ADbasic

Echtzeit-Entwicklungstool für *ADwin*-Systeme

***ADbasic* Version 6.00**

Jul. 2022

License Key:

ADwin - die schnellsten Echtzeitsysteme unter Windows

Hier finden Sie immer einen Ansprechpartner für Ihre Fragen:

Hotline:	(0 62 51) 9 63 20
Fax:	(0 62 51) 9 63 299
E-Mail:	info@ADwin.de
Internet	www.ADwin.de

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

Inhaltsverzeichnis

Inhaltsverzeichnis III

Konventionen 2

1 Einführung 3

2 Neues in *ADbasic 6* 5

3 Entwicklungsumgebung nutzen 9
3.1 Grundlegende Schritte 9
3.1.1 Entwicklungsumgebung erstmals starten 9
3.1.2 Lizenzschlüssel prüfen und ändern 10
3.1.3 ADwin-Betriebssystem laden 11
3.1.4 Grundelemente der Entwicklungsumgebung 12
3.2 Quelltexte erstellen 16
3.2.1 Online-Hilfe aufrufen 17
3.2.2 Kontextmenü im Quelltextfenster 18
3.2.3 Editor-Leiste 19
3.3 Quelltext übersetzen 20
3.4 Quelltext formatieren 21
3.4.1 Syntax hervorheben 21
3.4.2 Automatisch formatieren 22
3.4.3 Textzeilen einrücken 22
3.4.4 Kommentare positionieren 22
3.4.5 Zeilen in Kommentar ändern 23
3.4.6 Eigene Befehle und Variablen dokumentieren 23
3.4.7 Faltbaren Textbereich definieren 26
3.4.8 Textbereiche ein- / ausfalten 27
3.5 Suchen, markieren und ersetzen 28
3.5.1 Text schnell suchen 28
3.5.2 Text suchen und ersetzen 28
Beispiele für das Suchen von Text 32
Beispiele für das Ersetzen von Text 33
3.5.3 Reguläre Ausdrücke 34

3.5.4 Kontrollstruktur markieren 36
3.5.5 Textmarken nutzen 36
3.5.6 Zu einer Programmzeile springen 37
3.5.7 Zur Deklaration eines Befehls oder Variablen springen 37
3.5.8 Zwischen Sprungzielen wechseln 38
3.6 Programme leichter schreiben 39
3.6.1 Befehle und Variablen automatisch vervollständigen . . 39
3.6.2 Textbausteine einfügen 40
3.6.3 Befehlsparameter anzeigen 41
3.6.4 Deklaration eines Befehls oder Variablen anzeigen . . . 42
3.6.5 Deklarationen einer Datei anzeigen 42
3.6.6 Verwendete globale Variablen und Felder anzeigen . . . 42
3.6.7 Aufrufkürzel zu Dateien / Ordnern einfügen 43
3.7 Programmfehler finden 44
3.8 Bootloader nutzen 44
3.8.1 Bootloader programmieren 44
3.8.2 Prozesse aus dem Bootloader zurücklesen 46
3.8.3 Bootloader aktivieren / deaktivieren 46
3.9 Projekte verwalten 47
3.10 Menüs 51
3.10.1 Das Menü „File“ 52
3.10.2 Das Menü „Edit“ 53
3.10.3 Das Menü „View“ 53
3.10.4 Das Menü „Build“ 54
3.10.5 Das Menü „ADsim“ 55
3.10.6 Das Menü „Options“ 56
Dialogfenster „Compiler Options“ 56
Dialogfenster „Process Options“ 59
Dialogfenster „Settings“ 62
Dialogfenster „Project Settings“ 66
3.10.7 Das Menü „Debug“ 69
Option Timing Analyzer 69
Option Debug mode 69
Option Stack Test 71
Stack Test Analysis 72
Create Call tree Info File 72
Show Call tree 72

3.10.8 Das Menü „Tools“ 73
3.10.9 Das Menü „Window“ 74
3.10.10 Das Menü „Help“ 74
3.11 Fenster 76
3.11.1 Toolbox 76
3.11.2 Projektfenster 76
3.11.3 Parameterfenster 79
3.11.4 Prozessfenster 81
3.11.5 Quelltextfenster 83
3.11.6 Statusleiste 84
3.11.7 Fenster Call Tree 85
3.11.8 Fenster Stack Test Analysis 87
3.12 Info-Bereich 89
3.12.1 Infofenster 89
3.12.2 ToDo-Liste 91
3.12.3 Fenster „Timing Analyzer“ 91
3.12.4 Fenster „Global Variables“ 95
3.12.5 Fenster „Declarations“ 96
3.13 ADtools 98
3.14 *ADbasic* und *ADsim* 100

4 Prozesse programmieren 103
4.1 Programmaufbau 103
4.1.1 Die Programmabschnitte 105
4.1.2 Befehle für den Hardware-Zugriff 105
4.1.3 Benutzerdefinierte Befehle und Variablen 106
4.2 Variablen und Felder 108
4.2.1 Übersicht 108
4.2.2 Datenstrukturen 109
4.2.3 Datentypen 110
4.2.4 Zahlenwerte eingeben 113
4.2.5 Mit Bitmustern arbeiten 114
4.2.6 Globale Variablen (Parameter) 116
4.2.7 Globale Felder (Arrays) 117
4.2.8 System-Variablen 119
4.2.9 Lokale Variablen und Felder 120
4.3 Variablen und Felder – Details 121

4.3.1 Variablen und Felder im Datenspeicher 121
4.3.2 Speicherbereiche (T9…T11) 122
4.3.3 Speicherbereiche (T12/T12.1) 124
4.3.4 2-dimensionale Felder 126
4.3.5 Die Datenstruktur FIFO 127
4.3.6 Strings 129
Normale Zuweisung. 130
Zuweisung per Escape-Sequenz. 132
Nicht empfohlene Arten der Zuweisung 133
4.4 Berechnungsausdrücke 134
4.4.1 Auswertung von Operatoren 134
4.4.2 Typkonvertierung 135
4.5 Bedingungen, Schleifen und Module 138
4.5.1 Unterprogramm- und Funktions-Makros 138
4.5.2 Include-Dateien 139
4.5.3 Bibliotheken (Libraries) 139

5 Prozesse optimieren 143
5.1 Bearbeitungszeit messen 143
5.1.1 Besonderheiten beim T12/T12.1 145
5.2 Verschiedene Tipps 146
5.2.1 Zugriff auf Hardware-Adressen 146
5.2.2 Konstanten anstelle von Variablen 146
5.2.3 Schnellere Messfunktion 147
5.2.4 Wartezeit genau einstellen 147
5.2.5 Wartezeiten nutzen 150
5.2.6 Optimierung mit dem Prozessor T11 151
5.3 Debugging und Analyse 152
5.3.1 Laufzeitfehler erkennen (Debug-Modus) 152
5.3.2 Zeitverhalten prüfen (Timing-Modus) 153
Anzahl und Priorität von Prozessen prüfen 154
Optimales Zeitverhalten eines Prozesses 155
5.3.3 Stack-Überlauf vermeiden (Stack Test) 156
5.3.4 Programmstruktur analysieren 156
5.3.5 Speicherintensive Prozeduren optimieren 157

6 Prozesse im Betriebssystem 159
6.1 Prozessverwaltung. 160
6.1.1 Prozessarten 160
6.1.2 Prozesse mit der Priorität „Hoch“ 161
6.1.3 Prozesse mit der Priorität „Niedrig“ 161
6.1.4 Kommunikationsprozess 162
6.1.5 Speicherfragmentierung 162
6.2 Zeitverhalten von Prozessen 164
6.2.1 Processdelay 164
6.2.2 Zeitlich exakter Aufruf von Prozesszyklen 165
6.2.3 Niederpriore Prozesse bei T11 und T12/T12.1 166
6.2.4 Auslastung des ADwin-Systems 167
6.2.5 Verschiedene Betriebszustände im Betriebssystem . . .168
6.3 Kommunikation 170
6.3.1 Datenaustausch zwischen Prozessen 170
6.3.2 Kommunikation zwischen PC und ADwin-System 171
6.3.3 Die Device No 172
6.3.4 Kommunikation mit Entwicklungsumgebungen 172

7 Befehlsreferenz 173
7.1 Befehlssyntax. 173
7.2 Befehle L16, X-A20, Gold, Pro. 174
7.3 Mathematik-Befehle 333
7.4 FFT-Library 335

8 Was tun bei Problemen? 367

Anhang A-1
A.1 Tastaturkürzel in ADbasic A-1
A.2 ASCII-Zeichensatz. A-4
A.3 Lizenzvertrag A-5
A.4 Kommandozeilen-Aufruf A-9
A.5 Obsolete Programmteile A-17
A.6 Liste der Debug-Fehlermeldungen A-20
A.7 Index A-22
A.8 Befehle in diesem Handbuch. A-43

Liebe Leserin, lieber Leser!

ADbasic 6 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.
Wir setzen voraus, dass Sie Ihr *ADwin*-System bereits installiert haben; anderenfalls beachten Sie bitte das Handbuch ADwin-Installation.

Die Entwicklungsumgebung *ADbasic* 6 bietet den gewohnten Komfort bei der Handhabung und beim Editieren und unterstützt die neuen Prozessoren T12 und T12.1 (siehe „Neues in ADbasic 6“ auf). In der Befehlsreferenz finden Sie die Befehle für Rechenoperationen des Prozessors. Alle Befehle für Zugriffe auf Ein- und Ausgänge oder Schnittstellen sind im jeweiligen Handbuch der *ADwin*-Hardware beschrieben.

Wenn Sie *ADbasic* das erste Mal verwenden, empfehlen wir Ihnen den schnellen Einstieg mit Kapitel 1 und 4. 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 3 beschreibt die Bedienung der 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:

Das Achtung-Zeichen 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 ähnlich dargestellt wie in 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:

Bitnr.	15	14	13	…	1	0
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.

Konzept des *ADwin*-Systems

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 4 ist erklärt, wie Sie *ADbasic*-Programme aufbauen.

Mit nur wenigen Anweisungen können Sie beispielsweise:

- Messgrößen bis zu Abtastfrequenzen von 800 kHz erfassen
- schnelle digitale Regler mit Abtastraten bis zu 400 kHz 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 6.

Prozesse entwickeln in *ADbasic*

Die Entwicklungsumgebung *ADbasic* unterstützt Sie bei der Umsetzung Ihrer Aufgabe in einen Prozess. Zunächst erstellen Sie den Quelltext in einer erwei-

terten Basic-Syntax; mit den Befehlen und Funktionen können Sie die Hardware Ihres *ADwin*-Systems komfortabel programmieren (siehe Kapitel 4).

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 3).

Ein *ADwin*-System vom PC aus bedienen

Die Oberfläche *ADbasic* wird nicht mehr benötigt, sobald die Echtzeit-Prozesse zu Ihrer Zufriedenheit laufen.
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 Kallisté, TestPoint, LabVIEW, Diadem, HP-VEE, Intouch und Matlab nutzen.
Schließlich stehen auch Treiber für die Plattformen Linux, MacIntosh und Java zur Verfügung.

Aus Simulink® heraus können Sie Echtzeitprozesse für ein *ADwin*-System erzeugen: mit dem Programmpaket *ADsim* lassen Sie Simulink®-Modelle auf *ADwin*-Hardware laufen. Dabei können Sie vom PC aus Modellwerte und *ADwin*-Variablen beobachten und ändern, entweder mit der Bedienoberfläche *ADsimDesk* oder mit Hilfe der *ADwin*-Treiber aus allen gängigen Entwicklungsumgebungen.

2 Neues in *ADbasic 6*

Neu an der Echtzeit-Entwicklungsumgebung *ADbasic 6* ist, dass sie die Prozessoren T12 und T12.1 unterstützt. Die sonstige Funktionalität der Bedienoberfläche entspricht im wesentlichen der Version 5.

Im folgenden sind die besonderen Eigenschaften im Zusammenhang mit den Prozessoren T12 und T12.1 beschrieben:

– **Technische Daten des T12 / T12.1**:
  XILINX ZYNQ™ mit Dual-Core ARM Cortex-A9
  Taktgeschwindigkeit 1 GHz (T12) / 0,66 GHz (T12.1)
  1 Gigabyte Hauptspeicher für Code und Daten
  1 Gigabit-Ethernet-Schnittstelle

  Der Prozessor T12.1 ist weitgehend identisch zum T12. Hinweise für den Prozessor T12 gelten also auch für T12.1, Abweichungen für T12.1 sind in der Dokumentation separat angegeben.

– **Boot-Vorgang**: Beim Prozessor T12.1 dauert der Boot-Vorgang (siehe Kapitel 3.1.3 auf Seite 11) etwa vier Sekunden, in seltenen Einzelfällen auch bis zu fünfzehn Sekunden.

– **Zugriff auf ADwin-Hardware**: Das Prozessor-Modul Pro II-T12 kann nur auf Module am Pro II-Bus zugreifen, aber nicht auf Module am Pro I-Bus. Der Prozessor T12.1 hat keine Schnittstelle zu Pro-Modulen.

– **Speicher**: Die Speicherverwaltung unterscheidet nicht mehr nach Programm- und Datenspeicher (siehe Kapitel 4.3.3). Stattdessen beschleunigt ein „Cache“ – ein besonders schneller Pufferspeicher – den Zugriff auf Programm und Daten.

  Sie können deklarieren, dass der Prozessor auf ausgewählte `Data`-Felder nicht über den Cache zugreifen kann (`Uncacheable`). Damit bleibt der Cache für andere Daten verfügbar (`Cacheable`).

  Im Fenster Stack Test Analysis kann geprüft werden, ob die eingestellte Größe des Stapelspeichers für einen Prozess ausreicht. Die Prüfung ist insbesondere für Libraries wichtig.

– **Fließkomma-Werte**: Der Prozessor kann Fließkommawerte mit 32 Bit (einfache Genauigkeit) und mit 64 Bit (doppelte Genauigkeit) verarbeiten. Dafür erhält *ADbasic* die neuen Datentypen `Float32` und `Float64` (siehe Datentypen).

Der bisherige Datentyp `Float` wird vom Compiler automatisch durch `Float64` ersetzt. So können Sie auch diesbezüglich vorhandene Programme ohne Änderung weiter verwenden, und Sie erhalten eine bessere Rechengenauigkeit als beim T11 (Fließkommawerte mit 40 Bit).

Ganzzahlige Werte (`Long`) werden weiterhin als 32 Bit-Integer verarbeitet.

– **Strings**: Jedes Zeichen belegt nur 1 Byte im Speicher, bisher waren es jeweils 4 Bytes.

  Es darf nicht mehr auf das Element 1 eines String zugegriffen werden (enthielt bisher die Stringlänge). Wie bisher ermitteln Sie die Stringlänge mit **`StrLen`**.

– **For-Schleifen**: Der Prozessor T12/T12.1 prüft zuerst die Bedingung einer For-Schleife, bevor er sie das erste Mal ausführt, siehe `For … To … {Step … } Next`.

  Bis zum Prozessor T11 wurde die For-Schleife auf jeden Fall einmal ausgeführt, auch wenn der Startwert größer als der Endwert war.

– **Zeitmessung**: Wenn Sie Zeitabläufe im Programm ausmessen wollen, müssen Sie beachten, dass die Zeitmessung ausgewählter Programmteile deutlich mehr von den Rahmenbedingungen abhängt als bei früheren Prozessoren.

  Nach wie vor ist es einfach, die Dauer eines Prozesszyklus (Abschnitt Event) zu messen. Für die Dauer von Programmteilen können Sie auch Durchschnittswerte einfach ermitteln; für exakte Werte benötigen Sie darüber hinaus ein gutes Verständnis für die Abarbeitung von Befehlen und die Prozessorarchitektur, um mögliche Fehlereinflüsse zu berücksichtigen.

  Näheres finden Sie im Kapitel Bearbeitungszeit messen, Abschnitt „Besonderheiten beim T12/T12.1“.

– **Auslastung beachten**: Achten Sie streng darauf, dass die Prozessorauslastung (`Busy` in der Statusleiste) deutlich von 100% entfernt bleibt. Nur so ist gewährleistet, dass jeder einzelne Prozesszyklus im korrekten Zeitraster ausgeführt wird.

  Im Unterschied zu früheren Prozessoren kann der T12/T12.1 auch bei zeitweiliger Überlastung in der Regel noch weiter arbeiten (siehe Auslastung des ADwin-Systems). Das ist allerdings nur ausnahmsweise sinnvoll, wenn in Ihrem Prozess verlorene Prozesszyklen akzeptabel

sind. Sie können dann im Prozessfenster die Anzahl verlorener Prozesszyklen beobachten.

– **Zeitgesteuerte Prozesse**: Das Zeitverhalten ist bei mehreren Prozessen nun identisch zum Verhalten eines einzelnen Prozesses. Die Änderung betrifft nur das Verhalten bei einer Überlastung des Prozessors. Näheres siehe Einzelner zeitgesteuerter Prozess.

  Bei einem einzelnen zeitgesteuerten Prozess und beim extern gesteuerten Prozess bleibt das Zeitverhalten unverändert.

3 Entwicklungsumgebung nutzen

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.

3.1 Grundlegende Schritte

3.1.1 Entwicklungsumgebung erstmals starten

Zum Starten gehen Sie vor wie folgt:

1. Sie starten die Entwicklungsumgebung, indem Sie im Windows-Startmenü `Programs` ▸ `ADwin` ▸ `ADbasic` wählen.

   Beim ersten Starten kann es einige Sekunden dauern, bis die Oberfläche erscheint, weil dabei auch das Windows-Programmpaket .Net-Framework gestartet wird.

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

2. Beim ersten Start erscheint das Fenster `License key`. Geben Sie hier Ihren Lizenzschlüssel ein. Sie finden den `License key` auf dem Deckblatt dieses *ADbasic*-Handbuchs.

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

   Ohne Eingabe des gültigen License key befindet sich *ADbasic* im Demo-Modus. In diesem Modus ist das Arbeiten mit der Entwicklungsumgebung nur zu Prüf-, Demonstrations- und Bewertungszwecken erlaubt. Sie können beispielsweise keine Binärdateien erzeugen.

   Weitere Informationen zu *ADwin*-Software-Lizenzen finden Sie im Kapitel 3.1.2 auf Seite 10.

3. Stellen Sie nun das von Ihnen verwendete *ADwin*-System und den Prozessor im Menü `Options\Compiler` ein.

   Die Entwicklungsumgebung speichert Ihre Angaben. Ein Ändern ist erst wieder nötig, wenn Sie eine andere *ADwin*-Hardware verwenden.

3.1.2 Lizenzschlüssel prüfen und ändern

Um einen Lizenzschlüssel für *ADwin*-Software wie *ADbasic* zu prüfen oder zu ändern, gehen Sie vor wie folgt:

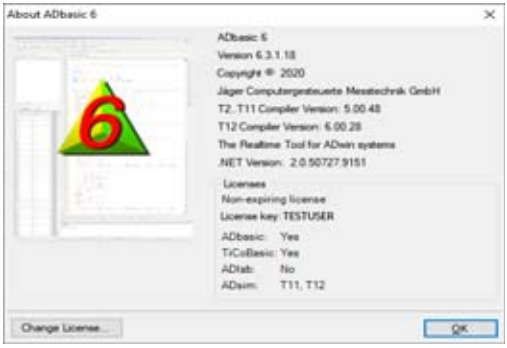
1. Legen Sie den Lizenzschlüssel bereit; er wird mit der *ADwin*-Software geliefert. Sie finden den Lizenzschlüssel in der Regel auf dem Deckblatt des Handbuchs oder auf dem Lieferschein.

   Es gibt Lizenzschlüssel für folgende Software:
   - *ADbasic* 6, arbeitet mit allen *ADwin*-Prozessoren
   - *ADbasic* 5, arbeitet mit *ADwin*-Prozessoren bis Version T11
   - *ADbasic* 3.0, arbeitet mit *ADwin*-Prozessoren bis Version T9
   - *ADbasic* 2.0, arbeitet mit *ADwin*-Prozessoren bis Version T8
   - *TiCoBasic*
   - *ADlab* (Matlab-Treiber für *ADwin*)
   - *ADsim*-Paket (Simulink-Treiber für *ADwin*)

   Die Lizenzen für *ADbasic*, *TiCoBasic* und *ADlab* können kombiniert vergeben werden. Für *ADsim* ist immer ein eigener Lizenzschlüssel erforderlich.

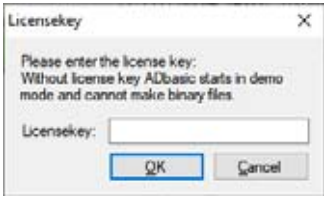
2. Wählen Sie den Menüeintrag `Help` ▸ `About` (siehe auch Kapitel 3.10.10).

   Das Fenster `About ADbasic` öffnet sich. Dort sind neben der Version der Entwicklungsumgebung unter `Licenses` die bereits aktivierten Lizenzen angegeben (mögliche Lizenzen siehe oben).



3. Zum Eingeben oder Ändern der Lizenzen klicken Sie die Schaltfläche `Change License`.

   Das Eingabefenster `License key` öffnet sich.



4. Geben Sie den Lizenzschlüssel ein.

   Der Lizenzschlüssel wird Ihnen mit der *ADwin*-Software geliefert.

Es sind folgende Lizenzen möglich:

– Keine Lizenz (`Demo mode`)

  Ohne Eingabe eines gültigen `License key` befindet sich *ADbasic* bzw. die entsprechende *ADwin*-Software im Demo-Modus. In diesem Modus ist das Arbeiten nur zu Prüf-, Demonstrations- und Bewertungszwecken erlaubt und freigegeben. Sie können beispielsweise keine Binärdateien erzeugen.

– Zeitlich begrenzte Lizenz (`Evaluation license`)

  Die Lizenz schaltet alle Funktionen der Entwicklungsumgebung für einen definierten Zeitraum frei. Anschließend arbeitet die *ADwin*-Software wieder im Demo-Modus (siehe oben).

– Zeitlich unbegrenzte Lizenz für den Lizenznehmer

  Die Lizenzbedingungen für *ADbasic* sind im Lizenzvertrag (Anhang Seite A-5) beschrieben.

3.1.3 ADwin-Betriebssystem laden

Sie übertragen das *ADwin*-Betriebssystem in Ihr *ADwin*-System, indem Sie auf die Schaltfläche B 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 ¼g 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).

3.1.4 Grundelemente der Entwicklungsumgebung

Die Entwicklungsumgebung besteht aus mehreren Leisten und Fenstern (siehe Abb. 1); Sie können die Höhe der Fenster frei anpassen.

Sie erhalten Hilfe zu einem Fenster oder zu dem aktuell markierten Kennwort, wenn Sie die Taste [F1] drücken (siehe auch Online-Hilfe aufrufen). Mit der Schaltfläche öffnen Sie das Indexverzeichnis der Hilfe.

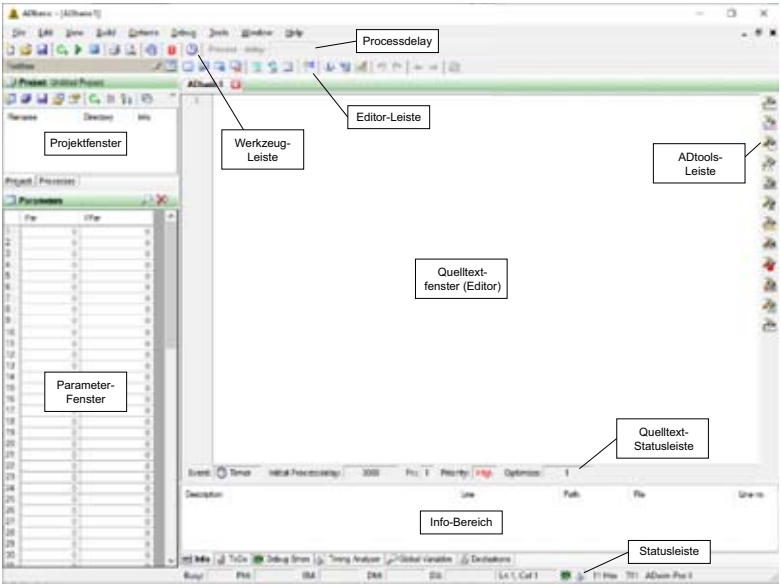


Abb. 1 – Elemente der *ADbasic*-Entwicklungsumgebung

Sie finden die Befehle der Entwicklungsumgebung über:

– die Werkzeugleiste und die Editorleiste (siehe Abb. 2).

– die Projektleiste im Projektfenster

– die Kontextmenüs der Fenster (rechte Maustaste).

– die Menüleiste.

– die Tastaturkürzel in ADbasic (siehe Anhang).

– die ADtools-Leiste.
  Hier können Sie auch eigene Aufrufkürzel einfügen (siehe Kapitel 3.6.7 auf Seite 43).

Wenn Sie einen Befehl anwenden, sehen Sie am linken Rand der Statusleiste eine Befehlserklärung.

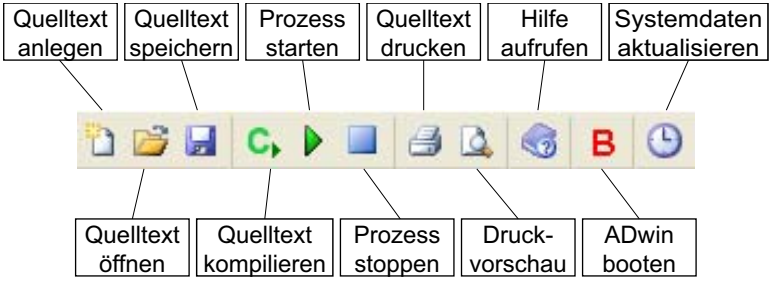


Abb. 2 – Die Werkzeugleiste

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. Viele 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 an.

– Die Quelltext-Statusleiste zeigt die von Ihnen eingestellten Prozess-Optionen.

Ein Doppelklick auf die Quelltext-Statusleiste öffnet das Dialogfenster „Process Options“.

– Im Parameterfenster (siehe Kapitel 3.11.3 auf Seite 79) können Sie durch Drücken der Schaltfläche `Scan Global Variables` (einmalig) markieren lassen, welche globalen Parameter Sie im aktiven Quelltext oder Projekt verwenden; siehe Verwendete globale Variablen und Felder anzeigen auf Seite 42.

– Im Infobereich unten werden in verschiedenen Fenstern eine Reihe von Informationen angezeigt:
  - Infofenster: Fehlermeldungen (rot hinterlegt) und Warnungen des Compilers (siehe Kapitel 3.12.1 auf Seite 89).
  - ToDo-Liste: Eine einfache To-Do-Liste aus Kommentarzeilen (siehe Kapitel 3.12.2 auf Seite 91).
  - Suchergebnisse bei einer Suche über alle Dateien eines Projekts (siehe Kapitel 3.5.2 auf Seite 28).
  - Debug-Informationen, wenn der Debug-Modus aktiv ist (siehe Option Debug mode, Seite 69).
  - Fenster „Timing Analyzer“: Das Zeitverhalten der Prozesse auf dem *ADwin*-System (siehe Kapitel 3.12.3 auf Seite 91).
  - Fenster „Global Variables“: Eine Liste der verwendeten globalen Variablen und Felder (siehe Kapitel 3.12.4 auf Seite 95).
  - Fenster „Declarations“: Eine Liste aller zu einer Quelltextdatei gehörenden Deklarationen (siehe Kapitel 3.12.5 auf Seite 96).

Beachten Sie, dass das Editieren im Quelltextfenster durch eine Reihe von Hilfsfunktionen unterstützt wird (siehe Seite 16).

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

Einige Daten Ihres *ADwin*-Systems werden kontinuierlich ausgelesen und angezeigt (nur, wenn der PC mit dem *ADwin*-System kommuniziert):

– Das Processdelay (Prozess-Zykluszeit) zu der Prozessnummer, die Sie für das aktive Quelltextfenster eingestellt haben, dargestellt rechts von der Werkzeugleiste.

– Die Werte der globalen Variablen im Parameterfenster; wenn Sie einen dieser Werte verändern, wird er sofort zum *ADwin*-System übertragen.

– Der Zustand der laufenden Prozesse im Prozessfenster (Seite 81).

– Informationen zur Speicherauslastung in der Statusleiste (siehe Kapitel 3.11.6).

3.2 Quelltexte erstellen

Öffnen Sie für jeden Prozess-Quelltext ein eigenes Fenster (mit `File` ▶ `New` oder [CTRL]+[N]).

Sie können die Reihenfolge der Reiter der Quelltextfenster verändern. Dazu drücken Sie die [ALT]-Taste, klicken mit der Maus auf einen Reiter und ziehen ihn an die neue Position.

Wenn Sie für eine Aufgabe mehrere Dateien verwenden, empfehlen wir, diese gemeinsam in einer Projektdatei zu verwalten (siehe Seite 47: Projekte verwalten).

Um einen Quelltext auf der *ADwin*-Hardware zu testen, müssen Sie den Quelltext übersetzen, siehe Seite 20.

Der Editor und der *ADbasic*-Compiler unterscheiden nicht zwischen Groß- und Kleinschreibung. In unseren Beispielen verwenden wir allerdings zur besseren Unterscheidung eine einheitliche Schreibweise.

Bei Unklarheiten oder Problemen sollten Sie die Online-Hilfe aufrufen (siehe unten).

Beim Editieren im Quelltextfenster können Sie eine Reihe von Hilfsfunktionen nutzen. Sie rufen die Funktionen über das Kontextmenü im Quelltextfenster (Seite 18) oder über die Editor-Leiste (Seite 19) auf:

Sie können Zahlenwerte im Quelltext in dezimaler, hexadezimaler, binärer und Exponential-Schreibweise eingeben (siehe Kapitel 4.2.4 „Zahlenwerte eingeben“).

Weitere Editor-Funktionen finden Sie unter den Rubriken:

– Quelltext formatieren, Seite 21

– Suchen, markieren und ersetzen, Seite 28

– Programme leichter schreiben, Seite 39

3.2.1 Online-Hilfe aufrufen

Das Menü „Help“ (Seite 74) erlaubt den Aufruf von ausgesuchten Hilfe-Seiten, z.B. das Inhaltsverzeichnis oder sortierte Befehlslisten.

Mit [F1] öffnen Sie eine Hilfeseite zu dem gerade geöffneten Dialogfenster oder zu dem Befehl an der Cursor-Position.

Wenn an der Cursor-Position ein ungültiger Befehl steht, öffnet die Hilfe das Indexverzeichnis. Gründe hierfür sind in der Regel:

– Es handelt sich nicht um einen Befehl, sondern um eine vom Benutzer definierte Deklaration: Variable / Feld, symbolischer Name, Makro (Sub, Function). Hierzu kann es keine Hilfeseiten geben.

– Der Befehl ist falsch geschrieben, z.B. `Digin_Wrod` anstatt **`Digin_Word`**.
  Wenn Sie den Fehler korrigiert haben, wird der Befehl wieder farbig hervorgehoben.

– Stellen Sie die passende *ADwin*-Hardware ein (siehe Dialogfenster „Compiler Options“) oder verwenden Sie einen für eingestellte Hardware gültigen Befehl.

– Im Quelltext fehlt die (benutzerdefinierte) Include- oder Library-Datei, in der der Befehl definiert ist.
  Fügen Sie die entsprechende Zeile am Anfang des Quelltexts ein.

Das Hilfefenster zeigt links mehrere Registerkarten, rechts die aufgerufene Hilfeseite. Die Registerkarten führen Sie zum Inhaltsverzeichnis (`Contents`), Index-Register (`Index`), Such-Fenster (`Search`) und zu sortierten Befehlslisten (`Commands`).

Sie können mehrere Hilfeseiten nebeneinander anzeigen:

– Klicken Sie mit der rechten Maustaste auf einen Link und wählen Sie im Kontextmenü den Eintrag `In neuem Fenster öffnen`; der Link öffnet sich dann in einem neuen Tab.

– Alternativ: Über die Schaltfläche + rechts oben (`Add new tab`) öffnen Sie beliebig viele (zunächst leere) Tab. Sobald Sie einen Link anklicken, wird das Link-Ziel im nächsten leeren Tab geöffnet. Wenn alle Tabs gefüllt sind, wird das Link-Ziel im aktiven Fenster angezeigt.

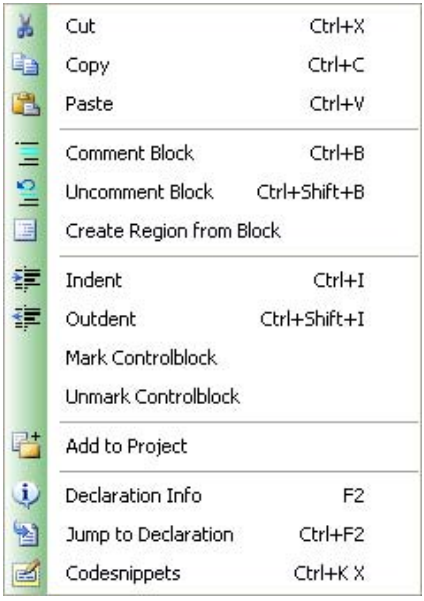
Sie können viele Vektorgrafiken in der Online-Hilfe (z.B. Pinbelegungen) zum Erkennen von Details vergrößern, indem Sie das Hilfefenster am Rahmen größer ziehen.

Bei der Suche innerhalb der Hilfe sind nur folgende Zeichen erlaubt: 0-9, A-Z, - (Minus) und Umlaute; Groß-/Kleinschreibung wird nicht unterschieden. Bei

mehreren Wörtern dient das Leerzeichen als Trennzeichen, gefunden werden nur Seiten, die alle Wörter enthalten. Die Suche nach ganzen Wörtern ist optional.

3.2.2 Kontextmenü im Quelltextfenster

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



Die folgenden Befehle verwenden die Cursor-Position oder die aktive Markierung:

- `Cut`: Markierung löschen und in die Zwischenablage kopieren.
- `Copy`: Markierung in die Zwischenablage kopieren.
- `Paste`: Markierung löschen und durch den Inhalt der Zwischenablage ersetzen.
- `Comment Block`, `Uncomment Block`: Zeilen in Kommentar ändern, Seite 23.
- `Create Region from Block`: Faltbaren Textbereich definieren, Seite 26.
- `Indent`, `Outdent`: Textzeilen einrücken, Seite 22.
- `Mark Control block`, `Unmark Control block`: Kontrollstruktur markieren, Seite 36.
- `Declaration Info`: Deklaration eines Befehls oder Variablen anzeigen, Seite 42.
- `Jump to Declaration`: Zur Deklaration eines Befehls oder Variablen springen, Seite 37.

Ohne vorherige Markierung sind folgende Befehle verfügbar:

– `Add to Project`: Datei dem aktiven Projekt zufügen.
– `Declaration Info`: Deklarationen einer Datei anzeigen, Seite 42.
– `Code snippets`: Textbausteine einfügen, Seite 40.

3.2.3 Editor-Leiste

Im Quelltextfenster können Sie verschiedene Hilfsfunktionen über die Editor-Leiste erreichen.

Textmarken nutzen, Seite 36.

Zeilen in Kommentar ändern, Seite 23.

Faltbaren Textbereich definieren, Seite 26.

Textbereiche ein- / ausfalten, Seite 27.

Deklaration eines Befehls oder Variablen anzeigen, Seite 42.

Zur Deklaration eines Befehls oder Variablen springen, Seite 37.

Textbausteine einfügen, Seite 40.

Einen Befehl rückgängig machen oder wieder herstellen.

Zwischen Sprungzielen wechseln, Seite 38.

Quellcode für geänderte Simulink-Library aktualisieren, Seite 100.

