenn Daten im Ausgangs-FIFO liegen.
- Sender: Der Controller kann Daten auf den Bus legen, die von anderen Teilnehmern gelesen werden können.
- Sender/Empfänger: Der Controller kann Daten auf den Bus legen und gleichzeitig zurücklesen. Dadurch ist eine Überprüfung der ausgegebenen Daten möglich.

### Siehe auch

CHECK\_SHIFT\_REG, GET\_RS, RS\_INIT, RS\_RESET, SET\_RS

### Gültig für

Gold-CAN

**Beispiel**

-/-

## SET\_RS

RSxxx: **SET\_RS** schreibt einen Wert in ein bestimmtes Register.

### Syntax

```
#INCLUDE ADWGCAN.INC  
SET_RS (reg_addr, value)
```

### Parameter

reg_addr	Nummer des zu beschreibenen Registers	LONG
value	Wert, der in das Register geschrieben werden soll	LONG

### Bemerkungen

Benutzen Sie diesen Befehl nur, wenn Sie sich bereits eingehend mit dem eingesetzten Controller vertraut gemacht haben (Datenblatt des Herstellers: TL16C754 von Texas Instruments). Für allgemeine Anwendungen stehen Ihnen komfortablere Befehle aus der Include-Datei zur Verfügung.

### Siehe auch

GET\_RS, RS\_INIT, RS\_RESET

### Gültig für

Gold-CAN

### Beispiel

-/-

## WRITE\_FIFO

RSxxx: **WRITE\_FIFO** schreibt einen Wert in den Sende-FIFO einer bestimmten Schnittstelle.

### Syntax

```
#INCLUDE ADWGCAN.INC  
  
ret_val = WRITE_FIFO(interface,value)
```

### Parameter

interface	Nummer (1, 2) der Schnittstelle, deren Sende-FIFO beschrieben wird	LONG
value	Wert der ins Sende-FIFO geschrieben werden soll	LONG
ret_val	Statusmeldung: 0: Daten wurden erfolgreich geschrieben. 1: Daten konnten nicht geschrieben werden, Sende-FIFO ist voll.	LONG

### Bemerkungen

prüft zuerst, ob noch mindestens ein Speicherplatz im Sende-FIFO frei ist. Ist dies der Fall, wird der übergebene Wert ins FIFO geschrieben (Rückgabewert 0); anderenfalls wird eine 1 zurückgeliefert, die angibt, dass das FIFO voll ist und ein Schreiben nicht möglich war.

### Siehe auch

CHECK\_SHIFT\_REG, READ\_FIFO, RS\_INIT, RS\_RESET, RS485\_SEND

### Gültig für

Gold-CAN



### Beispiel

```
#INCLUDE ADWGCAN.INC
DIM val AS LONG

INIT:
  RS_RESET()
  RS_INIT(1,9600,0,8,0,1)'Initialisierung von Schnittstelle 1
                           'mit 9600 Baud, keine Parität,
                           '8 Datenbits, 1 Stoppbit und
                           'Hardware-Handshake.

EVENT:
  PAR_1 = WRITE_FIFO(1,val)'Ist das FIFO nicht voll, wird
                           'val ins FIFO geschrieben.
                           'Wenn das FIFO-Feld voll ist, wird dies
                           'mit dem Wert 1 in PAR_1 angezeigt.
```

## SSI\_MODE

SSI: **SSI\_MODE** stellt den Modus aller SSI-Decoder ein, entweder „single shot“ (einzeln lesen) und „continuous“ (kontinuierlich lesen).

### Syntax

```
#INCLUDE ADWGCNT.INC      'nur für ADwin-Gold
#include ADWL16.INC       'nur für ADwin-light-16

SSI_MODE(pattern)
```

### Parameter

**pattern** Betriebsmodus der SSI-Decoder, angegeben als Bitmuster. Jedem Decoder ist ein Bit zugeordnet (siehe Tabelle). LONG

Bit = 0: Modus „Single shot“, der Encoder wird einmal ausgelesen.

Bit = 1: Modus „Continuous“, der Encoder wird kontinuierlich ausgelesen.

Bitnr.	31:2	3	2	1	0
SSI-Decoder Gold	–	4	3	2	1
SSI-Decoder L16	–	–	–	–	1

### Bemerkungen

Wenn Sie den Modus „continuous“ wählen, startet das Auslesen des entsprechenden Encoders sofort. **SSI\_START** ist hierzu nicht erforderlich.

Manche Encoder-Typen liefern im Modus „continuous“ gelegentlich den falschen Messwert 0 (Null) anstelle des korrekten Messwerts zurück. Im Modus „single shot“ tritt dieser Fehler nicht auf.

### Siehe auch

SSI\_READ, SSI\_SET\_BITS, SSI\_SET\_CLOCK, SSI\_START, SSI\_STATUS

### Gültig für

Gold-CAN, L16-DIO2

### Beispiel

```
#INCLUDE ADWGCAN.INC
```

#### INIT:

```
SSI_SET_CLOCK(1,200) 'Taktrate für Decoder 1 = 50 kHz
SSI_SET_CLOCK(2,200) 'Taktrate für Decoder 2 = 50 kHz
SSI_MODE(11b)        'Continuous-Mode einstellen
                     'für Encoder 1+2
SSI_SET_BITS(1,23)   '23 Encoder-Bits auf Encoder 1
SSI_SET_BITS(2,23)   '23 Encoder-Bits auf Encoder 2
```

#### EVENT:

```
PAR_1 = SSI_READ(1) 'Positionswert (Encoder 1) auslesen
PAR_2 = SSI_READ(2) 'Positionswert (Encoder 2) auslesen
```

## SSI\_READ

SSI: **SSI\_READ** gibt den zuletzt gespeicherten Zählerstand eines bestimmten SSI-Zählers zurück.

### Syntax

```
#INCLUDE ADWGCNT.INC      'nur für ADwin-Gold
#include ADWL16.INC       'nur für ADwin-light-16

ret_val = SSI_READ(dcdr_no)
```

### Parameter

<code>dcdr_no</code>	Nummer (Gold: 1...4; L16: 1) des SSI-Decoders, dessen Zählerstand auszulesen ist.	LONG
<code>ret_val</code>	Letzter Zählerstand des SSI-Zählers (= Abolutwert-Position des Encoders)	LONG

### Bemerkungen

Ein Encoder-Wert wird dann gespeichert, wenn die durch **SSI\_SET\_BITS** angegebene Anzahl von Bits eingelesen wurde.

### Siehe auch

SSI\_MODE, SSI\_SET\_BITS, SSI\_SET\_CLOCK, SSI\_START, SSI\_STATUS

### Gültig für

Gold-CAN, L16-DIO2

### Beispiel

```
#INCLUDE ADWGCAN.INC
DIM m, n, y AS LONG

INIT:
    SSI_SET_CLOCK(1,50)    'Taktrate für Decoder 1 = 200 kHz
    SSI_MODE(1)            'Continuous-Mode setzen (Encoder 1)
    SSI_SET_BITS(1,23)     '23 Encoder-Bits auf Encoder 1

EVENT:
    PAR_1 = SSI_READ(1)    'Positionswert (Encoder 1) auslesen

    REM Wert von Gray-Code in Binärwert wandeln:
    m = 0                  'Werte der letzten Wandlung löschen
    y = 0                  ' _"-_
    FOR n = 1 TO 32        'Alle 32 mögl. Bits durchgehen
        m = (SHIFT_RIGHT(PAR_1, (32 - n)) AND 1) XOR m
        y = (SHIFT_LEFT(m, (32 - n)) OR y
    NEXT n
    PAR_9 = y              'Das Ergebnis der Gray-/Binär-
                           'Wandlung in PAR_9
```

## SSI\_SET\_BITS

SSI: **SSI\_SET\_BITS** stellt für einen bestimmten SSI-Zähler die Anzahl der zu Bits ein, die einen vollständigen Encoder-Wert bilden.

Die Zahl der Bits sollte mit der Auflösung des Encoders identisch sein.

### Syntax

```
#INCLUDE ADWGCNT.INC      'nur für ADwin-Gold
#INCLUDE ADWL16.INC       'nur für ADwin-light-16

SSI_SET_BITS (dcdr_no, bit_no)
```

### Parameter

<code>dcdr_no</code>	Nummer (Gold: 1...4; L16: 1) des SSI-Decoders, dessen Auflösung einzustellen ist.	LONG
<code>bit_no</code>	Anzahl der Bits (1...32) der zu lesenden Bits für einen Encoder-Wert (entspricht der Encoder-Auflösung).	LONG

### Bemerkungen

Die Auflösung (Anzahl der Bits) des SSI-Encoders sollte mit der Anzahl der zu übertragenden Bits übereinstimmen.



Achten Sie darauf, dass die zu lesenden Bits mit der Encoder-Auflösung genau übereinstimmen.

### Siehe auch

SSI\_MODE, SSI\_READ, SSI\_SET\_CLOCK, SSI\_START, SSI\_STATUS

### Gültig für

Gold-CAN, L16-DIO2

### Beispiel

```
#INCLUDE ADWGCAN.INC

INIT:
    SSI_SET_CLOCK(1,50)    'Taktrate für Decoder 1 = 200 kHz
    SSI_SET_CLOCK(2,50)    'Taktrate für Decoder 2 = 200 kHz
    SSI_MODE(11b)          'Continuous-Mode einstellen
                           'für Encoder 1+2
    SSI_SET_BITS(1,10)     '10 Encoder-Bits für Encoder 1
    SSI_SET_BITS(2,25)     '25 Encoder-Bits für Encoder 2

EVENT:
    PAR_1 = SSI_READ(1)    'Positionswert (Encoder 1) auslesen
    PAR_2 = SSI_READ(2)    'Positionswert (Encoder 2) auslesen
```

## SSI\_SET\_CLOCK

SSI: **SSI\_SET\_CLOCK** stellt die Taktrate (ca. 40kHz bis 1MHz) ein, mit der der Encoder getaktet wird.

### Syntax

```
#INCLUDE ADWGCNT.INC      'nur für ADwin-Gold
#INCLUDE ADWL16.INC       'nur für ADwin-light-16

SSI_SET_CLOCK(dcd_r_no,prescale)
```

### Parameter

dcd_r_no	Nummer (Gold: 1...4; L16: 1) des SSI-Decoders, dessen Taktrate einzustellen ist.	LONG
prescale	Teilerfaktor (10...255) zur Einstellung der Taktrate nach der Formel: Taktrate = 10MHz / prescale	LONG

### Bemerkungen

Teilerfaktoren kleiner 10 werden automatisch auf den Wert 10 korrigiert; bei Werten über 255 werden die niederwertigsten 8 Bits als Teilerfaktor verwendet.

Die mögliche Taktfrequenz ist abhängig von Kabellänge, Kabeltyp und den jeweils verwendeten Sende- und Empfangsbausteinen des Encoders und des Decoders. Grundsätzlich gilt als Regel: Je höher die Taktfrequenz, desto kürzer die mögliche Kabellänge.

### Siehe auch

SSI\_MODE, SSI\_READ, SSI\_SET\_BITS, SSI\_START, SSI\_STATUS

### Gültig für

Gold-CAN, L16-DIO2



### Beispiel

```
#INCLUDE ADWGCAN.INC

INIT:
    SSI_SET_CLOCK(1,10)  'Taktrate für Decoder 1 = 1 MHz
    SSI_SET_CLOCK(2,20)  'Taktrate für Decoder 2 = 0,5 MHz
    SSI_MODE(11b)        'Continuous-Mode setzen
                        'für Encoder 1+2
    SSI_SET_BITS(1,23)    '10 Encoder-Bits auf Encoder 1
    SSI_SET_BITS(2,23)    '25 Encoder-Bits auf Encoder 2

EVENT:
    PAR_1 = SSI_READ(1)   'Positionswert (Encoder 1) auslesen
    PAR_2 = SSI_READ(2)   'Positionswert (Encoder 2) auslesen
```

## SSI\_START

SSI: **SSI\_START** startet das Auslesen eines oder beider SSI-Encoder (nur im Modus single shot).

### Syntax

```
#INCLUDE ADWGCNT.INC      'nur für ADwin-Gold
#include ADWL16.INC       'nur für ADwin-light-16

SSI_START(dcdr_no)
```

### Parameter

**dcdr\_no**      Bitmuster zur Auswahl der SSI-Decoder, die LONG gestartet werden sollen:  
 Bit = 0: keine Funktion  
 Bit = 1: Auslesen des SSI-Decoders starten

Bitnr.	31:2	3	2	1	0
SSI-Decoder Gold	–	4	3	2	1
SSI-Decoder L16	–	–	–	–	1

### Bemerkungen

Im Modus continuous ist diese Anweisung ohne Funktion, weil dann die Encoder-Werte ohnehin kontinuierlich ausgelesen werden.

Ein Encoder-Wert wird dann gespeichert, wenn die durch **SSI\_SET\_BITS** angegebene Anzahl von Bits eingelesen wurde.



Es wird immer ein vollständiger Encoder-Wert übertragen, auch wenn währenddessen der Modus umgestellt wird.

### Siehe auch

SSI\_MODE, SSI\_READ, SSI\_SET\_BITS, SSI\_SET\_CLOCK, SSI\_STATUS

### Gültig für

Gold-CAN, L16-DIO2

**Beispiel**

```
#INCLUDE ADWGCAN.INC
```

**INIT:**

```
SSI_SET_CLOCK(1,250) 'Taktrate für Decoder 1 = 40 kHz
SSI_SET_CLOCK(2,250) 'Taktrate für Decoder 2 = 40 kHz
SSI_MODE(0)          'Single shot-Mode einstellen
                     'für alle Zähler

SSI_SET_BITS(1,23)   '23 Encoder-Bits auf Encoder 1
SSI_SET_BITS(2,23)   '23 Encoder-Bits auf Encoder 2
```

**EVENT:**

```
SSI_START(11b)       'Positionswert von Encoder 1&2 lesen
DO
  'Für Encoder 1:
  UNTIL (SSI_STATUS(1) = 0) 'Wenn Positionswert komplett
    'gelesen ist, dann ...
  PAR_1 = SSI_READ(1)   'Positionswert auslesen
DO
  'Für Encoder 2:
  UNTIL (SSI_STATUS(2) = 0) 'Wenn Positionswert komplett
    'gelesen ist, dann ...
  PAR_1 = SSI_READ(2)   'Positionswert auslesen
```

## SSI\_STATUS

SSI: **SSI\_STATUS** liefert für einen bestimmten Decoder den aktuellen Lese-Status zurück.

### Syntax

```
#INCLUDE ADWGCNT.INC      'nur für ADwin-Gold
#include ADWL16.INC       'nur für ADwin-light-16

ret_val = SSI_STATUS(dcdr_no)
```

### Parameter

<code>dcdr_no</code>	Nummer (Gold: 1...4; L16: 1) des SSI-Decoders, dessen Status gefragt ist.	LONG
<code>ret_val</code>	Lese-Status des Decoders: 0: Decoder ist bereit, d.h. ein vollständiger Wert wurde gelesen. 1: Decoder liest einen Encoder-Wert ein.	LONG

### Bemerkungen

Verwenden Sie die Status-Abfrage nur im SSI-Modus „single shot“. Im Modus „continuous“ ist eine Status-Abfrage nicht sinnvoll.

### Siehe auch

SSI\_MODE, SSI\_READ, SSI\_SET\_BITS, SSI\_SET\_CLOCK, SSI\_START

### Gültig für

Gold-CAN, L16-DIO2

### Beispiel

```
#INCLUDE ADWGCAN.INC

INIT:
  SSI_SET_CLOCK(1,250) 'Taktrate für Decoder 1 = 40 kHz
  SSI_SET_CLOCK(2,250) 'Taktrate für Decoder 2 = 40 kHz
  SSI_MODE(0)          'Single shot-Mode einstellen
                        'für alle Zähler

  SSI_SET_BITS(1,23)   '23 Encoder-Bits auf Encoder 1
  SSI_SET_BITS(2,23)   '23 Encoder-Bits auf Encoder 2

EVENT:
  SSI_START(11b)       'Positionswert von Encoder 1&2 lesen
  DO
  UNTIL (SSI_STATUS(1) = 0) 'Wenn Positionswert komplett
                        'gelesen ist, dann ...
  PAR_1 = SSI_READ(1)   'Positionswert auslesen
  DO
  UNTIL (SSI_STATUS(2) = 0) 'Wenn Positionswert komplett
                        'gelesen ist, dann ...
  PAR_1 = SSI_READ(2)   'Positionswert auslesen
```



## 6.6 ADwin-L16 Rev. B

Dieser Abschnitt beschreibt zusätzliche Befehle für *ADwin-light-16* Rev. B. Für die Befehle dieses Abschnitts benötigen Sie die Include-Datei `ADWL16.INC`.

Außer den Befehlen dieses Abschnitts sind für *ADwin-light-16* Rev. B auch die Befehle aus Kapitel 6.3 (ab Seite 235) und Kapitel 6.4 (ab Seite 265) anwendbar.

## L16\_MODE

**L16\_MODE** stellt den Betriebsmodus für *ADwin-light-16* Rev. B ein.

### Syntax

**L16\_MODE** (*mode*)

### Parameter

*mode* Bitmuster zur Einstellung des Betriebsmodus. LONG

Bits in <i>mode</i>	Bedeutung
Bit 0:	Bit = 0: Standardbetrieb (Default). Bit = 1: Schnellbetrieb.
Bits 1...31:	reserviert

### Bemerkungen

Im Modus „Standardbetrieb“ arbeitet das Gerät vollständig kompatibel zu der Revision A. Nach dem Einschalten ist das Gerät immer im Modus „Standardbetrieb“.

Im Modus „Schnellbetrieb“ arbeitet der A/D-Wandler mit einer maximalen Abtastrate von 500kHz.

### Siehe auch

-/-

### Gültig für

L16 Rev. B

### Beispiel

```
INIT:
REM Modus "Schnellbetrieb" aktivieren
L16_MODE(1)
```



## SEQ\_INIT

**SEQ\_INIT** initialisiert die Ablaufsteuerung.

Eingestellt werden der Arbeitsmodus, der Verstärkungsfaktor, die Kanäle und die Einschwingzeit des Multiplexers.