3.3 Quelltext übersetzen

Sie übersetzen (kompilieren) einen Quelltext mit *ADbasic* in eine Binärdatei. Die Binärdatei kann auf die *ADwin*-Hardware übertragen und dort als Prozess ausgeführt werden.

Dazu stehen Ihnen folgende Möglichkeiten zur Verfügung:

– Quelltext im Entwicklungsprozess testen
  - Einzelne Quelltext-Datei übersetzen und ausführen, `Build` ▸ `Compile` oder Schaltfläche in der Werkzeugleiste.
  - Ausgewählte Projekt-Quelltexte kompilieren und ausführen, Schaltfläche in der Projektleiste.

– Fertigen Quelltext als Binärdatei speichern
  - Einzelne Quelltext-Datei übersetzen und als Binärdatei speichern, `Build` ▸ `Make Bin`.
  - Ausgewählte Projekt-Quelltexte kompilieren und als Binärdateien speichern, Schaltfläche in der Projektleiste.
  - Alle Projekt-Quelltexte kompilieren und als Binärdateien speichern, Schaltfläche in der Projektleiste.

  Quelltext übersetzen und als Binärdatei speichern: `Build` ▸ `Make Bin`.

– Vor und nach dem Kompilieren Kommandozeilen-Aufrufe durchführen.

  Kommandozeilen werden im Dialogfenster „Project Settings“ definiert, Reiter Prebuild / Postbuild (Seite 67).

Sie können erzeugte Binärdateien auf verschiedene Weise nutzen:

– Mehrere Binärdateien eines Projekts aus der Oberfläche *ADbasic* auf *ADwin*-Hardware übertragen und ausführen: Schaltfläche in der Projektleiste.

– Binärdateien aus einem externen Programm auf *ADwin*-Hardware übertragen, ausführen und steuern, siehe Kommunikation mit Entwicklungsumgebungen.

  Es stehen Schnittstellen für gängige Entwicklungsumgebungen (z.B. Java, Visual Basic, C#.NET) und zahlreiche Bedienoberflächen (wie DIAdem, MATLAB) zur Verfügung.

– Binärdateien dauerhaft im Bootloader der *ADwin*-Hardware speichern und beim Einschalten automatisch starten: Handbuch „ADwin-Bootlader“.

Beim Prozessor T12.1 wird der Bootloader mit der Entwicklungsoberfläche programmiert, siehe Kapitel 3.8 „Bootloader nutzen“, Seite 44.

Ergänzend verweisen wir auf die Möglichkeit, Library-Funktionen aus einem Quelltext als Binärdatei zu erzeugen, siehe Bibliotheken (Libraries). Library-Funktionen können in anderen Quelltexten eingebunden und genutzt werden.

3.4 Quelltext formatieren

Quelltext kann (meist automatisch) formatiert werden, um die Übersicht über die Programmstruktur zu zeigen:

– Syntax hervorheben, Seite 21

– Automatisch formatieren, Seite 22

– Textzeilen einrücken, Seite 22

– Kommentare positionieren, Seite 22

– Zeilen in Kommentar ändern, Seite 23

– Eigene Befehle und Variablen dokumentieren, Seite 23

– Faltbaren Textbereich definieren, Seite 26

– Textbereiche ein- / ausfalten, Seite 27

Hier finden Sie weitere Editor-Funktionen:

– Quelltexte erstellen, Seite 16

– Suchen, markieren und ersetzen, Seite 28

– Programme leichter schreiben, Seite 39

3.4.1 Syntax hervorheben

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.

Der Editor unterscheidet die von Ihnen eingegebenen Zeichenketten in mehrere Gruppen von Syntaxelementen, die unterschiedlich dargestellt werden. Sie können die Farbgebung unter `Options` ▸ `Settings`, Editor - Syntax Colors (siehe Seite 63) nach eigener Vorstellung verändern; dort ist auch eine Übersicht der Syntax-Gruppen angegeben.

Sobald eine Include-Datei mit `#Include` in einen Quelltext eingebunden ist, werden die darin definierten Befehle im Quelltext hervorgehoben. Auch ver-

schachtelte Einbindungen werden berücksichtigt. Zu Besonderheiten siehe Include-Dateien.

Die vollständige Syntax-Hervorhebung erfordert, dass die Option `Parse Declarations` unter Editor - General (siehe Seite 62) aktiv ist.

3.4.2 Automatisch formatieren

Wenn Sie eine Befehlszeile eingegeben haben, korrigiert der Editor automatisch die Leerzeichen, so dass ein einheitliches Bild entsteht. So erhalten z. B. Operatoren wie „`=`“ oder Kennwörter wie „`If`“ rechts und links ein Leerzeichen.

Wenn Sie lieber manuell formatieren, müssen Sie zunächst unter Editor - General, `Smart format` das automatische Formatieren abschalten (siehe Seite 62).

3.4.3 Textzeilen einrücken

Wenn Sie eine Befehlszeile eingegeben haben, rückt der Editor die Zeilen automatisch so ein, dass sich ein übersichtliches Bild ergibt. Wegen der Automatik ist manuelles Einrücken dann nicht möglich.

Wenn Sie lieber manuell formatieren, müssen Sie zunächst unter Editor - General, `AutoIndent` die automatische Einrückung abschalten. Anschließend können Sie Einrückungen mit Tabulator [TAB] oder Leerzeichen [SPACE] erzeugen. Mehrere markierte Zeilen können gemeinsam ein- oder ausgerückt werden, indem im Kontextmenü des Quelltextfensters (Klick mit rechter Maustaste) die Einträge `Indent` oder `Outdent` gewählt werden.

Unter dem Menüpunkt `Options` ▸ `Settings`, Editor - General, `Tabsize` wird eingestellt, wie viele Leerzeichen eine Einrückung hat.

3.4.4 Kommentare positionieren

Wenn Sie einen Kommentar mit `'` am Zeilenende eingeben, setzt der Editor den Kommentaranfang automatisch auf eine definierte Position. Das soll die Lesbarkeit der Kommentare verbessern.

Mit doppelten Kommentarzeichen " können Sie einen Kommentar auch bei automatischer Positionierung manuell positionieren.

Sie können die automatische Positionierung unter Editor - General, `Align comments` an- oder abschalten und die Kommentarposition einstellen (siehe Seite 62).

3.4.5 Zeilen in Kommentar ändern

Mehrere markierte Zeilen können gemeinsam in Kommentarzeilen umgewandelt werden, indem im Kontextmenü des Quelltextfensters (Klick mit rechter Maustaste) der Eintrag `Comment Block` gewählt wird oder mit [CTRL]-B. Als Folge fügt der Editor an jedem Zeilenanfang ein Kommentarzeichen ein, so dass der Compiler diese Zeilen überspringt.

In gleicher Weise löscht `Uncomment Block` oder [CTRL]-[SHIFT]-B ein Kommentarzeichen am Zeilenanfang wieder.

3.4.6 Eigene Befehle und Variablen dokumentieren

Sie können selbst geschriebene Befehle (`Sub … EndSub`, `Function … EndFunction`), Variablen, Felder und symbolische Namen (**`#Define`**) so mit Hinweisen dokumentieren, dass die Hinweistexte beim Programmieren als Tooltips und bei der automatischen Vervollständigung angezeigt werden. Mehr darüber ist bei den Funktionen „Befehle und Variablen automatisch vervollständigen“ (Seite 39), „Befehlsparameter anzeigen“ (Seite 41) und „Deklaration eines Befehls oder Variablen anzeigen“ (Seite 42) beschrieben.

Beispiel

Die folgenden Beispiele zeigen, wie Sie die Hinweistexte aufbauen.

```
''' Hier beginnt der Hinweistext zur Funktion my_func;
'''+alle Folgezeilen, die mit 3 Kommentar- und einem Plus-
'''+Zeichen beginnen, gehören zum ersten Textabschnitt,
'''+der den Zweck der Funktion und des Rückgabewerts
'''+beschreibt.
''' val1: Ein Einzeiler für den ersten Übergabeparameter
''' Der zweite Übergabeparameter val2. Wie man sieht, kann
'''+die Zeile auch ohne Parameternamen beginnen, mit ist
'''+es aber übersichtlicher.
''' Diese Zeilen starten einen Bereich, der keinem
'''+Parameter zugeordnet werden kann und daher nicht
'''+verwendet wird.
Function my_func(val1, val2) As Long
 my_func = (val1 + val2) / 2
EndFunction

'''Hier folgt die Beschreibung einer Variablen.
'''Für mehrzeilige Bereiche ist das Plus-Zeichen nicht
'''+ erforderlich, es schadet aber auch nicht.
Dim var As Long

'''Beschreibung für einen symbolischen Namen
#Define max Par_1
```

So erscheint der Tooltip beim Verwenden der Funktion:

```
27  Event:
28     max = my_func(
```

```
Function my_func(val1, val2) As Long
val1: Ein Einzeiler für den ersten Übergabeparameter
```

Beachten Sie folgende Regeln zum Anlegen eigener Hinweise:

– Sie können folgende selbst definierten Elemente mit Hinweistexten dokumentieren:
  - symbolische Namen (`#Define`)
  - Variablen und Felder (`Dim`)
  - Makros (`Sub`, `Function`)
  - Bibliotheken (`Lib_Sub`, `Lib_Function`)

– Fügen Sie alle Hinweistexte in Kommentarzeilen ein, die mit 3 Kommentarzeichen `'''` beginnen. Eine Zeile mit weniger als 3 Kommentarzeichen beendet den Hinweisbereich.

– Hinweiszeilen müssen direkt über der Zeile stehen, die eines der oben genannten Kennwörter enthält (Definitionszeile). Bei richtiger Eingabe wird der Hinweistext als faltbarer Textbereich angezeigt (siehe Seite 26).

  Wenn zwischen Definition und Hinweiszeilen eine andere Zeile steht, ist der Hinweisbereich vollständig deaktiviert.

– Bei Makros und Bibliotheken können Sie für den Befehl und jeden Übergabeparameter jeweils einen Unterbereich anlegen.

– Ein Unterbereich kann ein- oder mehrzeilig sein:
  - Die erste Zeile eines Unterbereichs beginnt mit 3 Kommentarzeichen `'''`, jede weitere Zeile des Unterbereichs erhält zusätzlich ein Plus, beginnt also mit `'''+`.
  - Die Anzahl der Zeilen in einem Unterbereich ist unbegrenzt.
  - In einem Unterbereich eines Übergabeparameters beginnt der Text mit dem Parameternamen, direkt gefolgt von einem Doppelpunkt. Alternativ kann der Name auch entfallen.

Die Anzeige der Hinweise (z.B. in Tooltips) ist nur möglich, wenn die Option `Parse Declarations` unter Editor - General (siehe Seite 62) aktiv ist.

3.4.7 Faltbaren Textbereich definieren

Sie können beliebige Quelltextzeilen als faltbaren Bereich definieren. Ein faltbarer Bereich wird durch eine graue Linie am Zeilenanfang markiert, mit einem Minuszeichen in der ersten Zeile des Bereichs. Sie falten den Bereich mit einem Klick auf das Minuszeichen in der ersten Zeile.

Der Editor erkennt auch Strukturen wie Bedingungen und Schleifen, Programmabschnitte, Makros und Bibliotheks-Module als faltbare Textbereiche.

Sie definieren einen eigenen faltbaren Bereich mit den Schlüsselwörtern `<Region>` und `<EndRegion>` auf folgende Weise:

– Markieren Sie die gewünschten Quelltextzeilen.

  Achten Sie darauf, dass die Markierung vorhandene Kontrollstrukturen immer vollständig oder gar nicht einschließt. Das Einschließen (oder Überlappen) eines anderen `<Region>`-Bereichs ist nicht erlaubt.

– Wählen Sie im Kontextmenü des Quelltextfensters den Menüeintrag `Create Region from Block`.

  Ein Eingabefenster öffnet sich.

– Geben Sie eine kurze Beschreibung für den faltbaren Bereich unter `Region description` an und bestätigen Sie mit `OK`.

– Der markierte Bereich ist nun von den Schlüsselwörtern `<Region>` und `<EndRegion>` umgeben und kann ein- und ausgefaltet werden.

  Sie können die Schlüsselwörter auch manuell eingeben.

```
12  '<REGION> #Defines and Dims
13  #Define x PAR_10
14  #Define x1 PAR_11
15  #Define x2 PAR_12
16  #Define x3 PAR_13
17
18  Dim x, x1, x2, x3 As Float
19  '<ENDREGION>
20
```

```
12  '<REGION> #Defines and Dims ...
20
```

Mit der Schaltfläche `Toggle Folding` können alle faltbaren Textbereiche auf einmal ein- oder ausgefaltet werden.

Faltbare Textbereiche werden nur erkannt, wenn die Option `Parse Declarations` unter Editor - General (siehe Seite 62) aktiv ist.

3.4.8 Textbereiche ein- / ausfalten

Der Editor erkennt Kontrollstrukturen wie Bedingungen und Schleifen, Programmabschnitte, Makros und Bibliotheks-Module als faltbare Textbereiche. Außerdem können Sie beliebige Quelltextzeilen als Faltbaren Textbereich definieren (siehe Kapitel 3.4.7). Ein solcher Bereich wird durch eine graue Linie am Zeilenanfang markiert, mit einem Minuszeichen in der ersten Zeile des Bereichs.

Sie falten einen einzelnen Bereich mit einem Klick auf das Minuszeichen in der ersten Zeile; im Beispiel unten würden Sie links neben `Function sumsquare` klicken.

```
4   Function sumsquare(w1,w2,w3) As Long
5     Rem Quadrat der Summe
6     Dim sum As Long
7     sum = w1+w2+w3
8     sumsquare = sum * sum
9   EndFunction
10
11  Dim x, x1, x2, x3 As Long
12
```

```
4   Function sumsquare(w1,w2,w3) As Long...
10
11  Dim x, x1, x2, x3 As Long
```

Mit der Schaltfläche `Toggle Folding` können alle faltbaren Textbereiche auf einmal ein- oder ausgefaltet werden.

Faltbare Textbereiche werden nur erkannt, wenn die Option `Parse Declarations` unter Editor - General (siehe Seite 62) aktiv ist.

3.5 Suchen, markieren und ersetzen

Suchen, markieren oder ersetzen Sie einen beliebigen Text mit diesen Funktionen:

– Text schnell suchen, Seite 28
– Text suchen und ersetzen, Seite 28
– Reguläre Ausdrücke, Seite 34
– Kontrollstruktur markieren, Seite 36
– Textmarken nutzen, Seite 36
– Zu einer Programmzeile springen, Seite 37
– Zur Deklaration eines Befehls oder Variablen springen, Seite 37
– Zwischen Sprungzielen wechseln, Seite 38

Hier finden Sie weitere Editor-Funktionen:

– Quelltexte erstellen, Seite 16
– Quelltext formatieren, Seite 21
– Programme leichter schreiben, Seite 39

3.5.1 Text schnell suchen

Mit dem Tastenkürzel [CTRL]-[F3] können Sie Text schnell suchen. Für die Rückwärtssuche verwenden Sie das Tastenkürzel [CTRL]-[SHIFT]-[F3].

Gesucht wird der markierte Text oder, falls kein Text markiert ist, das Wort an der Cursor-Position. Folgende Suchoptionen sind fest eingestellt:

– Groß-Kleinschreibung spielt keine Rolle.
– Der Suchausdruck kann auch ein Teil eines Wortes sein.
– Es werden auch eingeklappte Textbereiche durchsucht.
– Alle offenen Dokumente werden durchsucht.

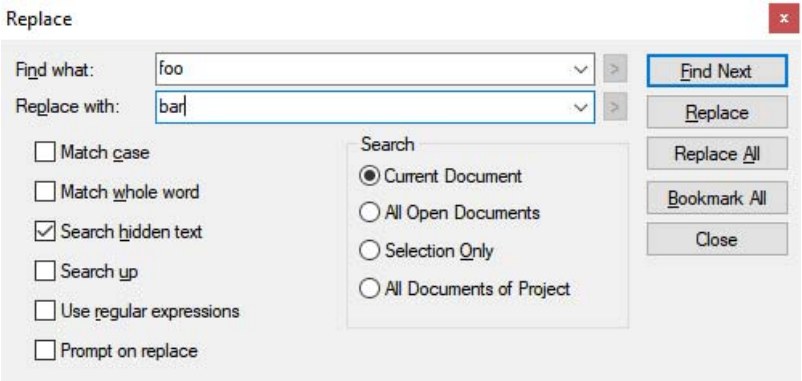
Bei der Schnellsuche können Sie keine regulären Ausdrücke verwenden und auch keine Textmarken erzeugen.

3.5.2 Text suchen und ersetzen

Sie können nach jedem Auftreten einer beliebigen Kombination von Zeichen suchen, zum Beispiel auch nach Groß- und Kleinbuchstaben, ganzen Wör-

tern, Teilen von Wörtern und regulären Ausdrücken (siehe Reguläre Ausdrücke auf Seite 34).

1. Wählen Sie den Menüeintrag `Edit` ▸ `Find` für eine Suche oder `Edit` ▸ `Replace` zum Ersetzen. Es erscheint ein Dialogfeld, das geöffnet bleibt, bis Sie es schließen.



2. Geben Sie im Eingabefeld `Find what` den gewünschten Suchausdruck an. Sie können auch einen der zuletzt benutzten Ausdrücke aus der Liste wählen.
3. Nur bei Ersetzen: Geben Sie im Eingabefeld `Replace With` den gewünschten Ersetzungsausdruck an. Sie können auch einen der zuletzt benutzten Ausdrücke aus der Liste wählen.
4. Stellen Sie die Optionen für die Suche ein:

Option	Beschreibung
`Match case`	Option aktiviert: Es wird nur nach Text gesucht, der genau die Groß-/Kleinschreibung des Suchausdrucks hat. Option deaktiviert: Die Groß-/Kleinschreibung spielt bei der Suche keine Rolle.
`Match whole word`	Option aktiviert: Es wird nur nach Text gesucht, bei dem der Suchausdruck ein ganzes Wort ist. Option deaktiviert: Der Suchausdruck kann auch ein Teil eines Wortes sein.

Option	Beschreibung
`Search hidden text`	Die Option bezieht sich auf Faltbaren Textbereich definieren (siehe Seite 26). Option aktiviert: Es werden auch eingeklappte Textbereiche durchsucht. Option deaktiviert: Eingeklappte Textbereiche werden übersprungen.
`Search up`	Option aktiviert: Es wird in Richtung zum Dateianfang hin gesucht. Option deaktiviert: Es wird in Richtung zum Dateiende hin gesucht.
`Use regular expressions`	Die Option bezieht sich auf Reguläre Ausdrücke (siehe Seite 34). Option aktiviert: Der eingegebene Suchausdruck ist ein regulärer Ausdruck; das ist eine Methode, die variable Textmuster beschreibt. Option deaktiviert: Der eingegebene Suchausdruck ist konstanter Text.
`Prompt on replace`	Die Option ist nur in Verbindung mit `Replace All` einsetzbar. Option aktiviert: Bei jedem Vorkommen erscheint ein Dialogfenster, mit dem Sie das Ersetzen steuern. Option deaktiviert: Alle Vorkommen werden ohne Rückfrage ersetzt.

5. Stellen Sie einen Bereich für die Suche ein:

Option	Beschreibung
`Current Document`	Die Suche beginnt im aktuellen Quelltext an der Cursor-Position. Wenn Text markiert ist, steht der Cursor hinter der Markierung.
`All open Documents`	Alle geöffneten Dateien werden durchsucht, beginnend mit dem aktuellen Quelltext.
`Selection only`	Nur der markierte Bereich wird durchsucht. Falls keine Markierung vorhanden ist, startet die Suche an der Cursor-Position.
`All Documents of Project`	Alle Dateien des Projekts[a] werden durchsucht, auch wenn der aktuelle Quelltext nicht zum Projekt gehört. Die Option ist für das Ersetzen nicht verfügbar. Die Suchergebnisse werden im Fenster `Find` im Info-Bereich angezeigt. Mit einem Doppelklick auf ein Ergebnis springt der Cursor in die entsprechende Zeile. Alternativ springen Sie mit den Pfeil-Schaltflächen im Fenster `Find` von einem Ergebnis zum nächsten.

a. Dateien aus dem Bereich `Other Files` sind von der Suche in jedem Fall ausgenommen.

6. Starten Sie die gewünschte Aktion durch eine der Schaltflächen.
   - `Find Next`: markiert das nächste Vorkommen des Suchausdrucks.
   - `Replace`: ersetzt die aktuelle Markierung durch den Ersetzungsausdruck und markiert das nächste Vorkommen des Suchausdrucks.
   - `Replace All`: ersetzt alle Vorkommen des Suchausdrucks durch den Ersetzungsausdruck.
   - `Bookmark All`: markiert alle Zeilen, in denen der Suchausdruck vorkommt, mit einer `Textmarke`.

7. Schließen Sie das Dialogfeld, indem Sie auf `Close` klicken. Sie können das Fenster auch geöffnet lassen und im Quelltext weiter arbeiten.

   Mit der Option `All Documents of Project` schließt das Dialogfeld automatisch. Die Suchergebnisse finden Sie im Fenster `Find` (im Infobereich unten).

Anmerkungen

– Der Menüeintrag `Edit` ▸ `Find Next` sucht das nächste Vorkommen des Suchausdrucks. Die aktuellen Suchoptionen sind dabei gültig, auch wenn das Suchen-Dialogfenster geschlossen ist.

– Die Aktion `Replace` ersetzt die aktuelle Markierung nur dann, wenn die Markierung dem Suchausdruck entspricht.

– Vorsicht ist angebracht beim Ersetzen mit regulären Ausdrücken, die auch auf „nichts“ passen, z. B. „`.+`“ oder „`a*`“. In solchen Fällen können Schleifen entstehen, in denen so lange ersetzt wird, bis die Zeile zu lang ist.

☞ – Tipp: Wenn Sie mit regulären Ausdrücken eine große Anzahl von Ersetzungen in einem oder gar allen geöffneten Dokumenten vornehmen wollen, können Sie sich zunächst mit `Find Next` und `Replace` vergewissern, dass Sie den Such- und den Ersetzungsausdruck korrekt formuliert haben, bevor Sie mit `Replace All` den Rest ersetzen lassen.

Beispiele für das Suchen von Text

Beispiele für das Suchen mit regulären Ausdrücken.

– Alle Leerzeichen oder Tabulatoren am Ende einer Zeile finden:
  `[ ]+$`

  Der Suchausdruck findet ein oder mehrere Leerzeichen oder Tabulatoren, auf die das Ende der Zeile folgt.

– Alles, was in einer Zeile steht, finden:
  `^.+`

  Der Suchausdruck findet den Anfang einer Zeile, auf den ein oder mehrere beliebige Zeichen folgen, bis zum Ende der Zeile.

– `$12.34` finden:
  `\$12\.34`

Vor `.` und `$` steht jeweils der Backslash (`\`), weil diese beiden Zeichen Metazeichen sind. Metazeichen werden nicht als Zeichen selbst, sondern als Bestandteil eines Musters interpretiert. Der Backslash (als „Escape-Code“) hebt diese besondere Wirkung auf, so dass tatsächlich nach einem Punkt und nach einem Dollar-Zeichen gesucht wird.

– Eine Zeichenfolge finden, die in *ADbasic* als Variablenname gültig ist:
  `\b[a-z][_a-z0-9]*`

  Der Suchausdruck findet ein Wort, das mit einem Groß- oder Kleinbuchstaben beginnt, worauf kein, ein oder mehrere Buchstaben, Ziffern oder Unterstriche folgen.

– Bei verschachtelten Klammern den innersten Ausdruck finden:
  `\([^\(\)]*\)`

  Der Suchausdruck findet eine öffnende Klammer, auf die kein, ein oder mehrere beliebige Zeichen außer der öffnenden und der schließenden Klammer folgen, auf die wiederum eine schließende Klammer folgt.

– Wiederholungen finden:
  `([0-9]+)-\1`

  Das Suchmuster in Klammern `(…)` steht für eine oder mehrere Ziffern; durch die Klammern wird das Muster gespeichert. Anschließend folgen ein Bindestrich und mit `\1` wieder das gespeicherte Muster. Dieser reguläre Ausdruck würde also etwa `14-14` und `08-08` finden, aber nicht zum Beispiel `08-15`.

Beispiele für das Ersetzen von Text

Beispiele für das Ersetzen mit regulären Ausdrücken.

– Finde zwei Zahlen, die durch ein oder mehrere Leerzeichen getrennt sind:
  `([0-9]+) +([0-9]+)`

  vertausche sie, und trenne sie durch einen Doppelpunkt:
  `$2:$1`

– Um die folgenden Veränderungen simultan durchzuführen:

  aus `X100000` soll werden `X100.000`, aus `Y100123` wird `Y100.123`

  aus `Z600` soll werden `Z.600`

  Suche nach: `([XYZ])([0-9]*)([0-9][0-9][0-9])`

  Ersetze durch: `$1$2.$3`

3.5.3 Reguläre Ausdrücke

Ein regulärer Ausdruck ist ein Suchausdruck, bei dem so genannte Metazeichen (siehe Liste unten) ein variables Suchmuster beschreiben. Die Metazeichen sind nur für das Suchen gültig, nicht für das Ersetzen.

Um mit einen regulären Ausdruck beim Suchen/Ersetzen zu verwenden, müssen Sie die Option `Use regular expressions` im Dialogfenster aktivieren. Rechts neben den Eingabefeldern werden dadurch Schaltflächen „`>`“ aktiv, über die Sie Metazeichen eingeben können.

Die Syntax für reguläre Ausdrücke ist im .NET-Framework 2.0 definiert. Eine ausführliche Beschreibung finden Sie im Internet unter der Adresse `https://docs.microsoft.com` (nach „reguläre Ausdrücke“ suchen).

Metazeichen:	Bedeutung:
.	Ein beliebiges einzelnes Zeichen. Beispiel: Das Muster `Ma.s` trifft unter anderem zu auf `Mais`, `Mars` und `Maus`, nicht aber auf `Mas`.
[ ]	Ein beliebiges Zeichen von denen, die 1. explizit in den eckigen Klammern angegeben sind, oder 2. eines aus dem Bereich von Zeichen, der durch den Bindestrich (`-`) definiert ist. Beispiele: Der Suchausdruck `h[aeiou][a-z]t` findet unter anderem: `hart`, `halt`, `heft`, `holt` und `hut`. Das Muster `[A-Za-z]` entspricht einem beliebigen Buchstaben. Der reguläre Ausdruck `x[0-9]` entspricht `x0`, `x1` usw. bis `x9`.
[^]	Ein beliebiges Zeichen außer denen, die hinter dem Caret (^, „Dach“) stehen. Beispiel: Das Muster `h[^uo]t` passt unter anderem auf `hat` und `hit`, aber nicht auf `hut` oder `hot`.
^	Der Anfang einer Zeile (Spalte 1). Beispiel: Der Suchausdruck `^Anfang` entspricht nur dann dem Text `Anfang`, wenn dieser ganz vorne in einer Zeile steht.
$	Das Ende einer Zeile (letzte Spalte). Benutzen Sie dieses Zeichen und nicht das Zeilenschaltzeichen `\n`, wenn Sie nach einem Ausdruck suchen, der am Ende einer Zeile steht. Beispiel: Der Suchausdruck `Ende$` findet die Zeichenfolge `Ende` nur dann, wenn `Ende` das letzte Wort einer Zeile ist.

Meta-zeichen:	Bedeutung:
\b	Der Anfang eines Wortes.
\B	Das Ende eines Wortes.
\n	Zeilenschaltzeichen (Newline). Zu verwenden bei Ausdrücken, die sich über mehrere Zeilen erstrecken. Hinter `\n` dürfen nicht die Operatoren `*`, `+` oder `{}` stehen. Um ein Suchmuster am Ende einer Zeile zu finden, sollten Sie unbedingt den Operator `$` und nicht das Zeilenschaltzeichen verwenden.
( )	Der Text innerhalb der Klammern wird als Muster in internen Registern abgelegt. Der Inhalt eines Registers kann an anderer Stelle im Suchausdruck oder im Ersetzungsausdruck wieder abgerufen werden. Bis zu 9 gespeicherte Muster sind zulässig. Sie werden in der Reihenfolge ihres Auftretens in dem regulären Ausdruck durchnummeriert. Der Rückbezug auf das Muster lautet `$x` im Ersetzungsausdruck und `\x` im Suchausdruck, mit `x` = 1…9. Beispiel: Der Ausdruck `([a-z]+) ([a-z]+)` findet `isses so`. `$2 $1` würde ihn ersetzen durch `so isses`.
*	Beliebig häufiges Auftreten des vorangehenden Zeichens oder Ausdrucks – kein Mal, einmal oder mehrmals. Beispiel: `hu*f` findet unter anderem `hf`, `huf` und `huuuf`.
?	Kein oder genau ein Auftreten des vorangehenden Zeichens oder Ausdrucks. Beispiel: `hu?f` findet `hf` und `huf`, nicht aber `huuuf`.
+	Mindestens ein (d.h. ein- oder mehrmaliges) Auftreten des vorangehenden Zeichens oder Ausdrucks. Beispiel: `hu+f` findet `huf` und `huuf`, nicht aber `hf`.
|	Entweder der Ausdruck auf der linken Seite des Striches oder der auf der rechten Seite. Beispiel: `huf|huuf` findet `huf` oder `huuf`.

Meta-zeichen:	Bedeutung:
\	Hebt die besondere Wirkung aller Operatoren auf, so dass sie wie normaler, konstanter Text behandelt werden. Um nach dem Backslash \ zu suchen, muss also \\ verwendet werden. Beispiel: ^a bedeutet, dass nach einem a am Anfang einer Zeile gesucht wird, dagegen sucht \^a nach der Zeichenfolge ^a.

3.5.4 Kontrollstruktur markieren

Sie können die Zeilen einer Kontrollstruktur oder eines Programmabschnitts auf einmal markieren, z.B. um verschachtelte Strukturen optisch zu prüfen. Hierzu setzen Sie den Cursor auf eines der Kennwörter der Struktur und wählen im Kontextmenü des Quelltextfensters (Klick mit rechter Maustaste) den Eintrag `Mark Control block`.
Es kann nur eine Kontrollstruktur gleichzeitig markiert werden.

Die Markierung wird mit `Unmark Control block` (im Kontextmenü) wieder gelöscht. Hierbei ist es gleichgültig, an welcher Stelle der Cursor steht.

Folgende Kontrollstrukturen werden erkannt:

- Programmabschnitte **`Init:`**, **`LowInit:`**, **`Event:`**, **`Finish:`**
- `Do` … `Until`
- `For` … `Next`
- `If` … `EndIf`
- `SelectCase` … `EndSelect`
- `Function` … `EndFunction`
- `Sub` … `EndSub`
- `Lib_Function` … `Lib_EndFunction`
- `Lib_Sub` … `Lib_EndSub`

Alle Kontrollstrukturen sind auch faltbare Bereiche (siehe Faltbaren Textbereich definieren, Seite 26).

3.5.5 Textmarken nutzen

Mit Textmarken können Sie, ähnlich einem Lesezeichen, bestimmte Zeilen in einem Quelltext kennzeichnen. Derart markierte Zeilen können einfach angesprungen werden.

Folgende Aktionen stehen zur Verfügung:

– Textmarke setzen

  Sie können eine Textmarke auf eine Zeile setzen, indem Sie entweder in der Editor-Leiste den Befehl `Toggle Bookmark` wählen oder im Dialogfeld `Replace` auf `Bookmark All` klicken (siehe Text suchen und ersetzen).

  Mit `Toggle Bookmark` werden gesetzte Textmarken wieder einzeln entfernt.

– Zur nächsten Textmarke springen

  Wählen Sie in der Editor-Leiste den Befehl `Next Bookmark`.

– Zur vorherigen Textmarke springen

  Wählen Sie in der Editor-Leiste den Befehl `Previous Bookmark`.

– Alle Textmarken entfernen

  Wählen Sie in der Editor-Leiste den Befehl `Delete all Bookmarks`.

  Einzeln werden Textmarken mit `Toggle Bookmark` entfernt.

Textmarken werden mit der Datei zusammen abgespeichert.

3.5.6 Zu einer Programmzeile springen

Sie können zu einer bestimmten Programmzeile im Quelltext springen, indem Sie auf die Zeilennummer in der Statusleiste doppelklicken oder im Menü `Edit` den Eintrag `Goto Line` wählen. Es öffnet sich ein Dialogfenster, in dem Sie die Nummer der gewünschten Programmzeile eingeben.

Um Zeilennummern anzuzeigen, muss die Option `show linenumbers` unter Editor - General (siehe Seite 62) aktiv sein.

3.5.7 Zur Deklaration eines Befehls oder Variablen springen

Sie können von einem Variablennamen aus direkt zu deren Deklaration springen. Diese Möglichkeit gilt für alle selbst deklarierten Namen: lokale Variablen, Felder, Befehle (`Sub`, `Function`) und symbolische Namen (`#Define`).

Sie springen zur Deklaration, indem Sie den Cursor auf den selbst deklarierten Namen setzen und dann entweder im Kontextmenü (rechte Maustaste) den Eintrag `Jump to Declaration` aufrufen, oder die Schaltfläche `Jump to Declaration` in der Editor-Leiste aufrufen.

Der Sprung zur Deklaration ist nur möglich, wenn die Option `Parse Declarations` unter Editor - General (siehe Seite 62) aktiv ist.

Für Befehle aus den Standard-Include-Dateien sowie für globale Variablen **`Par`** / **`FPar`** steht die Funktion nicht zur Verfügung.

3.5.8 Zwischen Sprungzielen wechseln

Sie können zwischen bereits verwendeten Sprungzielen hin- und herwechseln, indem Sie auf die Pfeile `Jump backward` / `Jump forward` in der Editor-Leiste klicken.

Die Oberfläche erstellt automatisch einen Verlauf der verwendeten Sprungziele. Folgende Sprungaktionen werden in den Verlauf aufgenommen:

- Zur Deklaration eines Befehls oder Variablen springen
- Zu einer Programmzeile springen
- Doppelklick in einem der folgenden Fenster des Info-Bereichs:
  - Infofenster (Compiler-Fehler und -Warnungen)
  - ToDo-Liste
  - Fenster „Global Variables“
  - Fenster „Declarations“
- Doppelklick in der Aufrufliste im Fenster Call Tree
- Text suchen und ersetzen

Angesprungene Textmarken werden nicht in den Verlauf aufgenommen, siehe Textmarken nutzen.

Wenn Sie mitten aus dem Verlauf heraus ein neues Ziel anspringen, bleiben die vorherigen Sprungziele im Verlauf erhalten und das neue Ziel wird zum letzten Sprungziel im Verlauf.

Wenn Sie ein Projekt öffnen, wird der Verlauf gelöscht.

3.6 Programme leichter schreiben

Die folgenden Funktionen erleichtern das Programmieren erheblich:

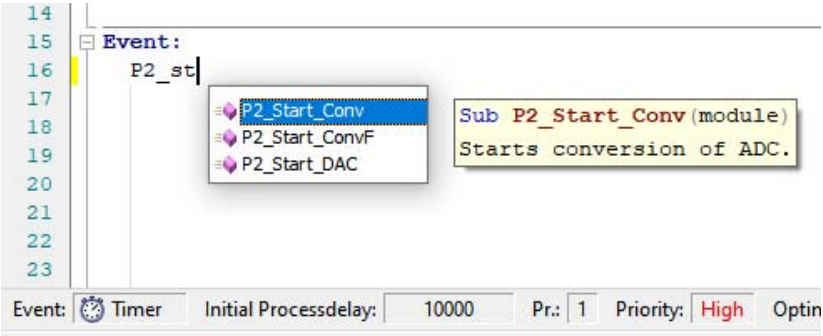
- Befehle und Variablen automatisch vervollständigen, Seite 39
- Befehlsparameter anzeigen, Seite 41
- Textbausteine einfügen, Seite 40
- Eigene Befehle und Variablen dokumentieren, Seite 23
- Deklaration eines Befehls oder Variablen anzeigen, Seite 42
- Deklarationen einer Datei anzeigen, Seite 42
- Verwendete globale Variablen und Felder anzeigen, Seite 42
- Aufrufkürzel zu Dateien / Ordnern einfügen, Seite 43

Hier finden Sie weitere Editor-Funktionen:

- Quelltexte erstellen, Seite 16
- Quelltext formatieren, Seite 21
- Suchen, markieren und ersetzen, Seite 28

3.6.1 Befehle und Variablen automatisch vervollständigen

Sie können Schlüsselwörter, Befehls- und Variablennamen und sogar Textbausteine automatisch vervollständigen, indem Sie die Tastenkombination [CTRL]-[SPACE] drücken.



Durch die automatische Vervollständigung müssen Sie in den meisten Fällen Schlüsselwörter, Befehle und Variablen nicht vollständig ausschreiben.

Um die automatische Vervollständigung zu nutzen, gehen Sie vor wie folgt:

1. Schreiben Sie die ersten Buchstaben des Wortes und drücken [CTRL]-[SPACE].

   Es wird eine Wortliste angezeigt, deren Einträge den bisher eingegebenen Text vervollständigen können.

   Wenn Sie die Vervollständigung nach einem Leerzeichen aufrufen, enthält die Liste alle verfügbaren Kennwörter.

2. Wählen Sie den gewünschten Listeneintrag mit der Maus oder mit den Pfeiltasten aus.

   Rechts neben dem gerade markierten Listeneintrag wird nach einem Moment eine von mehreren Erläuterungen angezeigt:
   - die Deklaration des Befehls oder der Variablen
   - die Angabe „Reserved Keyword“
   - der vollständige Textbaustein (siehe unten Textbausteine einfügen)

3. Wenn Sie weiteren Text eingeben, wird die Liste nicht automatisch aktualisiert. Drücken Sie zur Aktualisierung der Liste erneut die Tastenkombination [CTRL]-[SPACE].

4. Sie übernehmen den markierten Eintrag aus der Liste am besten durch Eingeben einer Klammer (bei einem Befehl) oder eines Leerzeichens.

   Sie können stattdessen aber auch die [EINGABE]-Taste verwenden oder ein anderes, nicht-alphanumerisches Zeichen eingeben.

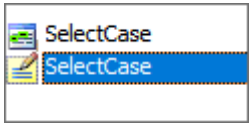
Automatisches Vervollständigen ist nur möglich, wenn die Option `Parse Declarations` unter Editor - General (siehe Seite 62) aktiv ist.

3.6.2 Textbausteine einfügen

Der Editor bietet Ihnen die Möglichkeit, mit Textbausteinen (code snippets) zu arbeiten, welche in einer vorgegebenen Sammlung zusammengefasst sind. Je nach Definition kann ein Textbaustein wenige Zeichen, mehrere Zeilen oder ein ganzes Programm einfügen.

Sie fügen einen Textbaustein an der Cursor-Position ein, indem Sie eine der folgenden Aktionen ausführen:

– Sie geben die ersten Zeichen eines Textbaustein-Kennworts ein, z.B. `Sele` für eine SelectCase-Struktur, drücken die Tastenkombination [CTRL]-[SPACE] und



wählen aus der Liste den Textbaustein `SelectCase`.
Siehe auch Befehle und Variablen automatisch vervollständigen.

– Sie rufen über Kontextmenü oder Editor-Leiste die Funktion `Code-snippets` auf. Es erscheint eine Auswahlliste mit Ordnern, die jeweils mehrere Textbausteine (oder weitere Ordner) enthalten.

  Sie navigieren durch die Menüs per Maus oder per Tastatur. Folgende Tasten sind belegt:
  - Pfeil oben/unten: Listeneintrag markieren.
  - Eingabe: Markierten Textbaustein einfügen oder neue Menüebene öffnen.
  - Backspace: In vorige Menüebene zurückkehren.

  Sobald Sie einen Textbaustein markiert haben, erscheint rechts davon das zugehörige Tastaturkürzel.

– Sie geben das Tastaturkürzel des Textbausteins ein, gefolgt von [TAB].

Um die Liste der Textbausteine und der Tastaturkürzel zu sehen, öffnen Sie die Datei `codesnippets.xml` im Ordner `C:\ADwin\ADbasic\Common\` mit einem Browser.

3.6.3 Befehlsparameter anzeigen

Bei Befehlen werden die zugehörigen Übergabeparameter mit einer Beschreibung automatisch als Tooltip angezeigt, sobald Sie nach dem Befehlsnamen eine öffnende Klammer eingeben. Im Tooltip wird derjenige Übergabeparameter fett dargestellt, den Sie gerade eingeben.

```
27  Event:
28      max = my_func(
```

```
Function my_func(val1, val2) As Long
val1: Ein Einzeiler für den ersten Übergabeparameter
```

Der Tooltip verschwindet, sobald der Cursor außerhalb der Klammern um die Parameter steht. Sie können die Anzeige mit [CTRL]-[SHIFT]-[SPACE] wieder aktivieren, wenn der Cursor innerhalb der Klammern steht.

Auch selbstdefinierte Befehle und ihre Parameter können Sie mit Hinweistexten dokumentieren und auf gleiche Weise anzeigen lassen, siehe „Eigene Befehle und Variablen dokumentieren“ auf Seite 23.

Die automatische Anzeige von Befehlsparametern ist nur möglich, wenn die Option `Parse Declarations` unter Editor - General (siehe Seite 62) aktiv ist.

3.6.4 Deklaration eines Befehls oder Variablen anzeigen

Sie können zu Befehlen, Variablen und anderen deklarierten Kennwörtern die zugehörige Deklaration sowie Hinweise als Tooltip anzeigen, indem Sie

– den Mauszeiger auf das Wort positionieren.

  Die Deklaration wird nur dann automatisch angezeigt, wenn die Option `Display quick info on mouse over` unter `Editor - General` (siehe Seite 62) aktiv ist.

– den Cursor auf das Kennwort setzen und die Taste [F2] drücken.

– den Cursor auf das Kennwort setzen und in der Editor-Werkzeugliste oder im Kontextmenü den Eintrag `Declaration Info` wählen.

Diese Möglichkeit gilt für alle Kennwörter, die zum Sprachumfang von *ADbasic* gehören oder selbst deklariert wurden: lokale und globale Variablen, Felder, Befehle (`Sub`, `Function`) und symbolische Namen (`#Define`).

Die Anzeige der Deklaration ist nur möglich, wenn die Option `Parse Declarations` unter Editor - General (siehe Seite 62) aktiv ist.

3.6.5 Deklarationen einer Datei anzeigen

Sie können alle zu einer Quelltextdatei gehörenden Deklarationen, Include- und Library-Dateien anzeigen, indem Sie im Info-Bereich das Fenster „Declarations“ in den Vordergrund stellen. Deklarationen aus anderen Quelltextdateien – auch innerhalb eines Projekts – werden nicht berücksichtigt.

Die Deklarationen können nur angezeigt werden, wenn die Option `Parse Declarations` unter Editor - General (siehe Seite 62) aktiv ist.

3.6.6 Verwendete globale Variablen und Felder anzeigen

Um globale Variablen und Felder anzuzeigen, die im aktiven Quelltext und im zugehörigen Projekt (falls vorhanden) verwendet werden, drücken Sie die Schaltfläche `Scan Global Variables` im Parameterfenster (siehe auch Seite 79) drücken.

Sie erhalten zwei Anzeigen:

– im Fenster „Global Variables“ werden alle verwendeten globalen Variablen und Felder (siehe Seite 95) angezeigt.

– im Parameterfenster werden alle verwendeten globalen Variablen farbig hervorgehoben (globale Felder nicht).

Die Hervorhebung ist farbig je nach Verwendung der Parameter:

- Grün: Parameter wird nur im aktiven Quelltext verwendet. 2 1336227
- Rot: Parameter wird im aktiven Quelltext verwendet, aber auch in einem anderen Quelltext des Projekts. 1 707279
- Blau Parameter wird im Projekt verwendet, jedoch nicht im aktiven Quelltext. 9 23314

Mit der Schaltfläche `Clear Scan` werden beide Anzeigen wieder gelöscht.

Bei Änderungen im Quelltext werden die Anzeigen nicht automatisch aktualisiert. Rufen Sie stattdessen `Scan Global Variables` erneut auf.

3.6.7 Aufrufkürzel zu Dateien / Ordnern einfügen

Sie können Links zu externen Dateien, Programmen oder Ordner als Aufrufkürzel unterhalb der ADtools-Leiste einfügen.

Zum Einfügen klicken sie mit der rechten Maustaste auf die *ADtools*-Leiste und wählen im Kontextmenü den Menüeintrag `Add file shortcut` oder `Add folder shortcut`. Im Dateiauswahlfenster legen Sie fest, zu welcher Datei der neu angelegte Link führen soll.

Mit einem Doppelklick auf den Link wird die Datei oder der Ordner aktiviert; das Betriebssystem des Rechners führt dann die Aktion aus, die mit dem Dateityp verknüpft ist.

Zum Löschen klicken sie mit der rechten Maustaste auf den Link und wählen im Kontextmenü den Menüeintrag `Remove: <file name>`.

3.7 Programmfehler finden

Die Entwicklungsumgebung stellt mehrere Hilfsmittel zur Verfügung, um Fehler im Programm zu finden. Die Hilfsmittel sind in Kapitel 5.3 „Debugging und Analyse“ (Seite 152) beschrieben:

– Laufzeitfehler erkennen (Debug-Modus), Seite 152

– Zeitverhalten prüfen (Timing-Modus), Seite 153

– Laufzeitfehler erkennen (Debug-Modus), Seite 152

– Programmstruktur analysieren, Seite 156

– Speicherintensive Prozeduren optimieren, Seite 157

3.8 Bootloader nutzen

Dieser Abschnitt bezieht sich auf den Prozessor T12.1; für andere Prozessoren ist der Umgang mit dem Bootloader im Handbuch „ADwin-Bootlader“ beschrieben.

Wie arbeitet der Bootloader? Beim Einschalten der *ADwin*-Hardware startet der Bootloader das Betriebssystem und lädt bis zu 10 *ADbasic*-Prozesse automatisch; anschließend wird der Prozess 10 gestartet. Der Bootloader kann aktiviert und deaktiviert werden.

3.8.1 Bootloader programmieren

Sie programmieren den Bootloader, indem Sie Binärdateien (als *ADbasic*-Prozesse) und das Betriebssystem in die *ADwin*-Hardware übertragen. Dazu gehen Sie vor wie folgt:

– Stellen Sie den Prozessor T12.1 im Dialogfenster „Compiler Options“ ein, siehe Seite 56.

– Erzeugen Sie die benötigten Binärdateien, beispielsweise mit `Build` ▶ `Make Bin File`; siehe Seite 54.

  Stellen Sie dabei die Eigenschaften (Priorität, Prozessnummer, Prozesstyp etc.) für jeden Prozess ein.

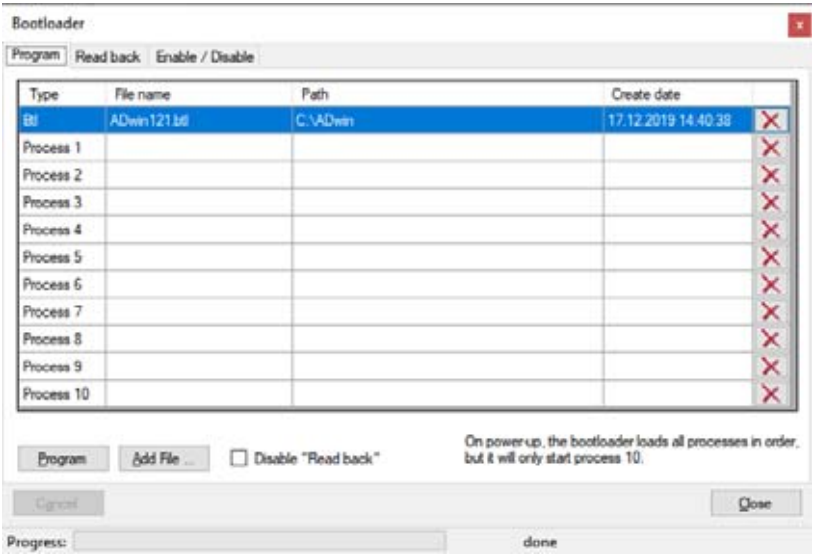
– Öffnen Sie das Dialogfenster `Bootloader`.

  Sie haben zwei Möglichkeiten:

  - Markieren Sie im Projektfenster Quelltextdateien, und wählen Sie anschließend in der Projektleiste die Schaltfläche `Bootloader` Die ausgewählten Dateien werden im automatisch im Fenster `Bootloader` eingetragen.

- Wählen Sie im Menü `Tools` den Eintrag `Bootloader...`, um das Fenster `Bootloader` direkt zu öffnen. Sie wählen die Binärdateien anschließend einzeln aus.

Das Dialogfenster `Bootloader` öffnet sich. Im Tab `Program` ist bereits die Betriebssystemdatei (`btl`) in der Tabelle eingetragen sowie beim Aufruf über die Projektleiste auch ausgewählte Binärdateien.



Zur Klarheit enthält die Tabelle den Pfad und das Erstellungsdatum der BTL und jeder Binärdatei. Prüfen Sie unbedingt, ob die Binärdateien aktuell sind.

– Fügen Sie mit der Schaltfläche `Add file` für jeden gewünschten Prozess die passende Binärdatei hinzu, bis alle benötigten Binärdateien in der Tabelle angezeigt werden. Sie können mehrere Dateien auf einmal hinzufügen.

  Mit der Schaltfläche `Delete` (letzte Spalte) können Sie Binärdateien wieder aus der Tabelle löschen.

  Achten Sie darauf, den Prozess 10 zu verwenden, der als einziger Prozess vom Bootloader automatisch gestartet wird. Alle weiteren Abläufe ergeben sich daraus, wie Sie den Prozess 10 programmiert haben; beispielsweise können andere geladene Prozesse gestartet werden.

– Stellen Sie mit der Option `Disable "read back"` ein, ob das Rücklesen der Binärdateien aus der *ADwin*-Hardware erlaubt ist, siehe Prozesse aus dem Bootloader zurücklesen.

– Mit der Schaltfläche `Program` starten Sie den Programmiervorgang.

  Der Balken ganz unten zeigt den Fortschritt der Übertragung der Binärdateien in den Bootloader an.

  Nach dem Programmieren wird der Bootloader automatisch aktiviert.

– Schließen Sie das Fenster wieder mit `Close`.

3.8.2 Prozesse aus dem Bootloader zurücklesen

Sie können die Dateien aus dem Bootloader Gehen Sie vor wie folgt:

– Lesen Sie die Inhalte des Bootloaders mit der Schaltfläche `Read back` im gleichnamigen Tab.

  Wenn das Rücklesen verboten war, erhalten Sie einen entsprechenden Hinweis und die Schaltfläche `Save files` bleibt deaktiviert.

  Die Dateiinformationen werden mit Name, Originalpfad und Erstellungsdatum in der Tabelle angezeigt.

– Mit der Schaltfläche `Delete` (letzte Spalte) können Sie Binärdateien aus der Tabelle löschen.

– Mit der Schaltfläche `Save files` speichern Sie alle angezeigten Dateien in einem frei wählbaren Verzeichnis.

3.8.3 Bootloader aktivieren / deaktivieren

Der Bootloader ist nach dem Programmieren automatisch aktiviert, siehe Bootloader programmieren. Sie können den Bootloader jederzeit im Tab `Enable/Disable` mit den Schaltflächen `Enable bootloader` und `Disable bootloader` aktivieren und deaktivieren.

Mit der Schaltfläche `Read status` stellen Sie fest, ob der Bootloader aktiviert oder deaktiviert ist.

3.9 Projekte verwalten

Sie können beliebig viele Quelltexte, Include-Dateien, Library-Dateien sowie Dateien anderen Typs gemeinsam als Projekt verwalten, beispielsweise wenn Sie eine Anwendung mit mehreren Prozessen programmieren. Die Dateien werden im Projektfenster (Seite 76) dargestellt. Es kann jeweils nur ein einziges Projekt geöffnet sein.

Mit dem Projekt wird die Darstellung der Bedienoberfläche gespeichert: Fenstergröße, Fensterposition, geöffnete Projektdateien. Außerdem (siehe Dialogfenster „Project Settings“) können Sie weitere Programmeinstellungen mit dem Projekt speichern lassen sowie Programmaufrufe vor und nach dem Kompilieren (`Prebuild` / `Postbuild`). Beim Öffnen des Projekts werden die gespeicherten Einstellungen wieder hergestellt.

Folgende Funktionen stehen in einem Projekt zur Verfügung.

– Projekt verwalten
  - Neues Projekt erstellen: `File` ▸ `New Project`.
  - Gespeichertes Projekt öffnen: `File` ▸ `Open Project`.
  - Projekt speichern: `File` ▸ `Save Project` / `Save Project As`.
    Es werden nur die Projekteinstellungen gespeichert. Änderungen in Projektdateien können z.B. mit `Save all Files of Project` gespeichert werden.
  - Projekt schließen: `File` ▸ `Close Project`.
    Mit dem Projekt werden auch die zum Projekt gehörenden Dateien geschlossen.
  - Projekteinstellungen ändern: `Options` ▸ `Project Settings`

    oder Schaltfläche `Project Settings` in der Projektleiste; siehe Dialogfenster „Project Settings“.

– Dateien ins Projekt einbinden
  - Eine Datei in das Projekt einbinden: `File` ▸ `Add to Project` oder `Add to Project` im Kontextmenü des Quelltextfensters.
  - Alle geöffneten Dateien in das Projekt einbinden: `Add Open Files To Project` im Kontextmenü des Projektfensters, siehe Seite 76.
  - Eine Simulink-Modelldatei in ein *ADsim*-Projekt einbinden:

    Schaltfläche `Begin/Add ADsim Project` in der Projektleiste oder
    im Kontextmenü des Projektfensters, siehe Seite 76.
    Die Datei wird im Projektfenster unter `Mathworks Simulink` aufgeführt.

- Eine Datei eines anderen Typs in das Projekt einbinden: `Add other File to Project` im Kontextmenü des Projektfensters, siehe Seite 76.
  Die Datei wird im Projektfenster unter `Other Files` aufgeführt.

Wir empfehlen, die Projektdateien im gleichen Ordner zu speichern wie das Projekt (`*.abp`) oder einem zugehörigen Unterordner. Damit sind beim Kopieren des Ordners alle Dateien im neuen Projekt enthalten und die Dateien können unabhängig vom alten Projekt benutzt werden.

Beim Einbinden einer Datei ins Projekt wird der Dateipfad relativ zum Projektordner gespeichert. Nur wenn die Datei auf einem anderen Laufwerk liegt, wird der Dateipfad absolut gespeichert.

– Dateien eines Projekts verwalten
  - Eine Datei des Projekts öffnen und zum aktiven Quelltext machen, mit einem Doppelklick auf die Datei im Projektfenster, siehe Seite 76.
  - Markierte Dateien aus dem Projekt entfernen: Taste [ENTF] oder `Remove from Project` im Kontextmenü des Projektfensters, siehe Seite 76.
  - Alle Dateien des Projekts auf einmal speichern, mit `Save all Files of Project` im Kontextmenü des Projektfensters, siehe Seite 76.

– Projekt-Quelltexte erzeugen (Simulink)
  - Zu den im Projektfenster markierten Simulink-Modelldateien einen Quelltext erzeugen (nur mit *ADsim*-Lizenz):
    Wählen Sie die Schaltfläche `Import ADsim Library as Template` in der Projektleiste (siehe ADbasic und ADsim auf Seite 100).
    Der erzeugte Quelltext wird in das Projekt eingefügt.
  - Einen Quelltext mit Bezug zu einer im Projekt enthaltenen Simulink-Modelldatei aktualisieren (nur mit *ADsim*-Lizenz):
    Wählen Sie die Schaltfläche `Update Project from Simulink library` in der Projektleiste (siehe ADbasic und ADsim auf Seite 100).
    Die Informationen aus der Modelldatei werden in den Quelltext

importiert. Beachten Sie dabei die Einstellung `Import block captions` unter Reiter General.

– Projekt-Quelltexte übersetzen
  - Markierte Dateien des Projekts kompilieren: Schaltfläche `Compile selected Files` in der Projektleiste.
    Der Menüeintrag `Build` ▸ `Compile` bezieht sich nicht auf das Projekt, sondern nur auf die aktive Datei.
  - Binärdateien aus markierten Dateien des Projekts erzeugen:
    Schaltfläche `Make Bin Files from selected Files` in der Projektleiste.
    Der Menüeintrag `Build` ▸ `Make Bin File` bezieht sich nicht auf das Projekt, sondern nur auf die aktive Datei.
  - Alle Binärdateien des Projekts erzeugen:
    `Build` ▸ `Make all Bin Files of Project` oder
    Schaltfläche `Make All Bin Files of Project` in der Projektleiste.
  - Markierte Binärdateien des Projekts in die *ADwin*-Hardware übertragen: Schaltfläche `Load selected Bin Files into ADwin system` in der Projektleiste.
  - Die Binärdateien aller im Projektfenster markierten Dateien in den Bootloader übertragen (nur mit Prozessor T12.1):
    Wählen Sie die Schaltfläche `Program selected Files into the Bootloader` in der Projektleiste (siehe Bootloader programmieren auf Seite 44).
  - Kommandozeilen-Aufrufe automatisch vor jedem Übersetzen (`Prebuild`) bzw. danach (`Postbuild`) ausführen, siehe Reiter Prebuild / Postbuild.

  Wenn im Dialogfenster „Compiler Options“ die Option `Autostart` aktiv ist, werden Binärdateien automatisch in die *ADwin*-Hardware übertragen und gestartet.

– Alle Dateien des Projekts durchsuchen

  Sie können auch die nicht geöffneten Dateien des Projekts durchsuchen. Ausgenommen sind Dateien aus dem Projektbereich `Other Files`).

  Aktivieren Sie zum Durchsuchen aller Dateien die Option `All Documents of Project` im Suchen-Fenster (siehe Kapitel 3.5.2 „Text su-

chen und ersetzen“).
Diese Option ist für das Ersetzen nicht verfügbar.

– Globale Variablen im Projekt hervorheben (siehe Kapitel 3.6.6).

– Windows Explorer mit dem Pfad der markierten Datei öffnen, mit `Open Path in Explorer Window` im Kontextmenü des Projektfensters.

Die Befehle zur Projektverwaltung finden Sie im Kontextmenü des Projektfensters (siehe auch „Projektfenster” auf Seite 76), in der Projektleiste des Projektfensters oder im Menü `File` (Kapitel 3.10.1).

3.10 Menüs

Sie finden in der Menüleiste folgende Funktionen:

–	File:	Dateien und Projekte verwalten.	(Seite 52)
–	Edit:	Quelltexte editieren.	(Seite 53)
–	View:	Fenster und Leisten anzeigen.	(Seite 53)
–	Build:	Ausführbare Programme erzeugen.	(Seite 54)
–	ADsim:	*ADsim*-Projekte organisieren.	(Seite 55)
–	Options:	Optionen aller Art einstellen.	(Seite 56)
–	Debug:	Hilfe zur Fehlersuche.	(Seite 69)
–	Tools:	Verschiedene Hilfsfunktionen.	(Seite 73)
–	Window:	Quelltextfenster anordnen.	(Seite 74)
–	Help:	Hilfe, Versions- und Lizenz-Informationen.	(Seite 74)

3.10.1 Das Menü „File“

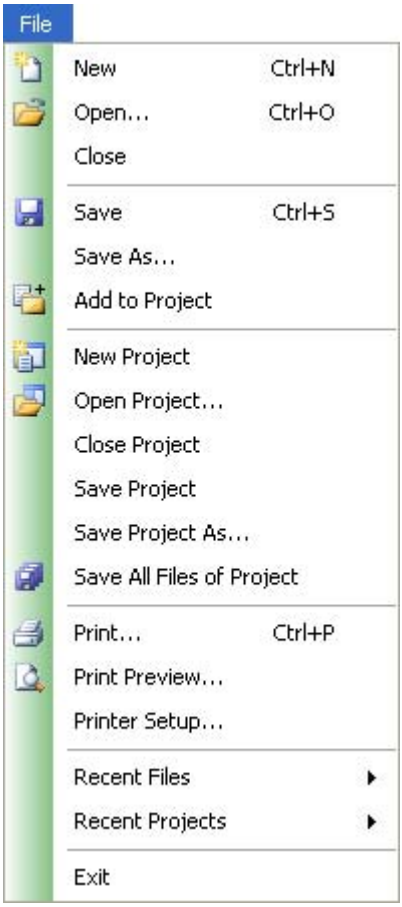
Das Menü `File` enthält Befehle zum Verwalten von Dateien und Projekten.

Mit den Befehlen können Sie Dateien öffnen, neu anlegen, speichern oder schließen. Sie können beliebig viele Quelltexte bearbeiten, aber höchstens 10 Prozesse gleichzeitig auf Ihr *ADwin*-System laden.

In gleicher Weise wie Dateien können Sie auch ein Projekt öffnen, speichern und neu erzeugen; es kann nur ein einziges Projekt gleichzeitig geöffnet sein kann. Weitere Befehle sind über das Projektfenster zugänglich (siehe Kapitel 3.11.2).

Das Menü enthält auch die Druckfunktionen (Drucken, Druckvorschau und Druckereinrichtung).

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



Quelltextdateien werden mit einem eigenen Format gespeichert. Um Dateien mit *ADbasic 4* zu nutzen, kann man sie mit `Save as` in den Dateityp `ADbasic4 Bas-File` konvertieren.

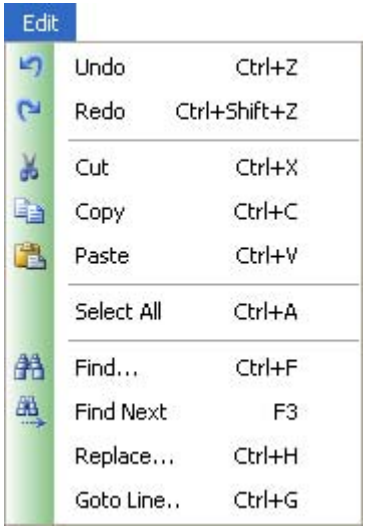
☞ Beachten Sie: Wenn Sie einen Quelltext als Library-Datei (Lib-File) speichern, empfehlen wir, an den Dateinamen die Folge `_lib` anzuhängen, z.B. `functions_lib.bas`. Dagegen darf ein Library-Dateiname nicht mit der Folge `.<Ziffer>` enden (z.B. `functions.1.bas`).

3.10.2 Das Menü „Edit“

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

Darüber hinaus enthält das Menü eine Such-Funktion (`Find`, `Find Next`) sowie eine Ersetzen-Funktion (`Replace`); siehe Text suchen und ersetzen.

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

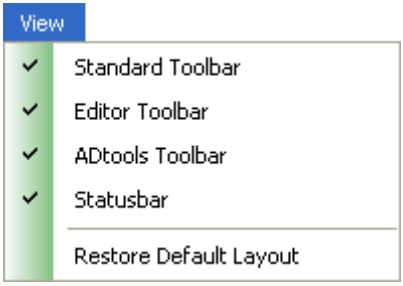


3.10.3 Das Menü „View“

Im Menü `View` stellen Sie ein, welche Leisten der Entwicklungsumgebung angezeigt werden:

- die Werkzeugleiste (Standard Toolbar)
- die Editor-Leiste
- die ADtools-Leiste
- die Statuszeile

Näheres zum Prozessfenster finden Sie in Kapitel 3.11.4 auf Seite 81, zur Werkzeugleiste siehe Abb. 2.



Mit `Restore Default Layout` stellen Sie mit einem Tastendruck die Standard-Ansicht wieder her, die beim ersten Öffnen des Programms *ADbasic* aktiv war, darunter auch die Einstellungen der Toolbox.

3.10.4 Das Menü „Build“

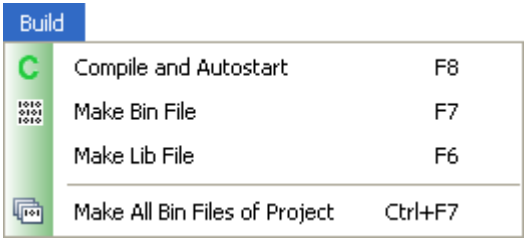
Im Menü `Build` können Sie Quelltext übersetzen:

– mit `Compile` in einen Prozess.

– mit `Make Bin File` in eine Binärdatei.

– mit `Make Lib File` in eine Library.

– mit `Make all Bin Files of Project` alle Dateien des Projekts in Binärdateien.



Bitte beachten Sie: Vor dem Kompilieren werden automatisch alle geänderten Quelltexte, Library- und Include-Dateien gespeichert. (AutoSave)
Eine Änderung kann auch durch automatisches Einrücken von Textzeilen (siehe Kapitel 3.4.3 auf Seite 22) stattfinden, beispielsweise beim Öffnen einer vorher nicht formatierten Datei.

`Compile` ist der umfassendste Befehl des Menüs: Er übersetzt den aktiven Quelltext und überträgt den erzeugten Binärcode als Prozess auf das *ADwin*-System.

Der Prozess wird außerdem auch 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 Abb. 10).

Fehler und Warnungen beim Kompilieren werden im Infofenster angezeigt. Ein Doppelklick auf die Fehlermeldung markiert die betreffende Zeile rot.

`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 auch Das Menü „Options“, Dialogfenster „Process Options“).

Eine Binärdatei mit der Endung `*.TC3` beispielsweise können Sie auf ein *ADwin*-System mit dem Prozessor T12 übertragen, der den Prozess unter der Nummer 3 startet. Sie können Binärdateien aus allen Entwicklungsumgebungen (wie C oder Visual Basic) an das *ADwin*-System übertragen (siehe Kapitel 6.3.2 auf Seite 171).

`Make Lib File` ist nur für lizenzierte *ADbasic*-Benutzer verfügbar. Der Befehl übersetzt den aktiven Quelltext – die Datei muss mit dem Dateityp `Lib-File` gespeichert sein – 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 `.LIx`. 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 4.5.3 auf Seite 139).

Beachten Sie beim Prozessor T12, dass zu jeder Library neben der Datei `*.LIC` außerdem die Dateien `*.O` und `*.H` gehören. Alle drei Dateien müssen sich im gleichen Verzeichnis befinden, damit die Library korrekt arbeitet.

`Make All Bin Files of Project` ist nur für lizenzierte *ADbasic*-Benutzer verfügbar. Der Befehl ist eine Kombination der Befehle `Make Bin File` und `Make Lib File`: Er übersetzt alle Quelltexte des Projekts und erzeugt dabei sowohl Library-Dateien als auch Binärdateien.
Beachten Sie, dass die Dateien in der Reihenfolge kompiliert werden, wie sie im Projektfenster angezeigt werden. Wenn Sie bei Library-Dateien eine andere Reihenfolge benötigen, müssen Sie die Dateien einzeln mit `Make Lib File` kompilieren.

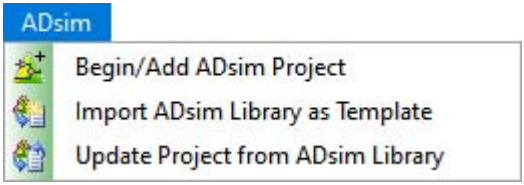
3.10.5 Das Menü „ADsim“

Im Menü `ADsim` können Sie *ADsim*-Projekte organisieren:

– `Begin/Add ADsim Project`: Eine Simulink-Modelldatei zum Projekt hinzufügen.

– `Import ADsim Library as Template`: Zu den im Projektfenster markierten Simulink-Modelldateien einen Quelltext erzeugen.

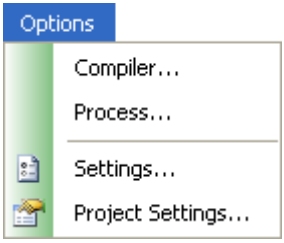
– `Update Project from ADsim Library`: Einen Quelltext mit Bezug zu einer im Projekt enthaltenen Simulink-Modelldatei aktualisieren.



Weitere Informationen finden Sie im Abschnitt ADbasic und ADsim.

3.10.6 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 die Einstellungen in diesem Dialogfenster für jedes System neu anpassen.

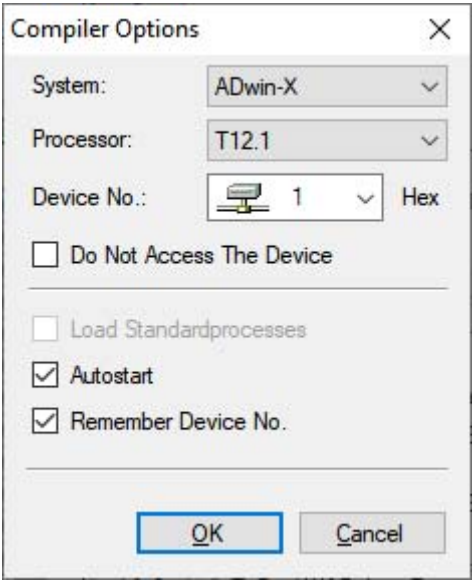


Abb. 3 – Das Dialogfenster `Compiler Options`

– `System`: Wählen Sie den Eintrag aus, der Ihrem *ADwin*-System entspricht.

– `Processor`: Wählen Sie den Prozessor des *ADwin*-Systems.

  Die Kurzbezeichnungen der Prozessor-Typen entsprechen den folgenden Vollbezeichnungen:

Kurzbezeichnung	T8	T5	T4	T2
Vollbezeichnung	T805	T450	T400	T225

Kurzbezeichnung	T12/ T12.1	T11	T10	T9
Vollbezeichnung	Xilinx Zynq 7z030	ADSP TS101S	ADSP 21160	ADSP 21062

Abb. 4 – Prozessorbezeichnungen

– `Device No.`: Wählen Sie die Gerätenummer, mit der das gewünschte *ADwin*-System angesprochen werden kann.

  Sie vergeben die Gerätenummer mit dem Programm `<ADconfig.exe>`. Die Werkseinstellung ist `1 Hex`.

– `Do Not access the device`: Bei deaktivierter Option ist der Zugriff auf *ADwin*-Hardware freigegeben. Insbesondere wird die beim Kompilieren erzeugte Binärdatei automatisch zur Hardware übertragen. Die Hardware muss daher für das Kompilieren erreichbar sein.

  Bei aktiver Option können Quelltexte auch dann für die eingestellte *ADwin*-Hardware kompiliert werden, wenn sie nicht mit dem PC verbunden ist. Einige Menüeinträge und Schaltflächen sind deaktiviert, um versehentliche Zugriffe auf *ADwin*-Hardware zu vermeiden.

– `Load standard processes`: Bei aktiver Option werden während des Boot-Vorgangs die Standard-Prozesse 11, 12 und 15 (siehe Kapitel 6.1.1 auf Seite 160) in das *ADwin*-System übertragen. Bei deaktivierter Option wird die Übertragung der Prozesse 11 und 12 unterdrückt.

  Die Einstellung ist nur für *ADwin-Gold* und *ADwin-light-16* verfügbar.