### Syntax

```
#INCLUDE ADWL16.inc

SEQ_INIT(mode, gain, channels, muxtime)
```

### Parameter

<b>mode</b>	Arbeitsmodus der Ablaufsteuerung:	LONG
	0: Normaler Modus (Default), Einzelmessung.	
	1: Modus „single shot“, einfacher Messzyklus.	
	2: Modus „continuous“, regelmäßiger Messzyklus.	
	3: Modus „continuous max“ mit max. Geschwindigkeit.	
<b>gain</b>	Verstärkungsfaktor (nur für die Modi 1 ... 3):	LONG
	0 Faktor = 1, Spannungsbereich -10V...+10V.	
	1 Faktor = 2, Spannungsbereich -5V...+5V.	
	2 Faktor = 4, Spannungsbereich -2,5V...+2,5V.	
	3 Faktor = 8, Spannungsbereich -1,25V...+1,25V.	
<b>channels</b>	Bitmuster, das die zu wandelnden Kanäle bestimmt.	LONG
	Bit = 0: Kanal nicht wandeln.	
	Bit = 1: Kanal wandeln.	

Bitnr.	31:15	14	13	12	...	3	2	1	0
Kanalnr.	–	15	–	11	...	–	3	–	1

<b>muxtime</b>	Anzahl der Zeiteinheiten, aus der sich die Einschwingzeit der Ablaufsteuerung ergibt:	LONG
	0: Werkseinstellung ( $200 \hat{=} 5\mu\text{s}$ ) für die Wartezeit.	
	200...2 <sup>31</sup> : Einschwingzeit in Einheiten von 25ns.	

### Bemerkungen

Nach dem Einschalten des Geräts ist Modus 0 aktiv.

Die Modi 1 ... 3 aktivieren die Ablaufsteuerung, die an mehreren Kanälen nacheinander eine Wandlung durchführt, je nach Modus einmal

oder in zyklischen Abständen. Die Steuerung bezieht sich immer nur auf die mit `channels` definierte Auswahl an Kanälen.

Die Modi unterscheiden sich wie folgt:

Modus	Art der Messung
0 Normal:	Standard: Einzelne Messung an einem Kanal, siehe <b>ADC</b> .
1 single shot:	Die Ablaufsteuerung wird mit <b>START_CONV</b> gestartet; die Ablaufsteuerung endet, sobald die gewählten Kanäle je einmal gewandelt sind.  Das Ende der Ablaufsteuerung wird mit <b>WAIT_EOC</b> abgefragt und die Messwerte mit <b>SEQ_READ</b> eingelesen.
2 continuous:	Die Ablaufsteuerung wandelt für jeden Prozesszyklus einen Satz neuer Messwerte.  Die Wandlung wird im Abschnitt <b>INIT</b> : gestartet, mit <b>START_CONV</b> als letztem Befehl. Das Wandlungsende (für alle Kanäle) wird automatisch mit dem Beginn des nächsten Prozesszyklus synchronisiert. Daher können – und sollten auch – alle Messwerte zu Beginn des Prozesszyklus mit <b>SEQ_READ</b> gelesen werden.
3 continuous max:	Die Ablaufsteuerung wandelt ununterbrochen und mit maximaler Wandlungsrate neue Messwerte an den gewählten Kanälen, d.h. Wandlung und Prozesszyklus laufen asynchron.  Die Wandlung wird mit <b>START_CONV</b> im Abschnitt <b>INIT</b> : gestartet. Im Prozesszyklus wird mit <b>SEQ_READ</b> der jeweils neueste Messwert gelesen.

Die Einschwingzeit (Parameter `muxtime`) legt den Zeitabstand zwischen 2 Wandlungen der Ablaufsteuerung fest. Wir empfehlen, den angegebenen Wertebereich nicht zu unterschreiten, weil eine zu kurze Einschwingzeit zu ungenaueren bis falschen Messwerten führt.



Wenn der Innenwiderstand der Spannungsquelle des Messsignals zu groß ist, ist die Werkseinstellung des Multiplexers zu kurz für eine genaue Messung. Stellen Sie dann mit dem Parameter `muxtime` eine längere Einschwingzeit für den Multiplexer ein.

### Siehe auch

ADC, SEQ\_READ, START\_CONV, WAIT\_EOC

### Gültig für

L16 Rev. B

### Beispiel

```
#INCLUDE ADWL16.inc
```

```
DIM DATA_1[8] AS LONG AT DM_LOCAL
DIM i AS LONG
```

#### INIT:

```
REM Ablaufsteuerung: Continuous Mode, Verstärkung 2
REM Kanäle 1, 3, ..., 15, Standard-Einschwingzeit
SEQ_INIT(3,1,5555h,0)
START_CONV(1)           'Messzyklus starten
```

#### EVENT:

```
REM Die Wandlung ist für alle gewählten Kanäle gerade
REM beendet, die Messwerte werden abgeholt.
FOR i = 1 TO 8
    DATA_1[i] = SEQ_READ(i*2-1) 'Messwerte holen
NEXT i
REM Messwerte verarbeiten
```

## SEQ\_READ

**SEQ\_READ** gibt den zuletzt gespeicherten Messwert des angegebenen Kanals zurück.

### Syntax

```
#INCLUDE ADWL16.inc  
ret_val = SEQ_READ(channel)
```

### Parameter

<code>channel</code>	Kanalnummer (1, 3, ..., 15).	LONG
<code>ret_val</code>	Messwert (0...65535) des angegebenen Kanals.	LONG

### Bemerkungen

Der Rückgabewert ist nur sinnvoll, wenn vorher mit **SEQ\_INIT** die Ablaufsteuerung aktiviert und dabei der angegebene Kanal mit ausgewählt wurde.

Im Modus „single shot“ muss zuerst das Ende der Ablaufsteuerung mit **WAIT\_EOC** abgefragt werden, bevor die Messwerte gelesen werden.

### Siehe auch

SEQ\_INIT, START\_CONV, WAIT\_EOC

### Gültig für

L16 Rev. B

### Beispiel

```
#INCLUDE ADWL16.inc
```

```
DIM DATA_1[400] AS LONG AT DM_LOCAL
```

#### INIT:

```
REM Ablaufsteuerung: Single shot, Verstärkung 1
REM Kanäle 5, 7, 13, 15, Standard-Einschwingzeit
SEQ_INIT(1,0,101000001010000b,0)
START_CONV(1)           'Messzyklus starten
```

#### EVENT:

```
WAIT_EOC(1)             'Wandlungsende abwarten
REM Kanäle 5, 7, 13, 15 lesen
DATA_1[1] = SEQ_READ(5)
DATA_1[2] = SEQ_READ(7)
DATA_1[3] = SEQ_READ(13)
DATA_1[4] = SEQ_READ(15)
START_CONV(1)           'nächsten Messzyklus starten
```



## 6.7 FFT-Library

Die FFT-Library enthält *ADbasic*-Befehle für die schnelle Fourier-Transformation (Fast Fourier Transformation).

Die Library arbeitet mit Prozessoren ab dem Typ T9.

### Hinweise zur Anwendung der Library

Sie können die Rechenzeit deutlich verringern, wenn Sie die Felder im internen Speicher (**AT DM\_LOCAL**) deklarieren. So dauert die Berechnung einer FFT mit 1024 Werten nur noch etwa 23ms anstatt 35ms (mit Prozessor T9).



Rufen Sie Anweisungen der FFT-Library nur in einem Programm mit niedriger Priorität auf oder im Programmabschnitt **LOWINIT**: oder **INIT**:. Wenn die Berechnung einer FFT in einem hochprioren Programm sehr lange dauert, nimmt der PC einen Fehler an und bricht die Kommunikation zum *ADwin*-System mit einer entsprechenden Fehlermeldung ab.



Im Ordner <C:\ADwin\ADbasic\lib\FFT\_doc+demo> finden Sie alle Beispiele zu den Library-Befehlen.

### Fast-Fourier Transformationen

Die Fast-Fourier-Transformation (FFT) ist ein Algorithmus zur schnellen Berechnung einer diskreten Fouriertransformation. Die FFT ist in der Signalverarbeitung für viele Aufgaben geeignet, so. z. B. zur

- Berechnung digitaler Filter.
- Umrechnung eines Zeitverlaufs in der Schwingungsmesstechnik in eine Frequenzdarstellung.
- Berechnung von Spektrogrammen (Diagramme mit der Darstellung der Amplituden von den jeweiligen Frequenzanteilen).
- Näherungsweise Bestimmung der Frequenzen in einem abgetasteten Signal.

### Befehle der Library

Name	Funktion
<b>FFT</b>	FFT führt eine komplexe Fast-Fourier-Transformation mit komplexen Eingangs- und Ausgangsdaten durch. 379
<b>FFT_MAG</b>	FFT_MAG gibt die Absolutbeträge komplexer Werte zurück. 383

Name	Funktion
<b>FFT_SCALE</b>	FFT_SCALE normiert das Ergebnis einer FFT- 381 Berechnung auf die Größenordnung der einzelnen Signalkomponenten der Quelldaten.
<b>FFT_PHASE</b>	FFT_PHASE gibt die Phasenlagen komplexer Werte 385 zurück.
<b>FFT_MAG_ SCALE</b>	FFT_MAG_SCALE gbt die skalierten Absolutbeträge 387 komplexer Werte zurück.
<b>FFT_INIT</b>	FFT_INIT initialisiert 2 Hilfsfelder für die Berech- 388 nung von Fast-Fourier-Transformationen.
<b>FFT_CALC</b>	FFT_CALC berechnet eine Fast-Fourier-Transforma- 389 tion nach vorheriger Initialisierung.
<b>FFT_CALC_DM</b>	FFT_CALC_DM berechnet eine Fast-Fourier-Trans- 391 formation nach vorheriger Initialisierung und ist für den Prozessor T10 optimiert.
<b>FFT_CALC_DX</b>	FFT_CALC_DX berechnet eine Fast-Fourier-Trans- 393 formation nach vorheriger Initialisierung und ist für den Prozessor T10 optimiert.



## FFT

**FFT** führt eine komplexe Fast-Fourier-Transformation mit komplexen Eingangs- und Ausgangsdaten durch.

### Syntax

```
IMPORT FFT.LI*          '*.LI9 für T9, *.LIA für T10,
                        '*.LIB für T11

FFT(src_re[],src_im[],res_re[],res_im[],
    array1[],array2[],pts)
```

### Parameter

<code>src_re[]</code>	Realteile der Quelldaten.	<div>FLOAT</div> <div>ARRAY</div>
<code>src_im[]</code>	Imaginärteile der Quelldaten.	<div>FLOAT</div> <div>ARRAY</div>
<code>res_re[]</code>	Ergebnis: Realteile (Index 1...n/2) der transformierten Daten. Feldgröße: 4 × pts.	<div>FLOAT</div> <div>ARRAY</div>
<code>res_im[]</code>	Ergebnis: Imaginärteile (Index 1...n/2) der transformierten Daten. Feldgröße: 4 × pts.	<div>FLOAT</div> <div>ARRAY</div>
<code>array1[]</code> , <code>array2[]</code>	Felder für interne Berechnungen. Feldgröße: 4 × pts.	<div>FLOAT</div> <div>ARRAY</div>
<code>pts</code>	Punktzahl (≥ 2) der Quelldaten. Die Anzahl der Punkte muss eine Potenz von 2 sein.	<div>LONG</div>

### Bemerkungen

Die Fourier-Transformation liefert korrekte Ergebnisse, wenn die Frequenzkomponenten  $f_i$  der Quelldaten innerhalb des folgenden Bereichs – bezogen auf die Abtastfrequenz  $f_{\text{Abtast}}$  – liegen:

$$0 \leq f_i \text{ und } f_i < f_{\text{Abtast}}/2$$

Die transformierten Daten, die komplexen Amplituden des Frequenzspektrums, befinden sich nur in den Elementen 1...pts/2 der Felder `res_re` und `res_im`. Die restlichen Feldelemente (bis 4 × pts) werden für interne Berechnungen benötigt und enthalten Zwischenergebnisse.

Das Ergebnis der Transformation ist nicht normiert auf die Größenordnung der einzelnen Signalkomponenten der Quelldaten. Sie können die transformierten Daten mit der Anweisung **FFT\_SCALE** normieren.

Die folgende Tabelle zeigt, wie das berechnete Frequenzspektrum den Elementen der Felder **res\_re** und **res\_im** zugeordnet ist (Normierung der Frequenzachse). Dabei ist  $t_{\text{total}}$  die gesamte Abtastzeit, in der die Quelldaten abgetastet wurden.

Im Beispiel unten ist die Abtastzeit  $t_{\text{total}} = 0,1\text{ s}$ ; dem Element [1024] ist daher die Frequenz  $(1024-1) / 0,1\text{ s} = 10230\text{ Hz}$  zugeordnet.

Elementindex	[ 1 ]	[ 2 ]	...	[ i ]	...	[ pts/2 ]
Frequenz [Hz]	0	$\frac{1}{t_{\text{total}}}$	...	$\frac{i-1}{t_{\text{total}}}$	...	$\frac{\text{pts}/2-1}{t_{\text{total}}}$



Wenn Sie mehrere FFT mit der *gleichen* Anzahl an Quelldaten berechnen, kann die Berechnungszeit verringert werden: Rufen Sie anstelle von **FFT** zuerst die Anweisung **FFT\_INIT** und anschließend mehrfach **FFT\_CALC** auf.

### Siehe auch

FFT\_MAG, FFT\_SCALE, FFT\_PHASE, FFT\_MAG\_SCALE

FFT\_INIT, FFT\_CALC, FFT\_CALC\_DM, FFT\_CALC\_DX

### Beispiel

Das Beispielprogramm (für *ADwin-Gold* und *ADwin-light-16*)

```
<C:\ADwin\ADbasic\lib\FFT_doc+demo\FFT_demo.bas>
```

liest das Analogsignal am Eingang 1 (2048 Messpunkte in 0,1s) und berechnet daraus eine FFT. Wenn beispielsweise ein Sinussignal von 1000Hz anliegt, werden die Maximalwerte in **DATA\_3[101]** (Realteil) und **DATA\_4[101]** (Imaginärteil) gespeichert.

## FFT\_SCALE

**FFT\_SCALE** normiert das Ergebnis einer FFT-Berechnung auf die Größenordnung der einzelnen Signalkomponenten der Quelldaten.

### Syntax

```
IMPORT FFT.LI*          '*.LI9 für T9, *.LIA für T10,
                        '*.LIB für T11

FFT_SCALE(data[],data_scal[],n)
```

### Parameter

<code>data[]</code>	Nicht normierte Daten einer FFT-Berechnung.	<div>FLOAT</div> <div>ARRAY</div>
<code>data_scal[]</code>	Ergebnis: Normierte Daten.	<div>FLOAT</div> <div>ARRAY</div>
<code>n</code>	Anzahl der Daten.	<div>LONG</div>

### Bemerkungen

Die Anweisung arbeitet nach der Formel:

$$\text{data\_scal}[i] = \begin{cases} i \neq 1: & \text{data\_scal}[i] = \text{data}[i]/n \\ i = 1: & \text{data\_scal}[i] = \text{data}[i]/(n \cdot 2) \end{cases}$$

Wenn Sie **FFT\_SCALE** direkt auf die Ergebnisfelder der Anweisung **FFT** anwenden, muss  $n = \text{pts} / 2$  sein (**pts**: Parameter von **FFT**).

**FFT\_SCALE** normiert das Ergebnis einer FFT-Berechnung auf die Größenordnung der einzelnen Signalkomponenten der Originaldaten. Dagegen normiert **FFT\_SCALE** nicht die Frequenzachse des Spektrums (siehe Erläuterungen hierzu unter **FFT**).

### Siehe auch

FFT, FFT\_MAG, FFT\_PHASE, FFT\_MAG\_SCALE

### Beispiel

Das Beispielprogramm (für alle ADwin-Systeme)

```
<C:\ADwin\ADbasic\lib\FFT_doc+demo\FFT_scale_demo.bas>
```

generiert ein Signal aus mehreren Sinussignalen, tastet das Signal ab, berechnet die FFT, den Absolutbetrag und normiert den Absolutbetrag.

Das Quellsignal entsteht aus:

- einem Sinussignal mit 60 Hz und der Amplitude 0,7
- einem Sinussignal mit 30 Hz und der Amplitude 1,0
- einem konstanten Signal mit der Amplitude 1,5

Die Amplituden des normierten Frequenzspektrums (siehe Grafik unten) zeigen exakt die Größen der überlagerten Quellsignale:

DATA\_6[7] = 0.7

Index 7: 60 Hz

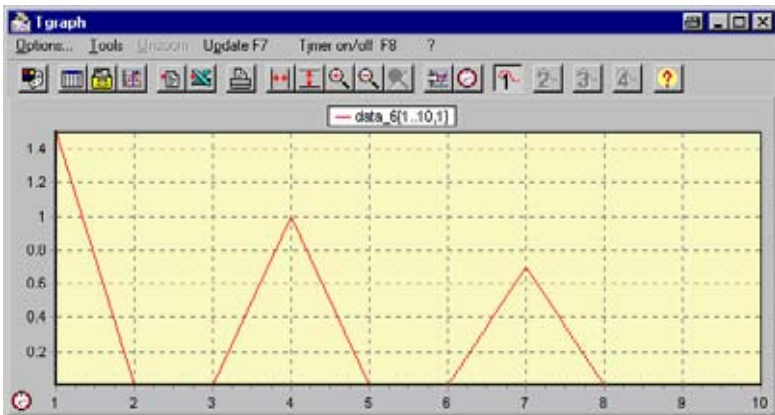
DATA\_6[4] = 1

Index 4: 30 Hz

DATA\_6[1] = 1.5

Index 1: Gleichspannungsanteil

Alle weiteren Amplituden haben den Wert 0.



## FFT\_MAG

**FFT\_MAG** gibt die Absolutbeträge komplexer Werte zurück.

### Syntax

```
IMPORT FFT.LI*          '*.LI9 für T9, *.LIA für T10,
                        '*.LIB für T11

FFT_MAG(cmplx_re[],cmplx_im[],magnitude[],n)
```

### Parameter

<code>cmplx_re[]</code>	Realteile der komplexen Werte.	<div>FLOAT</div> <div>ARRAY</div>
<code>cmplx_im[]</code>	Imaginärteile der komplexen Werte.	<div>FLOAT</div> <div>ARRAY</div>
<code>magnitude[]</code>	Ergebnis: Absolutbeträge der komplexen Werte.	<div>FLOAT</div> <div>ARRAY</div>
<code>n</code>	Anzahl der komplexen Werte.	<div>LONG</div>

### Bemerkungen

Der Betrag eines komplexen Werts wird errechnet mit der Formel:

$$\text{magnitude}[i] = \sqrt{\text{cmplx\_re}[i]^2 + \text{cmplx\_im}[i]^2}$$

Die Anweisung **FFT** berechnet die Amplituden des Frequenzspektrums als komplexe Werte. Die Anweisungen **FFT\_MAG** und **FFT\_PHASE** rechnen die komplexen Amplituden in Betrag und Phase um.