– `Autostart`: Bei aktiver Option wird die beim Kompilieren erzeugte und ins *ADwin*-System übertragene Binärdatei sofort als Prozess gestartet. Bei deaktivierter Option müssen Sie den Prozess manuell mit der Schaltfläche ▶ aus der Werkzeugleiste oder im Prozessfenster starten.

– `Remember Device No.`: Bei aktiver Option wird die zuletzt verwendete Device No. (siehe oben) beim Schließen des Programms gespeichert; beim Neustart wird diese Nummer automatisch eingestellt.

  Bei deaktivierter Option wird die Device No. nicht gespeichert. Beim Start von *ADbasic* wird die zuletzt – bei aktiver Option – gespeicherte Device No. verwendet.

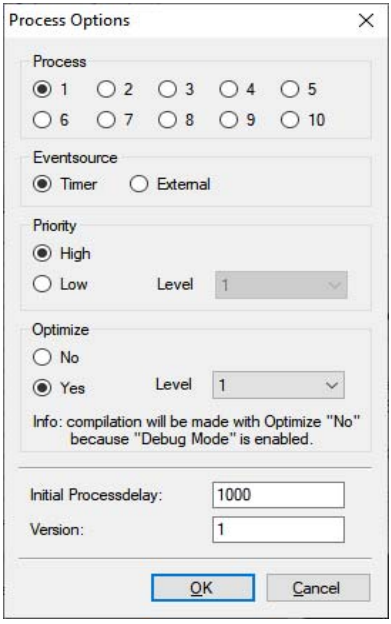
Dialogfenster „Process Options“

Sie legen in diesem Dialogfenster Compiler-Optionen für das aktive Quelltextfenster fest, d.h. Sie bestimmen Eigenschaften des Prozesses, der aus dem aktiven Quelltext übersetzt und zur *ADwin*-Hardware übertragen wird.

Dies gilt sinngemäß auch für Library-Dateien, bei denen aber nur die Option `Optimize` eingestellt werden kann.

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). Sie können das Dialogfenster mit einem Doppelklick auf die Statuszeile im Quelltextfenster öffnen. Das Dialogfenster für die Prozessortypen T4, T5 oder T8 weicht vom Standardfenster in wenigen Punkten ab und ist im Anhang A.5 beschrieben.

Einstellungen für Quelltext



Einstellungen für Library

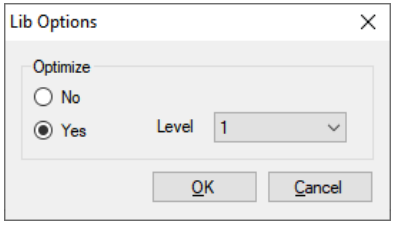


Abb. 5 – Das Dialogfenster `Process Options`

– `Process`: Prozessnummer

  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.
  - `Timer`
    regelmäßige Impulse des internen Zählers dienen als Event-Signal. Mit der Systemvariablen **`Processdelay`** legen Sie fest, in welchen Zeitabständen der Zähler ein Event-Signal auslöst.
  - `External`
    Ein Signal am Event-Eingang der *ADwin*-Hardware startet den Prozess. Dies könnte beispielsweise ein Impuls eines Messaufnehmers sein. Ein solcher Prozess läuft immer mit hoher Priorität ab.
    Wie Sie bei einem *ADwin-Pro*-System einen externen Event-Eingang nutzen können, lesen Sie bitte in der Software-Dokumentation unter dem Befehl **`EventEnable`** nach.

– `Priority`: Priorität des Prozesses.

  Stellen Sie die Priorität des Prozesses ein, mit der er im System laufen soll. Weitere Informationen zu diesem Thema finden Sie in Kapitel 6.1 „Prozessverwaltung“. Der Eintrag `Level` (-10…+10) definiert die Priorität innerhalb der *niederprioren* 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 höherer Wert unter `Level` führt zu kürzeren Ausführungszeiten. Beim Prozessor T12/T12.1 gibt es zusätzlich die Einstellungen `fast` und `size`, die eine besondere Optimierung auf Schnelligkeit oder geringe Speichergröße erlauben.

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

Bei Prozessoren bis T11 wird die Optimierung automatisch deaktiviert, solange der Debug-Modus aktiv ist.

– `Initial Processdelay`: temporäre Anfangs-Zykluszeit.

  Sie stellen hier temporär das Processdelay (Zykluszeit) ein, mit dem der Prozess beginnt. Die Einstellung bleibt in der Oberfläche unverändert, so lange *ADbasic* geöffnet ist. Stellen Sie den Wert passend zum Prozessortyp ein.

  Für eine dauerhafte Einstellung können Sie die Variable Processdelay im Quelltext setzen, z. B. im Abschnitt **`Init:`**.

– `User Version`: Ganzzahliger Wert, der als Versionsnummer des *ADbasic*-Programms dient.

  Die Versionsnummer wird in der erzeugten Binärdatei abgelegt. Sie können die Versionsnummer direkt aus der Binärdatei auslesen; wenden Sie sich dazu bitte an unseren Support (support@adwin.de).

  Beim T12/T12.1 können Sie die Version in *ADbasic* mit **`User_Version`** abfragen.

– `Stack size`, nur Prozessor T12/T12.1: Größe des Programm-Stapelspeichers (Stack) in Bytes.

  Die Größe ist auf 1000 Byte voreingestellt.Für Libraries kann es erforderlich sein, einen größeren Wert anzugeben. Die Anzahl der lokalen Variablen und Arrays in der Library bestimmt den Platzbedarf des Programm-Stapelspeichers (Stack). Eine ungenügende Stack-Größe kann zu Datenverlust oder Programmabbruch führen. Mit der Option Stack Test (Seite 71) prüfen Sie, ob die Größe des Stapelspeichers ausreicht.

– `Number of Loops`: Die Einstellung ist nur bei den Prozessoren T2…T8 verfügbar.

Dialogfenster „Settings“

Das Dialogfenster hat mehrere Seiten, die Sie über das Baumdiagramm im linken Teilfenster anwählen:

– Editor
  - Editor - General
  - Editor - Syntax Colors
  - Editor - Print Settings

– `General`

– `Language`

– `Directories`

– `ADtools`

Editor - General

`Parse and format`: Der Editor kann den Quelltext automatisch formatieren, z.B. einrücken und die Syntax hervorheben. Hierfür ist es nötig, dass der Editor alle Quelltexte kontinuierlich durchsucht (engl.: to parse). Die gefundenen Informationen dienen auch als Basis für komfortable Funktionen wie: Befehle und Variablen automatisch vervollständigen, Deklarationen einer Datei anzeigen oder Befehlsparameter anzeigen.

☞ Beachten Sie, dass das ständige Durchsuchen der Quelltexte auf langsamen Rechnern zu Geschwindigkeitseinbußen führen kann.

`Parse Declarations`: Der Editor durchsucht die Quelltexte kontinuierlich. Davon abhängig ist eine Reihe komfortabler Funktionen.

`Autoindent`: Der Quelltext wird automatisch eingerückt (engl.: to indent). Die Einrückpositionen werden über `Tabsize` festgelegt. Siehe auch „Textzeilen einrücken“ auf Seite 22.

`Indent ADbasic sections`: Programmabschnitte werden zusätzlich um eine Stufe eingerückt.

`Smart format`: Zeilen automatisch formatieren, siehe „Automatisch formatieren“ auf Seite 22.

`Align comments at specified position`: Kommentar hinter Quellcode wird automatisch auf die angegebene `Position` gesetzt.

Beachten Sie: Mit doppelten Kommentarzeichen `''` können Sie einen Kommentar dennoch manuell positionieren.

`Display quick info on mouse over`: Wenn Sie den Mauszeiger auf ein Wort setzen, erscheinen zugehörige Deklaration und Hinweise als Tooltip. Siehe auch „Deklaration eines Befehls oder Variablen anzeigen“ auf Seite 42.

`Tabsize`: Geben Sie ein, um wieviele Leerzeichen ein einzelner Tabulatorsprung eine Zeile einrückt. Zum Einrücken werden grundsätzlich nur Leerzeichen verwendet.

`Font size`: Geben Sie ein, mit welcher Zeichengröße der Quelltext angezeigt wird, siehe auch Editor - Syntax Colors.
Beachten Sie: Andere Texte in der Bedienoberfläche bleiben unverändert.

“`#If Processor = …`“ `graying`: Ein bedingter Anweisungsblock wird grau dargestellt, wenn die aktuelle Prozessoreinstellung nicht der Bedingung mit dem Systemparameter `Processor` entspricht; siehe auch `#If … Then … {#Else … } #EndIf` auf Seite 241. Andere Bedingungen werden nicht berücksichtigt.

`Show line numbers`: Im Randstreifen (engl.: gutter) links vom Quelltext werden die Zeilennummern des Quelltexts angezeigt. Siehe auch „Zu einer Programmzeile springen“ auf Seite 37.

`Column mark, visible`: An der angegebenen `Position` wird eine Linie angezeigt, so dass man eine selbst gewählte Zeilenlänge leicht einhalten kann. Auf diese Weise kann man z.B. zu lange Zeilen für den Druck vermeiden.

Editor - Syntax Colors

Der Editor hebt die verschiedenen Syntax-Elemente farbig hervor; siehe auch Kapitel 3.4.1 „Syntax hervorheben“ auf Seite 21. Die vollständige Syntax-Hervorhebung erfordert, dass die Option `Parse Declarations` unter Editor - General aktiv ist.

Sie können die Hervorhebung für jedes Syntaxelement (Definition siehe unten) separat einstellen:

– `Color`: Textfarbe

– `Bold`: Schriftschnitt **`Fett`**

– `Italic`: Schriftschnitt *Kursiv*

Der Beispieltext oberhalb wird mit den aktuellen Einstellungen formatiert.

`Set to Default` löscht alle Änderungen und setzt wieder die Standard-Einstellungen ein.

Sie können die Schriftgröße des Quelltexts unter Editor - General, `Font size` ändern.

Der Editor unterscheidet folgende Syntax-Elemente:

– *ADbasic*-Syntax (`System related`):
  - `ADbasic sections`: Die Kennwörter **`Init:`**, **`LowInit:`**, **`Event:`** und **`Finish:`** für die Programmabschnitte.
  - `Compiler Directives`: Befehle für den Pre-Compiler wie **`#Define`**, die mit **`#`** beginnen.
  - `Reserved Keywords`: Die Basis-Befehle in *ADbasic* wie `Dim`.
  - `Global Variables`: Die globalen Variablen **`Par_1`** … **`Par_80`**, **`FPar_1`** … **`FPar_80`** und **`Data_1`** … **`Data_200`**.
  - `External Keywords`: *ADbasic*-Befehle für den Zugriff auf Ein- und Ausgänge wie **`P2_ADC`**. Diese Befehle werden in der Regel in mitgelieferten Include- oder Library-Dateien deklariert.
  - `Symbols`: Rechenzeichen wie Klammern, `+` oder `=`.

– Benutzerdefiniert (`User related`):
  - `Defined Names`: Symbolische Namen wie `myName`, die mit **`#Define`** deklariert sind.
  - `Local Variables`: Mit `Dim` deklarierte lokale Variablen wie `myVar`.
  - `Sub Names`: Namen (wie `mySub`) von benutzerdefinierten Modulen, die mit `Sub` oder `Lib_Sub` deklariert sind.
  - `Function Names`: Namen (wie `myFunction`) von benutzerdefinierten Modulen, die mit `Function` oder `Lib_Function` deklariert sind.

– Sonstige (`Other`):
  - `Numbers`: Zahlenkonstanten in dezimaler (**`15`**), hexadezimaler (**`0Fh`**) und binärer Schreibweise (**`1111b`**).
  - `Strings`: Zeichenketten in doppelten `"Hochkommas"`.

– `Comments`: Kommentare nach *`Rem`* oder nach Kommentarzeichen `'`.

– `Standard Text`: Alle Elemente, die nicht zu anderen Gruppen gehören, z.B. ungültige Befehle wie `Eixt` (anstelle von `Exit`).

Editor - Print Settings

Die Einstellungen betreffen den Ausdruck eines Quelltexts.

`Header` bezieht sich auf die Kopfzeile des Ausdrucks.

`Print Header`: Bei aktiver Option erscheint auf jeder Seite des Ausdrucks eine Kopfzeile.

`Header text`: Der Text der Kopfzeile.

`Layout` legt fest, welche Elemente der Bildschirmansicht in den Ausdruck übernommen werden.

`Color`: Bei inaktiver Option ist der Ausdruck schwarz/weiß.

`Syntax Highlight`: Bei aktiver Option erscheint die Syntax-Hervorhebung im Ausdruck.

`Line numbers`: Bei aktiver Option werden die Zeilennummern am linken Rand gedruckt.

`Font size`: Legt die Schriftgröße des Ausdrucks fest.

`General`

Nur mit Prozessor T12 / T12.1: Wählen Sie aus, mit wie vielen Mantissen-Stellen Fließkommazahlen (FPar) im Parameterfenster angezeigt werden. Zur Wahl stehen `7 digits` oder `15 digits`.

Bei allen anderen Prozessoren werden Fließkommazahlen (FPar) mit 7 Mantissen-Stellen angezeigt.

`Language`

Wählen Sie aus, in welcher Landessprache die Fehlermeldungen des Compilers und die Online-Hilfe ausgegeben werden. Zur Wahl stehen `Deutsch` oder `English`.

`Directories`

Legen Sie die Verzeichnisse fest, in denen die Entwicklungsoberfläche und der Compiler nach Dateien suchen:

– `BTL Directory`: Hier sucht die Entwicklungsumgebung nach den Betriebssystem-Dateien `*.btl`, die beim Boot-Vorgang in das *ADwin*-System übertragen werden (siehe Kapitel 3.1.3 auf Seite 11).

– `Include Directory`: In diesem Verzeichnis sucht der Compiler nach Dateien `*.Inc`, die Sie mit **`#Include`** (und ohne Pfadangabe) in einen Quelltext einbinden.

– `Lib Directory`: In diesem Verzeichnis sucht der Compiler nach Library-Dateien `*.lib`, die Sie mit `Import` (und ohne Pfadangabe) in einen Quelltext einbinden.

– `Default working directory`: In diesem Verzeichnis sucht die Entwicklungsumgebung, wenn eine Datei oder ein Projekt geöffnet wird.

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

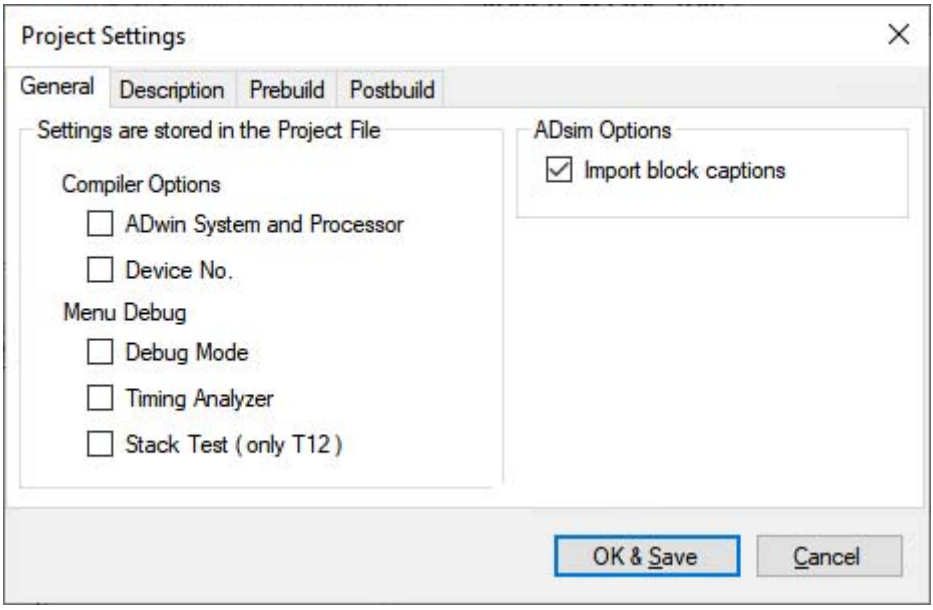
`ADtools`

Die Hilfsprogramme *ADtools* (Beschreibung siehe Kapitel 3.13 auf Seite 98) können über eine eigene Symbolleiste aufgerufen werden. Bei aktiver Option erscheint das jeweilige Hilfsprogramm in der *ADtools*-Leiste.

Dialogfenster „Project Settings“

Sie legen im Dialogfenster Einstellungen zum aktuellen Projekt fest. Die Einstellungen werden mit der Projektdatei gespeichert. Eine gespeicherte Einstellung wird beim Laden der Projektdatei wieder hergestellt; vorherige Einstellungen gehen verloren.

– Reiter General
– Reiter Description
– Reiter Prebuild / Postbuild



Reiter `General`

Die links markierten Einstellungen werden mit der Projektdatei gespeichert.

– Dialogfenster `Compiler Options`
  - `ADwin System and Processor`
  - `Device No.`

– Menü `Debug`
  - `Debug Mode`
  - `Timing Analyzer`
  - `Stack Test (T12 only)`

Die Einstellungen unter `ADsim Options` sind für das Programmpaket *ADsim* und werden immer mit der Projektdatei gespeichert.

– `Import block captions`: Bezeichnungen der *ADwin*-Blöcke als **`#Define`** aus dem Simulink-Modell importieren.

Eine gespeicherte Einstellung wird beim Laden der Projektdatei wieder hergestellt; vorherige Einstellungen gehen damit verloren.

Reiter `Description`

Sie können eine Beschreibung erstellen und mit der Projektdatei speichern. Die Beschreibung hat keine weitere Funktion in *ADbasic*.

Reiter `Prebuild` / `Postbuild`

Sie können Betriebssystem-Aufrufe eingeben, die automatisch vor jedem Übersetzen (`Prebuild`) bzw. danach (`Postbuild`) ausgeführt werden. Die Aufrufe werden mit der Projektdatei gespeichert. Die Syntax eines Aufrufs entspricht der beim Ausführen eines Befehls im Windows-Startmenü (Eingabe im Suchfeld).

Wenn Sie Kommandozeilen verwenden wie in der Eingabeaufforderung, empfehlen wir eine Batch-Datei, in der die Kommandozeilen abgelegt sind; das gilt insbesondere bei komplexeren Abläufen und bei Fallunterscheidungen und Schleifen.

Jeder Compiler-Aufruf (siehe unten, `$BUILDACTION`) löst die Ausführung der Betriebssystem-Aufrufe aus; für die Menüeinträge im Menü `Build` gilt das jedoch nur, wenn die zu kompilierende Quelltext-Datei in die Projektdatei eingebunden ist.

Die Betriebssystem-Aufrufe werden nacheinander dem Betriebssystem übergeben, wobei das Projektverzeichnis als Arbeitsverzeichnis dient. Anschließend wartet *ADbasic* jeweils auf die erfolgreiche Abarbeitung des Aufrufs.

Wenn die Abarbeitung fehlschlägt und nicht mehr reagiert, müssen Sie die Abarbeitung über den Taskmanager beenden.

In einem Betriebssystem-Aufruf können folgende Variablen verwendet werden:

– `$PROJECTNAME`: Projektdatei ohne Pfad und Dateierweiterung.

– `$PROJECTPATH`: Pfad des Projektverzeichnisses (ohne Projektdatei).

– `$PROJECT`: Projektdatei (ohne Pfad).

– `$BUILDACTION`: Zeichenkette, die den auslösenden Compiler-Aufruf angibt, so dass man entsprechend auf den Auslöser reagieren kann.
  - `C`: Menüeintrag `Build` ▶ `Compile`
  - `MB`: Menüeintrag `Build` ▶ `Make Bin File`
  - `ML`: Menüeintrag `Build` ▶ `Make Lib File`
  - `CS`: Schaltfläche `Compile Selected Files` (Projektleiste)
  - `MAB`: Schaltfläche `Make all Bin Files of Project` (Projektleiste)
  - `MSB`: Schaltfläche `Make Bin Files from Selected Files` (Projektleiste)

Pfadangaben sollten zur Sicherheit mit Anführungszeichen umgeben werden (wegen Leerzeichen).

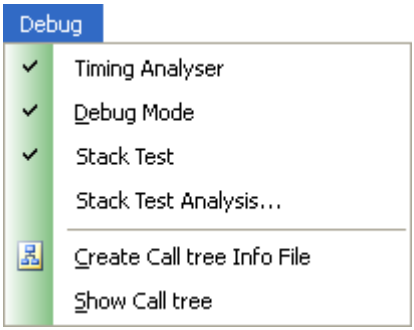
Beispiel: Batch-Datei `example.bat` aufrufen:

```
"$PROJECTPATH\example.bat" "$PROJECTPATH" $BUILDACTION
"$PROJECTNAME" "$PROJECT"
```

Wenn eine Zeile mit # beginnt, wird sie als Kommentar gewertet und nicht weiter beachtet.

3.10.7 Das Menü „Debug“

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



Option Timing Analyzer

Aktivieren Sie die Compiler-Option `Timing Analyzer`, um nach dem Kompilieren eines Quelltexts zusätzliche Informationen zum Zeitverhalten des Prozesses verfügbar zu machen. Die Timing-Informationen werden im Fenster „Timing Analyzer“ (Seite 91) angezeigt.

Die Compiler-Option wird auch in der Statusleiste angezeigt, die Einstellung für einen laufenden Prozess dagegen im Prozessfenster.

Die Option benötigt pro Event-Zyklus und Prozess etwa 60 Taktzyklen (bei den Prozessoren T9, T10 und T11) 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. Die Einstellung dieser Option wird beim Verlassen der Entwicklungsoberfläche nicht gespeichert.

Option Debug mode

Wenn Sie die Compiler-Option `Debug mode` im Menü `Debug` aktivieren und anschließend einen Quelltext kompilieren, werden zusätzliche Sicherheitsabfragen in den Prozess eingebaut (siehe auch Kapitel 5.3.1 auf Seite 152).

Die Compiler-Einstellung wird auch in der Statusleiste angezeigt, die Einstellung für einen laufenden Prozess dagegen im Prozessfenster.

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 die Option nur während der Programmentwicklung nutzen.
Mit **`#Begin_`**/**`#End_Debug_Mode_Disable`** können Sie den Modus zeitweise deaktivieren.

Wenn ein Laufzeitfehler im *ADwin*-System auftritt, wird die Fehlermeldung im Fenster „`Debug Errors`“ angezeigt (unten im Info-Bereich), und zwar mit dem Zeitpunkt des ersten Auftretens. Beim ersten Auftreten eines Fehlers wird das Fenster automatisch in den Vordergrund gesetzt.

Ein Neustarten des Prozesses setzt die Fehlermeldungen nicht zurück, das Rücksetzen geschieht jedoch bei einem Neuladen des Prozesses.

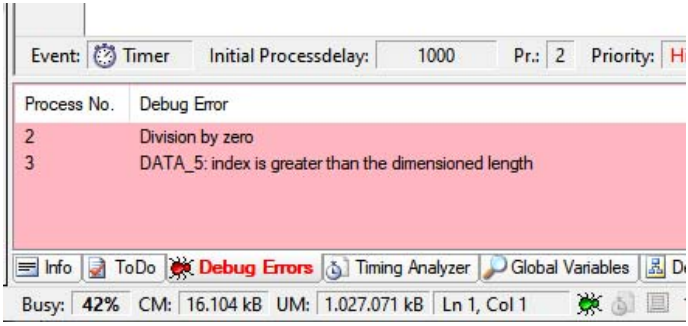


Abb. 6 – Das Fenster `Debug Errors`

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

Die folgende Tabelle zeigt einige allgemeine Fehler und welche Korrektur dazu jeweils erfolgt. Die vollständige Liste der Debug-Fehlermeldungen – auch solche, bei denen keine Korrekturen vorgenommen werden – finden Sie im Anhang auf Seite A-20, Kapitel A.6 „Liste der Debug-Fehlermeldungen“.

Fehlermeldung, Ursache	Korrekturmaßnahme
`Division durch Null`	Das Ergebnis einer Fließkomma-Division wird durch `+3.40282E+38` ersetzt, das einer ganzzahligen Division durch `+2147483647`.
`Wurzel aus negativer Zahl`	Das Ergebnis des Wurzelziehens wird durch den Wert `0` ersetzt.

Fehlermeldung, Ursache	Korrekturmaßnahme

`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.
	Der Zugriff wird nicht ausgeführt.

`FIFO-Index ist kein FIFO` Das Feld mit dieser Nummer ist nicht als FIFO oder gar nicht deklariert.	Befehl (**`FIFO_Clear`**, **`FIFO_Full`**, **`FIFO_Empty`**) wird nicht ausgeführt.
`Adresse des Pro II Moduls ist >15 oder <1`	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. Der Debug-Modus für den Befehl **`MemCpy`** ist nur für den Prozessor T11 verfügbar.

Option Stack Test

Beim Prozessor T12/T12.1 können Sie im Dialogfenster „Process Options“ mit dem Parameter `Stacksize` die Größe des Programm-Stapelspeichers einstellen. Die Option `Stack Test` erlaubt Ihnen zu prüfen, ob die eingestellte Größe für den Prozess ausreicht.

Wenn Sie die Compiler-Option `Stack Test` im Menü `Debug` aktivieren und anschließend einen Quelltext kompilieren, wird zusätzlicher Code in den Prozess eingebaut (siehe auch Kapitel 5.3.1 auf Seite 152). Sie starten die Prüfung des Stapelspeichers im Fenster Stack Test Analysis (Seite 72).

Das Einschalten der Option verlängert die Programm-Ausführungszeit (siehe Stack Test Analysis …) und vergrößert den Speicherbedarf. Sie sollten daher die Option nur während der Programmentwicklung nutzen.

Die Compiler-Einstellung wird auch in der Statusleiste angezeigt, die Einstellung für einen laufenden Prozess auf der *ADwin*-Hardware dagegen im Prozessfenster.

Stack Test Analysis …

Der Menüeintrag `Debug / Stack Test Analysis` (nur mit Prozessor T12/ T12.1) öffnet das Fenster Stack Test Analysis (siehe Seite 87).

Im Fenster können Sie die Prüfung des Stapelspeichers starten und sich das Ergebnis anzeigen lassen. Die Prüfung umfasst alle laufenden Prozesse auf der *ADwin*-Hardware, die mit der Option `Stack Test` kompiliert wurden.

Create Call tree Info File

Der Menüeintrag `Create Call tree Info File` erzeugt oder aktualisiert einen Aufrufgraph und öffnet das Fenster Call Tree (siehe Seite 85).

Der Aufrufgraph (call tree) ist ein einfaches Analyseprogramm zum besseren Verständnis von großen und komplexen *ADbasic*-Programmen; siehe Programmstruktur analysieren, Seite 156 und Speicherintensive Prozeduren optimieren, Seite 157.

Show Call tree

Der Menüeintrag `Debug / Show Call tree` zeigt einen vorher erzeugten Aufrufgraphen im Fenster Call Tree an (siehe Seite 85).

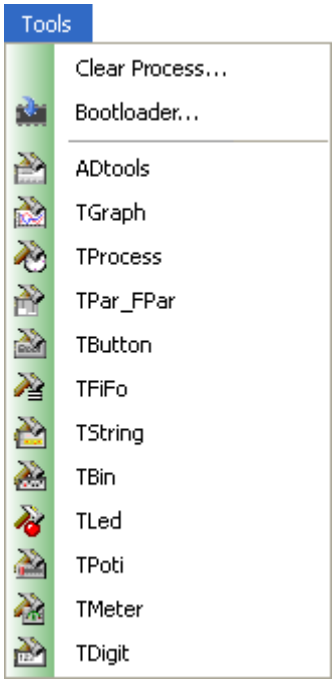
Der Aufrufgraph (call tree) ist ein einfaches Analyseprogramm zum besseren Verständnis von großen und komplexen *ADbasic*-Programmen; siehe Programmstruktur analysieren, Seite 156 und Speicherintensive Prozeduren optimieren, Seite 157.

3.10.8 Das Menü „Tools“

Im Menü `Tools` rufen Sie Hilfsprogramme auf.

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 `Bootloader...` (nur für Prozessor T12.1) öffnet das gleichnamige Fenster, von dem aus die Eigenschaften des Bootloaders festgelegt werden. Der Bootloader kann einen Prozess beim Einschalten der *ADwin*-Hardware automatisch starten. Näheres ist in Kapitel 3.8 „Bootloader nutzen“, Seite 44 beschrieben.



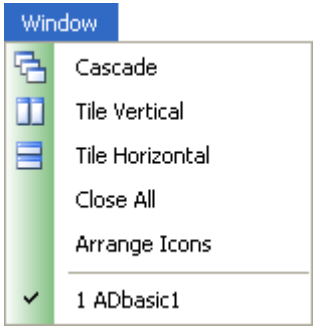
Die Menüeinträge `ADtools` und folgende starten jeweils ein Hilfsprogramm. Die Programme sind in Kapitel 3.13 auf Seite 98 kurz beschrieben.

3.10.9 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.

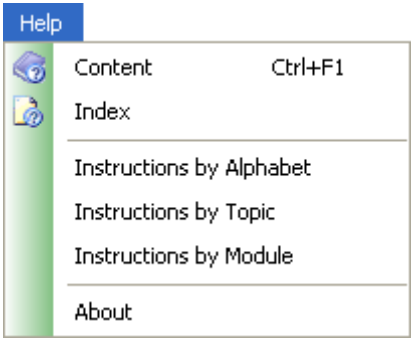
Unten im Menü können Sie über die Dateinamen einen der geöffneten Quelltexte zum aktiven Fenster machen. Der aktive Quelltext ist mit einem Haken gekennzeichnet; im Beispiel rechts ist dies `ADbasic1.bas`.



3.10.10 Das Menü „Help“

Mit dem Menü `Help` rufen Sie die Online-Hilfe der Entwicklungsumgebung auf:

- `Content`: Inhaltsverzeichnis
- `Index`: Indexverzeichnis
- `Instructions by`
  - `Alphabet`: Befehlsliste alphabetisch
  - `Topic`: Befehlsliste nach Themen.
  - `Module`: Befehlsliste nach Modulen (nur *ADwin-Pro*).



Die Befehlslisten beziehen sich auf das *ADwin*-System, das im Dialogfenster „Compiler Options“ (Seite 56) eingestellt ist.

Alternativ können Sie auch die Schaltfläche verwenden. Mit der Taste [F1] erhalten Sie Hilfe zu einem geöffneten Fenster oder zu einem markierten Befehlswort.

`About` öffnet ein Fenster, das die Version der Entwicklungsumgebung und die freigegebenen Lizenzschlüssel anzeigt. Über die Schaltfläche `Change License` können Sie einen `License key` ändern (siehe auch Kapitel 3.1.1 auf Seite 9).

Ohne Eingabe eines gültigen `License key` befindet sich *ADbasic* im Demo-Modus. In diesem Modus ist das Arbeiten mit der Entwicklungsumgebung nur zu Prüf-, Demonstrations- und Bewertungszwecken erlaubt.

3.11 Fenster

3.11.1 Toolbox

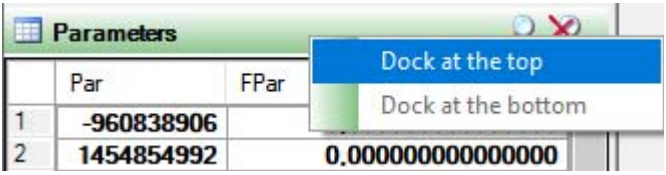
Die Toolbox ist der Bereich in der Oberfläche links, in dem das Projektfenster, das Parameterfenster und das Prozessfenster angezeigt werden.

Die Toolbox teilt sich in einen oberen und einen unteren Anzeigebereich. Die zugehörigen Fenster können den beiden Anzeigebereichen frei zugeordnet werden. Ein verdecktes Fenster wird durch einen Klick auf den zugehörigen Reiter in den Vordergrund geholt.

Um ein Fenster einem Bereich zuzuordnen, gehen Sie vor wie folgt:

– Öffnen Sie mit einem Rechtsklick auf die Kopfleiste des Fensters das Kontextmenü.

– Wählen Sie den Anzeigebereich oben (top) oder unten (bottom).



– Es ist möglich, alle Fenster dem gleichen Anzeigebereich zuzuordnen. Dadurch ist nur eines der Fenster im Vordergrund.

Die Standardeinstellung kann durch den Menüeintrag `View` ▸ `Restore default layout` wieder eingestellt werden.

Die Toolbox kann über die Symbole in der Kopfzeile als frei verschiebbares Fenster angezeigt oder ganz ausgeblendet werden.

3.11.2 Projektfenster

Das Projektfenster zeigt an, ob ein Projekt geöffnet und welche Quelltext-, Library-, Include- oder sonstigen Dateien eingebunden sind. Bei Quelltextdateien werden in der Spalte `Info` auch das Event-Signal (Timer, extern), die Prozessnummer und die Prozesspriorität (rot=hoch, blau=niedrig) angezeigt.

Das Projektfenster ist Teil der Toolbox (siehe Seite 76).

Sie können im Projektfenster folgende Aktionen ausführen (siehe auch Projekte verwalten, Seite 47):

– Alle offenen Dateien in das Projekt einbinden:
  Wählen Sie `Add Open Files to Project` aus dem Kontextmenü im Projektfenster.

– Eine andere Datei zum Projekt hinzufügen:
  Wählen Sie `Add Other File to Project` aus dem Kontextmenü im Quelltextfenster.

– Eine Simulink-Modelldatei zum Projekt hinzufügen:
  Wählen Sie die Schaltfläche `Begin/Add ADsim Project` aus dem der Projektleiste oder dem Kontextmenü im Projektfenster.

– Eine oder mehrere Dateien aus dem Projekt löschen:
  Markieren Sie die Dateien per Mausklick im Projektfenster und
  - drücken die Taste [ENTF] oder
  - wählen `Remove from Project` aus dem Kontextmenü.

– Eine Datei öffnen und zum aktiven Quelltext machen:
  - Klicken Sie doppelt (linke Maustaste) auf die Datei oder
  - Markieren Sie eine Datei im Projektfenster und wählen `Open` aus dem Kontextmenü.

– Alle Dateien des Projekts speichern:
  Wählen Sie `Save all Files of Project` aus dem Kontextmenü im Projektfenster oder die Schaltfläche in der Projektleiste.

– Alle im Projektfenster markierten Dateien kompilieren:
  Wählen Sie die Schaltfläche `Compile selected Files of Project` in der Projektleiste.

– Die Binärdateien zu den im Projektfenster markierten Dateien erzeugen:
  Wählen Sie die Schaltfläche `Make Bin Files from selected Files` in der Projektleiste.

– Die Binärdateien zu allen Dateien im Projektfenster erzeugen:
  Wählen Sie die Schaltfläche `Make All Bin Files of Project` in der Projektleiste.

– Die Binärdateien aller im Projektfenster markierten Dateien ins ADwin-System laden:
  Wählen Sie die Schaltfläche `Load selected Bin Files into ADwin system` in der Projektleiste.

– Die Binärdateien aller im Projektfenster markierten Dateien in den Bootloader übertragen (nur mit Prozessor T12.1):
  Wählen Sie die Schaltfläche `Program selected Files into the Bootloader` in der Projektleiste (siehe Bootloader programmieren auf Seite 44).

– Zu den im Projektfenster markierten Simulink-Modelldateien einen Quelltext erzeugen (nur mit *ADsim*-Lizenz):
  Wählen Sie die Schaltfläche `Import ADsim Library as Template` in der Projektleiste (siehe ADbasic und ADsim auf Seite 100).

– Windows Explorer mit dem Pfad der markierten Datei öffnen:
  Wählen Sie `Open Path in Window Explorer` aus dem Kontextmenü im Projektfenster.