Wenn Sie **FFT\_MAG** direkt auf die Ergebnisfelder der Anweisung **FFT** anwenden, muss `n = pts / 2` sein (`pts`: Parameter von **FFT**).

### Siehe auch

FFT, FFT\_PHASE, FFT\_MAG\_SCALE

### Beispiel

Das Beispielprogramm (für *ADwin-Gold* oder *ADwin-light-16*)

```
<C:\ADwin\ADbasic\lib\FFT_doc+demo\FFT_mag_demo.bas>
```

liest das Analogsignal am Eingang 1 (2048 Messpunkte in 0,1s), berechnet daraus eine FFT und die Absolutbeträge. Wenn beispielsweise ein Sinussignal von 1500Hz anliegt, wird der maximale Betrag in `DATA_5[151]` gespeichert.

## FFT\_PHASE

**FFT\_PHASE** gibt die Phasenlagen komplexer Werte zurück.

### Syntax

```
IMPORT FFT.LI*          '*.LI9 für T9, *.LIA für T10,
                        '*.LIB für T11

FFT_PHASE(cmplx_re[],cmplx_im[],phase[],n)
```

### Parameter

<code>cmplx_re[]</code>	Realteile der komplexen Werte.	<div>FLOAT</div> <div>ARRAY</div>
<code>cmplx_im[]</code>	Imaginärteile der komplexen Werte.	<div>FLOAT</div> <div>ARRAY</div>
<code>phase[]</code>	Ergebnis: Phasenlagen der komplexen Werte.	<div>FLOAT</div> <div>ARRAY</div>
<code>n</code>	Anzahl der komplexen Werte.	<div>LONG</div>

### Bemerkungen

Die Phasenberechnung arbeitet nach folgender Formel (Details siehe `<math.inc>`):

$$\text{phase}[i] = \begin{cases} \text{cmplx\_re}[i] \neq 0: & \text{phase}[i] = \text{atan}(\text{cmplx\_im}[i]/\text{cmplx\_re}[i]) \\ \text{cmplx\_re}[i] = 0: & \text{phase}[i] = \text{sgn}(\text{cmplx\_im}[i]) \cdot \pi/2 \end{cases}$$

Die Anweisung **FFT** berechnet die Amplituden des Frequenzspektrums als komplexe Werte. Die Anweisungen **FFT\_MAG** und **FFT\_PHASE** rechnen die komplexen Amplituden in Betrag und Phase um.

Wenn Sie **FFT\_PHASE** direkt auf die Ergebnisfelder der Anweisung **FFT** anwenden, muss `n = pts / 2` sein (`pts`: Parameter von **FFT**).

### Siehe auch

FFT, FFT\_MAG, FFT\_MAG\_SCALE

### Beispiel

Das Beispielprogramm (für alle ADwin-Systeme)

```
<C:\ADwin\ADbasic\lib\FFT_doc+demo\FFT_phase_demo.bas>
```

generiert 2 (um  $\pi/2$ ) phasenverschobene Sinussignale, tastet die Signale ab, berechnet daraus die FFT, die normierten Absolutbeträge und die Phasenlagen.

Das berechnete Frequenzspektrum hat folgende Werte:

DATA\_6[4] = 1                      Index 4: 30 Hz

DATA\_7[4] = -0.018410      Phase etwa 0

DATA\_26[4] = 1                      Index 4: 30 Hz

DATA\_27[4] = 1.552389      Phase etwa  $\pi/2$

Alle weiteren Amplituden haben den Wert 0 und die zugehörigen Phasenlagen sind undefiniert.



## FFT\_MAG\_SCALE

**FFT\_MAG\_SCALE** gibt die skalierten Absolutbeträge komplexer Werte zurück.

### Syntax

```
IMPORT FFT.LI*          '*.LI9 für T9, *.LIA für T10,  
                        '*.LIB für T11  
  
FFT_MAG_SCALE(cmplx_re[],cmplx_im[],mag_scal[],n)
```

### Parameter

<code>cmplx_re[]</code>	Realteile der komplexen Werte.	<div>FLOAT</div> <div>ARRAY</div>
<code>cmplx_im[]</code>	Imaginärteile der komplexen Werte.	<div>FLOAT</div> <div>ARRAY</div>
<code>mag_scal[]</code>	Ergebnis: Normierte Absolutbeträge der komplexen Werte.	<div>FLOAT</div> <div>ARRAY</div>
<code>n</code>	Anzahl der komplexen Werte.	<div>LONG</div>

### Bemerkungen

Die Anweisung **FFT\_MAG\_SCALE** liefert das gleiche Ergebnis wie der Aufruf von **FFT\_MAG** und **FFT\_SCALE**, arbeitet aber schneller.

Wenn Sie **FFT\_MAG\_SCALE** direkt auf die Ergebnisfelder der Anweisung **FFT** anwenden, muss `n = pts / 2` sein.

### Siehe auch

FFT, FFT\_MAG, FFT\_SCALE

### Beispiel

Das Beispielprogramm <FFT\_scale\_demo\_opt.bas> (für alle ADwin-Systeme) entspricht dem Beispiel <FFT\_scale\_demo.bas> (siehe Seite 381), verwendet jedoch **FFT\_MAG\_SCALE**.

## FFT\_INIT

**FFT\_INIT** initialisiert 2 Hilfsfelder für die Berechnung von Fast-Fourier-Transformationen.

### Syntax

```
IMPORT FFT.LI*          '*.LI9 für T9, *.LIA für T10,
                        '*.LIB für T11

FFT_INIT(array1[],array2[],pts)
```

### Parameter

<code>array1[]</code> ,	Ergebnis: Hilfswerte für interne Berechnungen. Feldgröße: $4 \times \text{pts}$ .	<span style="border: 1px solid black; padding: 2px;">FLOAT</span>
<code>array2[]</code>		<span style="background-color: black; color: white; padding: 2px;">ARRAY</span>
<code>pts</code>	Punktzahl ( $\geq 2$ ) der Quelldaten. Die Anzahl der Punkte muss eine Potenz von 2 sein.	<span style="border: 1px solid black; padding: 2px;">LONG</span>

### Bemerkungen

Die Anweisung **FFT\_INIT** ist nur notwendig und sinnvoll, wenn im Anschluss eine der Anweisungen **FFT\_CALC**, **FFT\_CALC\_DM** oder **FFT\_CALC\_DX** aufgerufen wird.



Wenn Sie mehrere FFT mit der *gleichen* Anzahl `pts` an Quelldaten berechnen, kann die Berechnungszeit verringert werden: Rufen Sie anstelle von **FFT** zuerst die Anweisung **FFT\_INIT** und anschließend mehrfach **FFT\_CALC** auf.

### Siehe auch

FFT, FFT\_CALC, FFT\_CALC\_DM, FFT\_CALC\_DX

### Beispiel

Siehe Beispielprogramm <FFT\_scale\_demo\_opt.bas> (für alle ADwin-Systeme) im Ordner <C:\ADwin\ADbasic\lib\FFT\_doc+demo>.

## FFT\_CALC

**FFT\_CALC** berechnet eine Fast-Fourier-Transformation nach vorheriger Initialisierung.

### Syntax

```
IMPORT FFT.LI*          '*.LI9 für T9, *.LIA für T10,
                        '*.LIB für T11

FFT_CALC(src_re[],src_im[],res_re[],res_im[],
         array1[],array2[],pts)
```

### Parameter

<code>src_re[]</code>	Realteile der Quelldaten.	<div>FLOAT</div> <div>ARRAY</div>
<code>src_im[]</code>	Imaginärteile der Quelldaten.	<div>FLOAT</div> <div>ARRAY</div>
<code>res_re[]</code>	Ergebnis: Realteile (Index 1...n/2) der transformierten Daten. Feldgröße: 4 × pts.	<div>FLOAT</div> <div>ARRAY</div>
<code>res_im[]</code>	Ergebnis: Imaginärteile (Index 1...n/2) der transformierten Daten. Feldgröße: 4 × pts.	<div>FLOAT</div> <div>ARRAY</div>
<code>array1[]</code> , <code>array2[]</code>	Felder für interne Berechnungen. Feldgröße: 4 × pts.	<div>FLOAT</div> <div>ARRAY</div>
<code>pts</code>	Punktzahl (≥ 2) der Quelldaten. Die Anzahl der Punkte muss eine Potenz von 2 sein.	<div>LONG</div>

### Bemerkungen

Die Anweisung ist nur sinnvoll, wenn vorher **FFT\_INIT** aufgerufen wurde.

Wenn Sie mehrere FFT mit der *gleichen* Anzahl `pts` an Quelldaten berechnen, kann die Berechnungszeit verringert werden: Rufen Sie anstelle von **FFT** zuerst die Anweisung **FFT\_INIT** und anschließend mehrfach **FFT\_CALC** auf.



Nur Prozessor T10: Anstelle von **FFT\_CALC** können die Funktionen **FFT\_CALC\_DM** oder **FFT\_CALC\_DX** genutzt werden, die die FFT in kürzerer Zeit berechnen.

**Siehe auch**

FFT, FFT\_INIT, FFT\_CALC\_DM, FFT\_CALC\_DX

**Beispiel**

Siehe Beispielprogramm <FFT\_scale\_demo\_opt.bas> (für alle *ADwin*-Systeme) im Ordner <C:\ADwin\ADbasic\lib\FFT\_doc+demo>.

## FFT\_CALC\_DM

**FFT\_CALC\_DM** berechnet eine Fast-Fourier-Transformation nach vorheriger Initialisierung und ist für den Prozessor T10 optimiert.

### Syntax

```
IMPORT FFT.LIA

FFT_CALC_DM(src_re[], src_im[], res_re[], res_im[],
  array1[], array2[], pts)
```

### Parameter

src_re[]	Realteile der Quelldaten. Das Feld muss <b>AT DM_LOCAL</b> deklariert werden.	<div>FLOAT</div> <div>ARRAY</div>
src_im[]	Imaginärteile der Quelldaten. Das Feld muss <b>AT DM_LOCAL</b> deklariert werden.	<div>FLOAT</div> <div>ARRAY</div>
res_re[]	Ergebnis: Realteile (Index 1...n/2) der transformierten Daten. Das Feld muss <b>AT DM_LOCAL</b> mit der Größe: 4 × pts deklariert werden.	<div>FLOAT</div> <div>ARRAY</div>
res_im[]	Ergebnis: Imaginärteile (Index 1...n/2) der transformierten Daten. Das Feld muss <b>AT DM_LOCAL</b> mit der Größe: 4 × pts deklariert werden.	<div>FLOAT</div> <div>ARRAY</div>
array1[], array2[]	Felder für interne Berechnungen. Die Felder müssen <b>AT DM_LOCAL</b> mit der Größe: 4 × pts deklariert werden.	<div>FLOAT</div> <div>ARRAY</div>
pts	Punktzahl ( $\geq 2$ ) der Quelldaten. Die Anzahl der Punkte muss eine Potenz von 2 sein.	<div>LONG</div>

### Bemerkungen

Die Anweisung ist nur sinnvoll, wenn vorher **FFT\_INIT** aufgerufen wurde.

**FFT\_CALC\_DM** hat die gleiche Funktion wie **FFT\_CALC** (und **FFT\_CALC\_DX**), berechnet eine FFT aber mit dem Prozessor T10 schneller. Beim Prozessor T9 oder T11 ist die Optimierung nicht möglich.

**FFT\_CALC\_DM** darf nur verwendet werden, wenn die Felder im internen Speicher deklariert sind.

Mit dem Prozessor T10 dauert die Berechnung einer FFT mit 1024 Werten nur noch etwa 11 ms anstatt 14 ms mit **FFT\_CALC**. Für beide Zeitmessungen wurden die Felder im internen Speicher **DM\_LOCAL** deklariert.

**Siehe auch**

FFT, FFT\_INIT, FFT\_CALC, FFT\_CALC\_DX

**Beispiel**

Siehe Beispielprogramm `<FFT_scale_demo_opt.bas>` (für alle *ADwin*-Systeme) im Ordner `<C:\ADwin\ADbasic\lib\FFT_doc+demo>`.

## FFT\_CALC\_DX

**FFT\_CALC\_DX** berechnet eine Fast-Fourier-Transformation nach vorheriger Initialisierung und ist für den Prozessor T10 optimiert.

### Syntax

```
IMPORT FFT.LIA

FFT_CALC_DX(src_re[], src_im[], res_re[], res_im[],
            array1[], array2[], pts)
```

### Parameter

<code>src_re[]</code>	Realteile der Quelldaten. Das Feld sollte <b>AT DRAM_EXTERN</b> deklariert werden.	<div>FLOAT</div> <div>ARRAY</div>
<code>src_im[]</code>	Imaginärteile der Quelldaten. Das Feld sollte <b>AT DRAM_EXTERN</b> deklariert werden.	<div>FLOAT</div> <div>ARRAY</div>
<code>res_re[]</code>	Ergebnis: Realteile (Index 1...n/2) der transformierten Daten. Das Feld sollte <b>AT DRAM_EXTERN</b> mit der Größe $4 \times pts$ deklariert sein.	<div>FLOAT</div> <div>ARRAY</div>
<code>res_im[]</code>	Ergebnis: Imaginärteile (Index 1...n/2) der transformierten Daten. Das Feld sollte <b>AT DRAM_EXTERN</b> mit der Größe $4 \times pts$ deklariert sein.	<div>FLOAT</div> <div>ARRAY</div>
<code>array1[]</code> , <code>array2[]</code>	Felder für interne Berechnungen. Die Felder müssen <b>AT DRAM_EXTERN</b> mit der Größe: $4 \times pts$ deklariert werden.	<div>FLOAT</div> <div>ARRAY</div>
<code>pts</code>	Punktzahl ( $\geq 2$ ) der Quelldaten. Die Anzahl der Punkte muss eine Potenz von 2 sein.	<div>LONG</div>

### Bemerkungen

Die Anweisung ist nur sinnvoll, wenn vorher **FFT\_INIT** aufgerufen wurde.

**FFT\_CALC\_DX** hat die gleiche Funktion wie **FFT\_CALC** (und **FFT\_CALC\_DM**), berechnet eine FFT aber mit dem Prozessor T10 schneller. Beim Prozessor T9 oder T11 ist die Optimierung nicht möglich.

**FFT\_CALC\_DX** darf nur verwendet werden, wenn die Felder im externen Speicher deklariert sind.

Mit dem Prozessor T10 dauert die Berechnung einer FFT mit 1024 Werten nur noch etwa 49ms anstatt 53ms mit **FFT\_CALC**. Für beide Zeitmessungen wurden die Felder im externen Speicher **DRAM\_EXTERN** deklariert.

**Siehe auch**

FFT, FFT\_INIT, FFT\_CALC, FFT\_CALC\_DM

**Beispiel**

Siehe Beispielprogramm <FFT\_scale\_demo\_opt\_DX.bas> (für alle ADwin-Systeme) im Ordner <C:\ADwin\ADbasic\lib\FFT\_doc+demo>.



### 7 Was tun bei Problemen?


Wenn Sie bei der Installation Probleme haben, ziehen Sie bitte die Dokumentation zu Ihrem *ADwin*-System zu Rate. Überprüfen Sie, ob alle Einstellungen richtig und vollständig durchgeführt wurden. Prüfen Sie auch, ob im Menü „Options\Compiler“ die Basisadresse, der Prozessortyp etc. richtig sind.

Sollten Ihre Probleme dann immer noch bestehen, rufen Sie uns bitte an. Auch wenn Sie weitergehende Hilfe wünschen, stehen wir Ihnen gern zur Verfügung; Sie finden Adresse und Telefonnummer Ihres Ansprechpartners in der vorderen Umschlagseite des Handbuchs.