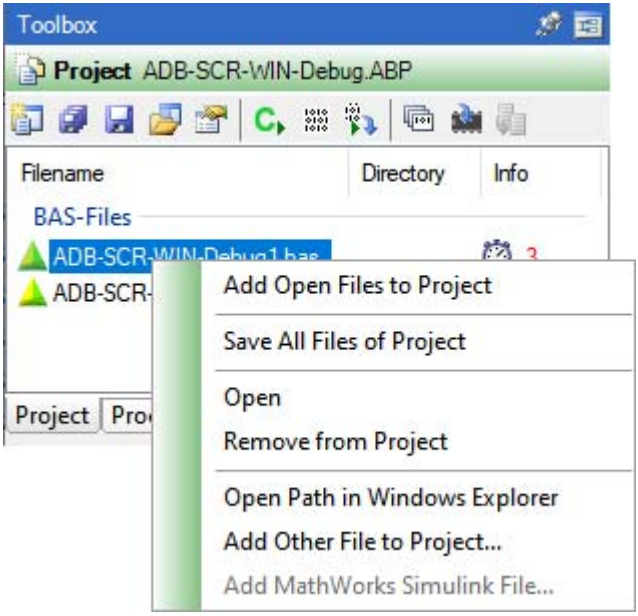


Abb. 7 – Das Projekt-Fenster mit Kontextmenü

Abb. 8 – Die Projektleiste

3.11.3 Parameterfenster

Das Parameterfenster `Parameters` zeigt die globalen Parameter **`Par_1`** … **`Par_80`** und **`FPar_1`** … **`FPar_80`** an. Mit dem Schiebebalken am rechten Rand können Sie den angezeigten Parameter-Bereich auswählen.

Das Parameterfenster ist Teil der Toolbox (siehe Seite 76).

Wenn die Kommunikation zwischen PC und *ADwin*-System aktiv ist (Schaltfläche `Enable Cyclic Update` 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 an, dass keine Kommunikation stattfindet.

Parameters

	Par	FPar
1	206	0,000000000000000
2	-1557758399	3,45212989511920E+028
3	5	2,66234000000000
4	0	0,000000000000000
5	0	0,000000000000000
6	0	0,000000000000000
7	0	0,000000000000000
8	707279	0,000000000000000
9	0	0,000000000000000
10	2147483647	0,000000000000000
11	0	0,000000000000000
12	0	0,000000000000000
13	0	0,000000000000000
14	0	0,000000000000000
15	0	0,000000000000000
16	0	0,000000000000000
17	0	0,000000000000000
18	0	0,000000000000000
19	0	0,000000000000000

Abb. 9 – Das Parameter-Fenster

Sie können die Werteanzeige eines ganzzahligen Parameters zwischen dezimal und hexadezimal umstellen (siehe **`Par_5`** in Abb. 9), indem Sie mit der Maus auf die Nummer der betreffenden Variable klicken (links vom Tabellenfeld). Mit einem Klick auf den Spaltentitel `Par` wird die Anzeige aller Parameter **`Par_1`**…**`Par_80`** auf einmal geändert.

Die Fließkommawerte **`FPar_1`** … **`FPar_80`** werden mit 7 Stellen angezeigt. Beim Prozessor T12/T12.1 können Sie stattdessen auch 15 Stellen anzeigen, siehe Dialogfenster „Settings“, Seite `General`.

Mit der Schaltfläche `Scan Global Variables` können Sie Verwendete globale Variablen und Felder anzeigen (siehe Seite 42).

3.11.4 Prozessfenster

Das Prozessfenster 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 ist das Fenster grau hinterlegt.

Das Prozessfenster ist Teil der Toolbox (siehe Seite 76). Sie öffnen das Prozessfenster mit einem Klick auf den Reiter `Processes`.

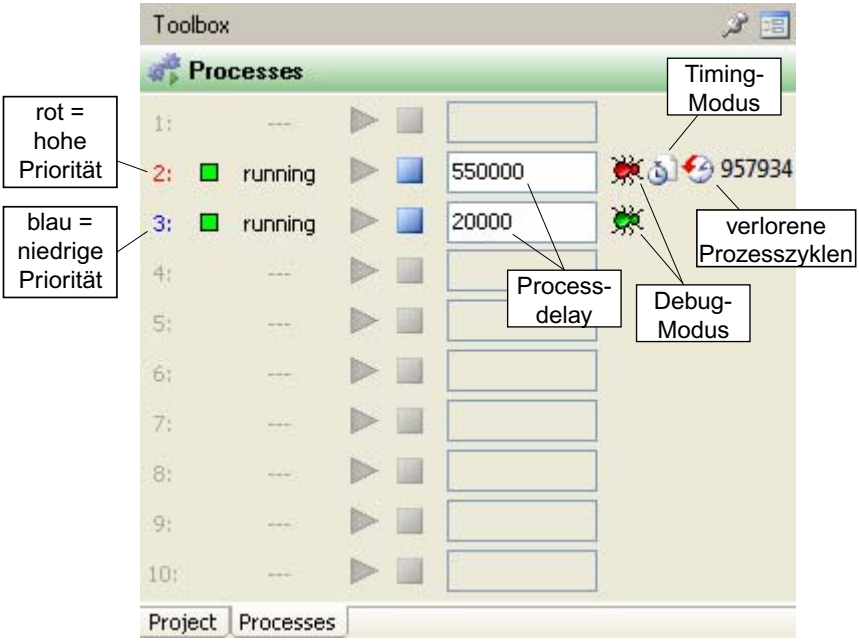


Abb. 10 – Das Prozess-Fenster

Für jeden Prozess werden folgende Informationen angezeigt:

– Prozess-Priorität

Die Farbe der Prozessnummer gibt die Priorität an.
- rot = hohe Priorität
- blau = niedrige Priorität.

– Prozess-Status
  - `running`: Prozess läuft.
  - `stopped`: Prozess wurde angehalten.
  - `---`: Prozess ist nicht vorhanden.

Sie können einen Prozess mit der Schaltfläche `Stop` beenden und mit `Start` wieder starten. Die Schaltflächen in der Werkzeugleiste haben die gleiche Funktion; sie beziehen sich auf den zum aktiven Quelltext gehörigen Prozess.

– Processdelay (Prozess-Zykluszeit)

Die Bedeutung und die Zeiteinheiten des Processdelay sind in Kapitel 6.2.1, Seite 164 erläutert.

Das zum aktiven Quelltext gehörige Processdelay ist auch oben in der Werkzeugleiste zu sehen.

Sie können die Zykluszeit ändern, indem Sie einen neuen Wert in das Eingabefeld schreiben. Sobald Sie das Feld verlassen, wird der Wert auf das *ADwin*-System übertragen. Achten Sie darauf, das System nicht durch zu kleine Werte überlasten.

– Prozess arbeitet im Debug-Modus

Das Icon wird angezeigt, wenn der Prozess im Debug-Modus arbeitet. Näheres zum Debug-Modus ist unter Option Debug mode beschrieben. Das Icon wird rot, wenn im Prozess ein Debug-Fehler erkannt wurde.

Die Compiler-Einstellung zum Debug-Modus wird in der Statusleiste angezeigt.

– Prozess arbeitet im Timing-Modus

Das Icon wird angezeigt, wenn der Prozess im Timing-Modus arbeitet. Näheres zum Timing-Modus ist unter der Option Timing Analyzer beschrieben (Seite 69). Die Timing-Informationen werden im Fenster „Timing Analyzer“ angezeigt.

Die Compiler-Einstellung zum Timing-Modus wird in der Statusleiste angezeigt.

– Nur T12/T12.1: Prozess arbeitet mit Stack-Test

  Das Icon wird angezeigt, wenn der Prozess im Modus `Stack Test` arbeitet. Näheres ist unter Option Stack Test und Fenster Stack Test Analysis beschrieben.

  Die Compiler-Einstellung zum Stack-Modus wird in der Statusleiste angezeigt.

– Nur T12/T12.1: Prozess verliert Prozesszyklen

  Die Zahl gibt an, wie viele Prozesszyklen der Prozess seit dem Start verloren hat. Verlorene Prozesszyklen (`lost events`) zeigen, dass im Zeitverhalten des Prozesses Fehler bestehen, die Sie beheben müssen. Es tritt zumindest zeitweise eine Überlastung des Prozessors auf (siehe Auslastung des ADwin-Systems).

  Um eine Überlastung zu vermeiden, halten Sie die Anzeige `Busy` in der Statusleiste immer deutlich unter 100%.

  Die Erkennung von `lost events` ist immer aktiviert, auch wenn der Prozess nicht im Timing-Modus arbeitet.

3.11.5 Quelltextfenster

Das Quelltextfenster ist das Hauptfenster der Entwicklungsumgebung, in dem Sie die Quelltexte von *ADbasic*-Programmen editieren.

Die verschiedenen Funktionen im Quelltextfenster sind in Kapitel 3 beschrieben. Wichtige Abschnitte sind folgende:

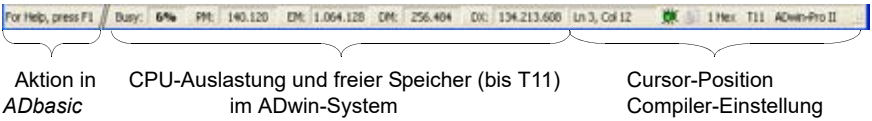
– Grundelemente der Entwicklungsumgebung, Seite 12.

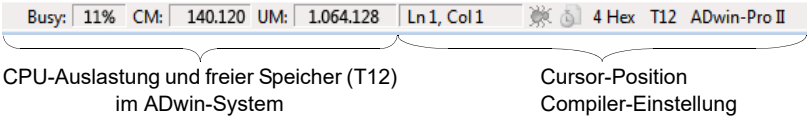
– Quelltexte erstellen, Seite 16.

– Kontextmenü im Quelltextfenster, Seite 18.

3.11.6 Statusleiste

Die Statusleiste befindet sich am unteren Rand der Bedienoberfläche.





– Links: Informationen zur zuletzt ausgeführten Aktion.

– Mitte: Die aktuelle Prozessorauslastung und der freie Speicher des *ADwin*-Systems (bei aktiver Verbindung zwischen PC und *ADwin*-System).

– Rechts: Die aktuelle Cursor-Position im Quelltextfenster (Zeile und Spalte); daneben die Einstellungen für den Compiler: Debug-Modus, Timing-Modus, Stack-Test (nur T12/T12.1), Device no., Prozessor, *ADwin*-Hardware.

  Ein Doppelklick auf die Compiler-Einstellungen öffnet das Dialogfenster „Compiler Options“ (Seite 56).

Die Angaben zu Prozessorauslastung und Speicherplatz bedeuten:

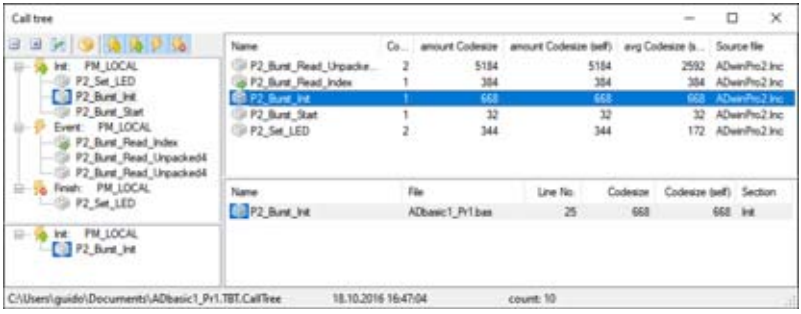
`Busy`:	Zeitliche Auslastung des Prozessors in Prozent, berechnet als: Rechenzeit / (Rechenzeit + Leerlaufzeit). Siehe auch Kapitel 6.2.4.
`PM`:	Freier Programmspeicher in Bytes.
`EM`:	nur T11: Freier Zusatzspeicher in Bytes.
`DM`:	bis T11: Freier interner Datenspeicher in Bytes.
`DX`:	bis T11: Freier externer Datenspeicher in Bytes.
`CM`:	nur T12/T12.1: Verfügbarer Hauptspeicher, der Daten an den Cache liefern kann, in Bytes.
`UM`:	nur T12/T12.1: Verfügbarer Hauptspeicher, der keine Daten an den Cache liefern kann, in Bytes.

3.11.7 Fenster Call Tree

Das Fenster `Call Tree` zeigt alle Prozeduren (`Sub`, `Function`, `Lib_Sub`, `Lib_Function`) an, die in einem *ADbasic*-Programm aufgerufen werden, sowie deren Speichergröße. Der Aufrufgraph ist statisch und wird mit dem Menüeintrag `Debug / Create Call tree Info File` erzeugt oder aktualisiert.

Der Aufrufgraph ist das Ergebnis einer einfachen Programmanalyse mit dem Zweck, komplexere Programme leichter zu verstehen (siehe Programmstruktur analysieren, Seite 156) und speicherintensive Prozeduren zu finden (siehe Speicherintensive Prozeduren optimieren, Seite 157).

Wenn ein im Quelltext vorhandener Aufruf einer Prozedur nicht angezeigt wird, liegt das meist daran, dass der Aufruf in einem bedingten Programmzweig stattfindet, der in keinem Fall durchlaufen wird. Dies geschieht, wenn eine Bedingung (`If` … `EndIf`, `#If` … `#EndIf`, `SelectCase` … `EndSelect`) eine Konstante verwendet und in jedem Fall „Unwahr“ ist.



Das Fenster `Call Tree` hat 4 Teilfenster:

- Aufrufgraph (links oben): Ein Baum, in dem jeder Knoten für den Aufruf einer Prozedur im Quelltext steht. Unterknoten sind (verschachtelte) Aufrufe innerhalb einer Prozedur.
- Wurzelpfad des ausgewählten Knotens (links unten).
- Liste aller aufgerufenen Prozeduren (rechts oben). Die Liste enthält Spalten für
  - `Name`: Name der Prozedur.
  - `Count`: Anzahl der Aufrufe einer Prozedur.
  - `Total code size`: Code-Größe, die sich aus der Summe der Aufrufe einer Prozedur ergibt, inklusive der Aufrufe innerhalb der Prozedur.

- `Code size`: Wie `Total code size`, jedoch ohne die Aufrufe innerhalb der Prozedur.
- `avg code size`: Durchschnittliche Code-Größe der Prozedur, d.h. Code-Größe (`Code size`) geteilt durch Anzahl der Aufrufe (`Count`).

– Aufrufliste der ausgewählten Prozedur (rechts unten): Eine Liste von Aufrufen für eine ausgewählte Prozedur mit
  - `Name`: Name der Prozedur.
  - `File`: Name der Datei, in der die Prozedur aufgerufen wird.
  - `Line no.`: Zeilennummer in der Datei, in der die Prozedur aufgerufen wird.
  - `Total code size`: Code-Größe, die sich aus der Summe der Aufrufe einer Prozedur ergibt, inklusive der Aufrufe innerhalb der Prozedur.
  - `Code size`: Wie `Total code size`, jedoch ohne die Aufrufe innerhalb der Prozedur.
  - `Section`: Programmabschnitt, in dem die Prozedur aufgerufen wird.

  Mit einem doppelten Mausklick auf eine Zeile der Aufrufliste springt der Cursor in die Quelltext-Zeile, in der die Prozedur aufgerufen wird.

In der Leiste oberhalb des Teilfensters mit dem Aufrufgraphen können Sie festlegen, welche Baumknoten in den Teilfenstern angezeigt werden:

– Ein-/ausfalten (nur für das Teilfenster „Aufrufgraph“):
  - Die Minus-Schaltfläche faltet alle Knoten ein, so dass nur noch die Programmabschnitte sichtbar sind.
  - Die Plus-Schaltfläche faltet alle Knoten aus.
  - Mit aktiver Option „Automatisch falten“ (`Autocollapse`) wird nur der Baumpfad ausgefaltet zum ausgewählten Knoten ausgefaltet, während alle anderen Knoten eingefaltet bleiben.

– Bibliotheks-Prozeduren : Die Bibliotheks-Option gibt die Anzeige von Bibliotheks-Funktionen und -Unterprogramme frei.

– Programmabschnitte: Die Optionen geben die Anzeige der Knoten eines Programmabschnitts frei oder sperren sie.
  Die Programmabschnitte heißen **`LowInit`**, **`Init`**, **`Event`** und **`Finish`**.

3.11.8 Fenster Stack Test Analysis

Im Fenster `Stack Test Analysis` (nur für Prozessor T12/T12.1) können Sie prüfen, ob die jeweils eingestellte Größe des Stapelspeichers (`Stack size`, siehe Dialogfenster „Process Options“) für laufende Prozesse auf der *ADwin*-Hardware ausreicht. Eine ungenügende Stack-Größe kann zum Überschreiben anderer Speicherbereiche und damit zu Datenverlust oder Programmabbruch führen.

Die Größe des Stapelspeichers ist insbesondere wichtig, wenn Sie Libraries (Lib_Sub … Lib_EndSub, Lib_Function … Lib_EndFunction) verwenden. Die Anzahl und Größe der lokalen Variablen und Arrays in der Library bestimmt den Platzbedarf für den Programm-Stapelspeicher eines Prozesses. Verschachtelte Library-Aufrufe können den Platzbedarf ebenfalls erheblich erhöhen.

Sie starten den Stack-Test mit der Schaltfläche `Start`. Die Prüfung ermittelt für jeden laufenden Prozess – der mit der Option Stack Test (Seite 71) kompiliert wurde – wie viele Bytes im Stapelspeicher seit dem Start des Prozesses genutzt wurden. Zusätzlich wird der genutzte Stack-Anteil in Prozent angegeben. Die Prüfung kann einige Sekunden dauern.
Um das Fenster wieder zu schließen, klicken Sie auf die Schaltfläche `OK`.

Beachten Sie, dass der Test die Auswirkungen des Programmabschnitts **`Finish`** auf den Stack nicht einbezieht.

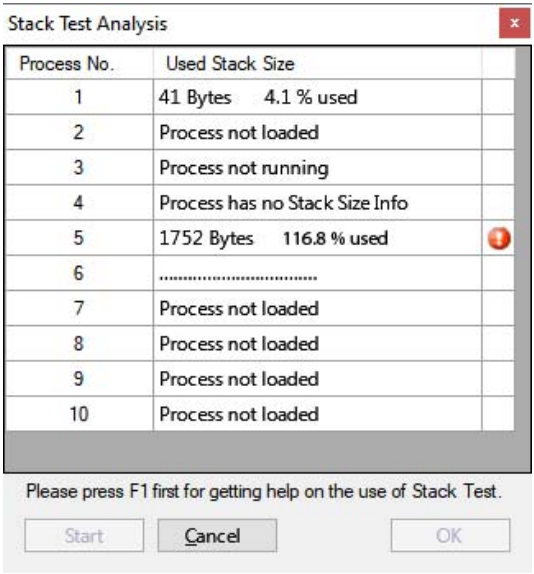


Abb. 11 – Das Fenster `Stack Test Analysis`

In der rechten Spalte wird bei einem genutzten Stack-Anteil >90% ein Warnsymbol, ab 100% ein Fehlersymbol gezeigt (siehe Abbildung 11, Prozess 5). In diesem Fall müssen Sie einen größeren Wert für den Stapelspeicher des Prozesses einstellen. Dazu öffnen Sie den Quelltext des Prozesses, stellen die `Stack size` im Dialogfenster „Process Options“ neu ein und kompilieren den Prozess erneut. Wir empfehlen, den Test anschließend noch einmal durchzuführen.
Beachten Sie, dass der Stapelspeicher im Speicher CM abgelegt wird und daher nicht allzu groß werden soll, siehe Speicherbereiche (T12/T12.1).

Die Anzeige . . . . (siehe Abbildung 11, Prozess 6) erscheint, wenn der Stack-Test auf seinen Start wartet. Der Stack-Test beginnt, sobald das nächste Mal das Ende des Programmabschnitts Event erreicht ist. Eine Verzögerung entsteht z. B. bei einem extern gesteuerten Prozess, der noch auf ein externes Event-Signal wartet, oder wenn die Bearbeitung eines niederprioren Prozesses sehr lange dauert.
Alternativ können Sie den Stack-Test mit `Cancel` abbrechen.

3.12 Info-Bereich

Der Info-Bereich ist der Bereich in der Oberfläche ganz unten, in dem folgende Fenster angezeigt werden:

– Infofenster (siehe unten)

– ToDo-Liste (siehe ToDo-Liste)

– Das Fenster Debug Errors (siehe Seite 70)

– Fenster „Timing Analyzer“ (siehe Seite 91)

– Fenster „Global Variables“ (siehe Seite 95)

– Fenster „Declarations“ (siehe Seite 96)

Weitere Fenster siehe Kapitel 3.11, Seite 76.

3.12.1 Infofenster

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

– Statusmeldung nach dem Kompilieren

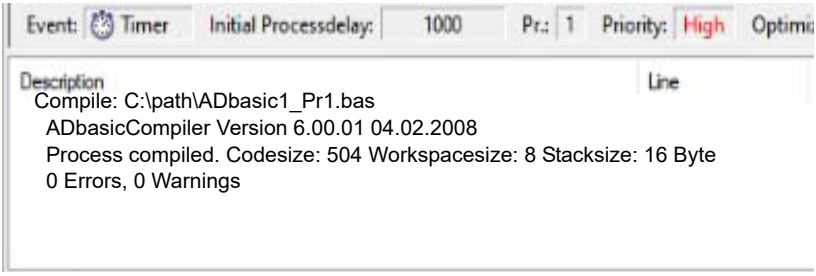
– Fehlermeldungen

– Warnungen

Warnungen und Fehlermeldungen werden mit dem Ort des Auftretens (Zeile, Dateiname und Pfad) angegeben. Ein Doppelklick auf die Meldung färbt die betreffende Quelltextzeile rot und setzt den Cursor in die Zeile.

Eine vollständige Beschreibung des Fehlers oder der Warnung können Sie in der Online-Hilfe lesen: Klicken Sie die Fehlermeldung an (kein Doppelklick) und drücken die Taste [F1].

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



Nur für Prozessoren bis T11: Die Werte der Statusmeldung geben Hinweise, wieviel Speicherplatz der Prozess benötigen wird.

– `Codesize`: Größe der erzeugten Binärdatei in Bytes; die Datei wird als Prozess im Programmspeicher (PM) abgelegt.

  Beim Prozessor T11 wird die Größe der Binärdatei für PM, EM und DX getrennt angegeben, siehe **`Init:`**, **`LowInit:`**, **`Event:`**, **`Finish:`**

– `Workspacesize`: Benötigter Speicherplatz in Bytes im lokalen Datenspeicher (DM).

  Der Platz wird benötigt für:
  - lokale Variablen und Felder
  - interne Zwecke (8 Byte)
  - globale Felder **`Data_x`**
    Zur Verwaltung werden je Feld 40 Byte benötigt, auch wenn die Daten im externen Datenspeicher liegen.
    Wenn das Feld in `DM_Local` liegt, belegt jedes Feldelement 4 Byte (Datentyp Long) oder 8 Byte (Datentyp Float).

– `Stacksize`: Größe des internen Stapels (DM), der für Libraries verwendet wird.

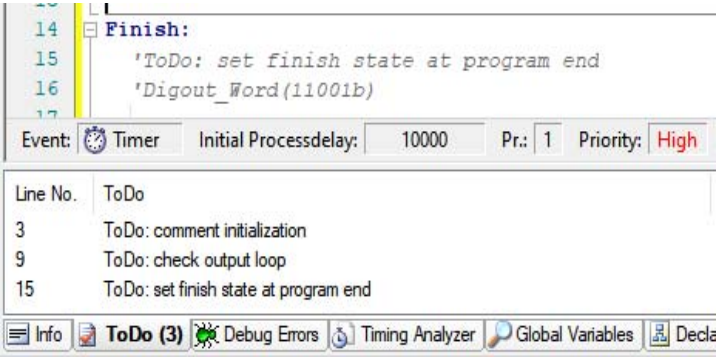
Es wird nicht angezeigt, wieviel Speicherplatz im externen Datenspeicher (DX) benötigt wird.

3.12.2 ToDo-Liste

Das Fenster `ToDo` dient als einfache ToDo-Liste: es werden Zeilen der aktuellen Quelltextdatei angezeigt, in denen der Text `ToDo:` als Kommentar enthalten ist. Durch entsprechende Kommentarzeilen können Sie noch nicht erledigte Arbeiten am Quelltext markieren und übersichtlich anzeigen. Mit einem Doppelklick auf einen Eintrag der ToDo-Liste springt der Cursor im Quelltext auf die entsprechende Zeile.

Wenn eine Aufgabe erledigt ist, löschen Sie einfach die entsprechende Kommentarzeile.

Das Fenster `ToDo` ist Teil des Info-Bereichs (siehe Seite 89).



Ein Doppelklick auf einen ToDo-Eintrag setzt den Cursor in die betreffende Zeile im Quelltext.

3.12.3 Fenster „Timing Analyzer“

Bei aktivem Timing-Modus zeigt das Fenster `Timing Analyzer` sieben Kennwerte, die das Zeitverhalten der Prozesse 1…10 seit dem Zeitpunkt des letzten Starts beschreiben. Nähere Hinweise über die Auswertung dieser Informationen gibt Ihnen Kapitel 5.3.2 „Zeitverhalten prüfen (Timing-Modus)“.

Sie aktivieren den Timing-Modus vor dem Kompilieren für jeden Prozess einzeln im Menü `Debug` mit der Option Timing Analyzer (Seite 69).

Das Fenster `Timing Analyzer` ist Teil des Info-Bereichs (siehe Seite 89).

Alle Zeitangaben sind in Taktzyklen des Prozessors angegeben (Einheiten siehe Abb. 20 auf Seite 164).

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

Bei Prozessen mit niedriger Priorität kann das Zeitverhalten durch andere Prozesse und Kommunikation stark verändert werden. Die Kennwerte müssen deswegen immer im Kontext mit allen anderen laufenden Prozessen interpretiert werden.

Event: Timer | Initial Processdelay: 10000 | Pr.: 1 | Priority: High | Optimize: Level 1 | Stacksize

	Process 1	Process 2	Process 3	Process 4	Process 5	Process 6	Process 7	Process 8
min. Length		11	27					
max. Length		11	33					
Ø Length		11.0	27.1					
max. Latency		0	57					
max(Latency + Length)		11	84					
count (Length > Delay)		0	0					
Critical timings		0	0					

Info | ToDo | Debug Errors | Timing Analyzer | Global Variables | Declarations

Abb. 12 – Das Fenster `Timing Analyzer`

`Length` bezeichnet die Dauer eines Prozesszyklus (Abschnitt **`Event:`**); diese Bearbeitungsdauer kann man auch ermitteln wie in Kapitel 5.1 „Bearbeitungszeit messen“ beschrieben. `Latency` ist die Zeitdauer zwischen dem Auftreten eines Event-Signals (extern oder vom internen Timer) und dem tatsächlichen Start des Prozesszyklus, in Abb. 13 am zeitgesteuerten Prozess 1 dargestellt.

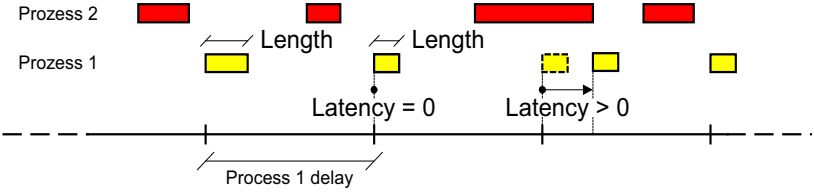


Abb. 13 – Latenz bei Prozess 1 durch einen zweiten Prozess

Im Fenster werden folgende Kennwerte angezeigt:

– `min. Length`: Minimale gemessene Dauer eines Prozesszyklus

– `max. Length`: Maximale gemessene Dauer eines Prozesszyklus

– ∅ `Length`: Durchschnittliche Dauer des Prozesszyklus

  Der Durchschnitt wird berechnet als gleitender Mittelwert aus vorhergehenden `length`-Werten.

$$\text{ØLength} = 0.999 \cdot \text{ØLength} + 0.001 \cdot \text{Length}$$

Nach dem Start eines Prozesses dauert es 7000 Zyklen, bis der Mittelwert einen gültigen Wert erreicht.

Gemeinsam mit `min. Length` und `max. Length` zeigt dieser der Mittelwert, 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 hochprioren 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, um so 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 6.2.5 „Verschiedene Betriebszustände im Betriebssystem“, Seite 168).

  Event-Signale können verloren gehen bei:
  - einem einzigen zeitgesteuerten, hochprioren Prozess (auch in Kombination mit dem extern gesteuerten Prozess)
  - dem extern gesteuerten Prozess (auch in Kombination mit einem oder mehreren zeitgesteuerten Prozessen)

  Bei Prozessoren bis einschließlich T11 gilt: 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:
    `max. latency+length` > 2 × Processdelay
  - bei extern gesteuerten Prozessen:
    Wenn die Bearbeitung des Abschnitts **`Event:`** gerade beendet ist, steht schon ein neues externes Event-Signal an (das während der Bearbeitung aufgetreten ist). Falls während dieser Bearbeitungszeit noch weitere Event-Signale aufgetreten sind, sind diese weiteren Events verloren.

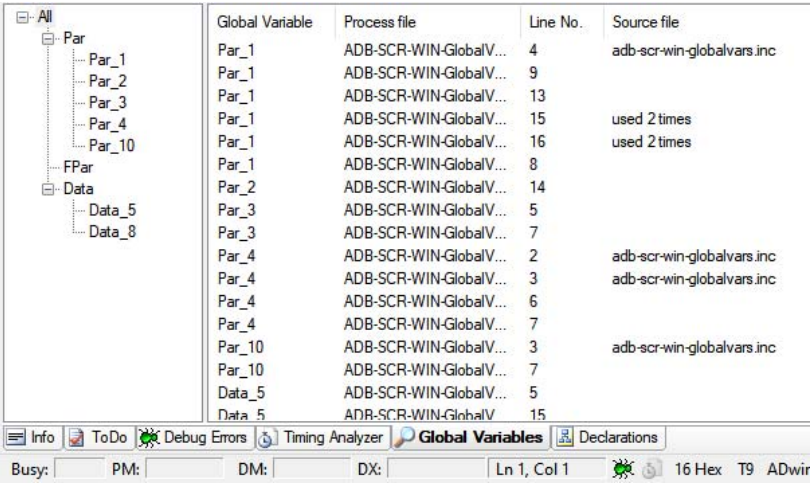
  Manchmal ist es möglich, dass trotz zutreffender Bedingung *kein* Event verloren geht. Daher liegen Sie auf der sicheren Seite, wenn Sie die Anzahl zutreffender Bedingungen so weit wie möglich reduzieren.

3.12.4 Fenster „Global Variables“

Das Fenster `Global Variables` zeigt an, welche globalen Variablen (**`Par_1`** … **`Par_80`**, **`FPar_1`** … **`FPar_80`**) und Felder (**`Data_1`** … **`Data_200`**) in einem Projekt oder in einem Quelltext verwendet werden.

Um die Anzeige zu starten oder zu aktualisieren, klicken Sie auf die Schaltfläche `Scan Global Variables` im Parameterfenster (siehe Verwendete globale Variablen und Felder anzeigen, Seite 42).

Das Fenster ist Teil des Info-Bereichs (siehe Seite 89).



Das markierte Element in der Baumanzeige links legt fest, welche Variablen im Fenster rechts angezeigt werden So zeigt die Liste nach einem Klick auf `Par` nur noch die Variablen **`Par_1`**…**`Par_80`** an (soweit verwendet).

Sie können die Zeilen im rechten Fenster mit einem Klick auf einen Spaltentitel alphabetisch sortieren.

Die Liste zeigt an:

– `Global Variable`: Name der verwendeten globalen Variablen oder des globalen Felds.

– `Process file`: Name der Hauptdatei des durchsuchten Prozesses.

– `Line no.`: Zeilennummer, wo die Variable genutzt oder aufgerufen wird.

– `Source file`: Dateiname, wo die Variable genutzt oder aufgerufen wird. Ein leerer Eintrag heißt, dass die Zeile sich auf die Hauptdatei des Prozesses bezieht (siehe Spalte `Process file`).

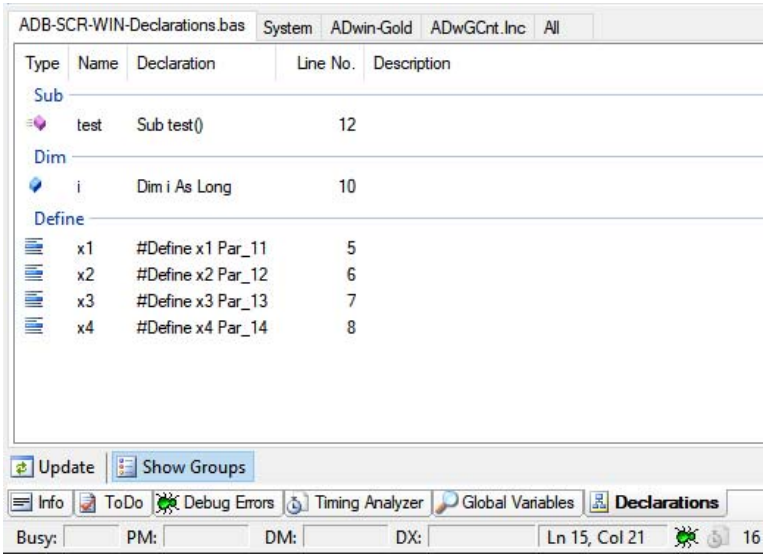
Wenn Sie den Quelltext ändern, wird das Fenster nicht automatisch aktualisiert. Verwenden Sie dazu Schaltfläche `Scan Global Variables` im Parameterfenster.

3.12.5 Fenster „Declarations“

Das Fenster `Declarations` zeigt alle zu einer Quelltextdatei und zu den verbundenen Include- und Library-Dateien gehörenden Deklarationen an. Zum Aktualisieren der Anzeige drücken Sie die Schaltfläche `Update`.

Deklarationen aus anderen als der aktiven Quelltextdatei – auch innerhalb eines Projekts – werden nicht berücksichtigt.

Das Fenster `Declarations` ist Teil des Info-Bereichs (siehe Seite 89).



Die Deklarationen werden unter verschiedenen Reitern, die den Ort der Deklaration repräsentieren, angezeigt:

– `[file].bas`: In der Quelltextdatei erstellte Deklarationen: lokale Variablen, Felder, Befehle (`Sub`, `Function`) und symbolische Namen (`#Define`).

– `System`: Systemvariablen und Befehle, die in *ADbasic* implementiert sind und die zu den aktuellen Compiler-Einstellungen passen.

  Die globalen Variablen **`Par`** und **`FPAR`** werden nicht angezeigt. Beachten Sie hierzu auch das Fenster „Global Variables“ (Seite 95) und die Funktion „Verwendete globale Variablen und Felder anzeigen“ (Seite 42).

– `ADwin-Gold`, `ADwin-light-16`: Befehle für Hardware-Zugriffe, die in *ADbasic* implementiert sind und die zu den aktuellen Compiler-Einstellungen passen.

– `[file].inc`: Variablen und Befehle, die in dieser Include-Datei deklariert sind. Ein solcher Reiter taucht nur auf, wenn im Quelltext eine Include-Datei mit **`#Include`** eingebunden ist.

– `[file].lib`: Variablen und Befehle, die in dieser Bibliotheksdatei deklariert sind. Ein solcher Reiter taucht nur auf, wenn im Quelltext eine Bibliotheksdatei mit `Import` eingebunden ist.

– `All`: Alle gültigen Deklarationen aus den oben aufgeführten Quellen.

Sie können die Deklarationen mit einem Klick auf einen Spaltentitel sortieren. Wenn Sie die Option `Show Groups` aktivieren, werden die Deklarationen nach Gruppen sortiert.

Wenn Sie den Quelltext ändern, wird das Fenster nicht automatisch aktualisiert. Verwenden Sie dazu Schaltfläche `Update`.

Die Deklarationen können nur angezeigt werden, wenn die Option `Parse Declarations` unter Editor - General (siehe Seite 62) aktiv ist.

3.13 ADtools

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

Starten Sie die *ADtools* einfach aus der senkrechten Leiste am rechten Rand.

Bitte beachten Sie: Sie können in die *ADtools*-Leiste auch Aufrufkürzel zu Dateien / Ordnern einfügen (Seite 43).

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 definierte Aktion aus, wie z.B. Booten, Variable ändern, Prozess laden oder starten.
TLed	zeigt einen Variablenwert an durch Leuchten, Blinken, Flackern oder akustischen Alarm und kann zur Überwachung dienen.
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. Variablenwerte können auch geändert werden.
TString	stellt den Inhalt von bis zu 5 String-Feldern dar.
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 jeweiligen Hilfsprogramm aufrufen.

3.14 *ADbasic* und *ADsim*

Dieser Abschnitt bezieht sich auf die Abläufe in *ADbasic* im Zusammenhang mit *ADsim* und mit Prozessor T12 / T12.1. Die entsprechenden Funktionen sind in *ADbasic* nur mit *ADsim*-Lizenz verfügbar.

Das Programmpaket *ADsim* erlaubt, Simulink®-Modelle auf *ADwin*-Hardware lauffähig zu machen; bei Hardware mit Prozessor T12 wird dazu *ADbasic* benötigt. Der Ablauf ist grundsätzlich wie folgt:

– *ADwin*-Blöcke sowie Inports/Outports in das Simulink-Modell einfügen.

– Modell aus Simulink als Modelldatei in C-Code exportieren.

– In *ADbasic*
  - Die Modelldatei in ein *ADsim*-Projekt einfügen.
  - Aus der Modelldatei eine Modell-Library erzeugen.
  - Die Modell-Library in *ADbasic*-Quellcode einbinden.
  - Benötigte Zugriffe auf *ADwin*-Hardware im Quellcode programmieren.
  - *ADbasic*-Prozess kompilieren, auf die *ADwin*-Hardware übertragen und starten.

– Sie können auf den laufenden Prozess und das darin integrierte Simulink-Modell aus *ADsimDesk* (Teil des Programmpakets *ADsim*) oder aus Entwicklungsumgebungen (siehe Kapitel 6.3.4) zugreifen.

Der genaue Ablauf ist im Handbuch „ADsim T12“ beschrieben.

Die Entwicklungsumgebung *ADbasic* stellt für *ADsim* folgende Funktionen zur Verfügung:

– Simulink-Modell in ein Projekt einfügen

  Schaltfläche `Begin/Add ADsim Project` in der Projektleiste oder im Kontextmenü des Projektfensters (oben links in der Toolbox).

– Simulink-Library erzeugen

  Folgende Schaltflächen im Projektfenster erzeugen eine Library auch für Simulink-Modelle (siehe auch Quelltext übersetzen):
  - `Make Bin Files from Selected Files`.
  - `Make All Bin Files of Project`.

Das Erzeugen einer Library ist immer dann nötig, wenn das Modell aus Simulink exportiert wird.

– Quellcode mit Simulink-Library erzeugen

Das Einbinden (mit `Import`) und Aufrufen der erzeugten Library folgt gemäß den Regeln in *ADbasic*.

Zur Vereinfachung verwenden Sie die Schaltfläche im Projektfenster `Import ADsim Library as Template`. Auf Basis der markierten Modelldatei `*.rtw` wird ein *ADbasic*-Quelltext erzeugt, in der die Library fertig eingebunden ist. Der Quelltext wird in das Projekt eingefügt.

Sie können den Quelltext nach Ihren Anforderungen frei verändern, insbesondere können Sie Zugriffe auf *ADwin*-Hardware einfügen.

Beachten Sie: Der Quelltext enthält mehrere Aktualisierungsbereiche, die von den Zeilen

```
'<Region> "ADsim_...
...
'<Endregion>
```

umschlossen sind. Die Bereiche enthalten Informationen, um die Library des Simulink-Modells korrekt aufzurufen. Der Inhalt der Bereiche kann (siehe unten) automatisch aktualisiert werden.

– Quellcode für geänderte Simulink-Library aktualisieren

Die Schaltfläche `Update Project from Simulink library` in der Editor-Leiste passt im aktiven Quellcode die Befehlszeilen in den Aktualisierungsbereichen an die Simulink-Library an (die entsprechende Modelldatei muss im Projekt enthalten sein). Die Aktualisierung funktioniert nur dann, wenn Sie die Aktualisierungsbereiche (siehe oben) unverändert lassen; manuelle Änderungen innerhalb der Bereiche gehen verloren.

Der Quellcode muss aktualisiert werden, wenn im Simulink-Modell die Inports oder Outports oder die Abtastfrequenz verändert wurden. Bei sonstigen Veränderungen im Modell ist die Aktualisierung nicht erforderlich, schadet aber auch nicht. Beachten Sie dabei die Einstellung `Import block captions` unter Reiter General.

Sie können die Library-Aufrufe auch manuell organisieren. In diesem Fall empfehlen wir, die Aktualisierungsbereiche aufzulösen, indem Sie

die Kommentarzeilen (`<Region> „ADsim ... <Endregion>`) um die Bereiche löschen.

4 Prozesse programmieren

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

4.1 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, siehe Quelltext übersetzen auf Seite 20.

Ein Quelltext besteht aus einer beliebigen Anzahl von Befehlszeilen, die jeweils einen Befehl oder eine Zuweisung enthalten; Ausnahme siehe : (Doppelpunkt). Eine Quelltextzeile darf bis zu 2048 (ASCII-) Zeichen enthalten; Ausnahme siehe **`#Include`**.

ADbasic akzeptiert bei Befehlen und Variablennamen Groß- und Kleinschreibung. In unseren Beispielen verwenden wir allerdings zur besseren Unterscheidung eine einheitliche Schreibweise.

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. 14 beim Programmaufbau ein.

Jedes Programm muss zumindest den Abschnitt **`Event:`** enthalten.

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

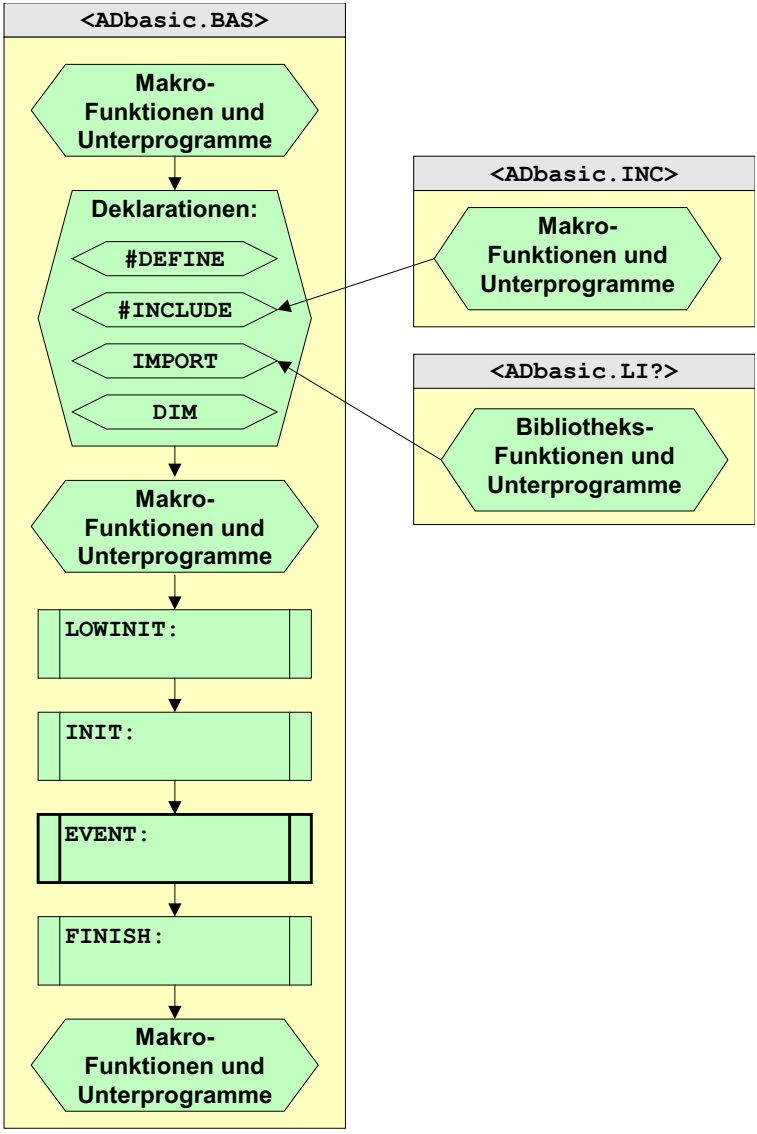


Abb. 14 – Aufbau eines *ADbasic*-Programms

4.1.1 Die Programmabschnitte

Die Programmabschnitte (siehe Abb. 14) beginnen jeweils mit einem Kennwort.

– **`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 seinen 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“ zur Initialisierung. 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.

4.1.2 Befehle für den Hardware-Zugriff

Befehle für den Zugriff auf *ADwin*-Hardware gehören zum Lieferumfang und werden in Include-Dateien (und Bibliotheken (Libraries)) bereit gestellt.

Die folgenden Include-Dateien enthalten die Befehle zur Ansteuerung der Hardware-I/Os. Sie können die Dateien mit einem Textbaustein (siehe Textbausteine einfügen, Seite 40) leicht in den Quelltext einbinden:

`ADwL16.inc`	*ADwin-light-16* Textbaustein einfügen mit IL[TAB].
`ADwin-X.inc`	*ADwin-X-A20* Textbaustein einfügen mit IX[TAB].

`ADwGCnt.inc` `ADwGCAN.inc`	*ADwin-Gold*: Befehle für die Erweiterungen *CO1* und *CAN*; Textbaustein einfügen mit IG[TAB].
`ADwinGoldII.inc`	*ADwin-Gold II*; Textbaustein einfügen mit IG2[TAB].
`ADwinPro_All.inc`	*ADwin-Pro*, *ADwin-Pro II*; Textbaustein einfügen mit IP[TAB].

Wenn Sie sowohl aus *ADbasic* als auch aus *TiCoBasic* auf *ADwin*-Hardware zugreifen, achten Sie darauf, dass Sie nicht auf die gleichen Schnittstellen zugreifen.

4.1.3 Benutzerdefinierte Befehle und Variablen

Sie können Eigene Befehle und Variablen dokumentieren (Seite 23), so dass die eingefügten Hinweistexte beim Programmieren automatisch als Tooltip angezeigt werden.

Das Fenster „Declarations“ (Seite 96) zeigt eine Übersicht aller definierten Befehle und Variablen an.

Alle benutzerdefinierten Namen (symbolische Namen ausgenommen) müssen mit einem Buchstaben beginnen und dürfen nur aus Buchstaben (a-z, A-Z), Ziffern (0-9) und dem Zeichen „_“ (Underscore) bestehen. Umlaute (ä, ö, ü) sind nicht erlaubt, Groß- und Kleinschreibung wird nicht unterschieden. Die Länge von Namen ist nur begrenzt durch die max. Zeilenlänge (2048 Zeichen).

Symbolische Namen

Mit der Anweisung **`#Define`** können Sie symbolische Namen definieren (siehe Seite 208). Gruppieren Sie alle diese Definitionen am Beginn der Datei und vor dem Start der Programmabschnitte.

Symbolische Namen werden häufig für Konstanten, globale Variablen und globale Felder verwendet, aber auch für Berechnungsausdrücke.

Felder und lokale Variablen

In *ADbasic* müssen Sie nur lokale Variablen und alle Felder vor der Benutzung mit `Dim` deklarieren (siehe Seite 210). Die globalen Variablen **`Par_n`** und **`FPar_n`** sind bereits vordefiniert 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 Daten auszutauschen.

Im Fenster „Global Variables“ (Seite 95) können alle verwendeten globalen Variablen und Felder angezeigt werden.

Makros

Makro-Funktionen `Function … EndFunction` und -Unterprogramme `Sub … EndSub` werden bei einem Aufruf im Programmtext an der aufrufenden Stelle eingefügt (siehe auch Kapitel 4.5.1 auf Seite 138). Makros müssen in jedem Fall außerhalb der Programmabschnitte definiert werden (siehe Abb. 14 auf Seite 104).

Die Gruppe der Makros und Libraries werden unter dem Oberbegriff Prozeduren zusammengefasst.

Libraries

Das Einbinden von Libraries muss vor dem Beginn der Programmabschnitte erfolgen. Library-Funktionen `Lib_Function … Lib_EndFunction` und -Unterprogramme `Lib_Sub … Lib_EndSub` sind Programm-Module mit geringerem Speicherbedarf (bei mehrfachem Aufruf) als die o.g. Makro-Funktionen und -Unterprogramme (siehe auch Kapitel 4.5.3 auf Seite 139).

Die Gruppe der Makros und Libraries werden unter dem Oberbegriff Prozeduren zusammengefasst.

4.2 Variablen und Felder

4.2.1 Übersicht

Datenstruktur	Name	Datentyp	Bemerkung
Globale Variablen und Felder			
Variable (Skalar)	**Par_1**...**Par_80**	Long	Vordefiniert, nicht deklarierbar, Speicherbereich DM (bis T11)
	FPar_1...**FPar_80**	Float, Float64	
System-Variable	**Processdelay**	Long	
	Process_Error	Long	
	Processn_Running	Long	
Ein- oder zweidimensionales Feld (Vektor)	**Data_1** [] []... **Data_200** [] []	Long, Float32/64, Float, String, FIFO	Name **Data_** nicht änderbar, nur Deklaration von Feldnummer und Dimension.
Lokale Variablen und Felder			
Variable (Skalar)	frei wählbar	Long, Float32/64, Float	muss deklariert werden
Eindimensionales Feld (Vektor)	frei wählbar	Long, Float32/64, Float, String	muss deklariert werden

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 4.3.1). Beim Prozessor T12/T12.1 werden Variablen standardmäßig im Speicherbereich Cacheable angelegt; bei Feldern entscheidet die Feldgröße, siehe Speicherbereiche (T12/T12.1).

Alle Datentypen haben eine Länge von 32 Bit, nur Float64 hat 64 Bit Länge.

4.2.2 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 4.3.5 auf Seite 127).

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

Der Compiler unterscheidet

– Globale Variablen (Parameter) und Globale Felder (Arrays):

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

  System-Variablen zählen zu den globalen Variablen (siehe Seite 119).

– Lokale Variablen und Felder (siehe Seite 120):

  Lokale Datenstrukturen sind nur innerhalb des Prozesses verfügbar, in dem sie deklariert wurden.

  Innerhalb einer Funktion oder eines Unterprogramms deklarierte Datenstrukturen sind nur lokal innerhalb der Funktion oder des Unterprogramms verfügbar.

Sie deklarieren Variablen und Felder mit der Anweisung `Dim`. Dadurch wird der Datentyp bestimmt, der erforderliche Platz im Speicher belegt und der Speicherplatz 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 wer-

den. Ausnahme: Nach dem Einschalten des *ADwin*-Systems werden globale Variablen automatisch mit Null initialisiert.

4.2.3 Datentypen

Bei der Deklaration von Variablen und Feldern muss der Datentyp angegeben werden.

Der Compiler verarbeitet folgende Datentypen:

`LONG` Ganzzahliger 32 Bit-Wert mit dem Wertebereich:
$-2147483648 \ldots +2147483647 = -2^{31} \ldots +2^{31}-1$

`FLOAT32` Fließkomma-Wert mit 32 Bit / 64 Bit ab Prozessor T12.

`FLOAT64` 32 Bit:

$-3{,}402823 \cdot 10^{+38} \ldots -1{,}175494 \cdot 10^{-38}$

$+1{,}175494 \cdot 10^{-38} \ldots +3{,}402823 \cdot 10^{+38}$

64 Bit:

$-1{,}797693134862315 \cdot 10^{+308} \ldots$
$-2{,}2250738585072014 \cdot 10^{-308}$

$+2{,}2250738585072014 \cdot 10^{-308} \ldots$
$+1{,}797693134862315 \cdot 10^{+308}$

Die Wertebereiche entsprechen nicht dem „IEEE-floating point“-Format.

Die Datentypen `Float32` / `Float64` ersetzen den Datentyp `Float` (siehe unten). Aus Gründen der Kompatibilität akzeptiert der Compiler `Float` weiterhin, ersetzt ihn aber durch `Float64`.

`FLOAT` Fließkomma-Wert bis T11; die Rechengenauigkeit hängt vom Prozessor ab:

32 Bit (T9, T10):

$$-3{,}402823 \cdot 10^{+38} \ldots -1{,}175494 \cdot 10^{-38}$$

$$+1{,}175494 \cdot 10^{-38} \ldots +3{,}402823 \cdot 10^{+38}$$

40 Bit (T11):

$$-3{,}402823668 \cdot 10^{+38} \ldots -1{,}175494351 \cdot 10^{-38}$$

$$+1{,}175494351 \cdot 10^{-38} \ldots +3{,}402823669 \cdot 10^{+38}$$

Bei T12/T12.1 wird `Float` durch `Float64` ersetzt, siehe oben.

Die Wertebereiche entsprechen nicht dem „IEEE-floating point"-Format.

Die Rechengenauigkeit 40 Bit beim Prozessor T11 gilt ausschließlich in folgenden Bereichen:

- Berechnungen innerhalb des *ADwin*-Prozessors.
- 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.

Im Speicher belegt eine 40 Bit-Variable 64 Bit.

`STRING` ASCII-Zeichenfolge, die in Feldern so gespeichert wird, dass jedes Feldelement ein Zeichen enthält (Details siehe Kapitel 4.3.6 auf Seite 129). Ein einzelnes Zeichen entspricht einem ganzzahligen 8 Bit-Wert im Bereich 0…255. Die Zeichen 0…127 sind eindeutig (siehe ASCII-Zeichensatz), die Zeichen 128…255 dagegen abhängig von der Ländereinstellung.

Unicode wird nicht unterstützt.

Die veralteten Datentypen `Short` und `Integer` – bei Prozessoren vor T9 verwendet – wurden durch den Datentyp `Long` ersetzt. Aus Gründen der Kompatibilität akzeptiert der Compiler die Datentypen weiterhin, ersetzt sie aber durch `Long`.

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” auf Seite 135.

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

Bezeichnung gemäß IEEE	Bitmuster-Bereich	32 Bit-Wert auf dem PC
+0	`00000000h`	0
Positive denormalized numbers	`00000001h` `007FFFFFh`	0
Positive normalized numbers	`00800000h` `7F7FFFFFh`	+1,175494 · 10-38 +3,402823 · 10+38
+∞ (Infinity, #INF)	`7F800000h`	3.402823E+38
Signalling Not a number (SNaN)	`7F800001h` `7FBFFFFFh`	3.402823E+38
Quiet Not a number (QNaN)	`7FC00000h` `7FFFFFFFh`	3.402823E+38
-0	`80000000h`	0
Negative denormalized numbers	`80000001h` `807FFFFFh`	0
Negative normalized numbers	`80800000h` `FF7FFFFFh`	-1,175494 · 10-38 -3,402823 · 10+38
-∞ (Infinity, #INF)	`FF800000h`	-3.402823E+38
Signalling Not a number (SNAN)	`FF800001h` `FFBFFFFFh`	3.402823E+38
Indeterminate	`FFC00000h`	3.402823E+38
Quiet Not a number (QNAN)	`FFC00001h` `FFFFFFFFh`	3.402823E+38

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

4.2.4 Zahlenwerte eingeben

Sie können 4 verschiedene Schreibweisen benutzen, wenn Sie einen Zahlenwert angegeben möchten. Die folgenden Beispiele weisen einer Variablen `x` den Wert 930 zu.

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

1. Dezimale Schreibweise:

   `x = 930` LONG

   `x = 930.0` FLOAT

   Beachten Sie den Unterschied: Die Zahl „`930`“ hat den Datentyp `Long`, die Zahl „`930.0`“ dagegen den Datentyp `Float`. Dies ist wichtig, wenn Sie beide Datentypen in einem Berechnungsausdruck miteinander verwenden (siehe Kapitel 4.4.2).

2. Exponential-Schreibweise:

   `x = 93E1` LONG

   `x = 9.3E2` FLOAT

   Hierbei steht · „`9.3E2`“ für $9{,}3 \times 10^2$, d.h. nach dem „`E`“ folgt der (max. 2-stellige) Exponent zur Basis 10.

3. Binäre Schreibweise (angehängtes „`b`“):

   `x = 111010001b` LONG

   Siehe auch: Mit Bitmustern arbeiten.

4. Hexadezimale Schreibweise (angehängtes „`h`“):

   `x = 3A2h` 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.

Bei einer Zahlenwert im Quelltext können Sie die Schreibweise zwischen dezimal, binär und hexadezimal einfach mit [CTRL]-[SPACE] ändern.

4.2.5 Mit Bitmustern arbeiten

Ein Bitmuster ist eine Aneinanderreihung von einzelnen Bits, ist also Information im Binärcode. Die Bits stellen Zustände mit ähnlichem Bezug dar, ein Bit steht z.B. für einen bestimmten Digitalkanal oder einen spezieller Fehler.

Bits in einem Bitmuster werden von 0 aufwärts nummeriert (von rechts nach links). Bei einer Gruppe von Bits – in der Tabelle die Bits 15:0 – wird die höchste Bitnummer zuerst geschrieben.

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

In *ADbasic* können Sie ein Bitmuster in binärer Schreibweise angeben, indem Sie den Bitwerten ein `b` anhängen: `10001b`.

Bitmuster werden in *ADbasic* in Variablen vom Datentyp `Long` (32 Bit) gespeichert, daher entspricht ein Bitmuster immer auch einem Zahlenwert (Wertzuordnung siehe Tabelle oben). Beachten Sie, dass Bitmuster kein Vorzeichen kennen, `Long`-Zahlenwerte aber schon. Ein Zahlenwert kann in *ADbasic* außer in binärer Schreibweise auch in dezimaler oder hexadezimaler Schreibweise angegeben werden: `10001b` = `17` = `11h`, siehe Kapitel 4.2.4.

Beachten Sie, dass auf dem PC die Bits eines Bitmusters möglicherweise in umgekehrter Reihenfolge (Little Endian / Big Endian) gespeichert sein können.

Arbeiten mit Bitmustern

In *ADbasic* verarbeiten Sie Bitmuster insbesondere mit Booleschen Operatoren (**`And`**, **`Not`**, **`Or`**, **`XOr`**) und mit bitweisen Verschiebungen (**`Shift_Left`**, **`Shift_Right`**).

– Bits setzen
  - Bitmaske verwenden, in der zu setzende Bits auf 1 gesetzt sind, alle anderen Bits auf 0.
  - Bitmuster mit der Bitmaske `Or`-verknüpfen.

```
Par_1 = 01001011b
'Bits 0, 2, 3 setzen (Bitmuster mit 8 Bits)
Par_1 = Par_1 Or 00001101b
'       01001011b Bitmuster
'   Or  00001101b Bitmaske
'       ---------
'    =  01001111b Ergebnis
```

– Bits löschen / ausmaskieren
  - Bitmaske verwenden, in der zu löschende Bits auf 0 gesetzt sind, zu erhaltende Bits auf 1.
  - Bitmuster mit der Bitmaske And-verknüpfen.

```
Par_1 = 01001011b
'Bits 0, 2, 3 löschen (Bitmuster mit 8 Bits)
Par_1 = Par_1 And 11110010b
'       01001010b Bitmuster
'  And  11110010b Bitmaske
'       ---------
'    =  01000010b Ergebnis
```

Zum Löschen weniger oder einzelner Bits kann ein komplementäres Bitmuster mit Not hilfreich sein.

```
'Bit 3 löschen
Par_11 = Par_10 And (Not(1000b))
```

– Bits umschalten
  - Bitmaske verwenden, in der die umzuschaltenden Bits auf 1 gesetzt sind, die zu erhaltenden Bits auf 0.
  - Bitmuster mit der Bitmaske XOr-verknüpfen.

```
Par_1 = 01001011b
'Bits 0, 2, 3 umschalten (Bitmuster mit 8 Bits)
Par_1 = Par_1 XOr 00001101b
'       01001010b Bitmuster
'  XOr  00001101b Bitmaske
'       ---------
'    =  01000111b Ergebnis
```

– einzelnes Bit an fester Position prüfen

```
'in Par_10 prüfen: Bit 5 = 1?
If (Par_10 And 100000b) > 0)
  Rem Bit 5 = 1
Else
  Rem Bit 5 = 0
EndIf
```

– einzelnes Bit an variabler Position prüfen

```
'in Par_10 prüfen: Bit 5 = 1?
Par_1 = 5                'Bitnummer
If ((Shift_Right(Par_10, Par_1) And 1b) = 1)
  Rem Bit 5 = 1
Else
  Rem Bit 5 = 0
EndIf
```

– Bitmaske erzeugen

  Um eine Bitmaske mit einem gesetzten Bit an einer bestimmten Position zu erhalten, verwenden Sie **Shift_Left**.

```
'Bitmaske mit gesetztem Bit 15 erzeugen
Par_10 = 15              'Bitnummer
Par_11 = Shift_Left(1, Par_10)
```

  Wenn in einer Bitmaske viele Bits gesetzt sein sollen, kann ein komplementäres Bitmuster mit **Not** hilfreich sein.

```
'Alle Bits der Bitmaske setzen bis auf Bit 15
Par_10 = 15              'Bitnummer
Par_11 = Not(Shift_Left(1, Par_10))
```

– Bitmasken zusammenfassen

  Bitmasken können zusammengefasst werden, z.B. um in einer Operation mehrere Bits auf einmal zu überprüfen oder zu setzen.

```
'Bitmaske mit gesetzten Bits 0 und 7 erzeugen
Par_10 = 0               'Bitnummer
Par_11 = 7               'Bitnummer
Par_12 = Shift_Left(1, Par_10) Or Shift_Left(1, Par_11)
```

4.2.6 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 (siehe auch Kapitel 6.3.1 „Datenaustausch zwischen Prozessen“). 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. Beim Prozessor T12/T12.1 haben globale Fließkomma-Variablen eine Länge von 64 Bit (`Float64`).

Die ebenfalls global verfügbaren System-Variablen sindauf Seite 119 beschrieben.

Beim Prozessor T12/T12.1 können globale Variablen etwas langsamer arbeiten als Lokale Variablen und Felder. Es kann daher sinnvoll sein, globale Variablen nur für den Datenaustausch mit anderen Prozessen oder dem PC zu verwenden.

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 Einschalten des *ADwin*-Systems werden globale Variablen automatisch mit Null initialisiert.

Die in einem Programm verwendeten globalen Variablen und Felder können im Fenster „Global Variables“ (Seite 95) angezeigt werden.

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 `Float64` für Fließkommawerte. Bei Prozessoren bis T11 ist der Datentyp `Float`.

Beispiel

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

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.

4.2.7 Globale Felder (Arrays)

Die globalen Felder ermöglichen Ihnen, große Datenmengen zwischen den Prozessen auf dem *ADwin*-System oder dem PC auszutauschen (siehe auch Kapitel 6.3.1 „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 (siehe `Dim`) und möglichst auch initialisieren (die Feldelemente haben sonst keinen definierten Wert).

Die in einem Programm verwendeten globalen Felder und Variablen können im Fenster „Global Variables“ (Seite 95) angezeigt werden.

Beim Prozessor T12/T12.1 können globale Felder etwas langsamer arbeiten als Lokale Variablen und Felder. Es kann daher sinnvoll sein, globale Felder nur für den Datenaustausch mit anderen Prozessen oder dem PC zu verwenden.

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 als die genannten 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

```
REM Feld 5 mit 20000 Elementen vom Typ Long deklarieren.
Dim DATA_5[20000] As Long
REM Feld 3 mit 7x5 Elementen vom Typ Float deklarieren.
Dim DATA_3[7][5] As Float
```

Näheres zu 2-dimensionalen Feldern steht in Kapitel 4.3.4 auf Seite 126.

Die maximale Größe der Felder richtet sich nur nach dem verfügbaren Speicherplatz. Beispielsweise kann auf einem *ADwin*-System mit 16 MiB Speicher ein Feld mit bis zu 4 Millionen Elementen vom Typ `Long` deklariert werden. Beachten Sie beim Prozessor T12/T12.1 die automatische Zuordnung zu Speicherbereichen nach der Feldgröße, siehe Speicherbereiche (T12/T12.1).

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
Rem  Wert des 200. Elements aus dem Feld 5 zugewiesen.
PAR_1 = DATA_5[200]

Rem Durch diese Anweisung erhält das 345. Element aus
Rem dem Feld Data_5 den Wert 4000.
DATA_5[345] = 4000

Rem Diese Anweisung weist dem 1. (!) Element aus dem
Rem 2 - dimensionalen 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.
Dim nummer1 As Long
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:

```
Dim num As Long          'überflüssige Variable
num = 2
DATA_num[300] = 20       'FALSCH !!
DATA_2[300] = 20         'RICHTIG
```

Der Compiler interpretiert `DATA_num` als Namen eines lokalen Felds, das (noch dazu) nicht deklariert wurde und daher nicht verfügbar ist. Verwenden Sie statt dessen die Schreibweise **`Data_2`**. Beachten Sie die unterschiedliche Syntax-Hervorhebung bei den Variablen.

Bei Prozessoren ab T12 bietet die Include-Datei `ADwin_utils.inc` zusätzliche Befehle, bei denen die Feldnummer mit einer Variablen übergeben wird.

4.2.8 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.

`Processn_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 288). Die Variable kann nur gelesen werden.

`Process_Error`

Zeigt bei aktiviertem Debug-Modus den zuletzt aufgetretenen Fehler des Prozesses `n` an (mit `n` = 1…16, siehe Seite 287). 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`** (siehe auch Seite 284) können Sie die Zykluszeit abfragen oder einstellen. Die Zykluszeit wird in Taktzyklen des Zählers gemessen.

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.

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

User_Version

Nur T12/T12.1: Mit der Konstanten **User_Version** (siehe auch Seite 327) können Sie die Versionsnummer abfragen. Sie stellen die Versionsnummer im Dialogfenster „Process Options“ (Seite 61) ein.

4.2.9 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.[1] Umlaute (ä, ö, ü) sind nicht erlaubt, Groß- und Kleinschreibung wird nicht unterschieden. Die Länge von Variablennamen ist nur begrenzt durch die max. Zeilenlänge (2048 Zeichen).

Bei (skalaren) Variablen sind als Datentypen ganzzahlige Werte in 32 Bit (`Long`) und Fließkomma-Werte in 32/64 Bit (`Float32`, `Float64`) verfügbar.

Beim Prozessor T12/T12.1 können lokale Variablen und Felder etwas schneller arbeiten als globale (siehe Kapitel 4.2.6 und 4.2.7). Beachten Sie auch die automatische Zuordnung zu Speicherbereichen nach der Feldgröße, siehe Speicherbereiche (T12/T12.1).