## Anhang

### A.1 Tastaturkürzel in ADbasic

Tastenkombination	Funktion	Gleiche Funktion wie
CTRL + F5	ADwin-System booten	Build ▶ Boot
F8	Quelltext kompilieren	Build ▶ Compile
CTRL + F8	Prozess starten	Build ▶ Start
F9	Prozess stoppen	Build ▶ Stop
CTRL + R	Verwendete Parameter markieren	Parameterfenster: Schaltfläche 
CTRL + B	Markierte Zeilen kommentieren	Quelltext-Kontextmenü: Comment Block
CTRL + SHIFT + B	Markierte Zeilen entkommentieren	Quelltext-Kontextmenü: Uncomment Block
TAB	Zeilen rechts einrücken	Quelltext-Kontextmenü: Indent
SHIFT + TAB	Zeilen links einrücken	Quelltext-Kontextmenü: Outdent
CTRL + N	Neue Quelltextdatei	File ▶ New
CTRL + O	Quelltextdatei öffnen	File ▶ Open
CTRL + S	Quelltextdatei sichern	File ▶ Save
CTRL + P	Quelltextdatei drucken	File ▶ Print
CTRL + Z	Eingabe zurücknehmen	Edit ▶ Undo
CTRL + Y	Eingabe wiederherstellen	Edit ▶ Redo
CTRL + X	Ausschneiden	Edit ▶ Cut
CTRL + C	Kopieren	Edit ▶ Copy
CTRL + V	Einfügen	Edit ▶ Paste
CTRL + A	Alles markieren	Edit ▶ Select All
CTRL + F	Text suchen	Edit ▶ Find
F3	Text erneut suchen	Edit ▶ Find Next
CTRL + H	Text ersetzen	Edit ▶ Replace
F1	Hilfe für markierten Befehl aufrufen (kontextsensitiv)	Help ▶ Help Topics

## A.2 ASCII-Zeichensatz

<b>NUL</b> 00h 0	<b>SOH</b> 01h 1	<b>STX</b> 02h 2	<b>ETX</b> 03h 3	<b>EOT</b> 04h 4	<b>ENQ</b> 05h 5	<b>ACK</b> 06h 6	<b>BEL</b> 07h 7
<b>BS<sup>1</sup></b> 08h 8	<b>TAB<sup>2</sup></b> 09h 9	<b>LF<sup>3</sup></b> 0Ah 10	<b>VT</b> 0Bh 11	<b>FF</b> 0Ch 12	<b>CR<sup>4</sup></b> 0Dh 13	<b>SO</b> 0Eh 14	<b>SI</b> 0Fh 15
<b>DLE</b> 10h 16	<b>DC1</b> 11h 17	<b>DC2</b> 12h 18	<b>DC3</b> 13h 19	<b>DC4</b> 14h 20	<b>NAK</b> 15h 21	<b>SYN</b> 16h 22	<b>ETB</b> 17h 23
<b>CAN</b> 18h 24	<b>EM</b> 19h 25	<b>SUB</b> 1Ah 26	<b>ESC</b> 1Bh 27	<b>FS</b> 1Ch 28	<b>GS</b> 1Dh 29	<b>RS</b> 1Eh 30	<b>US</b> 1Fh 31
<b>SPC<sup>5</sup></b> 20h 32	<b>!</b> 21h 33	<b>"</b> 22h 34	<b>#</b> 23h 35	<b>\$</b> 24h 36	<b>%</b> 25h 37	<b>&amp;</b> 26h 38	<b>'</b> 27h 39
<b>(</b> 28h 40	<b>)</b> 29h 41	<b>*</b> 2Ah 42	<b>+</b> 2Bh 43	<b>,</b> 2Ch 44	<b>-</b> 2Dh 45	<b>.</b> 2Eh 46	<b>/</b> 2Fh 47
<b>0</b> 30h 48	<b>1</b> 31h 49	<b>2</b> 32h 50	<b>3</b> 33h 51	<b>4</b> 34h 52	<b>5</b> 35h 53	<b>6</b> 36h 54	<b>7</b> 37h 55
<b>8</b> 38h 56	<b>9</b> 39h 57	<b>:</b> 3Ah 58	<b>;</b> 3Bh 59	<b>&lt;</b> 3Ch 60	<b>=</b> 3Dh 61	<b>&gt;</b> 3Eh 62	<b>?</b> 3Fh 63
<b>@</b> 40h 64	<b>A</b> 41h 65	<b>B</b> 42h 66	<b>C</b> 43h 67	<b>D</b> 44h 68	<b>E</b> 45h 69	<b>F</b> 46h 70	<b>G</b> 47h 71
<b>H</b> 48h 72	<b>I</b> 49h 73	<b>J</b> 4Ah 74	<b>K</b> 4Bh 75	<b>L</b> 4Ch 76	<b>M</b> 4Dh 77	<b>N</b> 4Eh 78	<b>O</b> 4Fh 79
<b>P</b> 50h 80	<b>Q</b> 51h 81	<b>R</b> 52h 82	<b>S</b> 53h 83	<b>T</b> 54h 84	<b>U</b> 55h 85	<b>V</b> 56h 86	<b>W</b> 57h 87
<b>X</b> 58h 88	<b>Y</b> 59h 89	<b>Z</b> 5Ah 90	<b>[</b> 5Bh 91	<b>\</b> 5Ch 92	<b>]</b> 5Dh 93	<b>^</b> 5Eh 94	<b>_</b> 5Fh 95
<b>`</b> 60h 96	<b>a</b> 61h 97	<b>b</b> 62h 98	<b>c</b> 63h 99	<b>d</b> 64h 100	<b>e</b> 65h 101	<b>f</b> 66h 102	<b>g</b> 67h 103
<b>h</b> 68h 104	<b>i</b> 69h 105	<b>j</b> 6Ah 106	<b>k</b> 6Bh 107	<b>l</b> 6Ch 108	<b>m</b> 6Dh 109	<b>n</b> 6Eh 110	<b>o</b> 6Fh 111
<b>p</b> 70h 112	<b>q</b> 71h 113	<b>r</b> 72h 114	<b>s</b> 73h 115	<b>t</b> 74h 116	<b>u</b> 75h 117	<b>v</b> 76h 118	<b>w</b> 77h 119
<b>x</b> 78h 120	<b>y</b> 79h 121	<b>z</b> 7Ah 122	<b>{</b> 7Bh 123	<b> </b> 7Ch 124	<b>}</b> 7Dh 125	<b>~</b> 7Eh 126	<b>□</b> 7Fh 127

<sup>1</sup> Backspace, <sup>2</sup> Tabulator, <sup>3</sup> Linefeed,<sup>4</sup> Carriage Return, <sup>5</sup> Space

### A.3 Baudraten für den CAN-Bus

*ADwin-light-16* DIO1 und *ADwin-Gold-CAN* besitzen Schnittstellen für den CAN-Bus, Version „high speed“. Dort können folgende Baudraten eingestellt werden:

Einstellbare Baudraten [Bit/s]				
1000000.0000	888888.8889	800000.0000	727272.7273	666666.6667
615384.6154	571428.5714	533333.3333	500000.0000	470588.2353
444444.4444	421052.6316	400000.0000	380952.3810	363636.3636
347826.0870	333333.3333	320000.0000	307692.3077	296296.2963
285714.2857	266666.6667	250000.0000	242424.2424	235294.1176
222222.2222	210526.3158	205128.2051	200000.0000	190476.1905
181818.1818	177777.7778	173913.0435	166666.6667	160000.0000
156862.7451	153846.1538	148148.1481	145454.5455	142857.1429
140350.8772	133333.3333	126984.1270	125000.0000	123076.9231
121212.1212	117647.0588	115942.0290	114285.7143	111111.1111
106666.6667	105263.1579	103896.1039	102564.1026	100000.0000
98765.4321	95238.0952	94117.6471	90909.0909	88888.8889
87912.0879	86956.5217	84210.5263	83333.3333	81632.6531
80808.0808	80000.0000	78431.3725	76923.0769	76190.4762
74074.0741	72727.2727	71428.5714	70175.4386	69565.2174
68376.0684	67226.8908	66666.6667	66115.7025	64000.0000
63492.0635	62500.0000	61538.4615	60606.0606	60150.3759
59259.2593	58823.5294	57971.0145	57142.8571	55944.0559
55555.5556	54421.7687	53333.3333	52631.5789	52287.5817
51948.0519	51282.0513	50000.0000	49689.4410	49382.7160
48484.8485	47619.0476	47337.2781	47058.8235	46783.6257
45714.2857	45454.5455	44444.4444	43956.0440	43478.2609
42780.7487	42328.0423	42105.2632	41666.6667	41025.6410
40816.3265	40404.0404	40000.0000	39215.6863	38647.3430
38461.5385	38277.5120	38095.2381	37037.0370	36363.6364
36199.0950	35714.2857	35555.5556	35087.7193	34782.6087
34632.0346	34482.7586	34188.0342	33613.4454	33333.3333

Einstellbare Baudraten [Bit/s]				
33057.8512	32921.8107	32388.6640	32258.0645	32000.0000
31746.0317	31620.5534	31372.5490	31250.0000	30769.2308
30651.3410	30303.0303	30075.1880	29629.6296	29411.7647
29304.0293	29090.9091	28985.5072	28673.8351	28571.4286
28070.1754	27972.0280	27777.7778	27681.6609	27586.2069
27210.8844	27027.0270	26936.0269	26755.8528	26666.6667
26315.7895	26143.7908	25974.0260	25806.4516	25641.0256
25396.8254	25078.3699	25000.0000	24844.7205	24767.8019
24691.3580	24615.3846	24390.2439	24242.4242	24024.0240
23809.5238	23668.6391	23529.4118	23460.4106	23391.8129
23255.8140	23188.4058	22988.5057	22857.1429	22792.0228
22727.2727	22408.9636	22222.2222	22160.6648	22038.5675
21978.0220	21739.1304	21680.2168	21621.6216	21505.3763
21390.3743	21333.3333	21276.5957	21220.1592	21164.0212
21052.6316	20833.3333	20779.2208	20671.8346	20512.8205
20460.3581	20408.1633	20202.0202	20050.1253	20000.0000
19851.1166	19753.0864	19704.4335	19656.0197	19607.8431
19512.1951	19323.6715	19230.7692	19138.7560	19047.6190
18912.5296	18867.9245	18823.5294	18648.0186	18604.6512
18518.5185	18433.1797	18390.8046	18306.6362	18181.8182
18140.5896	18099.5475	18018.0180	17857.1429	17777.7778
17738.3592	17582.4176	17543.8596	17429.1939	17391.3043
17316.0173	17241.3793	17204.3011	17094.0171	17021.2766
16949.1525	16913.3192	16842.1053	16806.7227	16771.4885
16666.6667	16632.0166	16563.1470	16528.9256	16460.9053
16393.4426	16326.5306	16260.1626	16227.1805	16194.3320
16161.6162	16129.0323	16000.0000	15873.0159	15810.2767
15779.0927	15686.2745	15625.0000	15594.5419	15503.8760
15473.8878	15444.0154	15384.6154	15325.6705	15238.0952
15180.2657	15151.5152	15122.8733	15094.3396	15065.9134
15037.5940	15009.3809	14842.3006	14814.8148	14705.8824

Einstellbare Baudraten [Bit/s]				
14652.0147	14571.9490	14545.4545	14519.0563	14492.7536
14414.4144	14336.9176	14311.2701	14285.7143	14260.2496
14184.3972	14109.3474	14035.0877	13986.0140	13937.2822
13913.0435	13888.8889	13840.8304	13793.1034	13722.1269
13675.2137	13605.4422	13582.3430	13559.3220	13513.5135
13468.0135	13445.3782	13377.9264	13333.3333	13289.0365
13223.1405	13157.8947	13136.2890	13114.7541	13093.2897
13071.8954	13008.1301	12987.0130	12903.2258	12882.4477
12820.5128	12800.0000	12759.1707	12718.6010	12698.4127
12578.6164	12558.8697	12539.1850	12500.0000	12422.3602
12403.1008	12383.9009	12345.6790	12326.6564	12307.6923
12288.7865	12195.1220	12158.0547	12121.2121	12066.3650
12030.0752	12012.0120	11994.0030	11922.5037	11904.7619
11851.8519	11834.3195	11764.7059	11730.2053	11695.9064
11661.8076	11627.9070	11611.0305	11594.2029	11544.0115
11494.2529	11477.7618	11428.5714	11396.0114	11379.8009
11363.6364	11347.5177	11299.4350	11220.1964	11204.4818
11188.8112	11111.1111	11080.3324	11034.4828	11019.2837
10989.0110	10943.9124	10928.9617	10884.3537	10869.5652
10840.1084	10810.8108	10796.2213	10781.6712	10752.6882
10695.1872	10666.6667	10638.2979	10610.0796	10582.0106
10540.1845	10526.3158	10457.5163	10430.2477	10416.6667
10389.6104	10335.9173	10322.5806	10296.0103	10269.5764
10256.4103	10230.1790	10204.0816	10101.0101	10088.2724
10062.8931	10025.0627	10012.5156	10000.0000	9937.8882
9925.5583	9876.5432	9852.2167	9828.0098	9803.9216
9791.9217	9768.0098	9756.0976	9696.9697	9685.2300
9661.8357	9615.3846	9603.8415	9569.3780	9523.8095
9456.2648	9433.9623	9411.7647	9400.7051	9367.6815
9356.7251	9324.0093	9302.3256	9291.5215	9259.2593
9227.2203	9216.5899	9195.4023	9153.3181	9142.8571

Einstellbare Baudraten [Bit/s]				
9090.9091	9070.2948	9049.7738	9039.5480	9009.0090
8958.5666	8928.5714	8918.6176	8888.8889	8879.0233
8869.1796	8859.3577	8771.9298	8743.1694	8714.5969
8695.6522	8658.0087	8648.6486	8620.6897	8602.1505
8592.9108	8556.1497	8547.0085	8510.6383	8483.5631
8474.5763	8465.6085	8456.6596	8421.0526	8403.3613
8385.7442	8333.3333	8281.5735	8264.4628	8255.9340
8230.4527	8205.1282	8196.7213	8163.2653	8130.0813
8113.5903	8105.3698	8097.1660	8088.9788	8080.8081
8064.5161	8000.0000	7976.0718	7944.3893	7936.5079
7905.1383	7843.1373	7812.5000	7804.8780	7797.2710
7774.5384	7751.9380	7736.9439	7729.4686	7714.5612
7692.3077	7662.8352	7655.5024	7619.0476	7590.1328
7575.7576	7561.4367	7547.1698	7532.9567	7518.7970
7469.6545	7441.8605	7421.1503	7407.4074	7400.5550
7386.8883	7352.9412	7326.0073	7285.9745	7272.7273
7259.5281	7246.3768	7187.7808	7168.4588	7142.8571
7136.4853	7130.1248	7111.1111	7098.4916	7092.1986
7054.6737	7017.5439	6993.0070	6956.5217	6944.4444
6926.4069	6902.5022	6896.5517	6861.0635	6820.1194
6808.5106	6802.7211	6791.1715	6779.6610	6734.0067
6688.9632	6683.3751	6666.6667	6611.5702	6578.9474
6568.1445	6562.7564	6557.3770	6535.9477	6530.6122
6493.5065	6456.8200	6451.6129	6441.2238	6410.2564
6400.0000	6379.5853	6349.2063	6324.1107	6289.3082
6274.5098	6269.5925	6250.0000	6245.1210	6211.1801
6172.8395	6163.3282	6153.8462	6144.3932	6102.2121
6060.6061	6046.8632	6037.7358	5997.0015	5961.2519
5952.3810	5925.9259	5895.3574	5865.1026	5847.9532
5818.1818	5797.1014	5772.0058	5747.1264	5714.2857
5702.0670	5681.8182	5649.7175	5614.0351	5610.0982



Einstellbare Baudraten [Bit/s]				
5555.5556	5521.0490	5517.2414	5464.4809	5434.7826
5423.7288	5376.3441	5333.3333	5291.0053	5245.9016
5208.3333	5161.2903	5079.3651	5000.0000	

## A.4 Lizenzvertrag

Zwischen dem Käufer von *ADbasic* – nachfolgend „Lizenznehmer“ genannt – und Jäger Computergesteuerte Messtechnik GmbH, Rheinstraße 2 - 4, 64653 Lorsch – nachfolgend „Jäger Messtechnik GmbH“ genannt – besteht der folgende Lizenzvertrag:

### 1. GEGENSTAND DES LIZENZVERTRAGES

- 1.1 Gegenstand des Lizenzvertrages ist die Software des Compilers und Entwicklungssystems *ADbasic* (im folgenden als „*ADbasic*-Software“ bezeichnet) und die gedruckte Benutzerdokumentation mit der Bezeichnung „*ADbasic*: Das Echtzeit-Entwicklungstool für *ADwin*-Systeme“ (im folgenden als „zugehöriges Schriftmaterial“ bezeichnet).
- 1.2 Die Jäger Messtechnik GmbH macht darauf aufmerksam, dass es nach dem Stand der Technik nicht möglich ist, Computersoftware so zu erstellen, dass sie in allen Anwendungen und Kombinationen fehlerfrei arbeitet. Gegenstand des Lizenzvertrages ist nur eine Computersoftware, die im Sinne der Benutzerdokumentation grundsätzlich brauchbar ist.

### 2. UMFANG DER BENUTZUNG

- 2.1 Die Jäger Messtechnik GmbH gewährt dem Lizenznehmer das einfache, nicht ausschließliche und persönliche Nutzungsrecht. Dies bedeutet, dass die beiliegende Kopie der *ADbasic*-Software nur auf einem einzelnen Computer und nur an einem Ort benutzt werden darf. Der Lizenznehmer darf die *ADbasic*-Software in körperlicher Form (d. h. auf einem Datenträger abgespeichert) von einem Computer auf einen anderen Computer übertragen, vorausgesetzt, dass sie zu jedem Zeitpunkt immer nur auf einem einzelnen Computer genutzt wird. Eine weitergehende Nutzung ist nicht zulässig.
- 2.2 Programme, die durch den Lizenznehmer mit der *ADbasic*-Software erstellt werden, dürfen uneingeschränkt verteilt und weiterverwendet werden.

### 3. BESONDERE BESCHRÄNKUNGEN

Dem Lizenznehmer ist es untersagt,

- a) ohne vorherige schriftliche Einwilligung der Jäger Messtechnik GmbH die *ADbasic*-Software oder das zugehörige schriftliche Material an einen Dritten zu übergeben oder einem Dritten sonstwie zugänglich zu machen,

- b) die *ADbasic*-Software von einem Computer über ein Netz oder einen Datenübertragungskanal auf einen anderen Computer zu übertragen,
- c) ohne vorherige schriftliche Einwilligung der Jäger Messtechnik GmbH die *ADbasic*-Software abzuändern, zu übersetzen, zurückzuentwickeln, zu entkompilieren oder zu disassemblieren,

#### 4. INHABERSCHAFT AN RECHTEN

- 4.1 Der Lizenznehmer erhält mit dem Erwerb des Produktes nur Eigentum an dem körperlichen Datenträger, auf dem die *ADbasic*-Software aufgezeichnet ist. Ein Erwerb von Rechten an der *ADbasic*-Software selbst ist damit nicht verbunden.
- 4.2 Die Jäger Messtechnik GmbH behält sich insbesondere alle Veröffentlichungs-, Vervielfältigungs-, Bearbeitungs- und Verwertungsrechte an der *ADbasic*-Software vor.

#### 5. VERVIELFÄLTIGUNG

- 5.1 Die *ADbasic*-Software und das zugehörige Schriftmaterial sind urheberrechtlich geschützt.

Zu Sicherungszwecken ist dem Lizenznehmer das Anfertigen einer einzigen Reservekopie der *ADbasic*-Software erlaubt. Er ist verpflichtet, auf der Reservekopie den Urheberrechtsvermerk der Jäger Messtechnik GmbH anzubringen. Der in der *ADbasic*-Software vorhandene Urheberrechtsvermerk darf nicht entfernt werden.

- 5.2 Es wird ausdrücklich untersagt, die *ADbasic*-Software, wie auch das zugehörige Schriftmaterial, ganz oder teilweise in ursprünglicher oder abgeänderter Form oder in mit anderer Software zusammengemischter oder in anderer Software eingeschlossener Form zu kopieren oder anders zu vervielfältigen.

#### 6. ÜBERTRAGUNG DES BENUTZUNGSRECHTES

- 6.1 Das Recht zur Benutzung der *ADbasic*-Software kann nur mit vorheriger schriftlicher Einwilligung der Jäger Messtechnik GmbH an einen Dritten übertragen werden. Der Lizenznehmer hat dann die bei ihm installierte Software vollständig zu löschen und dem Dritten die Software vollständig (Original-Datenträger mit Dokumentation, einschließlich der Sicherungskopie) zu übergeben. Das Nutzungsrecht darf ferner nur auf einen Dritten übertragen werden, wenn sich der Dritte zu Gunsten der Jäger Messtechnik GmbH mit den Bestimmungen dieses

Lizenzvertrages und den allgemeinen Geschäftsbedingungen der Jäger Messtechnik GmbH einverstanden erklärt.

- 6.2 Vermietung und Verleihung der *ADbasic*-Software sind ausdrücklich untersagt.

## 7. DAUER DES VERTRAGES

- 7.1 Der Lizenzvertrag läuft auf unbestimmte Zeit.

- 7.2 Das Recht des Lizenznehmers zur Benutzung der *ADbasic*-Software erlischt automatisch ohne Kündigung, wenn er eine Bedingung dieses Lizenzvertrages verletzt. Bei Beendigung des Nutzungsrechtes ist er verpflichtet, den Original-Datenträger und alle Kopien der *ADbasic*-Software einschließlich etwaiger abgeänderter Exemplare sowie das zugehörige Schriftmaterial zu vernichten.

## 8. SCHADENSERSATZ UND VERTRAGSSTRAFE BEI VERTRAGS- VERLETZUNG

- 8.1 Verletzt der Lizenznehmer Bestimmungen dieses Lizenzvertrages, so ist er zum Schadensersatz verpflichtet.

- 8.2 Unbeschadet dessen wird bei Verletzung des Urheberrechts der Jäger Messtechnik GmbH, unbefugter Benutzung der Software und unbefugter Weitergabe der Software an Dritte, eine Vertragsstrafe von 20.000,- EURO für jeden Fall der Zuwiderverhandlung vereinbart.

- 8.3 Unterlassungsansprüche werden von den Schadensersatzansprüchen und Vertragsstrafen nicht berührt.

## 9. ÄNDERUNGEN UND AKTUALISIERUNGEN

Die Jäger Messtechnik GmbH ist berechtigt, Aktualisierungen der *ADbasic*-Software nach eigenem Ermessen zu erstellen. Die Jäger Messtechnik GmbH ist nicht verpflichtet, Aktualisierungen der *ADbasic*-Software dem Lizenznehmer zur Verfügung zu stellen.

Für umfangreiche Aktualisierungen behält sich die Jäger Messtechnik GmbH vor, einen Kostenbeitrag zu erheben.

## 10. GEWÄHRLEISTUNG UND HAFTUNG DER JÄGER MESSTECHNIK GMBH

- a) Die Jäger Messtechnik GmbH gewährleistet gegenüber dem Lizenznehmer, dass zum Zeitpunkt der Übergabe der Datenträger, auf dem die *ADbasic*-Software aufgezeichnet ist, unter normalen Betriebsbe-

dingungen und bei normaler Instandhaltung in seiner Materialausführung fehlerfrei ist.

- b) Sollte der Datenträger fehlerhaft sein, so kann der Lizenznehmer Ersatzlieferung während der Gewährleistungszeit von 6 Monaten ab Lieferung verlangen. Er muss dazu den Datenträger einschließlich einer Kopie der Rechnung/Quittung an die Jäger Messtechnik GmbH oder an den Vertriebspartner, von dem das Produkt bezogen wurde, zurückgeben.
- c) Wird ein Fehler im Sinne von Ziffer 10 b) nicht innerhalb angemessener Frist durch eine Ersatzlieferung behoben, so kann der Lizenznehmer nach seiner Wahl die Herabsetzung des Erwerbspreises oder das Rückgängigmachen des Lizenzvertrages verlangen. Weitere Ansprüche gegen die Jäger Messtechnik GmbH entstehen nicht.
- d) Aus den vorstehend unter Punkt 1.2 genannten Gründen übernimmt die Jäger Messtechnik GmbH keine Haftung für die Fehlerfreiheit der *ADbasic*-Software. Insbesondere übernimmt die Jäger Messtechnik GmbH keine Gewähr dafür, dass die *ADbasic*-Software den Anforderungen und Zwecken des Lizenznehmers genügt oder mit anderen von ihm ausgewählten Programmen zusammenarbeitet. Die Verantwortung für die richtige Auswahl und die Folgen der Benutzung der *ADbasic*-Software, sowie der damit beabsichtigten oder erzielten Ergebnisse trägt der Lizenznehmer. Das gleiche gilt für das die *ADbasic*-Software begleitende zugehörige Schriftmaterial.
- e) Jäger Messtechnik GmbH haftet nicht für Schäden, es sei denn, dass ein Schaden durch Vorsatz oder grobe Fahrlässigkeit seitens der Jäger Messtechnik GmbH verursacht worden ist. Eine Haftung wegen eventuell von der Jäger Messtechnik GmbH zugesicherten Eigenschaften bleibt unberührt. Eine Haftung für Mangelfolgeschäden, die nicht von der Zusicherung umfasst sind, ist ausgeschlossen.
- f) Die Jäger Messtechnik GmbH übernimmt keine Haftung für Schäden durch Viren, die mit dem Datenträger übertragen werden. Der Lizenznehmer ist angehalten, die Datenträger auf Viren zu überprüfen, bevor er die *ADbasic*-Software auf seinem Computer installiert.

## 11. SCHLUSSBESTIMMUNGEN

Die Unwirksamkeit einzelner Bestimmungen berührt die Wirksamkeit des Lizenzvertrages im übrigen nicht.

Ergänzend zu den Bestimmungen dieses Lizenzvertrages gelten die allgemeinen Geschäftsbedingungen der Jäger Messtechnik GmbH.

## A.5 Kommandozeilen-Aufruf

Der *ADbasic*-Compiler kann nicht nur in der Bedienoberfläche aktiviert werden, sondern auch direkt aus Windows oder DOS aufgerufen werden (sogenannter „Kommandozeilen“-Aufruf). Der Compiler arbeitet in beiden Fällen auf gleiche Weise, kann also eine Quelltext-Datei kompilieren und daraus eine Binär- oder Library-Datei erzeugen.



Der Compiler-Aufruf wird nur ausgeführt, wenn Sie in *ADbasic* Ihren License key bereits eingegeben haben.

Der Begriff und die Funktionalität „Kommandozeilen-Aufruf“ stammen noch aus der DOS-Zeit, in der man Befehle an das Betriebssystem DOS in einer Kommandozeile eingeben musste. Solche Eingaben sind auch unter Windows nach wie vor möglich.

Sie haben mehrere Möglichkeiten, Kommandozeilen unter Windows einzugeben:

- Öffnen Sie unter Windows ein MS-DOS-Fenster (Windows Start-Menü, Verzeichnis *Programs / Accessories*). Jede Eingabe ist hier eine Kommandozeile.



Der Compiler-Aufruf erfordert in jedem Fall die Windows-Umgebung. Der Aufruf funktioniert also nur aus diesem Windows-Fenster, nicht aus der originären DOS-Ebene.

- Wählen Sie im Start-Menü den Menüeintrag „Run“ und geben Sie im Eingabefenster eine Kommandozeile ein.
- Legen Sie für öfter benötigte Kommandozeilen-Aufrufe jeweils ein Icon auf dem Desktop an. Bei der Erstellung eines Icon geben Sie eine Kommandozeile direkt ein.

Ein oder mehrere Kommandozeilen-Aufrufe können in einer Batch-Datei `<*.bat>` zusammengefasst werden, z. B. um mehrere Quelltexte eines Projekts mit nur einem Aufruf zu kompilieren.

Bei jedem Aufruf müssen Sie die jeweils passenden Schalter und Parameter übergeben. Nicht alle Compiler-Einstellungen sind mit dem Kommandozeilen-Aufruf einstellbar.

### A.5.1 Syntax

Ein Kommandozeilen-Aufruf besteht mindestens aus dem Namen des aufgerufenen Programms und der zu kompilierenden Datei (jeweils mit Pfad und Dateiname). Daran können Schalter anschließen, die mit dem Zeichen „/“

beginnen und teilweise einen Parameter haben.  
Der Aufruf wird in einer einzelnen Zeile geschrieben.

### Syntax

```
{ [LW:\] [Pfad\]}ADbasic{.exe} /L /M  
{ [LW:\] [Pfad\]}infile{.bas} {/Sx} {/Px}  
{/A{ [LW:\] [Pfad\]}outfile}
```

### Schalter

[LW:\]	Optional: Laufwerk oder Festplatte. Für das Programm <ADbasic.exe> ist dies in der Regel „C:\“.
[Pfad\]	Optional: Unterverzeichnis, in dem sich das Programm <ADbasic.exe> oder der Quelltext befindet. Bei Standard-Installation ist dies C:\ADwin\ADbasic\.
infile	Dateiname des zu kompilierenden Quelltextes.
/L	Quelltext kompilieren und eine Library-Datei (Bibliothek) erzeugen mit der Endung „LIx“ (schließt Schalter /M aus). „x“      Prozessortyp, auf dem die kompilierte Datei laufen soll (siehe Schalter „/P“).
/M	Quelltext kompilieren und eine Binär-Datei erzeugen mit der Endung „Txn“ (schließt Schalter /L aus). „x“      Prozessortyp, auf dem die kompilierte Datei laufen soll (siehe Schalter „/P“). „n“      Prozessnummer der kompilierten Datei (wird aus der Quelltextdatei gelesen).
/Sx	ADwin-System einstellen, für das die Datei kompiliert wird: /SC      Karten (ISA-Bus; 16 bit resolution = No) /SL      Light-16 (16 bit resolution = Yes) /SG      Gold (16 bit resolution = Yes); Default /SP      Pro (16 bit resolution = Yes)

/Px	Prozessortyp, für den die Datei kompiliert wird:
/P2	Prozessor T2
/P4	Prozessor T4
/P5	Prozessor T5
/P8	Prozessor T8
/P9	Prozessor T9 (ADSP); Default
/P10	Prozessor T10 (ADSP)
/P11	Prozessor T11 (ADSP)
/Aoutfile	Pfad und Name der zu erzeugenden Binär- oder Library-Datei <outfile> .

### A.5.2 Bemerkungen

Optionale Angaben sind in geschweifte Klammern gesetzt. Die Reihenfolge der Schalter ist beliebig. In Kommandozeilen wird Groß-/Kleinschreibung nicht unterschieden.

Die Option "Debug Mode" ist immer ausgeschaltet, wenn über Kommandozeile kompiliert wird.

Wenn der Schalter /A nicht verwendet wird, wird die erzeugte Binär- oder Library-Datei in dem Verzeichnis gespeichert, in dem sich der Quelltext befindet.

Folgende Schalter schließen sich gegenseitig aus:

Schalter	dadurch ausgeschlossen:
/L	/M
/M	/L
/SG, /SL	/P2, /P4, /P5, /P8
/SP	/P2

### A.5.3 Beispiele



C:\ADwin\ADbasic\ADbasic.exe /L Z:\Myfiles\test.bas

Diese Kommandozeile kompiliert den Quelltext <test.bas> und erzeugt die Library-Datei <test.li9> im Verzeichnis <Z:\Myfiles\>.

Da nichts anderes angegeben ist, wird die Standardeinstellung verwendet:

- Prozessor T9
- System Gold (16 Bit System = Yes)
- erzeugte Datei im Verzeichnis der Quelldatei speichern



Wenn Sie sich beim Aufruf im Verzeichnis <C:\ADwin\ADbasic> befinden, können Sie obige Zeile kürzer schreiben:

```
ADbasic.exe /L Z:\Myfiles\test.bas
```

Die kürzeste Aufruf-Variante ergibt sich, wenn auch der Quelltext im Verzeichnis <C:\ADwin\ADbasic> liegt (hier ohne Dateinamen-Erweiterung):

```
ADbasic /L test
```

```
C:\ADwin\ADbasic\ADbasic /L Z:\Myfiles\test.bas /SL
```



Diese Kommandozeile kompiliert den Quelltext <string.bas> in eine Library-Datei für ein *Light-16*-System mit Prozessor T9.

Der gleiche Aufruf, nur für den Prozessor T10, sieht so aus:

```
C:\ADwin\ADbasic\ADbasic /L Z:\Myfiles\test.bas /P10 /SL
```

```
C:\ADwin\ADbasic\ADbasic /M Z:\Myfiles\test.bas
```



```
C:\ADwin\ADbasic\samples_ADwin\bas_dmo6f /P9 /SG
```

Kompiliert die Demo-Datei <bas\_dmo6f.bas> in eine Binär-Datei für ein Gold-System mit T9 als Prozessor.

```
C:\ADwin\ADbasic\ADbasic /M
```



```
C:\ADwin\ADbasic\samples_ADwin\bas_dmo6 /P8 /SL
```

Kompiliert die Demo-Datei <bas\_dmo6.bas> in eine Binär-Datei für eine Light-16-Karte mit dem Prozessor T8.

```
C:\ADwin\ADbasic\ADbasic /M C:\user\my_file.bas /P4 /SC  
/Ayour_file
```



Diese Anweisung kompiliert die Datei <my\_file.bas> für eine *ADwin*-Karte mit Prozessor T4. Die erzeugte Binärdatei hat den Namen <your\_file.T41> und befindet sich im gleichen Verzeichnis wie der Quelltext: <C:\user>.

```
C:\ADwin\ADbasic\ADbasic.exe /M C:\user\my_file.bas  
/AY:\somewhere\your_file
```



Die Binärdatei heißt nun <your\_file.T91> und befindet sich jetzt im Verzeichnis <Y:\somewhere>.

## A.5.4 Spezielle Einstellungen und Meldungen

### Spezielle Einstellungen



Beachten Sie bitte, woher der Compiler seine Informationen für die Kompilierung bezieht. Manche Compiler-Einstellungen können Sie nicht über die Kommandozeile angeben, sondern nur in der Entwicklungsumgebung *ADbasic*:

- Die Einstellungen „Event“, „Process“, „Priority“, „Optimize“ speichert die Entwicklungsumgebung (unsichtbar) in der Quelltext-Datei.
- Die Pfade für „Include-Directory“ und „Lib-Directory“ werden in der Registry gespeichert.

Besonders die Registry-Einträge können schnell geändert werden, wenn Sie oder ein anderer Benutzer an Ihrem PC ein anderes Projekt bearbeitet haben. Wir empfehlen daher, diese Einstellungen zu prüfen, bevor Sie einen Kommandozeilen-Aufruf vornehmen.

### Warn- und Fehlermeldungen

Treten während der Kompilierung Warnungen oder Fehler auf, so werden diese in den Dateien <filename.WRN> und <filename.ERR> gespeichert. Die Fehlermeldungen sind identisch mit denjenigen, die *ADbasic* im Infofenster ausgeben würde.

Wir empfehlen, vor dem Kompilieren Dateien mit Warn- und Fehlermeldungen zu löschen, damit Sie nach dem Kompiliervorgang einfach prüfen können, ob dieser fehlerlos abgelaufen ist.

## A.6 Obsolete Programmteile

Die Entwicklungsumgebung *ADbasic* 4 enthält aus Gründen der Kompatibilität auch Einstellmöglichkeiten für *ADwin*-Systeme mit Transputer-Prozessoren (T4, T5, T8) sowie Hilfsprogramme, für die es bereits einen besseren und aktuellen Ersatz gibt. Diese sind nachfolgend beschrieben.

### A.6.1 Dialogfenster „Process Options“

In diesem Dialogfenster legen Sie Compiler-Optionen für das gerade aktive Quelltextfenster fest, d.h. Sie bestimmen Eigenschaften desjenigen Prozesses, der aus dem aktiven Quelltext übersetzt und ins *ADwin*-System übertragen wird.

Sie müssen für jedes Quelltextfenster separat die nötigen Einstellungen vornehmen, indem Sie das Dialogfenster jeweils neu aufrufen (es sei denn, Sie möchten die Voreinstellung verwenden).

Wenn Sie im Dialogfenster „Compiler Options“ den Prozessortyp T4, T5 oder T8 eingestellt haben, wird das in Abb. 1 gezeigte Dialogfenster geöffnet.

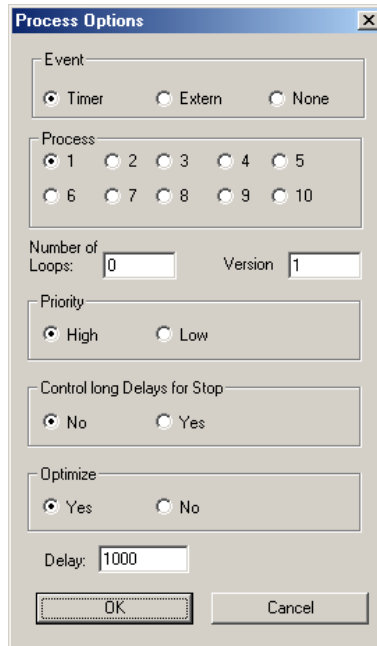


Abb. 1 – Dialogfenster „Process Options“ für Prozessoren T4...T8

- „Event“: Sie stellen hier ein, welches Event-Signal den Abschnitt **EVENT**: Ihres Prozesses starten soll.

Mit der Einstellung „Timer“ definieren Sie Impulse des internen Zählers als Event-Signal. In diesem Fall legen Sie mit der Systemvariablen **PROCESSDELAY** fest, in welchen Zeitabständen der Zähler ein Event-Signal auslöst.

Mit „Extern“ legen Sie fest, dass ein Signal am Event-Eingang Ihrer ADwin-Hardware den Prozess startet. Dies könnte beispielsweise ein Impuls eines Messaufnehmers sein. Ein solcher Prozess muss mit hoher Priorität ablaufen. Stellen Sie für diesen Fall die Option „Priority“ auf „High“.

Wie Sie bei einem *ADwin-Pro*-System einen externen Event-Eingang nutzen können, lesen Sie bitte in der zugehörigen Software-Dokumentation unter dem Befehl **EVENTENABLE** nach.


Mit der Einstellung „None“ startet der Prozess nach der Übertragung ins System sofort. Der Abschnitt **EVENT** wird, unabhängig von Event-Signalen, nach der Ausführung erneut gestartet (Endlos-Schleife). Insbesondere bei einem hochprioren Prozess müssen Sie dann dafür sorgen, dass der Prozess auch Rechenzeit für andere Aufgaben (z. B. Kommunikation mit dem PC) bereitstellt.

- „Process“: Stellen Sie die Nummer (1...10) ein, unter der der übertragene Prozess auf dem System angesprochen wird.

Wenn Sie mehrere Prozesse gleichzeitig auf einem *ADwin*-System ablaufen lassen, müssen Sie jedem Prozess eine eigene Nummer zuweisen.

- „Number of Loops“: Falls gewünscht, können Sie hier die Anzahl der Event-Durchläufe des Prozesses einstellen. Ist diese Anzahl erreicht, stoppt der Prozess automatisch. Eine geänderte Einstellung wird beim nächsten Start des Prozesses wirksam (also nicht bei einem laufenden Prozess), ohne dass Sie das Programm neu kompilieren müssen.

Wenn Sie den Wert „0“ eintragen, wird das Programm solange wiederholt, bis Sie den Prozess stoppen mit:

- dem Befehl **END**,
- dem Befehl **STOP\_PROCESS** oder
- der Schaltfläche  in der Entwicklungsumgebung.

- „Version“: Hier können Sie einen ganzzahligen Wert eingeben, um verschiedene Versionen Ihres Programms zu unterscheiden.
- „Priority“: Stellen Sie hier die Priorität des Prozesses ein, mit der er im System laufen soll. Weitere Informationen zu diesem Thema finden Sie in Kapitel 5.1 „Prozessverwaltung“. Der Eintrag „Level“ existiert für diesen Prozessortyp nicht.
- „Control long Delays for Stop“: Die Einstellung ist nur bei den Prozessoren T2...T8 verfügbar.

Das Stoppen eines Prozesses tritt verzögert ein, wenn er nur selten (Zykluszeit > 5 Millisekunden) aufgerufen wird. Wir empfehlen Ihnen in

diesem Fall, die Option zu aktivieren, denn sie beschleunigt den Stopp-Vorgang.

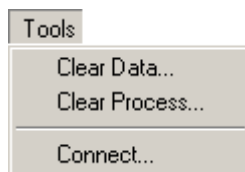
- „Optimize“: Die wahlweise einschaltbare Optimierung kann die Prozess-Ausführungszeit um bis zu 20% verkürzen. Eine höhere Einstellung unter „Level“ führt zu kürzeren Ausführungszeiten.

Wenn Sie unerwartete Compiler- oder Laufzeitfehler eines Prozesses feststellen, können Sie dies in Ausnahmefällen durch die Einstellung eines niedrigeren „Level“ beheben.

- „Delay“: Stellen Sie hier das Processdelay (Zykluszeit) ein, mit dem der Prozess beginnen soll.

### A.6.2 Der Menüeintrag „Connect“

Im Menü „Tools“ ist unter anderem der Menüeintrag „Connect“ enthalten. Dieser öffnet das gleichnamige Dialogfenster, in dem Sie Einstellungen für das (von uns nicht mehr aktualisierte) Programm *ADserver* vornehmen können. Sie richten damit eine Netzwerk-Verbindung zu einem *ADwin*-System ein.



Wir empfehlen Ihnen, das Programm *ADwin TCPIP-server* anstelle des Programms *ADserver* zu verwenden. Nehmen Sie für diesen Fall keine Einstellungen im Dialogfenster vor, sondern schließen Sie es! Sie finden nähere Informationen z. B. in der Online-Hilfe von *ADwin TCPIP-server*.

Wenn Sie dennoch das Programm *ADserver* verwenden möchten:

Sie können aus *ADbasic* über ein vorhandenes Netzwerk (LAN, ISDN, Internet, ...) auf ein *ADwin*-System zugreifen, das an einen beliebigen Netzwerkrechner angeschlossen ist. Auf diesem Netzwerkrechner muss zunächst das Programm *ADserver* gestartet werden. Anschließend geben Sie die Netzwerkdaten im Dialogfenster an.

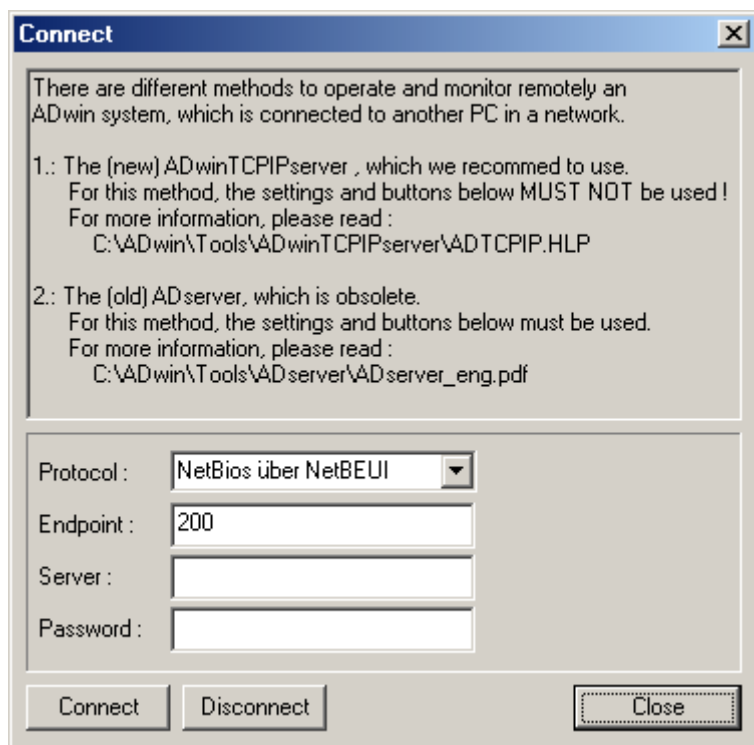


Abb. 2 – Das Dialogfenster „Connect“

- „Protocol“: Das vom Netzwerk verwendete Protokoll. Dieses muss auf Ihrem Rechner ordnungsgemäß installiert sein.
- „Endpoint“: Endpunkt für die Netzwerk-Kommunikation.
- „Server“: Name bzw. Adresse des Netzwerk-Rechners, zu dem Sie eine Verbindung aufbauen möchten.
- „Password“: Passwort zum Zugriff auf das Programm *ADserver*. Bei der Überprüfung des Passwortes wird auf Groß-/Kleinschreibung geachtet.

Die Einstellungen „Protocol“, „Endpoint“ und „Password“ müssen mit der Einstellung im Programm *ADserver* auf dem Netzwerkrechner identisch sein.

Sobald Sie die Schaltfläche „Connect“ drücken, wird die Verbindung zum Netzwerk-Rechner und dem dort angeschlossenen *ADwin*-System aufgebaut. Alle weiteren Aktionen der Entwicklungsumgebung werden an dieses System übertragen. Durch Drücken der Schaltfläche „Disconnect“ unterbrechen Sie diese Verbindung wieder.

## A.7 Index

### Symbole

- · 101
- # · 106
- #DEFINE · 127
- #ELSE · 156
- #ENDIF · 156
- #IF · 156
- #INCLUDE · 160
- \* · 102
- + · 99
- + (String) · 100
- .NET · 95
- / · 103
- : · 107
- < = > · 109
- = · 108
- ^ · 104
- ' (REM) · 195

### Zahlen

- 150h, *siehe* Device No.
- 2-dimensionale Felder · 57
- 40 Bit-Genauigkeit · 48

### A

- Abbruch *siehe* Prozess beenden
- ABSF · 110
- ABSI · 111
- Absolutwert
  - Fließkomma-Zahlen · 110
  - Ganze Zahlen · 111
- ActiveX · 95
  - aus Entwicklungsumgebung nutzen · 95
  - Kommunikation zum ADwin-System · 95

- ADbasic
  - Lizenzvertrag · 8
  - starten · 5
  - Verzeichnisse einstellen · 25
- ADC · 236
- ADC12, ADC14 · 239
- Add Open Files to Project · 35
- Add to Project · 10, 35
- Addition · 99
- ADtools · 40
- ADWIN\_SYSTEM · 156
- ADwin32.dll · 94
- aktivieren, Trace-Modus · 228
- Analoge Ein-/Ausgänge
  - ADC: Messung durchführen · 236
  - DAC: Wert ausgeben · 246
  - Gewandelten Wert lesen
    - 12 Bit, 14 Bit · 254
    - 16 Bit · 253
  - Multiplexer setzen · 258
  - Wandlung starten · 260
  - Wandlungsende abwarten · 262
- Analyse
  - Allgemein · 78
  - Laufzeitfehler · 78
  - Programmablauf · 81
  - Zeitverhalten · 78
- AND · 112
- Anhalten *siehe* Prozess beenden
- Anmerkungen *siehe* Kommentar
- Anzahl von Prozessen · 79
- Anzeige
  - aktuelle Informationen · 8
  - Auslastung: CPU, PM, EM, DM, DX · 40
  - Prozessoptionen · 7
- Arcus-Cosinus: ARCCOS · 114
- Arcus-Sinus: ARCSIN · 115
- Arcus-Tangens: ARCTAN · 116



## Arithmetische Funktionen

- · 101
- \* · 102
- + · 99
- / · 103
- ^ · 104
- DEC · 126
- EXP · 137
- INC · 159
- LN · 172
- LOG · 175
- SQRT · 207

Array-Index (lokal) zu groß / < 1, *siehe* Laufzeitfehler, erkennen

Arrays, *siehe* Felder

(DIM) AS · 129

ASC · 117

ASCII-Zeichensatz · 2

(DIM ...) AT · 129

Aufbau eines ADbasic-Programms · 44

## Auslastung

- Anzeige · 40
- Definition · 92
- Einfluss der Prozess-Anzahl · 79

## Automatische

Typkonvertierung · 66

## B

Backslash im String, Escape-Sequenz · 63

Basis e · 137

Baudraten für CAN-Bus · 3

Bearbeitungszeit messen · 71

Bedienoberfläche · 6

## Bedingter Sprung

- IF ... THEN · 154
- SELECTCASE · 198

Befehls-Separator (:) · 107

## Befehlszeile

- Groß-/Kleinschreibung · 43
- Zeilenlänge
  - mit #INCLUDE · 160
  - Standard · 43

## Berechnungsausdrücke

- auswerten · 64
- getrennte Auswertung · 67

Betriebsmodus L16 einstellen · 370

## Betriebssystem

- laden, *siehe* Booten
- Standardverzeichnis · 5
- Verzeichnis einstellen · 25

## Bibliothek

- Allgemeines · 69
- Einbinden · 158
- Erzeugen
  - aus ADbasic · 16
  - aus Kommandozeile · 13
- Funktion · 164
- Position im Programm · 46
- Rekursion verboten · 70
- Unterprogramm · 168
- Verzeichnis einstellen · 25

## Binärdatei

- siehe auch* Bibliothek
- siehe* Menü, Build
- Erzeugen
  - aus ADbasic · 16
  - aus Kommandozeile · 13
- nutzen in
  - Entwicklungsumgebung · 95

Binäre Schreibweise · 50

## Bits verschieben

- nach links · 201
- nach rechts · 202

Booten · 5

BTL-Datei: Verzeichnis einstellen · 25

Busy-Anzeige · 40

**C**

- C#.NET, C++ · 95
- CAN\_MSG
  - Gold CAN · 328
- CAN\_MSG (L16 DIO1) · 312
- CAN-Bus
  - Baudraten · 3
- CAN-Bus (L16 DIO1)
  - CAN\_MSG · 312
  - EN\_INTERRUPT · 314
  - EN\_RECEIVE · 315
  - EN\_TRANSMIT · 317
  - GET\_CAN\_REG · 318
  - INIT\_CAN · 319
  - READ\_MSG · 320
  - SET\_CAN\_BAUDRATE · 322
  - SET\_CAN\_REG · 324
  - TRANSMIT · 325
- Carriage Return im String, Escape-Sequenz · 63
- CASE, CCASE, CASEELSE (SELECTCASE ...) · 198
- CAST\_FLOATTOLONG · 118
- CAST\_LONGTOFLOAT · 119
- CHECK\_SHIFT\_REG · 344
- CHR · 120
- CLEAR\_DIGOUT · 242
- CNT\_CLEAR · 267
- CNT\_CLEARENABLE · 269
- CNT\_ENABLE · 271
- CNT\_GETSTATUS · 273
- CNT\_INPUTMODE · 276
- CNT\_LATCH · 278
- CNT\_MODE · 280
- CNT\_READ · 282
- CNT\_READFLATCH · 286
- CNT\_READLATCH · 284
- CNT\_RESETSTATUS · 288
- CNT\_SE\_DIFF · 290

- CNT\_SET · 292
- Code size · 39
- Comment Block · 10
- Compiler
  - Aufruf · 15
  - Kommandozeilen-Aufruf · 12
  - Optionen einstellen · 17
  - Statusmeldung · 39
  - Wait for stop · 15
- Compiler-Anweisungen · 106
  - #DEFINE · 127
  - #IF ... THEN · 156
  - #INCLUDE · 160
- CONF\_DIO · 244
- CONF\_DIO\_E · 294
- Controlblock · 10
- Cosinus: COS · 121
- CPU\_SLEEP · 122
- Cursor-Position · 40

**D**

- DAC · 246
- DATA\_n
  - dimensionieren · 129
  - Globale Felder · 51
  - Globale Felder, 2-dimensional · 57
  - Übersicht · 124
- Data-Index (global) zu groß / <1,  
*siehe* Laufzeitfehler, erkennen
- Dateiname
  - Bibliothek · 16
  - Binärdatei · 16
- Datenaustausch
  - mit dem PC · 94
  - mit der
    - Entwicklungsumgebung · 95
    - zwischen Prozessen · 94

- Datenspeicher
  - siehe auch* Speicher
  - 2-dim. Felder im ~ · 57
  - Übersicht, intern, extern · 55
  - Zusatzbedarf durch
    - Debug-Modus · 78
    - Timing-Modus · 81
    - Trace-Modus · 81
- Datenspeicher, intern (DM) · 56
- Datenstrukturen
  - FIFO · 58
  - Globale Felder · 51
  - Globale Felder, 2-  
dimensional · 57
  - Globale Variablen · 50
  - Lokale Variablen und Felder · 53
  - Übersicht · 47
- Datentypen
  - String · 61
  - Typkonvertierung · 65
  - Übersicht · 48
- Datenverlust
  - beim Booten · 5
  - FIFO · 59
- Datenwort: Nummerierung von  
Bits · 2
- deaktivieren
  - Trace-Modus · 227
- Debug
  - Allgemein · 78
  - Debug-Modus anwenden · 78
  - Menü · 26
  - Timing-Fenster · 26
  - Timing-Modus aktivieren · 26
  - Timing-Modus anwenden · 78
  - TRACE\_MODE\_PAUSE · 227
  - TRACE\_MODE\_RESUME · 228
  - Trace-Fenster · 30
  - Trace-Modus anwenden · 81
  - Trace-Modus einrichten · 30
- DEC · 126
- DEFINE *siehe* #DEFINE
- Dekadischer Logarithmus · 175
- Deklaration, *siehe* Dimensionierung
- Dekrementieren · 126
- Delphi · 95
- Device No.
  - Definition · 95
  - einstellen · 18
- Dezimale Schreibweise · 49
- Dezimaltrennzeichen · 49
- DIAdem · 95
- DIGIN · 247
- DIGIN\_LONG\_E (L16 DIO2) · 299
- DIGIN\_WORD · 249
- DIGIN\_WORD1\_E (L16-DIO) · 295
- DIGIN\_WORD2\_E (L16  
DIO1) · 297
- Digitale Ein-/Ausgänge
  - Alle Ausgänge setzen · 251
  - Alle Eingänge lesen · 249
  - Einen Ausgang löschen · 242
  - Einen Ausgang setzen · 256
  - Einen Eingang lesen · 247
  - Konfigurieren · 244
- DIGOUT\_LONG\_E (L16  
DIO2) · 311
- DIGOUT\_RESET1\_E (L16  
DIO1) · 301
- DIGOUT\_RESET2\_E (L16  
DIO1) · 303
- DIGOUT\_SET1\_E (L16  
DIO1) · 305
- DIGOUT\_SET2\_E (L16  
DIO1) · 307
- DIGOUT\_WORD · 251
- DIGOUT\_WORD1\_E (L16  
DIO1) · 309
- DIGOUT\_WORD2\_E (L16 DIO1,  
DIO2) · 310
- DIM · 129

Dimensionierung  
  Befehl DIM · 129  
  Position im Programm · 45  
  Speicherbereich · 54  
Directory *siehe* Verzeichnis  
Disable Trace · 10  
Division  
  durch 2 · 202  
  einfache · 103  
Division durch Null, *siehe* Laufzeit-  
  fehler, erkennen  
DM, *siehe* Datenspeicher, intern  
DM\_LOCAL (DIM ...) · 129  
DO ... UNTIL · 132  
DRAM\_EXTERN (DIM ...) · 129  
DX, *siehe* Externer Speicher

**E**

e-Funktion: EXP · 137  
Einbinden  
  Include-Datei · 160  
  Library · 158  
Einschwingzeit, *siehe*  
  Multiplexer · 258  
ELSE (IF ...) · 154  
EM, *siehe* Zusatzspeicher  
EM\_LOCAL (DIM ...) · 129  
EN\_CAN\_INTERRUPT · 330  
EN\_INTERRUPT (L16 DIO1) · 314  
EN\_RECEIVE  
  Gold CAN · 331  
EN\_RECEIVE (L16 DIO1) · 315  
EN\_TRANSMIT  
  Gold CAN · 333  
EN\_TRANSMIT (L16 DIO1) · 317  
Enable Trace · 10  
END · 133  
ENDFUNCTION · 151

ENDIF (IF ...) · 154  
ENDSELECT (SELECTCASE  
  ...) · 198  
ENDSUB · 223  
Entwicklungsumgebung  
  Leisten und Fenster · 6  
  starten · 5  
  Tastaturkürzel · 1  
Escape-Sequenz · 63  
Event  
  externes Signal: löschen · 196  
  verlorenes Signal  
    ein Prozess zeitgesteuert · 92  
    extern gesteuerter  
      Prozess · 93  
    mehrere Prozesse  
      zeitgesteuert · 93  
    prüfen · 28  
    Zeitdifferenz messen · 72  
EVENT: · 45, 134  
EXIT · 136  
Exklusiv-ODER-Verknüpfung · 233  
Exponential-Funktion EXP · 137  
Exponential-Schreibweise · 50  
Externer Speicher (DX) · 56  
externer Speicher (SDRAM) · 55

**F**

F1: Hilfe aufrufen · 9  
Farbe einstellen · 23  
Fehler  
  durch Cut&Paste erzeugt · 14  
  geringere Optimierung  
    einstellen · 21  
Fehlermeldung  
  Laufzeitfehler · 32  
  Time out · 88  
  Wait for stop · 15

- Felder
    - 2-dimensional · 57
    - DATA\_n · 124
    - FIFO · 138
    - globale · 51
      - erstes Element · 52
    - initialisieren · 45
    - kopieren · 178
    - lokale · 53
      - erstes Element · 54
    - Speicherbereich festlegen · 54
    - Übersicht · 47
  - Fenster
    - Compiler Options · 17
    - Info-Fenster · 39
    - Parameter · 36
    - Process Options · 20
    - Projekt · 35
    - Quelltext-Informationen · 7
    - Quelltext-Statusleiste · 7
    - Statusleiste · 40
    - Übersicht · 6
  - FFT · 379
  - FFT\_CALC · 389
  - FFT\_CALC\_DM · 391
  - FFT\_CALC\_DX · 393
  - FFT\_INIT · 388
  - FFT\_MAG · 383
  - FFT\_MAG\_SCALE · 387
  - FFT\_PHASE · 385
  - FFT\_SCALE · 381
  - FIFO
    - Aufbau der Datenstruktur · 58
    - belegte Elemente abfragen · 143
    - Datenverlust · 59
    - dimensionieren · 129
    - Elementanzahl prüfen · 59
    - freie Elemente abfragen · 142
    - initialisieren · 140
    - Übersicht · 138
  - FIFO\_CLEAR · 140
  - FIFO\_EMPTY · 142
  - FIFO\_FULL · 143
  - FINISH: · 45, 144
  - Fließkomma-Zahlen
    - Dezimale Schreibweise · 49
    - Exponential-Schreibweise · 50
    - Wertebereich · 48
  - FLO40TOSTR · 147
  - Float, *siehe* Fließkomma-Zahlen
  - FLOTOSTR · 145
  - Font einstellen · 23
  - FOR ... NEXT · 149
  - Fourier-Transformation
    - FFT · 379
    - FFT\_CALC · 389
    - FFT\_CALC\_DM · 391
    - FFT\_CALC\_DX · 393
    - FFT\_INIT · 388
    - FFT\_MAG · 383
    - FFT\_MAG\_SCALE · 387
    - FFT\_PHASE · 385
    - FFT\_SCALE · 381
  - FPAR\_n, globale Variablen · 50
  - FUNCTION · 151
  - Funktion
    - Allgemeines zu Bibliotheken · 69
    - Allgemeines zu Makros · 68
    - Bibliothek
      - (LIB\_FUNCTION) · 164
    - Makro · 151
    - Position im Programm · 46
- ## G
- Ganze Zahlen
    - Binäre Schreibweise · 50
    - Hexadezimale Schreibweise · 50
    - Typkonvertierung · 65
    - Wertebereich · 48
  - Gerätenummer, Definition · 95
  - GET\_CAN\_REG
    - Gold CAN · 335

GET\_CAN\_REG (L16 DIO1) · 318  
GET\_RS · 346  
gleich = · 109  
GLOBALDELAY · 190  
globale Felder, *siehe* Felder, globale  
le  
globale Variablen, *siehe* Variablen, globale  
Groß-/Kleinschreibung · 9  
größer als >, >= · 109

## H

Halt *siehe* Prozess beenden  
Hardware-Zugriff  
Lesen · 188  
Schreiben · 189  
Hexadezimale Schreibweise · 50  
Hilfe  
für *ADbasic*-Befehle (F1) · 9  
kontextsensitiv · 6  
Hilfsprogramme (*ADtools*<>) · 40

## I

IEEE-Floating point-Format · 48  
IF · 154  
*siehe auch* #IF · 156  
IMPORT · 158  
INC · 159  
INCLUDE · 160  
Include-Datei  
Allgemeines · 69  
einbinden · 160  
Verzeichnis einstellen · 25  
Info-Fenster · 39  
INIT: · 45, 162  
INIT\_CAN  
Gold CAN · 336  
INIT\_CAN (L16 DIO1) · 319

Initialisierung  
Booten · 5  
umfangreiche · 45  
Inkrementieren · 159  
Installation,  
Standardverzeichnis · 5  
Integer, Typkonvertierung · 65  
Interner Speicher  
Datenspeicher (DM) · 56  
Gesamt (SRAM) · 55  
Programm (PM) · 56  
Zusatzspeicher (EM) · 56

## K

kleiner als <, <= · 109  
Kommandozeilen-Aufruf · 12  
Kommentar · 195  
Kommunikation  
mit dem PC · 94  
mit der  
Entwicklungsumgebung · 95  
Prozess im ADwin-System · 88  
Time out · 88  
zwischen Prozessen · 94  
kompilieren *siehe* Compiler  
Kontextmenü · 10  
Projektfenster · 35  
Kontrollstrukturen · 68  
Konvertierung, *siehe*  
Wandlung · 260

## L

L16: Betriebsmodus  
einstellen · 370  
L16\_MODE · 370  
latency (Timing-Fenster) · 27  
Laufzeiten von Befehlen · 77

Laufzeitfehler  
     *siehe auch* Debug-Modus  
 Anzeige · 32  
     erkennen · 78  
 length (Timing-Fenster) · 27  
 LIB\_ENDFUNCTION · 164  
 LIB\_ENDSUB · 168  
 LIB\_FUNCTION · 164  
 LIB\_SUB · 168  
 Library  
     *siehe auch* Bibliothek  
     Funktion · 164  
     IMPORT · 158  
     Unterprogramm · 168  
 Lib-Verzeichnis einstellen · 25  
 Line Feed im String, Escape-  
     Sequenz · 63  
 Lizenzvertrag · 8  
 LN · 172  
 LNGTOSTR · 173  
 LOG · 175  
 Logarithmus  
     dekadischer · 175  
     natürlicher · 172  
 Logische Funktionen  
     AND · 112  
     NOT · 181  
     OR · 182  
     SHIFT\_LEFT · 201  
     SHIFT\_RIGHT · 202  
     XOR · 233  
 Long, *siehe* Ganze Zahlen  
 LOWINIT: · 45, 176

## M

Makro  
     Allgemeines · 68  
     Funktion · 151  
     Position im Programm · 46  
 Mark Controlblock · 10  
 Matlab · 95

Matrix, 2-dimensional · 57  
 Maximale Zeilenlänge  
     mit #INCLUDE · 160  
     Standard · 43  
 MEMCOPY · 178  
 Menü  
     auswählen · 7  
     Build · 15  
     Debug · 26  
     Edit · 14  
     File · 13  
     Help · 35  
     Leiste · 12  
     Options · 17  
     Tools · 34  
     View · 15  
     Window · 34  
 Menüleiste · 12  
 Messwertverlauf darstellen · 40  
 Multiplexer: setzen · 258  
 Multiplikation  
     einfache · 102  
     mit 2 · 201

## N

Nachkommastellen  
     abschneiden · 66  
 Namen, lokale Variablen · 54  
 Natürlicher Logarithmus · 172  
 negatives Vorzeichen · 65  
 NEXT (FOR ...) · 149  
 NICHT · 181  
 niederrpriore Prozesse beim  
     T11 · 91  
 NOP · 180  
 NOT · 181

## O

ODER-Verknüpfung · 182

- Operatoren
    - auswerten · 64
    - Exklusiv-ODER · 233
    - negatives Vorzeichen · 65
    - Priorität · 65
  - Optimales Zeitverhalten
    - ein Prozess · 80
    - mehrere Prozesse · 79
  - Optimierung
    - siehe auch* Debug
    - Allgemein · 71
    - Bearbeitungszeit messen · 71
    - Konstanten statt Variablen · 73
    - Polynom schneller
      - berechnen · 105
    - Registerzugriff · 72
    - schneller messen · 73
    - T11 Speicherzugriff · 77
    - Wartezeit einstellen · 74
    - Wartezeit nutzen · 76
  - Optionen einstellen
    - Allgemein (Settings) · 22
    - Compiler · 17
    - Editor · 22
    - Prozess · 20
    - Sprache · 24
    - strukturierte
      - Befehlsdarstellung · 23
    - Verzeichnisse · 25
  - OR · 182
  - Ordner *siehe* Verzeichnis
- P**
- P1\_SLEEP · 184
  - P2\_SLEEP · 186
  - PAR\_n, globale Variablen · 50
  - Parameter, *siehe* Variablen, globale
  - Parameterfenster · 36
  - PEEK · 188
  - PM, *siehe* Programmspeicher
  - POKE · 189
  - Polynom, schneller
    - berechnen · 105
  - Potenz · 104
    - in Polynom ersetzen · 105
    - zur Basis e · 137
  - Präprozessor-Anweisungen · 106
    - #DEFINE · 127
    - #IF ... THEN · 156
    - #INCLUDE · 160
  - Priorität
    - niederpriorie Prozesse mit T11 · 91
    - Operatoren · 65
    - Prozess, *siehe* Prozess, Priorität von Prozessen prüfen · 79
  - PROCESSDELAY · 190
  - Programm verbessern, *siehe* Optimierung
  - Programmablauf verfolgen · 81
  - Programmabschnitte
    - Event: · 45
    - Finish: · 45
    - Init: · 45
    - Lowinit: · 45
    - Übersicht · 45
  - Programmspeicher
    - Zusatzbedarf durch
      - Debug-Modus · 78
      - Timing-Modus · 81
      - Trace-Modus · 81
  - Programmspeicher (PM) · 56



- Programmstruktur
  - Übersicht · 68
  - Bibliothek
    - LIB\_FUNCTION · 164
    - LIB\_SUB · 168
    - Übersicht · 69
  - Include-Datei · 69
  - Kommentar REM · 195
  - Makros
    - Funktion FUNCTION · 151
    - Übersicht · 68
    - Unterprogramm SUB · 223
  - Schleife
    - DO ... UNTIL · 132
    - FOR ... NEXT · 149
  - Sprung
    - IF ... THEN · 154
    - SELECTCASE · 198
- Projektfenster · 35
- Projektverwaltung
  - Allgemeines · 11
  - benutzte Variablen
    - markieren · 37
  - Fenster · 35
- Prozess
  - Anzahl · 86
  - Bearbeitungszeit · 89
  - beenden
    - andere · 211
    - sich selbst (in EVENT:) · 133
    - sich selbst (in INIT:, LOWINIT:, FINISH:) · 136
  - Betriebszustände beim
    - Zeitverhalten · 92
  - Kommunikation zwischen
    - ~en · 94
  - Kommunikationsprozess · 88
  - mehrere · 90
  - Optimales Zeitverhalten
    - ein Prozess · 80
    - mehrere Prozesse · 79
  - Optionen
    - einstellen · 20
  - Optionen, Anzeige · 7
  - Priorität
    - Hoch · 87
    - Kommunikation · 88
    - Niedrig · 87
    - Niedrig mit T11 · 91
    - Übersicht · 86
  - prüfen auf Anzahl, Priorität · 79
  - Standardprozesse 11, 12 · 87
  - starten
    - anderen Prozess · 208
    - gleichen Prozess erneut · 197
    - verzögert · 209
  - Status ermitteln · 193
  - Zeitverhalten · 88
  - Zyklus, *siehe* Prozesszyklus
- Prozess optimieren, *siehe* Optimierung
- PROZESSn\_RUNNING · 193
- PROZESSOR · 156

Prozess-Steuerung  
  END · 133  
  EXIT · 136  
  PROZESSn\_RUNNING · 193  
  RESET\_EVENT · 196  
  RESTART\_PROCESS · 197  
  START\_PROCESS · 208  
  START\_PROCESS\_DELAYED · 209  
  STOP\_PROCESS · 211  
Prozesszyklus  
  Aufruf  
    mit Event-Signal · 85  
    Zeiteinheit · 89  
    zeitlich exakter Aufruf · 90  
  Punkt vor Strich, *siehe* Operatoren

## Q

Quadratwurzel · 207  
Quelltext  
  erstellen · 9  
  im Projekt verwenden · 35  
  strukturiert darstellen · 9  
Quelltext-Statusleiste · 7

## R

READ\_FIFO · 347  
READ\_MSG  
  Gold CAN · 337  
READ\_MSG (L16 DIO1) · 320  
READ\_TIMER · 194  
READADC · 253  
READADC12 · 254  
Rechenzeit sparen  
  Konstanten statt Variablen · 73  
  Registerzugriff · 72  
  schneller messen · 73  
  Wartezeit einstellen · 74  
  Wartezeit nutzen · 76  
Registerzugriff · 72

Rekursion bei Bibliothek · 70  
REM · 195  
RESET\_EVENT · 196  
RESTART\_PROCESS · 197  
Ringspeicher · 58  
RS\_INIT · 348  
RS\_RESET · 350  
RS485\_SEND · 351

## S

Save All Files of Project · 35  
(Bits) schieben  
  nach links · 201  
  nach rechts · 202  
Schreibweise von Zahlen · 49  
Schriftart einstellen · 23  
SDRAM, *siehe* Externer Speicher  
SELECTCASE · 198  
Separator · 107  
SEQ\_INIT  
  L16 Rev. B · 371  
SEQ\_READ  
  L16 Rev. B · 374  
SET\_CAN\_BAUDRATE  
  Gold CAN · 339  
SET\_CAN\_BAUDRATE (L16 DIO1) · 322  
SET\_CAN\_REG  
  Gold CAN · 341  
SET\_CAN\_REG (L16 DIO1) · 324  
SET\_DIGOUT · 256  
SET\_MUX · 258  
SET\_RS · 353  
SHIFT\_LEFT · 201  
SHIFT\_RIGHT · 202  
Short-Cuts · 1  
Sinus: SIN · 204  
SLEEP · 205

- Speicher
  - siehe auch* Datenspeicher
  - Auslastung · 40
  - Bedarf bestimmen · 39
  - Bereich festlegen · 54
  - Bereiche (PM, DM, EM, DX) · 55
  - String · 61
  - Zusatzbedarf durch
    - Bibliotheken · 70
    - Debug-Modus · 78
    - Makros · 68
    - Timing-Modus · 81
    - Trace-Modus · 81
- Sprung, bedingter
  - IF ... THEN · 154
  - SELECTCASE · 198
- SQRT · 207
- SRAM, *siehe* Interner Speicher, Gesamt
- SSI\_MODE · 356
- SSI\_READ · 358
- SSI\_SET\_BITS · 360
- SSI\_SET\_CLOCK · 362
- SSI\_START · 364
- SSI\_STATUS · 366
- Stack size · 39
- START\_CONV · 260
- START\_PROCESS · 208
- START\_PROCESS\_DELAYED · 209
- Starten, ADbasic · 5
- Statusleiste · 40
- Statusmeldung Compiler · 39
- STEP (FOR ...) · 149
- Steuerzeichen in Strings · 63
- STOP\_PROCESS · 211
- Stopp *siehe* Prozess beenden
- STRCOMP · 215
- String · 213
  - Definition des Datentyps · 49
  - Escape-Sequenz · 63
  - nicht empfohlene Zuweisung · 64
  - Steuerzeichen · 63
  - Variablenaufbau · 61
  - Werte normal zuweisen · 62
- String-Anweisung
  - Addition · 100
  - ASCII-Wert in Zeichen · 120
  - Dimensionierung · 213
  - Float in String · 145
  - Float in String (40 Bit) · 147
  - Länge eines Strings · 218
  - Long in String · 173
  - String in Float · 229
  - String in Long · 231
  - Teilstring
    - linker · 216
    - mittlerer · 219
    - rechter · 221
  - Vergleichen · 215
  - Zeichen in ASCII-Wert · 117
- STRLEFT · 216
- STRLEN · 218
- STRMID · 219
- STRRIGHT · 221
- Strukturieren
  - Farbige Befehlsdarstellung · 9
  - Programmabschnitte · 68
  - Zeilen einrücken · 10
- SUB · 223
- Subtraktion · 101
- Systemvariablen
  - PROCESSDELAY · 190
  - PROZESSn\_RUNNING · 193
  - Übersicht · 53

**T**

- T11
  - niederpriorie Prozesse · 91
  - Wartezeit einstellen · 74
- Tabulator
  - im String, Escape-Sequenz · 63
  - Schrittweite einstellen · 22
- Tangens: TAN · 226
- Tastatur
  - Anzeige von Einstellungen · 40
  - Kürzel · 1
- Terminierung *siehe* Prozess beenden
- Testpoint · 95
- THEN (IF ...) · 154
- Time out · 88
- Timer *siehe* Zähler
- Timing, *siehe* Zeitverhalten
- Timing-Modus
  - aktivieren · 26
  - anwenden · 78
  - Fenster · 26
  - zusätzliche Prozessorzeit · 81
- TO (FOR ...) · 149
- Tools
  - TBin · 40
  - TButton · 40
  - TDigit · 40
  - TFifo · 40
  - TGraph · 40
  - TLed · 40
  - TMeter · 40
  - TPar\_FPar · 40
  - TPoti · 40
  - TProcess · 40
- TRACE\_MODE\_PAUSE · 227
- TRACE\_MODE\_RESUME · 228

## Trace-Modus

- aktivieren · 228
- anwenden · 81
- deaktivieren · 227
- einrichten · 30
- Fenster · 30
- im Programm anwenden · 83
- Informationen aktualisieren · 83
- TRACE\_MODE\_PAUSE · 227
- TRACE\_MODE\_RESUME · 228
- zusätzliche Prozessorzeit und Speicherbedarf · 81

## TRANSMIT

- Gold CAN · 342
  - TRANSMIT (L16 DIO1) · 325
  - Transputer, Einstellungen · 16
- Trigonometrische Funktionen
- ARCCOS · 114
  - ARCSIN · 115
  - ARCTAN · 116
  - COS · 121
  - SIN · 204
  - TAN · 226

## Typkonvertierung

- ASCII-Wert in Zeichen · 120
- automatische · 65
- Float in Long (nur Datentyp) · 118
- Float in String · 145
- Float in String (40 Bit) · 147
- Long in Float (nur Datentyp) · 119
- Long in String · 173
- String in Float · 229
- String in Long · 231

**U**

- Uncomment Block · 10
- UND-Verknüpfung · 112
- ungleich <> · 109
- Unmark Controlblock · 10

Unterprogramm  
 Allgemeines zu Bibliotheken · 69  
 Allgemeines zu Makros · 68  
 Bibliothek (LIB\_SUB) · 168  
 Makro (SUB) · 223  
 Position im Programm · 46  
 UNTIL (DO ...) · 132

## V

VALF · 229  
 VALI · 231

Variablen  
 Anzeige · 36  
 benutzte markieren · 37  
 globale · 50  
   kopieren, große Anzahl · 178  
 initialisieren · 45  
 Initialisierung beim Booten · 5  
 lokale · 53  
   Länge des Namens · 54  
   Speicherbereich festlegen · 54  
 Übersicht · 47  
 vordefinierte Namen · 47  
 Werte hexadezimal anzeigen · 37  
*siehe auch* Systemvariablen  
 Verarbeitungszeiten · 77  
 Vergleich  
   < = > · 109  
   Strings · 215  
 Verzeichnis bei  
   Standardinstallation · 5  
 Verzeichnisse einstellen · 25  
 Visual Basic · 95

## W

Wagenrücklauf im String, Escape-  
 Sequenz · 63  
 WAIT\_EOC · 262

Wandlung  
 starten · 260

Warten  
 P1\_SLEEP: Pro I-Bus · 184  
 P2\_SLEEP: Pro II-Bus · 186  
 Prozessor T11:  
   CPU\_SLEEP · 122  
 Prozessor: NOP · 180  
 SLEEP: Prozessor bis T10 · 205  
 Wartezeit genau einstellen · 74

Werkzeuge (ADtools<>)> · 40  
 Werkzeugleiste · 7  
 Wertebereich · 48  
 Workspace size · 39  
 WRITE\_FIFO · 354  
 Wurzel · 207  
   aus negativer Zahl, *siehe* Lauf-  
   zeitfehler, erkennen

## X

XOR · 233

## Z

Zahlenwerte  
 Schreibweise · 49

Zähler  
 auslesen · 194  
 interner, Taktzyklus · 89

Zeichenkette *siehe* String

Zeilenlänge, max.  
 mit #INCLUDE · 160  
 Standard · 43

Zeilenvorschub im String, Escape-  
 Sequenz · 63

Zeit  
 exakter Aufruf · 90  
 Zykluszeit · 88  
 Zeitoptimierung, *siehe* Optimierung

- Zeitverhalten
  - Änderung durch
    - Debug-Modus · 78
    - Timing-Modus · 81
    - Trace-Modus · 81
  - Betriebszustände
    - allgemein · 92
    - ein Prozess zeitgesteuert · 92
    - extern gesteuerter Prozess · 93
    - mehrere Prozesse
      - zeitgesteuert · 93
- Informationen abfragen · 80
  - optimales
    - ein Prozess · 80
    - mehrere Prozessen · 79
  - prüfen, optimieren · 78
- Zusatzspeicher (EM) · 56
- zuweisen, Zahlen · 49
- Zuweisung (=) · 108

### A.8 Befehle für ADwin-Gold-Systeme

#### Symbole

< = > (Vergleich)  
+ (Addition)  
+ (String-Addition)  
- (Subtraktion)  
\* (Multiplikation)  
/ (Division)  
^ (Potenz)  
= (Zuweisung)  
: Doppelpunkt  
#DEFINE  
#IF ... THEN ... {#ELSE ...} #ENDIF  
#INCLUDE  
#..., Präprozessor-Anweisung

#### A

ABSF  
ABSI  
ADC  
ADC12, ADC14  
AND  
ARCCOS  
ARCSIN  
ARCTAN  
ASC

#### C

CAN\_MSG (CAN)  
CAST\_FLOATTOLONG  
CAST\_LONGTOFLOAT  
CHECK\_SHIFT\_REG (CAN)  
CHR  
CLEAR\_DIGOUT  
CNT\_CLEAR (CO1)  
CNT\_ENABLE (CO1)  
CNT\_GETSTATUS (CO1)  
CNT\_INPUTMODE (CO1)  
CNT\_LATCH (CO1)

CNT\_MODE (CO1)  
CNT\_READ (CO1)  
CNT\_READFLATCH (CO1)  
CNT\_READLATCH (CO1)  
CNT\_RESETSTATUS (CO1)  
CNT\_SET (CO1)  
CNT\_SE\_DIFF (CO1)  
CONF\_DIO  
COS

#### D

DAC  
DATA\_n  
DEC  
DIGIN  
DIGIN\_WORD  
DIGOUT\_WORD  
DIM  
DO ... UNTIL

#### E

END  
EN\_CAN\_INTERRUPT (CAN)  
EN\_RECEIVE (CAN)  
EN\_TRANSMIT (CAN)  
EVENT:  
EXIT  
EXP

#### F

FFT  
FFT\_CALC  
FFT\_INIT  
FFT\_MAG  
FFT\_MAG\_SCALE  
FFT\_PHASE  
FFT\_SCALE  
FIFO  
FIFO\_CLEAR

FIFO\_EMPTY  
FIFO\_FULL  
FINISH:  
FLOTOSTR  
FLOTOSTR\_X  
FOR ... TO ... {STEP ...} NEXT  
FUNCTION ... ENDFUNCTION

**G-L**

GET\_CAN\_REG (CAN)  
GET\_RS (CAN)  
IF ... THEN ... {ELSE ...} ENDIF  
IMPORT  
INC  
INIT:  
INIT\_CAN (CAN)  
LIB\_FUNCTION ...  
LIB\_ENDFUNCTION  
LIB\_SUB ... LIB\_ENDSUB  
LN  
LNGTOSTR  
LOG  
LOWINIT:

**M-Q**

NOP  
NOT  
OR  
PEEK  
POKE  
PROCESSDELAY  
PROZESSn\_RUNNING

**R**

READADC  
READADC12  
READ\_FIFO (CAN)  
READ\_MSG (CAN)  
READ\_TIMER  
REM

RESET\_EVENT  
RS485\_SEND (CAN)  
RS\_INIT (CAN)  
RS\_RESET (CAN)

**S**

SELECTCASE  
SET\_CAN\_BAUDRATE (CAN)  
SET\_CAN\_REG (CAN)  
SET\_DIGOUT  
SET\_MUX  
SET\_RS (CAN)  
SHIFT\_LEFT  
SHIFT\_RIGHT  
SIN  
SLEEP  
SQRT  
SSI\_MODE (CAN)  
SSI\_READ (CAN)  
SSI\_SET\_BITS (CAN)  
SSI\_SET\_CLOCK (CAN)  
SSI\_START (CAN)  
SSI\_STATUS (CAN)  
START\_CONV  
START\_PROCESS  
STOP\_PROCESS  
STRCOMP  
String " "  
STRLEFT  
STRLEN  
STRMID  
STRRIGHT  
SUB ... ENDSUB

**T-Z**

TAN  
TRACE\_MODE\_PAUSE  
TRACE\_MODE\_RESUME  
TRANSMIT (CAN)  
VALF



VALI  
WAIT\_EOC

WRITE\_FIFO (CAN)  
XOR

**A.9 Befehle für ADwin-light-16-Systeme****Symbole**

< = > (Vergleich)  
 + (Addition)  
 + (String-Addition)  
 - (Subtraktion)  
 \* (Multiplikation)  
 / (Division)  
 ^ (Potenz)  
 = (Zuweisung)  
 : Doppelpunkt  
 #DEFINE  
 #IF ... THEN ... {#ELSE ...} #ENDIF  
 #INCLUDE  
 #..., Präprozessor-Anweisung

**A**

ABSF  
 ABSI  
 ADC  
 AND  
 ARCCOS  
 ARCSIN  
 ARCTAN  
 ASC

**C**

CAN\_MSG (nur DIO1)  
 CAST\_FLOATTOLONG  
 CAST\_LONGTOFLOAT  
 CHR  
 CLEAR\_DIGOUT  
 CNT\_CLEAR  
 CNT\_CLEARENABLE (DIO1, DIO2)  
 CNT\_ENABLE  
 CNT\_GETSTATUS (DIO1, DIO2)  
 CNT\_INPUTMODE (DIO1, DIO2)  
 CNT\_LATCH

CNT\_MODE (DIO1, DIO2)  
 CNT\_READ  
 CNT\_READFLATCH (DIO1, DIO2)  
 CNT\_READLATCH  
 CNT\_SET (DIO1, DIO2)  
 CONF\_DIO\_E (DIO1, DIO2)  
 COS

**D**

DAC  
 DATA\_n  
 DEC  
 DIGIN  
 DIGIN\_LONG\_E (DIO2)  
 DIGIN\_WORD  
 DIGIN\_WORD1\_E (DIO1, DIO2)  
 DIGIN\_WORD2\_E (DIO1, DIO2)  
 DIGOUT\_LONG\_E (DIO2)  
 DIGOUT\_RESET1\_E (DIO1, DIO2)  
 DIGOUT\_RESET2\_E (DIO1, DIO2)  
 DIGOUT\_SET1\_E (DIO1, DIO2)  
 DIGOUT\_SET2\_E (DIO1, DIO2)  
 DIGOUT\_WORD  
 DIGOUT\_WORD1\_E (DIO1, DIO2)  
 DIGOUT\_WORD2\_E (DIO1, DIO2)  
 DIM  
 DO ... UNTIL

**E**

END  
 EN\_INTERRUPT (nur DIO1)  
 EN\_RECEIVE (nur DIO1)  
 EN\_TRANSMIT (nur DIO1)  
 EVENT:  
 EXIT  
 EXP

### F

FFT  
FFT\_CALC  
FFT\_INIT  
FFT\_MAG  
FFT\_MAG\_SCALE  
FFT\_PHASE  
FFT\_SCALE  
FIFO  
FIFO\_CLEAR  
FIFO\_EMPTY  
FIFO\_FULL  
FINISH:  
FLOTOSTR  
FLOTOSTR\_X  
FOR ... TO ... {STEP ...} NEXT  
FUNCTION ... ENDFUNCTION

### G-L

GET\_CAN\_REG (nur DIO1)  
IF ... THEN ... {ELSE ...} ENDIF  
IMPORT  
INC  
INIT:  
INIT\_CAN (nur DIO1)  
L16\_MODE (Rev. B)  
LIB\_FUNCTION ...  
LIB\_ENDFUNCTION  
LIB\_SUB ... LIB\_ENDSUB  
LN  
LNGTOSTR  
LOG  
LOWINIT:

### M-Q

NOP  
NOT  
OR  
PEEK  
POKE

PROCESSDELAY  
PROZESSn\_RUNNING

### R

READADC  
READ\_MSG (nur DIO1)  
READ\_TIMER  
REM  
RESET\_EVENT

### S

SELECTCASE  
SEQ\_INIT (Rev. B)  
SEQ\_READ (Rev. B)  
SET\_CAN\_BAUDRATE (nur DIO1)  
SET\_CAN\_REG (nur DIO1)  
SET\_DIGOUT  
SET\_MUX  
SHIFT\_LEFT  
SHIFT\_RIGHT  
SIN  
SLEEP  
SQRT  
SSI\_MODE (DIO2)  
SSI\_READ (nur DIO2)  
SSI\_SET\_BITS (nur DIO2)  
SSI\_SET\_CLOCK (nur DIO2)  
SSI\_START (nur DIO2)  
SSI\_STATUS (nur DIO2)  
START\_CONV  
START\_PROCESS  
STOP\_PROCESS  
STRCOMP  
String " "  
STRLEFT  
STRLEN  
STRMID  
STRRIGHT  
SUB ... ENDSUB

**T-Z**

TAN

TRACE\_MODE\_PAUSE

TRACE\_MODE\_RESUME

TRANSMIT (nur DIO1)

VALF

VALI

WAIT\_EOC

XOR

### A.10 Befehle für ADwin-Pro-Systeme

Die folgende Übersicht enthält nur diejenigen Befehle, die direkt in einem Pro-CPU-Modul bearbeitet werden.

Alle Hardware-bezogenen Befehle für das ADwin-Pro-System finden Sie (aus Platzgründen) in der separaten Dokumentation „ADwin-Pro Software“.

#### Symbole

< = > (Vergleich)  
+ (Addition)  
+ (String-Addition)  
- (Subtraktion)  
\* (Multiplikation)  
/ (Division)  
^ (Potenz)  
= (Zuweisung)  
: Doppelpunkt  
#DEFINE  
#IF ... THEN ... {#ELSE ...} #ENDIF  
#INCLUDE  
#..., Präprozessor-Anweisung

#### A

ABSF  
ABSI  
AND  
ARCCOS  
ARCSIN  
ARCTAN  
ASC

#### C

CAST\_FLOATTO LONG  
CAST\_LONGTO FLOAT  
CHR  
COS  
CPU\_SLEEP

#### D

DATA\_n  
DEC  
DIM  
DO ... UNTIL

#### E

END  
EVENT:  
EXIT  
EXP

#### F

FFT  
FFT\_CALC  
FFT\_CALC\_DM  
FFT\_CALC\_DX  
FFT\_INIT  
FFT\_MAG  
FFT\_MAG\_SCALE  
FFT\_PHASE  
FFT\_SCALE  
FIFO  
FIFO\_CLEAR  
FIFO\_EMPTY  
FIFO\_FULL  
FINISH:  
FLO40TOSTR  
FLOTOSTR  
FOR ... TO ... {STEP ...} NEXT  
FUNCTION ... ENDFUNCTION

**G-L**

IF ... THEN ... {ELSE ...} ENDIF  
IMPORT  
INC  
INIT:  
LIB\_FUNCTION ...  
LIB\_ENDFUNCTION  
LIB\_SUB ... LIB\_ENDSUB  
LN  
LNGTOSTR  
LOG  
LOWINIT:

**M-Q**

MEMCPY  
NOP  
NOT  
OR  
P1\_SLEEP  
P2\_SLEEP  
PEEK  
POKE  
PROCESSDELAY  
PROZESSn\_RUNNING

**R**

READ\_TIMER  
REM  
RESET\_EVENT  
RESTART\_PROCESS

**S**

SELECTCASE  
SHIFT\_LEFT  
SHIFT\_RIGHT  
SIN  
SLEEP  
SQRT  
START\_PROCESS  
START\_PROCESS\_DELAYED  
STOP\_PROCESS  
STRCOMP  
String " "  
STRLEFT  
STRLEN  
STRMID  
STRRIGHT  
SUB ... ENDSUB

**T-Z**

TAN  
TRACE\_MODE\_PAUSE  
TRACE\_MODE\_RESUME  
VALF  
VALI  
XOR

<b>Symbole</b>		CNT_RESETSTATUS	288	FFT_PHASE	385
< = > (Vergleich)	109	CNT_SET	292	FFT_SCALE	381
+ (Addition)	99	CNT_SE_DIFF	290	FIFO	138
+ (String-Addition)	100	CONF_DIO	244	FIFO_CLEAR	140
- (Subtraktion)	101	CONF_DIO_E	294	FIFO_EMPTY	142
* (Multiplikation)	102	COS	121	FIFO_FULL	143
/ (Division)	103	CPU_SLEEP	122	FINISH:	144
^ (Potenz)	104	<b>D</b>		FLO40TOSTR	147
= (Zuweisung)	108	DAC	246	FLOTOSTR	145
: Doppelpunkt	107	DATA_n	124	FOR ... TO ... {STEP ...}	
" " (String)	213	DEC	126	NEXT	149
#DEFINE	127	DIGIN	247	FUNCTION ... END-	
#IF ... THEN ... {#ELSE		DIGIN_LONG_E	299	FUNCTION	151
...} #ENDIF	156	DIGIN_WORD	249	<b>G-J</b>	
#INCLUDE	160	DIGIN_WORD1_E	295	GET_CAN_REG	
#..., Präprozessor-Anwei-		DIGIN_WORD2_E	297	Gold CAN	335
sung	106	DIGOUT_LONG_E	311	L16 DIO1	318
<b>A-B</b>		DIGOUT_RESET1_E	301	GET_RS	346
ABSF	110	DIGOUT_RESET2_E	303	IF ... THEN ... {ELSE ...}	
ABSI	111	DIGOUT_SET1_E	305	ENDIF	154
ADC	236	DIGOUT_SET2_E	307	IMPORT	158
ADC12, ADC14	239	DIGOUT_WORD	251	INC	159
AND	112	DIGOUT_WORD1_E	309	INIT:	162
ARCCOS	114	DIGOUT_WORD2_E	310	INIT_CAN	
ARCSIN	115	DIM	129	Gold CAN	336
ARCTAN	116	DO ... UNTIL	132	L16 DIO1	319
ASC	117	<b>E-F</b>		<b>K-L</b>	
<b>C</b>		END	133	L16_MODE	370
CAN_MSG		EN_CAN_INTERRUPT		LIB_FUNCTION ... LIB_	
Gold CAN	328	Gold CAN	330	ENDFUNCTION	164
L16 DIO1	312	EN_INTERRUPT		LIB_SUB ... LIB_END-	
CAST_FLOATTOLONG		L16 DIO1	314	SUB	168
118		EN_RECEIVE		LN	172
CAST_LONGTOFLOAT		Gold CAN	331	LNGTOSTR	173
119		L16 DIO1	315	LOG	175
CHECK_SHIFT_REG	344	EN_TRANSMIT		LOWINIT:	176
CHR	120	Gold CAN	333	<b>M-R</b>	
CLEAR_DIGOUT	242	L16 DIO1	317	NOP	180
CNT_CLEAR	267	EVENT:	134	NOT	181
CNT_CLEARENABLE	269	EXIT	136	OR	182
CNT_ENABLE	271	EXP	137	P1_SLEEP	184
CNT_GETSTATUS	273	FFT	379	P2_SLEEP	186
CNT_INPUTMODE	276	FFT_CALC	389	PEEK	188
CNT_LATCH	278	FFT_CALC_DM	391	POKE	189
CNT_MODE	280	FFT_CALC_DX	393	PROCESSDELAY	190
CNT_READ	282	FFT_INIT	388	PROZESSn_RUNNING	
CNT_READFLATCH	286	FFT_MAG	383	193	
CNT_READLATCH	284	FFT_MAG_SCALE	387	READADC	253

---

READADC12	254	SET_CAN_REG	215
READ_FIFO	347	Gold CAN	216
READ_MSG		L16 DIO1	218
Gold CAN	337	SET_DIGOUT	219
L16 DIO1	320	SET_MUX	221
READ_TIMER	194	SET_RS	223
REM	195	SHIFT_LEFT	201
RESET_EVENT	196	SHIFT_RIGHT	202
RESTART_PROCESS		SIN	204
197		SLEEP	205
RS485_SEND	351	SQRT	207
RS_INIT	348	SSI_MODE	356
RS_RESET	350	SSI_READ	358
<b>S</b>		SSI_SET_BITS	360
SELECTCASE	198	SSI_SET_CLOCK	362
SEQ_INIT	371	SSI_START	364
SEQ_READ	374	SSI_STATUS	366
SET_CAN_BAUDRATE		START_CONV	260
Gold CAN	339	START_PROCESS	208
L16 DIO1	322	START_PROCESS_	
		DELAYED	209
		STOP_PROCESS	211
		" " (String)	213
		STRCOMP	215
		STRLEFT	216
		STRLEN	218
		STRMID	219
		STRRIGHT	221
		SUB ... ENDSUB	223
		<b>T-Z</b>	
		TAN	226
		TRACE_MODE_PAUSE	227
		TRACE_MODE_RESU-	
		ME	228
		TRANSMIT	
		Gold CAN	342
		L16 DIO1	325
		VALF	229
		VALI	231
		WAIT_EOC	262
		WRITE_FIFO	354
		XOR	233