Beispiel

```
REM Definiere die Variable 'wert' mit dem Datentyp Long
Dim wert As Long
REM Definiere die Variablen 'wert1' und 'wert2'
REM mit dem Datentyp Float
Dim wert1, wert2 As 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.

1. Beachten Sie: Namen lokaler Felder dürfen nicht mit der Zeichenfolge `Data_` beginnen, anderenfalls entsteht ggf. ein Compiler-Fehler.

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.

4.3 Variablen und Felder – Details

4.3.1 Variablen und Felder im Datenspeicher

Dieser Abschnitt ist nur für die Prozessoren T9 … T11 relevant. Für den Prozessor T12/T12.1 beachten Sie bitte Kapitel 4.3.3.

Sie können für Felder und lokale Variablen explizit festlegen, in welchem Speicherbereich (siehe unten) 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. 15 sehen Sie Beispiele für Deklarationen, um Variablen und Felder in den verschiedenen Speicherbereichen anzulegen.

Variable / Feld	Speicher-bereich	Deklaration im Quelltext

Lokale Variable	intern (DM)	`Dim var As <VarType>`
		oder
		`Dim var As … At DM_Local`

	Zusatz (EM)	`Dim var As … At EM_Local`
	extern (DX)	`Dim var As … At DRam_Extern`
Feld	intern (DM)	`Dim array[5] As … At DM_Local`
	Zusatz (EM)	`Dim array[5] As … At EM_Local`

(global oder lokal)	extern (DX)	`Dim array[5] As …`
		oder
		`Dim array[5] As … At DRam_Extern`

Abb. 15 – Festlegung des Speicherbereichs bei Deklarationen

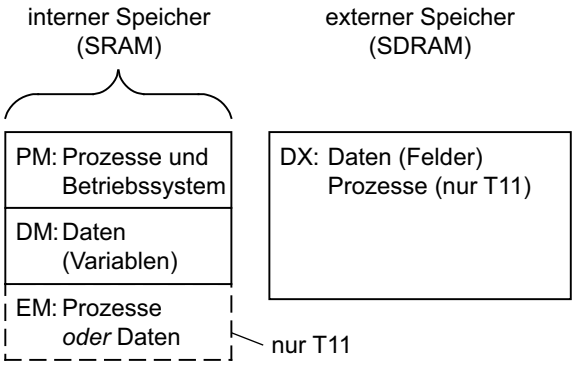
Die globalen Variablen **`Par_1`**…**`Par_80`** und **`FPar_1`**…**`FPar_80`** sind vordefiniert und stehen immer im internen Speicher DM zur Verfügung. Sie brauchen und können diese daher nicht neu (z.B. für den externen Speicher) deklarieren.

4.3.2 Speicherbereiche (T9…T11)

Dieser Abschnitt ist nur für die Prozessoren T9 … T11 relevant. Für den Prozessor T12/T12.1 beachten Sie bitte Kapitel 4.3.3.

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. Die Festlegung geschieht im Quelltext bei der Deklaration mit `Dim` oder beim Programmabschnitt (z.B. **`Init:`**) durch den Zusatz `At EM_Local`.



– Programmspeicher (PM)

  Der Programmspeicher belegt die Hälfte des internen SRAM und nimmt das Betriebssystem und Ihre Prozesse auf.

– Datenspeicher intern (DM)

  Der interne Datenspeicher belegt die Hälfte des internen SRAM und nimmt die globalen und lokalen Variablen 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.

– 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 (und EM) deutlich schneller zugreifen (etwa Faktor fünf) als auf Daten im externen Speicher DX. Die Zugriffszeiten auf die Speicherbereiche PM, EM und DM im internen SRAM sind gleich.

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

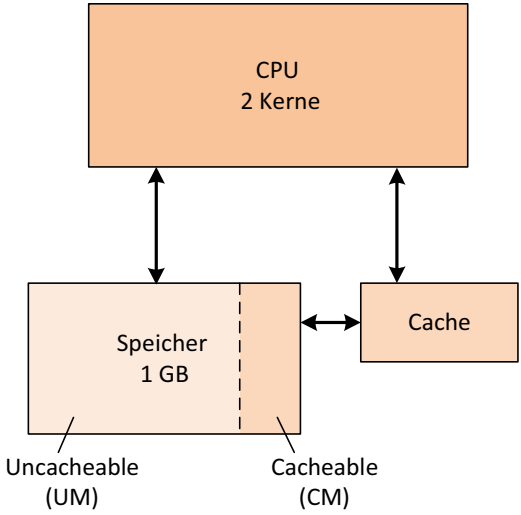
Entwicklungsumgebung, wieviel Speicher Ihnen in PM, DM, EM und DX noch zur Verfügung steht (angegeben in Bytes).

4.3.3 Speicherbereiche (T12/T12.1)

Dieser Abschnitt ist nur für den Prozessor T12/T12.1 relevant. Für die Prozessoren T9 … T11 beachten Sie bitte Kapitel 4.3.1 und Kapitel 4.3.2.

Der Prozessor T12/T12.1 hat einen externen Hauptspeicher mit einer Größe von 1 Gigabyte. Programmcode und Daten werden nicht getrennt, sondern gemeinsam im Hauptspeicher abgelegt.

Der Zugriff auf den Hauptspeicher ist relativ langsam. Zur Beschleunigung des Datenzugriffs nutzt die CPU einen schnellen, internen Speicherbereich, den Cache mit einer Größe von 512 kB. Die CPU füllt zunächst den Cache mit den als nächstes benötigten Werten und Programmbefehlen aus dem Hauptspeicher. Je nach Bedarf lädt die CPU Variablenwerte und Programmcode portionsweise in den Cache nach; wann das geschieht, ist in *ADbasic* nicht vorhersehbar.



Der Teil des Hauptspeichers, der Informationen an den Cache liefern kann, wird als cacheable (CM, 24 MB) bezeichnet, der größere Rest des Hauptspeichers als uncacheable (UM, 1.003 MB).

Etwa 6 MB des CM sind nach dem Booten vom *ADbasic*-Betriebssystem belegt. Im CM werden außerdem Variablen, kleine Felder und Programmcode

aus dem *ADbasic*-Quelltext abgelegt, große Felder dagegen im Bereich UM. In der Regel ist die automatische Speicherzuordnung eine gute Lösung.

Bei Feldern, die (mittels automatischer Speicherzuordnung) im CM abgelegt werden sollen, darf die Elementanzahl in Abhängigkeit vom Datentyp eine festgelegte Größe nicht überschreiten:

Datentyp	lokales Feld	DATA_x
`Long`, `Float32`	< 2.499	< 250.000
`Float64`	< 1.249	< 125.000
`String`	< 9.999	< 1.000.000

Wenn der Speicherplatz z. B. im CM für Programm und Daten nicht ausreicht, erhalten Sie eine Fehlermeldung, sobald der Prozess ins *ADwin*-System geladen wird (siehe auch Speicherfragmentierung). Dann kann es sinnvoll sein, dass Sie die Speicherzuordnung bei **`Data_x`**-Feldern selbst deklarieren, beispielsweise selten benötigte oder sehr große **`Data_x`**-Felder `At Uncacheable` (UM) und häufig genutzte **`Data_x`**-Felder `At Cacheable` (CM). Beachten Sie, dass Speicherdeklarationen indirekt auch die Speicherzuordnung anderer Felder beeinflussen.
Sie können die Speicherzuordnung bei lokalen Feldern nicht verändern.

Programmcode und Daten werden folgendermaßen den Speicherbereichen CM und UM zugeordnet:

Datenblock	CM	UM
ADbasic-Betriebssystem	x	–
Programmcode, lokale Variablen, lokale Felder (<10.000 Byte)	x	–
Lokale Felder >=10.000 Byte	–	x
`DATA_x` `At Cacheable`	x	–
`DATA_x` `At Uncacheable`	–	x
`DATA_x` <= 1.000.000 Byte **`DATA_x`** > 1.000.000 Byte	x –	– x

–

4.3.4 2-dimensionale Felder

Globale Felder `Data_n` können mit 1 oder 2 Dimensionen deklariert werden. Die grundsätzlichen Eigenschaften sind in Kapitel 4.2.7 „Globale Felder (Arrays)“ auf Seite 117 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 um so 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:

```
Dim DATA_1[3][2] As Float
Rem Hauptprogramm
DATA_1[3][1] = 17.2
setpar1(DATA_1)          'setzt FPar_1 = 17.2

Rem Library - Modul
Lib_Sub setpar1(ByRef array[] As Float)
  FPAR_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:

$$\mathbf{DATA_n}[\mathbf{i}][\mathbf{j}] \equiv \mathbf{DATA_n}[\mathbf{s} \cdot (\mathbf{i} - 1) + \mathbf{j}]$$

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

4.3.5 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.

Die grundsätzlichen Eigenschaften von globalen Feldern sind in Kapitel 4.2.7 „Globale Felder (Arrays)“ auf Seite 117 beschrieben.

Ein FIFO ist nicht zu verwechseln mit der Datenstruktur Ringspeicher im *TiCo*-Prozessor (`Ringbuffer`). Der *TiCo*-Ringspeicher ist im Handbuch *TiCo-Basic* beschrieben.

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`, `Float`, `Float32`, `Float64`.

Beispiel

```
Dim DATA_5[1003] As Long As FIFO
```

Diese Anweisung deklariert das globale Feld mit der Nummer 5 als FIFO-Ringspeicher mit 1003 Elementen vom Typ `Long`. Beachten Sie beim T11 die besondere Wahl der Feldgröße, siehe Seite 221.

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, so dass das FIFO-Feld leer ist.

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.

Während ein Prozess oder ein PC-Programm (siehe Kapitel 6.3.2 „Kommunikation zwischen PC und ADwin-System“) in ein FIFO-Feld schreibt, darf keine andere Instanz (Prozess oder PC-Programm) gleichzeitig in das FIFO-Feld schreiben. Für Lese-Zugriffe gilt dasselbe. Anderenfalls geraten die Datenzeiger durcheinander, Daten gehen verloren und der Füllstand stimmt nicht mehr.

Erlaubt und sinnvoll ist, dass eine Instanz in das FIFO-Feld schreibt, während eine andere zur gleichen Zeit daraus Daten liest.

Zugriffe mehrerer Instanzen können problemlos hintereinander stattfinden; die Koordination kann z.B. über ein Semaphor geregelt werden.

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
REM Schreibe den Wert 95 in den FIFO mit der Nummer 5.
DATA_5 = 95
REM Lies einen Wert aus dem FIFO und speichere ihn in der
REM globalen Variablen Par_7.
PAR_7 = DATA_5
```

Um sicherzustellen, dass noch Platz im FIFO ist, sollten Sie vor dem Schreiben die Funktion **`FIFO_Empty`** verwenden. In gleicher Weise prüft die Funk-

tion **FIFO_Full** vor dem Lesen, ob noch nicht gelesene Werte vorhanden sind.

Aus dem Konzept des Ringspeichers ergibt sich, dass Sie jeden Wert aus einem FIFO-Feld nur ein einziges Mal lesen können. Deswegen sollte nur ein einziger Prozess bzw. ein einziges Programm die Daten eines FIFO lesen.

Beispiel

```
Dim free,used,value1 As Long
Dim DATA_1[1003] 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
```

4.3.6 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:

Asc	ASCII-Nummer eines Zeichens bestimmen
Chr	Zeichen zu einer ASCII-Nummer bestimmen
FloToStr	Fließkomma-Wert in einen String wandeln
Flo40ToStr	40 Bit Fließkomma-Wert in einen String wandeln
LngToStr	Long-Wert in einen String wandeln
StrComp	2 Strings auf Gleichheit prüfen
StrLeft	Zeichen linksbündig aus einem String kopieren
StrLen	Länge eines Strings bestimmen
StrMid	Zeichenfolge aus einem String kopieren
StrRight	Zeichen rechtsbündig aus einem String kopieren
ValF	String in einen -Wert wandeln

`ValI`	String in einen Long-Wert wandeln
+ (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, `C` für T12; für T12.1 jedoch `String.1.LIC`). 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 Feldelement 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); für T12/T12.1 nicht verwendbar
`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 bzw. 1 Byte beim Prozessor T12/T12.1. Das erste Element `text[1]` und das letzte Element `text[7]` des Strings reserviert der *ADbasic*-Compiler automatisch.

Achtung: In *ADbasic* ist es nicht erlaubt, das Element 0 zu verwenden, hier also `text[0]`. Beim Prozessor T12/T12.1 ist es in *ADbasic* nicht erlaubt, die Elemente 0 und 1 zu verwenden, also `text[0]` und `text[1]`.

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); für T12/T12.1 nicht verwendbar
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-Zeichensatz, siehe auch Anhang) einsetzen, mit Ausnahme der folgenden Zeichen. Diese müssen Sie per Escape-Sequenz zuweisen (s.u.):

– einfaches Hochkomma ('): `\x27`

– doppeltes Hochkomma ("): `\x22`

– Backslash (\): `\x5C`

Zuweisung per Escape-Sequenz

Wenn Sie nicht nur Texte verarbeiten, sondern z.B. auch Steuerzeichen in einen String einbinden möchten, sollten Sie dafür Escape-Sequenzen einsetzen. Mit jeder Escape-Sequenz übergeben Sie einen einzelnen ASCII-Wert 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` der zu übergebende ASCII-Wert in hexadezimaler Schreibweise ist (siehe ASCII-Zeichensatz im Anhang). Jede Escape-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 ASCII-Werte von `00h` bis `FFh` hinzufügen können, eignen sich Escape-Sequenzen besonders für die Zuwei-

sung nicht darstellbarer Zeichen (wie Line Feed, Carriage Return, …). Die Zeichen `00h`…`7Fh` sind eindeutig (siehe ASCII-Zeichensatz), die Zeichen `80h`…`FFh` dagegen abhängig von der Ländereinstellung.

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\x4CO"
```

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 Editor - unerwünschte Aktionen (Zeilenumbruch o.ä.) und Programmfehler verursachen.

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

4.4 Berechnungsausdrücke

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.

– Befehlen aus dem *ADbasic*-Befehlssatz oder benutzerdefinierte Befehlen.

4.4.1 Auswertung von Operatoren

Für die Auswertung eines Berechnungsausdrucks (Definition siehe Kapitel 4.4) 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 niederer Priorität bearbeitet (siehe Abb. 17).

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

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

Abb. 16 –

Abb. 17 – Prioritäten von Operatoren-Kategorien
(von oben nach unten absteigende Priorität)

Beispiel

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

entspricht

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

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.

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
```

4.4.2 Typkonvertierung

Sie können in *ADbasic* im allgemeinen die Datentypen `Long` und `Float` (siehe auch Kapitel 4.2.3 „Datentypen“) gemeinsam verwenden, ohne auf passende Datentypen zu achten. Die Daten vom Typ `Long` werden, falls erforderlich, automatisch in den Typ `Float` konvertiert.
Die folgenden Angaben gelten auch für den Prozessor T12/T12.1, allerdings wird dann der Datentyp `Float` durch `Float64` ersetzt.

Verwechseln Sie die Typkonvertierung nicht mit den Befehlen **`Cast_FloatToLong`** oder **`Cast_LongToFloat`**, die eine ganz andere Funktion haben (siehe Seite 197).

Beachten Sie bitte folgende Besonderheiten:

Bei der Wandlung von `Int64` zu `Long` werden die Bits 63:32 abgeschnitten und nur die Bits 31:0 bleiben stehen. Damit kann ein ganz anderer Wert entstehen, was Fehler oder unerwartete Berechnungsergebnisse verursachen kann.

– Abschneiden von Nachkommastellen

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

Beim Prozessor T11 zeigt sich dieser Effekt manchmal beim Teilen von Fließkomma-Werten und es entstehen unerwartete Ergebnisse:

```
FPAR_1 = 6
FPAR_2 = 3
PAR_1 = FPAR_1 / FPAR_2 'unerwartetes Ergebnis: Par_1=1
PAR_2 = Round(FPAR_1 / FPAR_2)'korrekte Rechnung: Par_2=2
```

Es ist deshalb oft – insbesondere beim T11 – sinnvoll, vor dem Zuweisen erst eine Rundung mit dem Befehl **Round** (Seite 295) durchzuführen.

– 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_4 = 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
```

4.5 Bedingungen, Schleifen und Module

Wenn Sie Programme schreiben, strukturieren Sie diese in *ADbasic* mit folgenden Elementen:

– Kontrollstrukturen verkürzen aufwändige Abschnitte.
  - Schleifen für oft wiederholte Abschnitte:
    `Do` … `Until` oder
    `For` … `Next`.
  - Abfragen für fallweise Unterscheidungen:
    `If` … `EndIf`, `#If` … `#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`

– Bibliotheken (Libraries) von kompilierten Funktionen und Unterprogrammen, die Sie mit `Import` in den Quelltext einbinden:
  - Library-Unterprogramme: `Lib_Sub` … `Lib_EndSub`
  - Library-Funktionen: `Lib_Function` … `Lib_EndFunction`

– Sammlungen von Quelltext-Abschnitten und Programm-Makros in Include-Dateien, die Sie komplett in den Quelltext einbinden mit
  `#Include filename.Inc`

Sie finden nähere Erklärungen und Beispiele der Befehle in Kapitel 7 „Befehlsreferenz“.

4.5.1 Unterprogramm- und Funktions-Makros

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

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.

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 235: Function … EndFunction; Seite 323: Sub … EndSub).

4.5.2 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 Unterprogramm- und Funktions-Makros, die Befehle für den Hardware-Zugriff definieren; siehe Befehle für den Hardware-Zugriff auf Seite 105. Aus diesem Grund ist die korrekte Stelle für das Einbinden dieser Dateien der Anfang des Quelltexts (siehe Seite 104).

Rekursives Einbinden von Include-Dateien ist nicht erlaubt.

Sobald eine Include-Datei mit **`#Include`** in einen Quelltext eingebunden ist, werden die darin definierten Befehle im Quelltext hervorgehoben (siehe Syntax hervorheben). Auch verschachtelte Einbindungen werden berücksichtigt.

Wenn man die Syntaxhervorhebung für Befehle aus einer Include-Datei, die man (z. B. zum Vermeiden einer Rekursion) nicht mit **`#Include`** eingebunden hat, nutzen möchte, fügt man folgende Zeile ein:

```
'<#Include OtherIncFile.inc>
```

4.5.3 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`, speichern die Datei mit `File / Save as` als Dateityp `Lib-file *.bas` und kompilieren daraus die Library-Datei mit „`Build / Make lib file`“.
Beim Prozessor T12/T12.1 darf der Dateiname nicht mit Punkt und Ziffer (0…9) enden (in der Form `*.0.bas`).

Beachten Sie beim Prozessor T12/T12.1, dass zu jeder Library neben der Datei `*.lic`/`*.1.lic` außerdem die Dateien `*.o` und `*.h` gehören. Alle drei Dateien müssen sich im gleichen Verzeichnis befinden, damit die Library korrekt arbeitet.

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 4.5.1 „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. 18), sonst erhalten Sie eine Fehlermeldung des Compilers.

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

Prüfen Sie beim Prozessor T12/T12.1 im Zusammenhang mit Libraries, ob die Größe des Stapelspeichers ausreicht. Eine ungenügende Stack-Größe kann zu Datenverlust oder Programmabbruch führen. Näheres finden Sie im Fenster Stack Test Analysis (Seite 87).

Sie finden nähere Informationen zum Aufbau von Library-Modulen in der Befehlsreferenz (Seite 252: Lib_Function … Lib_EndFunction; Seite 257: Lib_Sub … Lib_EndSub).

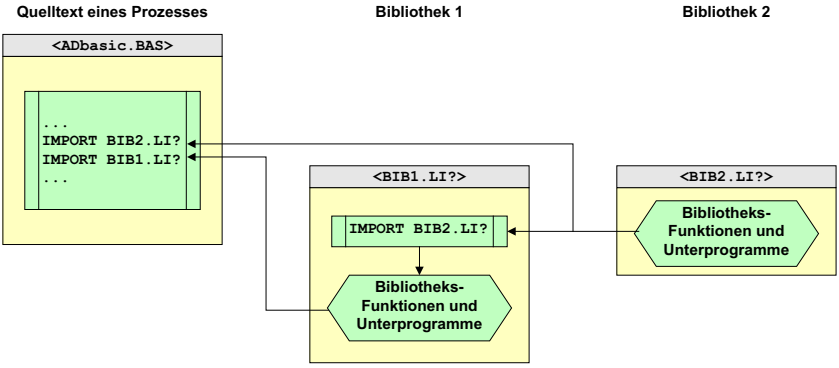


Abb. 18 – Verschachteltes Einbinden von Bibliotheken

Die folgenden Bibliotheken gehören zum Lieferumfang von *ADbasic*.

`String.li*`	Befehle für die String-Operationen.
`Math.li*`	Besondere mathematische Befehle.
`FFT.li*`	Befehle für FFT-Transformationen.

Makros kontra Bibliotheken

Was tun Sie, wenn Sie sowohl ein Makro als auch eine Bibliothek zur Lösung einer Aufgabe verwenden können? In diesem Fall müssen Sie zwischen Programmgeschwindigkeit und Programmgröße abwägen.

Ein Makro wird im Code so oft expandiert, wie es im Quelltext aufgerufen wird. Wenn Ihr Programm eine Makro-Funktion 100-mal aufruft, gibt es 100 Kopien dieses Makros im endgültigen Programm. Im Gegensatz dazu gibt es den Code einer Library-Funktion nur einmal. Deshalb wäre es hinsichtlich einer geringen Programmgröße besser, eine Library-Funktion zu wählen.

Wenn ein Programm eine Library-Funktion aufruft, erfordert das einen bestimmten Verarbeitungsaufwand, um die Programmausführung an die Bibliothek zu übergeben und später wieder in das aufrufende Programm zurückzukehren. Beim „Aufrufen“ eines Makros gibt es diesen Verarbeitungsaufwand nicht, da der Code bereits im Programm steht. Hinsichtlich einer höheren Ausführungsgeschwindigkeit liegen die Vorteile also bei den Makros.

Das Abwägen zwischen Programm-Größe und -Geschwindigkeit ist für den Programmieranfänger eher unerheblich. Wichtig werden diese Betrachtungen erst bei großen oder zeitkritischen Anwendungen.

5 Prozesse optimieren

Ihr *ADwin*-System ist dafür ausgelegt, Regel-, Steuer- und Messaufgabe schnell und präzise auszuführen. Je nach Anforderung kann es erforderlich werden, dass Sie Ihr *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.

5.1 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 hochpriore und für niederpriore Prozesse (siehe Abb. 20 auf Seite 164). 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 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 Anweisungen – 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:
  Rem …
  t1 = Read_Timer()
  Rem …                          'zeitkritischer Abschnitt
  t2 = Read_Timer()
  PAR_1 = t2 - t1- 4             'Bearb.zeit des zeitkritischen
                                 'Abschnitts in Zeittakten
                                 '(Offset = 4 Zeittakte)
  PAR_2 = Calc_TicksToNs(PAR_1)  'Zeit in Nanosekunden
```

Wenn **Par_1** im obigen Beispiel den Wert 37 erhält, hat der zeitkritische Abschnitt bei einem hochprioren Prozess mit dem Prozessor T10 $37 \times 25\,\text{ns} = 925\,\text{ns}$ benötigt.

Sie können die Zeitmessung auch benutzen, um beispielsweise die Zeitdifferenz zwischen zwei externen Event-Signalen zu messen. Im Beispiel wird diese (in Zählerzyklen) 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
  PAR_2 = Calc_TicksToNs(PAR_1)  'Zeit in Nanosekunden
  oldtime = time
```

5.1.1 Besonderheiten beim T12/T12.1

Beachten Sie beim Prozessor T12/T12.1, dass die Zeitmessung ausgewählter Anweisungen teilweise erheblich von den Rahmenbedingungen abhängt. Dazu gehören insbesondere die im folgenden beschriebenen Effekte. Keine Unterschiede zu bisher gibt es, wenn Sie die Zeitdifferenz zwischen zwei Event-Signalen zu messen.
Die Hinweise im vorigen Abschnitt gelten weiterhin.

– Der Compiler kann Berechnungen im *ADbasic*-Quelltext stark optimieren, um eine möglichst schnelle Verarbeitung zu erreichen. Dazu kann der Compiler unter anderem Berechnungsschritte zusammenfassen oder voneinander unabhängige Berechnungen in anderer Reihenfolge ausführen.

  Als Folge kann eine Zeitmessung mit zwei **`Read_Timer`**-Zeilen andere Befehlsschritte umfassen als es im Quelltext steht. Dieser Effekt ist um so geringer, je länger der zu messende Programmabschnitt ist.

  Es gilt aber auch: Zugriffe auf *ADwin*-Hardware wie **`ADC`** oder **`DAC`** werden immer in der Reihenfolge wie im Quelltext ausgeführt.
  Anweisungen mit globalen Variablen **`Par`**, **`FPar`**, **`Data`** und dem Befehl **`Read_Timer`** sind von der Optimierung (weitgehend) ausgeschlossen.

– Die Prozessorarchitektur erlaubt, mehrere Befehle parallel zu bearbeiten. Das reduziert die Gesamtbearbeitungszeit, aber gleichzeitig verliert die Bearbeitungszeit einzelner Befehle an Bedeutung.
  Die parallele Bearbeitung erschwert es außerdem, die Bearbeitungszeit eines Einzelbefehls zu bestimmen.

  Sie können daher die Bearbeitungszeit eines Programmabschnitts nicht einfach aus der Summe von Einzel-Bearbeitungszeiten berechnen. Wir empfehlen, die Zeitmessung über größere Programmteile hinweg vorzunehmen.

– Wenn der Prozessor auf *ADwin*-Hardware oder den Speicher zugreift, kann eine darauf folgende Zeitnahme mit **`Read_Timer`** schon stattfinden, während der Hardware-Zugriff noch in Bearbeitung ist.

  Der Hintergrund ist der Aufbau des Prozessors in mehreren Ebenen, die parallel zueinander arbeiten. Sobald der Prozessor den Hardware-Zugriff an die nächste Prozessorebene weiter gegeben hat, beginnt er schon mit der Bearbeitung des nächsten Befehls, um so die Bearbeitung insgesamt zu beschleunigen.

Je weniger Befehle eine Zeitmessung umfasst, um so mehr müssen Sie Bearbeitungszeiten von vorhergehenden oder nachfolgenden Befehlen berücksichtigen. Mit dem Befehl **Read_Timer_Sync** können Sie sicherstellen, dass ein Zugriff auf *ADwin*-Hardware oder den Hauptspeicher vollständig abgearbeitet ist, bevor die Zeitnahme stattfindet.
Read_Timer_Sync dauert länger als **Read_Timer**.

– Es macht einen Unterschied, ob Sie globale oder lokale Variablen verwenden. Globale Variablen **Par**, **FPar**, **Data** können langsamer sein, weil der Prozessor den Variablenwert bei jedem Zugriff mit dem Cache oder Hauptspeicher abgleicht und dabei eine Verzögerung auftreten kann; bei lokalen Variablen geschieht das nicht.

– Innerhalb des Prozessors existieren verschiedene Zwischenspeicher für Daten oder Befehle. Es hängt sehr von der Vorgeschichte des Programms ab, in welchem Zustand die Zwischenspeicher sich befinden. Verschiedene Messungen für den gleichen Programmabschnitt können daher unterschiedliche Bearbeitungszeiten ergeben.

5.2 Verschiedene Tipps

5.2.1 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 im entsprechenden Hardware-Handbuch.

5.2.2 Konstanten anstelle von Variablen

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

Beispiel

```
FPAR_1 = Sqrt(PAR_2)    'mit Par_2=17
FPAR_1 = Sqrt(17)
```

Für die erste Berechnung muss zur Laufzeit der Wert der Variablen **Par_2** ermittelt, die Wurzel berechnet und **FPar_1** zugewiesen wer-

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

5.2.3 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 Hardware-Handbuch zum *ADwin*-System).

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 II
Event:
  Rem Multiplexer der ADC auf Kanäle 1 und 2 setzen
  Set_Mux1(00000b)
  Set_Mux2(00000b)
  …                        'Einschwingzeit abwarten
  Start_Conv(11b)          'Wandlung an beiden ADC
                           'starten
  Wait_EOC(11b)            'Wandlungsende abwarten
  PAR_1 = Read_ADC(1)      'Auslesen von ADC1
  PAR_2 = Read_ADC(2)      'Auslesen von ADC2
```

Auf *ADwin-light-16* befindet sich nur ein ADC, daher ist das Beispiel nicht umsetzbar.

5.2.4 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.

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 später startet.

Das Überbrücken einer Multiplexer-Einschwingzeit von 14 µs 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:

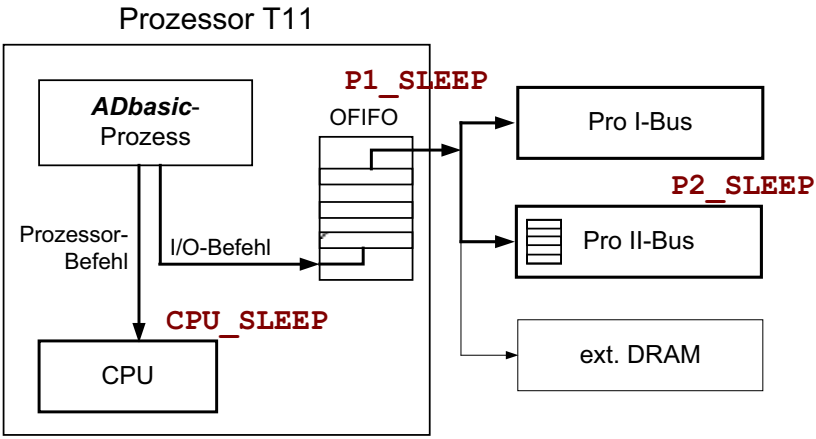
Für die Wartezeit gibt es 3 mögliche Befehle:

- **P1_Sleep** lässt den Pro I-Bus warten, aber auch Pro II-Bus und ext. DRAM.
- **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 14 µs auf einem Pro I-Modul könnte folgendermaßen aussehen:

```
Set_Mux(2,00000b)        'Mux auf Kanal 1 setzen
Rem 14µs auf dem Pro I-Bus warten. Beachten Sie die Zeiteinheit
P1_Sleep(1400)
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[2] 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[3] 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.
3. Genauer gesagt lässt der Befehl **`P1_Sleep`** nicht den Pro I-Bus warten, sondern den Zwischenspeicher `OFIFO`.

5.2.5 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 (16 Bit: Dauer 5µs) für die Kanäle 1+2 folgt das Setzen des Multiplexers (16 Bit: Dauer 5µs) für die Kanäle 3+4.

Beispiel

```
Rem Beispiel für Gold Rev. B, 16 Bit (für T11 nicht geeignet)

Init:
  Set_Mux(000000b)        'Mux für 1. Messung setzen, Kanäle 1 +
                          '2
  Sleep(65)               '6,5 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
  Sleep(15)               'restliche 1,5 µs warten

  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 im Abschnitt **Event:** ist es 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 5.2.4 „Wartezeit genau einstellen“.

5.2.6 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 DM 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 147), wobei Verzögerungen auftreten können. Zum anderen ist die Verwaltung des externen Speichers langsamer als die Verwaltung des internen Speichers.

Sie können auf Daten im internen Speicher DM (und EM) deutlich schneller zugreifen (etwa Faktor fünf) als auf Daten im externen Speicher DX. Die Zugriffszeiten auf die Speicherbereiche PM, EM und DM im internen SRAM sind gleich.

Legen Sie Daten (Variablen, Felder) und Programm „sortenrein“ in die verschiedenen Speicherbereiche, beispielsweise nur Daten in den Zusatzspeicher EM. Der Prozessor kann Daten und Programm parallel laden, wenn sie in verschiedenen Speicherbereichen liegen. Das erhöht die Geschwindigkeit bei der Prozessverarbeitung.

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.

5.3 Debugging und Analyse

Die Bedienoberfläche beinhaltet einige praktische Hilfsmittel zur statischen und dynamischen Fehlersuche und Programmanalyse.

Die dynamischen Analyse-Tools Debug-Modus, Stack-Test und Timing-Modus werden über das Menü „`Debug`“ aktiviert (siehe Kapitel 3.10.7 auf Seite 69) und entfalten ihre Hilfstätigkeit für solche Prozesse, die bei aktiviertem Modus kompiliert werden.

Eine statische Analyse bietet der `Call tree`, mit dem Sie die Programmstruktur analysieren und die Speicherbelegung optimieren können.

☞ 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.

5.3.1 Laufzeitfehler erkennen (Debug-Modus)

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

– 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`“ (siehe Option Debug mode) angezeigt. Außerdem wird der Fehler so behandelt, dass ein stabiler Betriebszustand erhalten bleibt.

Gefundene Fehler müssen in jedem Fall bereinigt werden; die automatische Fehlerbehandlung 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” auf Seite 69 beschrieben.

5.3.2 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 Analyzer“ angezeigt werden können (siehe Abb. 12).

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 6.2.5 „Verschiedene Betriebszustände im Betriebssystem“, Seite 168).

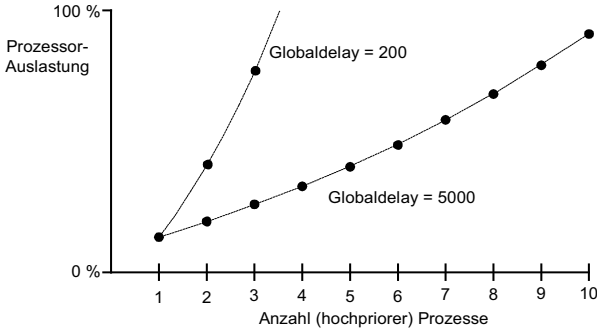
Sie erreichen ein insgesamt optimales Zeitverhalten in 2 Schritten:

1. Anzahl und Priorität von Prozessen prüfen
2. Optimales Zeitverhalten eines Prozesses anstreben, insbesondere bei einem hochprioren Prozess.

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 hochprioren, 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 hochpriorer 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 hochpriore Prozesse den niederprioren 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` ▸ `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` ▸ `Timing Analyzer` wieder, damit weitere Prozesse nicht unnötig Timing-Informationen erzeugen.
4. Zeigen Sie das Fenster „Timing Analyzer“ im Info-Bereich an.

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 6.2.5 „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 Erklärung der Werte finden Sie bei der Beschreibung zum Fenster „Timing Analyzer“, Seite 91.

Die (geringfügige) Änderung des Zeitverhaltens durch den Timing-Modus ist leider nicht vermeidbar und entsteht auch, wenn die Kennwerte nicht ange-

zeigt 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 niederprioren Prozess erheblich verlangsamen kann.

5.3.3 Stack-Überlauf vermeiden (Stack Test)

Beim Prozessor T12/T12.1 können Sie im Dialogfenster „Process Options“ mit dem Parameter `Stacksize` die Größe des Programm-Stapelspeichers einstellen. Ein zu kleiner Stapelspeicher kann insbesondere im Zusammenhang mit Libraries auftreten. Das kann zu einem Überschreiben anderer Speicherbereiche und damit zu Datenverlust oder Programmabbruch führen.

Mit der Option Stack Test können Sie prüfen, ob die eingestellte Größe des Stapelspeichers für den Prozess ausreicht. Sie starten die Prüfung des Stapelspeichers im Fenster Stack Test Analysis (Seite 87). Wenn eine Warnung oder ein Fehlersymbol angezeigt wird, muss der Parameter `Stacksize` des Prozesses entsprechend vergrößert werden.

5.3.4 Programmstruktur analysieren

Der Aufrufgraph (call tree) ist ein einfaches Analyseprogramm zum besseren Verständnis von großen und komplexen *ADbasic*-Programmen; Sie können damit auch Speicherintensive Prozeduren optimieren (Seite 157).

Das Fenster Call Tree (siehe Seite 85) zeigt alle Prozeduren (`Sub`, `Function`, `Lib_Sub`, `Lib_Function`), die in einem *ADbasic*-Quelltext aufgerufen werden, in einem statischen Aufrufgraphen an. Der Aufrufgraph ist nicht kontextsensitiv, d.h. im Baum existiert ein Knoten für jede Prozedur.

Mehr über den Umgang mit dem Aufrufgraphen finden Sie im Kapitel 3.11.7 „Fenster Call Tree“.

Eine einfache Anwendung des Aufrufgraphen ist, die Programmstruktur eines vorhandenen *ADbasic*-Projekts zu analysieren, das möglicherweise aus mehreren Dateien und Libraries besteht. Der Aufrufgraph zeigt die Verschachtelung der Aufrufe an und ermöglicht, die Reihenfolge der Aufrufe leicht zu verfolgen.

Der Aufrufgraph ist geeignet:

– sich leichter in die Logik eines fremden Programms einzuarbeiten,
– bei der eigenen Programmentwicklung den aktuellen Stand zu überblicken,
– die Schritte zur Umsetzung des Programmkonzepts in einer Übersicht zu überprüfen,
– im Rahmen von Teamarbeit die wesentliche Programmlogik verschiedener Autoren transparent darzustellen.

Der Graph dient außerdem als Basis für weitergehende Analysen, wie der Nachverfolgung des Datenflusses zwischen Prozeduren.

Der Aufrufgraph bezieht sich immer nur auf einen einzelnen Prozess.

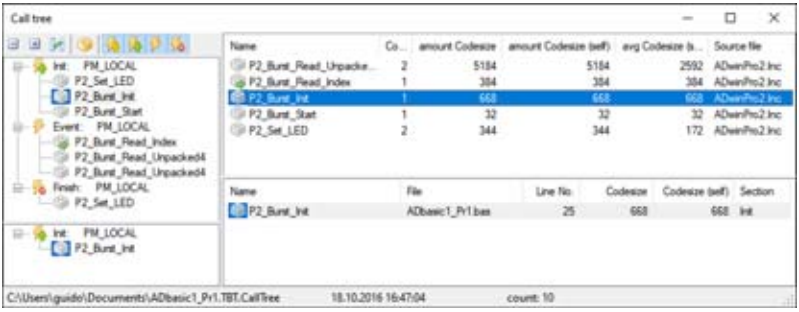
5.3.5 Speicherintensive Prozeduren optimieren

Wenn ein Programm die Speichergrenzen der *ADwin*-Hardware erreicht, können Sie mit folgenden Schritten den benötigten Speicherplatz verringern:

– Verwenden Sie den Aufrufgraph (call tree) um speicherintensive Prozeduren zu finden.
  Suchen Sie im Teilfenster oben rechts nach Prozeduren mit großer Code-Größe (`Total code size`) und nach Makros (`Sub`, `Function`) mit häufigen Aufrufzahlen (`Count`).

  Das Fenster Call Tree (siehe Seite 85) zeigt alle Prozeduren (`Sub`, `Function`, `Lib_Sub`, `Lib_Function`), die in einem ADbasic-Quelltext aufgerufen werden, in einem statischen Aufrufgraphen gemeinsam mit ihrer Speichergröße an.

  Mehr über den Umgang mit dem Aufrufgraphen finden Sie im Kapitel 3.11.7 „Fenster Call Tree“. Der Aufrufgraph ist auch geeignet, um die Programmstruktur analysieren (Seite 156).



– Verringern Sie die Anzahl der Makro-Aufrufe (`Sub`, `Function`) mit großer Code-Größe (`Total code size`), denn mit jedem Makro-Aufruf wird zusätzlicher Code expandiert. `Count` gibt an, wie oft eine Prozedur aufgerufen wird.

  Bei Bibliotheken ist die Anzahl der Aufrufe nicht relevant; siehe auch Makros kontra Bibliotheken auf Seite 141.

– Verringern Sie die Code-Größe von großen Prozeduren, insbesondere von häufig aufgerufenen Makros.

– Kompilieren Sie das Programm erneut und prüfen Sie den benötigten Speicherplatz erneut.

Programmabschnitte können in bestimmten Speicherbereichen abgelegt werden; wo dies geschieht, wird im Aufrufgraph in den Teilfenstern links hinter den Abschnittsnamen angezeigt.

Achten Sie beim Prozessor T11 darauf, dass in den Speicherbereichen Daten und Programmabschnitte „sortenrein“ vorliegen. Das gilt auch für den Zusatzspeicher EM. Wenn Sie dennoch Daten und Programm im Speicher EM mischen, werden die Informationen (je nach Programmablauf) sequentiell und damit langsamer verarbeitet.

6 Prozesse im Betriebssystem

Das *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 darin fest, wie und wann Ihr System analoge und digitale Daten verarbeitet und nach außen gibt.

Nach dem Start des Prozesses wird das Programm[4] im *ADwin*-System (typischerweise) zyklisch, also in regelmäßigen Abständen neu aufgerufen und abgearbeitet. Ein solcher Aufruf eines Prozesszyklus wird durch eines der 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: das ist ein zeitgesteuerter Prozess.
2. Externer Event: Ein externes Signal, das am „Event“-Eingang Ihres *ADwin*-Systems eingeht. Dies könnte beispielsweise ein Impuls eines Inkrementalgebers sein: das ist ein extern gesteuerter Prozess.

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.Mit der Bootloader-Option können Prozesse außerdem beim Starten der *ADwin*-Hardware automatisch gestartet werden. Näheres zum Programmieren des Bootloaders siehe Handbuch „*ADwin*-Bootlader“; für Prozessor T12.1 siehe Kapitel 3.8 „Bootloader nutzen“.

4. genauer: der Programmabschnitt **`Event:`**.

6.1 Prozessverwaltung

6.1.1 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 6.2.5).

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“

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

Abb. 19 – 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 162) 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 der vorher übertragene Prozess überschrieben wird.

6.1.2 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 300 ns.

– 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) je nach Prozessor in verschiedener Abstufung eingestellt werden (siehe Abb. 20 auf Seite 164).

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.

6.1.3 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.

Niederpriore Prozesse werden in die Prioritätsstufen -10 (niedrig) bis +10 (hoch) unterteilt. Prozesszyklen mit niedrigerer Stufe können jederzeit von solchen mit höherer Stufe unterbrochen werden. Die Prozessoren T11 und

T12/T12.1 beachten die Prioritätsstufen bei der Prozessverwaltung sehr strikt (siehe Kapitel 6.2.3 auf Seite 166).

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 2 ms (Prozessor T9), 1 ms (Prozessoren T10, T11), 100 µs (Prozessor T12), 150 µs (Prozessor T12.1).

Niederpriore Prozesse müssen immer zeitgesteuert sein. Die Zykluszeit (Processdelay) kann in diskreten Schritten eingestellt werden; Schrittweite siehe Abb. 20 auf Seite 164.

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

6.1.4 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 250 ms 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 6.3.2 auf Seite 171.

6.1.5 Speicherfragmentierung

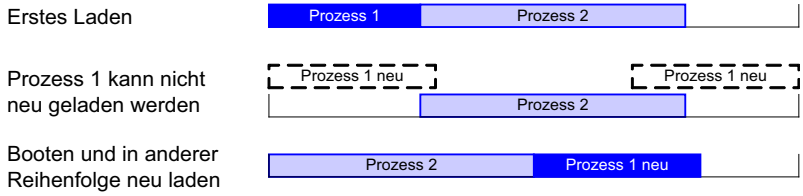
Das Betriebssystem des *ADwin*-Systems sorgt dafür, dass Prozesse, Felder und Variablen an der passenden Stelle im Speicher abgelegt werden und korrekt aufgerufen werden. Daher ist die Speicherverwaltung in aller Regel für den Benutzer problemlos und bedarf keiner weiteren Erklärung.

Unter bestimmten Bedingungen können Sie jedoch die Fehlermeldung „`Not enough memory or memory access error. Please reboot the ADwin`

`system.`“[5] erhalten. Der Grund hierfür ist oft eine Speicherfragmentierung, die entsteht, wenn Prozesse oder Datenfelder mehrmals und mit wachsender Größe in den Speicher geladen werden; ein typischer Vorgang z.B. für das Entwickeln neuer Prozesse.
Eine einfache Lösung ist, das *ADwin*-System zu booten und die Daten neu zu laden.

Bei einer Speicherfragmentierung liegt freier Speicher zerstückelt zwischen belegten Speicherblöcken. Wenn nun ein neuer Datenblock wie ein Prozess oder ein Feld geladen wird – und zwar nur als unzerteilte Einheit – kann es vorkommen, dass der Datenblock in keines der freien Speicherfragmente passt. Sie erhalten die obige Fehlermeldung und müssen den Speicher neu organisieren, um genügend große freie Speicherbereiche zu erhalten.
Booten und Neuladen ist hier geeignet, weil die Datenblöcke ohne Lücke hintereinander abgelegt werden und der freie Speicher damit ein zusammenhängender Bereich ist.

Beispiel: Zwei (recht große) Prozesse sind bereits im Speicher geladen. Prozess 1 soll durch neuen, längeren Code ersetzt werden, aber der neue Datenblock passt weder vor noch nach Prozess 2 in den Speicher und Sie erhalten eine Fehlermeldung. Nach dem Booten und Laden in neuer Reihenfolge kann Prozess 1 nun beliebig oft neu geladen werden, ohne dass der Speicher fragmentiert wird.



Als Alternative können Sie auch den Prozess 2 manuell löschen und beide Prozesse neu laden. Der Vorteil ist, dass damit die Werte der globalen Variablen und Felder erhalten bleiben; bei einem globalen Datenfeld bleiben die Werte nur erhalten, wenn die Größe des Felds unverändert bleibt. Die Schwierigkeit beim manuellen Löschen liegt darin, dass Sie bei vielen Prozessen von Anfang an die Übersicht behalten müssen, welcher Prozess im Speicher an welcher Stelle steht.

Ähnlich wie bei Prozessen kann es auch im Datenspeicher zur Speicherfragmentierung kommen, wenn globale Datenfelder nacheinander mit unter-

5. Bei den Prozessoren T9 und T10 erscheint keine Fehlermeldung, sondern das *ADwin*-System gibt eine Auslastung von 100% an.

schiedlicher Größe geladen werden, z. B. beim Entwickeln eines Prozesses. Beim Laden des Prozesses werden dann die Speicherbereiche der (jetzt anders dimensionierten) Felder freigegeben und jeweils ein neuer Speicherbereich gesucht, was zur Speicherfragmentierung führt. Booten und Neuladen des Prozesses ist auch hier eine einfache Lösung.
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 4.5.2 „Include-Dateien“).

6.2 Zeitverhalten von Prozessen

6.2.1 Processdelay

Der Zeitabstand, in dem *zeitgesteuerte* Prozesszyklen vom Zähler aufgerufen werden, ist die Zykluszeit des Abschnitts `Event:`. Die Zykluszeit wird in Taktzyklen des Zählers gemessen und dann als *Processdelay* bezeichnet (in früheren *ADbasic*-Versionen: Globaldelay). Sie können das Processdelay jedes Prozesses über den Wert der Systemvariablen `Processdelay` festlegen (siehe auch Seite 284).

Auch der Zeitabstand zwischen dem Ende des letzten Befehls im Abschnitt `Init:` und dem Beginn des Abschnitts `Event:` beträgt ein *Processdelay*.

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

Prozessor	Priorität	
	Hoch	Niedrig
T9	25 ns	100 µs
T10	25 ns	50 µs
T11	$3{,}\overline{3}$ ns	$3{,}\overline{3}$ ns = $0{,}00\overline{3}$ µs
T12	1 ns	1 ns
T12.1	1,5 ns	1,5 ns

Abb. 20 – 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 × 25 ns = 25000 ns = 25 µs, bei einem niederprioren Prozess

dagegen von 1000 × 100 µs = 100000 µs = 100 ms. Dies kann z. B. eingestellt werden mit der Programmzeile:

```
= 1000
```

Damit jeder (zeitgesteuerte) Prozesszyklus zu der (mit) 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).

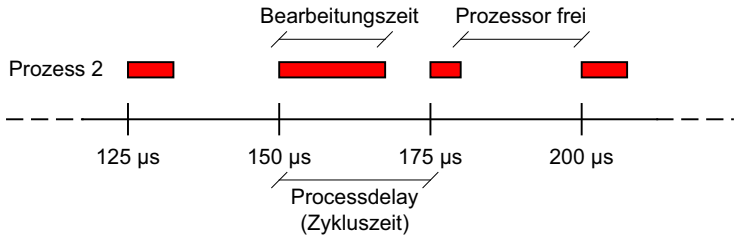


Abb. 21 – Processdelay und Bearbeitungszeit **bei hochprioren Prozesszyklen**

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.

6.2.2 Zeitlich exakter Aufruf von Prozesszyklen

Wenn Sie (wie in Abb. 21) genau einen hochprioren Prozess haben, so wird dieser exakt im Zeitraster aufgerufen und abgearbeitet.

Stellen Sie sicher, dass die Bearbeitungszeit eines hochprioren Prozesszyklus seine Zykluszeit (im Beispiel unten: 25 µs) 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.

Bei Prozessoren bis T11 gilt: Gibt es mehrere hochpriore Prozesse, kann der jeweils aktive Prozesszyklus das Zeitraster der übrigen Prozesszyklen beein-

flussen. In Abb. 22 kann z.B. der geplante Aufruf im Prozess 1 erst verzögert geschehen, wenn die Bearbeitung des aktiven Prozesses 2 beendet ist.

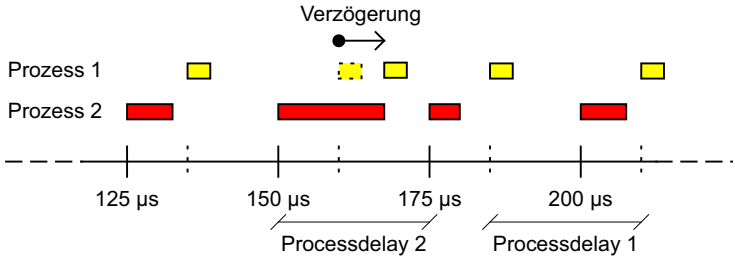


Abb. 22 – Verzögerung eines hochprioren Prozesszyklus

Halten Sie die Ausführungszeit von hochprioren 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.

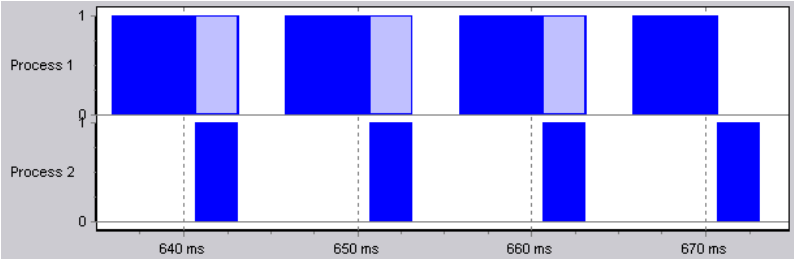
Ein niederpriorer Prozess ist abhängig vom Zeitverhalten aller anderen Prozesszyklen mit höherer oder gleicher Priorität. Jede Unterbrechung verringert das Zeitfenster, in dem der niederpriore Prozesszyklus Rechenleistung nutzen kann; im Extremfall wird er überhaupt nicht aufgerufen.

6.2.3 Niederpriore Prozesse bei T11 und T12/T12.1

Die Prozessoren T11 und T12/T12.1 verwalten niederpriore Prozesse strikt nach deren Prioritätsstufe -10…+10, siehe `Level` im Dialogfenster „Process Options“ auf Seite 59. Bei den Prozessoren T9 und T10 haben Prioritätsstufen nur geringe Bedeutung.
Der Vorrang von Kommunikationsprozess und hochprioren Prozessen vor allen niederprioren Prozessen bleibt davon unberührt.

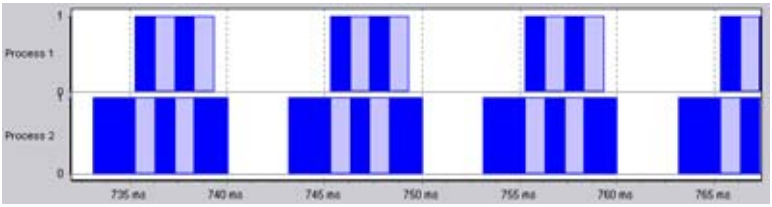
Bei niederprioren Prozessen ist zu unterscheiden:

– 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.

6.2.4 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 (siehe Kapitel 3.11.6). Die-

ser 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.

Auch beim Prozessor T12/T12.1 kann trotz einer `Busy`-Anzeige unter 90% kurzfristig eine Überlastung vorliegen. In diesem (absolut unerwünschten) Betriebszustand gehen Prozesszyklen verloren. In diesem Fall müssen Sie Ihren Prozess korrigieren oder das Processdelay anpassen, um die Auslastung zu senken.
Sie können im Prozessfenster (`Lost events`) ablesen, wie viele Prozesszyklen ein Prozess bereits verloren hat.

6.2.5 Verschiedene Betriebszustände im Betriebssystem

Das Betriebssystem unterscheidet beim Zeitverhalten für hochpriore Prozesse zunächst 2 Betriebszustände, je nachdem, ob nur 1 oder ob mehrere zeitgesteuerte (hochpriore) 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.

Nur für den T12/T12.1 gilt: Das Betriebssystem unterscheidet beim Zeitverhalten nur zwischen hochprioren (zeitgesteuert, extern gesteuert) und niederprioren Prozessen.

Einzelner zeitgesteuerter Prozess

Beim Prozessor T12/T12.1 werden mehrere zeitgesteuerte Prozesse genauso behandelt wie ein 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 6.2.1), 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

Dieser Abschnitt gilt nicht für den Prozessor T12/T12.1. Mehrere zeitgesteuerte Prozesse werden dort genauso behandelt wie ein Einzelner zeitgesteuerter Prozess (siehe vorherigen Abschnitt).

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 5.3.2 auf Seite 153).
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 91). Achten Sie deshalb in diesem Prozess ganz besonders auf kurze Prozesszyklen (im Abschnitt **`Event:`**).

6.3 Kommunikation

6.3.1 Datenaustausch zwischen Prozessen

Sie können Daten zwischen *ADbasic*-Prozessen über globale Variablen (**`Par_1`** … **`Par_80`**, **`FPar_1`** … **`FPar_80`**) oder über globale Felder (**`Data_n`**) austauschen. Für einen kontinuierlichen Datenfluss sind Fifo-Felder besonders geeignet. 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 4.5.2 „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.

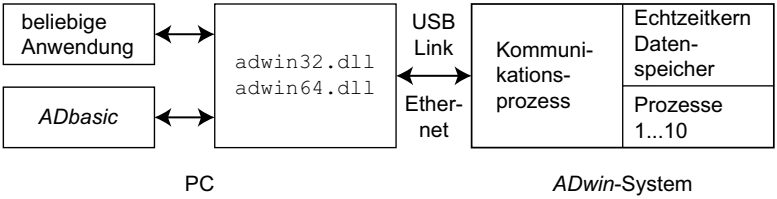
6.3.2 Kommunikation zwischen PC und ADwin-System

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

Daten werden immer über globale Variablen (**`Par_n`**, **`FPar_n`**) oder globale Felder (**`Data_n`**) ausgetauscht (auch beim Datenaustausch zwischen Prozessen, siehe oben).

Jede Kommunikation zum *ADwin*-System läuft unter Windows über die sogenannte `ADwin32.dll` bzw. `ADwin64.dll` (dynamic-link library), im *ADwin*-System übernimmt diese Aufgabe der Kommunikationsprozess (Seite 162).

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 `ADwin32/64.dll` oder erhält sie von dieser.



Die `ADwin32/64.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 Gerätenummer, die sogenannte „Device No“. Anhand dieser „Device No“ unterscheidet die `ADwin32/64.dll` die verschiedenen *ADwin*-Systeme und ordnet die entsprechenden Einstellungen zu.

Die Kommunikation funktioniert in gleicher Weise auch für Echtzeitprozesse, die mit Hilfe von *ADsim* aus einem Simulink®-Modell erzeugt werden. Weitere Details finden Sie im Handbuch *ADsim*.

6.3.3 Die Device No

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

Sie stellen die Device-No. mit dem Programm *ADconfig* ein: `Programs` ▸ `ADwin` ▸ `ADconfig`.

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

6.3.4 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, Java, Visual Basic, C++, Delphi oder C#.NET, oder eine 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 171).

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 3.10.4 auf Seite 54).

– 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.

7 Befehlsreferenz

Im folgenden sind die in *ADbasic* verfügbaren Befehle für *ADwin*-Prozessoren aufgeführt. Befehle zur Steuerung der Ein- und Ausgänge finden Sie in der Hardware-Dokumentation.

Die Befehle sind alphabetisch sortiert aufgeführt. Im Anhang gibt es eine Befehlsübersicht.

In Kapitel 7.4 und Kapitel 7.3 sind *ADbasic*-Befehle für die Anwendung der FFT-Library sowie die Mathematik-Befehle aufgeführt.

7.1 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 angegeben:

LONG	ganze Zahl, 32 Bit
FLOAT	Fließkomma-Zahl Beim Prozessor T12/T12.1 steht FLOAT für den Datentyp `Float64`.
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 135).

– 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`).

7.2 Befehle L16, X-A20, Gold, Pro

Die Befehle in diesem Abschnitt sind gültig für alle Typen von *ADwin*-Hardware: *ADwin-light-16*, *ADwin-X-A20*, *ADwin-Gold*, *ADwin-Gold II*, *ADwin-Pro*, *ADwin-Pro II*.

+ (Addition)

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

Syntax

```
val = val_1 + val_2
```

Parameter

val_1	Summand 1	FLOAT LONG
val_2	Summand 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

- (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, .LIC für T12,
                          '.1.LIC für T12.1

str[] = val_1[] + val_2[]
```

Parameter

str[]	Ziel-String	ARRAY STRING
val_1[]	Zeichenkette 1	ARRAY STRING
val_2[]	Zeichenkette 2	ARRAY 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, Strings

Beispiel

```
Import string.li9
  'T9: .LI9 / T10: .LIA / T11: .LIB / T12: .LIC / T12.1: .1.LIC

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"
  Rem Par_1 = 10 (Anzahl der Zeichen)
  Par_1 = StrLen(res_str)
```

- (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 4.4.1 auf Seite 134).

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 4.4.2 auf Seite 135). 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 4.4.1 „Auswertung von Operatoren“ auf Seite 134).

Siehe auch

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

Beispiel

```
Par_1 = 36 / 4          'Par_1 = 9
REM Par_2: ganzzahlige Rechnung
Par_2 = 2 / 4 * 5       'Par_2 = 0
Par_3 = 27 / (-Par_1)   'Par_3 = -3
Rem Beachten Sie die Klammersetzung in der letzten Zeile
```

^ (Potenz)

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

Syntax

```
val = val_1 ^ val_2
```

Parameter

val_1	Basis	FLOAT LONG
val_2	Exponent	FLOAT LONG

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 kann dies zu den bei Float üblicherweise zu erwartenden Rechenungenauigkeiten bei großen Zahlen führen.

Beispiel für Prozessor T9:

```
Par_2 = 31              ' variable
Par_1 = 2^Par_2         ' = 7FFFFFE2h
```

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 4.4.1 „Auswertung von Operatoren“ auf Seite 134). 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, Round, Sqrt

Beispiel

```
Par_1 = 9 ^ 4            'Par_1 = 6561
```

#… (Compiler-Anweisung)

Ein *ADbasic*-Befehl, der mit dem Zeichen „#“ beginnt, ist eine Anweisung für den Compiler, den Quelltext vor dem Erzeugen des Binärcodes auf eine bestimmte Weise zu bearbeiten.

Folgende Compiler-Anweisungen stehen Ihnen zur Verfügung:

`#Define`	Definition symbolischer Konstanten: Zeichenfolgen im Quelltext werden durch andere Zeichenfolgen ersetzt.
`#Include`	Datei einfügen: Eine Datei (mit Quelltext) wird in den Quelltext eingefügt.
`#If…#EndIf`	Bedingte Kompilierung: Bei erfüllter Bedingung werden die entsprechenden Quelltextzeilen kompiliert, anderenfalls gelöscht
`#Begin_Debug_Mode_Disable`	Unterbricht in einem laufenden Prozess den Debug-Modus.
`#End_Debug_Mode_Disable`	Hebt die Unterbrechung des Debug-Modus wieder auf.

: (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 2048 Zeichen beinhalten.

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.

AbsF ersetzt die Funktion **Abs** (ab Prozessor T11), die aus Kompatibilitätsgründen weiterhin gültig ist.

Syntax

```
ret_val = AbsF(value)
```

Parameter

value	Argument	FLOAT
ret_val	Betrag des Arguments	FLOAT

Bemerkungen

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

Siehe auch

AbsI, Round

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: $-(2^{31}-1) \ldots +2^{31}-1$.	LONG
ret_val	Betrag des Arguments $(0 \ldots +2^{31}-1)$.	LONG

Bemerkungen

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

Für den kleinsten negativen, ganzzahligen Wert -2^{31} gibt es in *ADbasic* keine positive Entsprechung; der Betrag von -2^{31} ist daher undefiniert.

Siehe auch

AbsF, Mod

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	LONG
expr1, expr2	Boolescher Ausdruck mit dem Wert „wahr“ oder „falsch“	LOGIC

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           '= 6
val3 = val_1 And val_2 'Bitweise Verknüpfung
Rem Ergebnis: val3 = 0100b = 4

Rem ODER:
Rem Boolesche Verknüpfung von Booleschen Ausdrücken
Dim fval_1 As Float
Dim 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

val	Argument (-1 … +1)	FLOAT
ret_val	Arcus-Cosinus des Arguments im Bogenmaß (0…π)	FLOAT

Bemerkungen

Für val < -1 wird der Wert π (3,14159…) zurückgegeben, für val > 1 der Wert 0 (Null).

Die Ausführungszeit des Befehls beträgt beim T9 bis zu 2,9µs, beim T10 bis zu 1,45µs, beim T11 bis zu 0,68µs.

Siehe auch

Sin, Cos, Tan, ArcSin, ArcTan, ArcTan2, Round

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)	FLOAT
ret_val	Arcus-Sinus des Arguments im Bogenmaß (-π/2 … +π/2)	FLOAT

Bemerkungen

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

Siehe auch

Sin, Cos, Tan, ArcCos, ArcTan, ArcTan2, Round

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

val	Argument (gesamter Wertebereich, siehe „Zahlenwerte eingeben“ auf Seite 113).	FLOAT
ret_val	Arcus-Tangens des Arguments im Bogenmaß (-π/2…π/2)	FLOAT

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, ArcTan2, Round

Beispiel

```
Dim val_1, val_2 As Float

Event:
  val_1 = 0.5
  val_2 = ArcTan(val_1)
  Rem Ergebnis: val_2 = 0.4636
```

ArcTan2

Ab Prozessor T12: **ArcTan2** liefert den Arcus-Tangens zu einer kartesischen Koordinate.

Syntax

```
ret_val = ArcTan2(y, x)
```

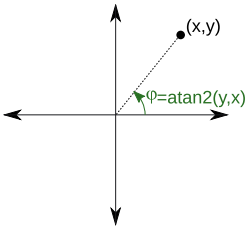
Parameter

y	Y-Wert der kartesischen Koordinate.	FLOAT
x	X-Wert der kartesischen Koordinate.	FLOAT
ret_val	Arcus-Tangens der kartesischen Koordinate im Bogenmaß (-π…π), auch Polarwinkel φ.	FLOAT

Bemerkungen

Die Funktion ist eine Erweiterung von **ArcTan** und wie diese eine Umkehrfunktion der Winkelfunktion **Tan**. Der Wertebereich des Rückgabewerts umfasst den Vollkreis von 360°, also alle vier Quadranten.

Der Rückgabewert gibt den Polarwinkel φ zwischen der positiven x-Achse und dem Strahl zu einem Punkt (x, y) an, siehe Skizze.



Ebenso ist der Rückgabewert der Winkel einer komplexen Zahl x + iy.

Siehe auch

Sin, Cos, Tan, ArcSin, ArcCos, ArcTan, Round

Beispiel

```
Dim x, y, val As Float

Event:
  x = 0.5
  y = 0.5
  val = ArcTan2(y, x)
  Rem Ergebnis: val = 0.7854
```

Asc

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

Syntax

```
ret_val = Asc(string)
```

Parameter

string	Zeichenfolge	STRING
ret_val	ASCII-Nummer (0…255) des (ersten) Zeichens	LONG

Bemerkungen

Die Zeichen 0…127 sind eindeutig (siehe ASCII-Zeichensatz), die Zeichen 128…255 dagegen abhängig von der Ländereinstellung.

Unicode wird nicht unterstützt.

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
```

#Begin_Debug_Mode_Disable

#Begin_Debug_Mode_Disable verhindert bei eingeschaltetem Debug-Modus das Erzeugen von Debug-Informationen.

Syntax

#Begin_Debug_Mode_Disable

Parameter

- / -

Bemerkungen

#Begin_Debug_Mode_Disable ist eine Präprozessor-Anweisung und hat nur eine Funktion, wenn der Prozess mit der Option Debug mode kompiliert wird (siehe Seite 69). Der Debug-Modus dient dazu, Laufzeitfehler zu erkennen, siehe Kapitel 5.3.1 auf Seite 152.

Die Unterbrechung des Debug-Modus wird mit **#End_Debug_Mode_Disable** wieder aufgehoben.

Sie können Unterbrechungen des Debug-Modus nicht verschachteln, d.h. jedes **#End_Debug_Mode_Disable** hebt die Unterbrechung in jedem Fall wieder auf. Achten Sie dabei besonders auf Unterbrechungen in Makros oder Bibliotheken.

Siehe auch

#End_Debug_Mode_Disable, #Include

Beispiel

```
Event:
  #Begin_Debug_Mode_Disable
  Rem unkritischer Quellcode, der nicht überwacht wird
  Rem …
  #End_Debug_Mode_Disable
  Rem kritischer Quellcode mit Überwachung auf
  Rem Laufzeitfehler
  Rem …
```

Cast_FloatToLong

Bis T11: **Cast_FloatToLong** wandelt den Datentyp eines Arguments von Float (Fließkomma) in Long (ganze Zahl).

Syntax

```
ret_val = Cast_FloatToLong(var)
```

Parameter

var	Bitmuster mit Datentyp Float (Fließkomma)	FLOAT
ret_val	Identisches Bitmuster mit Datentyp Long (ganze Zahl)	LONG

Bemerkungen

Für Prozessor T12/T12.1 siehe **Cast_Float32ToLong**.

Es handelt sich bei dieser Funktion **nicht** um die übliche Typkonvertierung eines Zahlenwerts (siehe Kapitel 4.4.2 „Typkonvertierung“, Seite 135). 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 4.2.3 auf Seite 110).

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

Bis T11: **Cast_LongToFloat** wandelt den Datentyp eines Bitmusters von Long (ganze Zahl) in Float (Fließkomma).

Syntax

```
ret_val = Cast_LongToFloat(val)
```

Parameter

val	Bitmuster mit Datentyp Long (ganze Zahl)	LONG
ret_val	Identisches Bitmuster mit Datentyp Float (Fließkomma 32 Bit)	FLOAT

Bemerkungen

Für Prozessor T12/T12.1 siehe **Cast_LongToFloat32**.

Es handelt sich bei dieser Funktion **nicht** um die übliche Typkonvertierung eines Zahlenwerts (siehe Kapitel 4.4.2 „Typkonvertierung“, Seite 135). 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 4.2.3 auf Seite 110).

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

Cast_Float32ToLong

Nur T12/T12.1: **Cast_Float32ToLong** wandelt den Datentyp eines Arguments von Float32 (Fließkomma) in Long (ganze Zahl).

Syntax

```
ret_val = Cast_Float32ToLong(var)
```

Parameter

var	Bitmuster mit Datentyp Float32 (Fließkomma 32 Bit)	FLT32
ret_val	Identisches Bitmuster mit Datentyp Long (ganze Zahl)	LONG

Bemerkungen

Es handelt sich bei dieser Funktion **nicht** um die übliche Typkonvertierung eines Zahlenwerts (siehe Kapitel 4.4.2 „Typkonvertierung“, Seite 135). Für die Zuweisung einer Fließkomma-Zahl an eine ganzzahlige Variable genügt der Operator „=“.

Die Funktion ist im Zusammenhang mit der Umkehrfunktion **Cast_LongToFloat32** sinnvoll einsetzbar, wenn ein Bitmuster eine Fließkomma-Zahl (32 Bit) 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 4.2.3 auf Seite 110).

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_Float32ToLong** von Float32 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_LongToFloat32** wieder vom Datentyp Long in Float32 gewandelt.

Siehe auch

Cast_LongToFloat32

Für Prozessoren bis T11 siehe **Cast_FloatToLong**.

Cast_LongToFloat32

Nur T12/T12.1: **Cast_LongToFloat32** wandelt den Datentyp eines Bitmusters von Long (ganze Zahl) in Float32 (Fließkomma).

Syntax

```
ret_val = Cast_LongToFloat32(val)
```

Parameter

val	Bitmuster mit Datentyp Long (ganze Zahl)	LONG
ret_val	Identisches Bitmuster mit Datentyp Float32 (Fließkomma 32 Bit)	FLT32

Bemerkungen

Es handelt sich bei dieser Funktion **nicht** um die übliche Typkonvertierung eines Zahlenwerts (siehe Kapitel 4.4.2 „Typkonvertierung“, Seite 135). Für die Zuweisung einer Fließkomma-Zahl an eine ganzzahlige Variable genügt der Operator „=“.

Die Funktion ist sinnvoll einsetzbar, wenn ein Bitmuster eine Fließkomma-Zahl (32 Bit) 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 4.2.3 auf Seite 110).

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_Float32ToLong** von Float32 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_LongToFloat32** wieder vom Datentyp Long in Float32 gewandelt.

Siehe auch

Cast_Float32ToLong

Für Prozessoren bis T11 siehe **Cast_LongToFloat**.

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, .LIC für T12,
                          '.1.LIC für T12.1

Chr(vascii,dest_text)
```

Parameter

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

Bemerkungen

Wenn eine String-Variable mehr als ein Zeichen (oder Element) hat, weist **Chr** die ASCII-Nummer nur dem ersten Zeichen zu.Die Zeichen 0…127 sind eindeutig (siehe ASCII-Zeichensatz), die Zeichen 128…255 dagegen abhängig von der Ländereinstellung.

Unicode wird nicht unterstützt.

Siehe auch

"" String, + (String-Addition), Asc, FloToStr, Flo40ToStr, LngToStr, StrComp, StrLeft, StrLen, StrMid, StrRight, ValF, ValI, Strings**Beispiel**

```
Import String.LI9
  'T9: .LI9 / T10: .LIA / T11: .LIB / T12: .LIC / T12.1: .1.LIC

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

angle	Winkel im Bogenmaß (-π…π)	FLOAT
ret_val	Cosinus des Winkels (-1…1)	FLOAT

Bemerkungen

Wenn Sie für den Winkel Werte außerhalb von -π…+π verwenden, nimmt der Berechnungsfehler mit wachsendem Wert zu.Die Ausführungszeit des Befehls beträgt beim T9 bis zu 1,3 µs, beim T10 bis zu 0,7 µs, beim T11 bis zu 0,31 µs.

Siehe auch

Sin, Tan, ArcCos, ArcSin, ArcTan, ArcTan2, Round

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

Ab Prozessor T11: **CPU_Sleep** lässt den Prozessor für eine bestimmte Zeit warten.

Syntax

CPU_Sleep(val)

Parameter

val	Anzahl der zu wartenden Zeiteinheiten in 10 ns. Der Wertebereich ist vom Prozessor abhängig: T11: 9…715827879 ≈ 2^{30} / 1,5T12: 9…214748364 ≈ 2^{30} / 5 T12.1: 9…322122547 ≈ 2^{30} * 0,3	LONG

Bemerkungen

Hinweis für *ADwin-Pro II*: Alternativ gibt es die Anweisungen **P1_Sleep** und **P2_Sleep** (siehe auch Kapitel 5.2.4 „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.
- Nur T11: 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

IO_Sleep, NOP, P1_Sleep, P2_Sleep, Processdelay, Sleep

Beispiel

```
Event:
  Rem Warten, um eine nachfolgende Messung genau 100 µs
  Rem nach dem externen Event-Signal zu starten.
  CPU_Sleep(10000)
  Rem …
```

Data_n

Mit `Dim Data_n[...] As` ... wird ein globales `Data`-Feld dimensioniert. Weitere Informationen zur Dimensionierung siehe `Dim` auf Seite 210.

Syntax

```
Dim Data_n[dim1] {, Data_n[dim2]} As <arr_type>
    {At <mem_type>}
Dim Data_n[dim1]{[dim2]} As <arr_type>
    {At <mem_type>}
```

Parameter

`Data_n`	Name des deklarierten `Data`-Felds mit `n`: 1…200.
`<arr_type>`	Datentyp: `Float32`, `Float64`, `Float` (bis T11), `Long`, `String`.
`dim1`, `dim2`	Feldgröße: Anzahl (1…2^31) der Elemente vom Typ `arr_type` im Feld. CONST LONG
`<mem_type>`	Speicher, in dem die Variablen abgelegt werden: `Cacheable`: Hauptspeicher CM, der Daten an den Cache liefern kann (nur T12/T12.1). `Uncacheable`: Hauptspeicher UM, der keine Daten an den Cache liefern kann (nur T12/T12.1). Bis T11: `DRAM_Extern`: externer Datenspeicher (Default). `DM_Local`:interner Datenspeicher. `EM_Local`: Zusätzlicher interner Programm- oder Datenspeicher (nur T11).

Bemerkungen

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

Wenn Sie beim Prozessor T12/T12.1 keinen Speicherbereich `<mem_type>` angeben, legt der Compiler den Speicherbereich allein anhand der Feldgröße fest:

Datentyp	CM	UM
Long, Float32	< 250.000	≥ 250.000
Float64	< 125.000	≥ 125.000
String	< 1.000.000	≥ 1.000.000

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

Ein globales Feld kann auch 2-dimensional deklariert werden. Näheres ist in „2-dimensionale Felder" auf Seite 126 beschrieben.

⚠ Wenn Sie die Include-Datei `EEPROM_Support.Inc` verwenden, dürfen Sie die Felder **`Data_199`** und **`Data_200`** nicht deklarieren oder für eigene Zwecke in *ADbasic*-Prozessen einsetzen. Die Felder **`Data_199`** und **`Data_200`** werden in der Include-Datei `EEPROM_Support.Inc` zum Datenaustausch mit dem ADwin Ethernet Interface deklariert und verwendet.

Siehe auch

Dim, FIFO, „Globale Felder (Arrays)" auf Seite 117, „Variablen und Felder im Datenspeicher" auf Seite 121

Beispiel

```
Rem Dimensioniere das globale Feld Data_15 mit
Rem 1000 Long-Elementen
Dim Data_15[1000] As Long

Rem Bis T11: Dimensioniere das globale Feld Data_5 mit
Rem 20 x 75 Float-Elementen (bis T11)
Dim Data_5[20][75] As Float
Rem Ab T12: Dimensioniere die Felder Data_6, Data_7 mit
Rem jeweils 8 x 1000 Float32/Float64-Elementen
Dim Data_6[8][1000] As Float32
Dim Data_7[8][1000] As Float64
```

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: val=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

`name`	Symbolischer Name, *ohne* Hochkommata. Sonderzeichen sind nicht erlaubt, nur alphanumerische Zeichen (a…z, A…Z, 0…9) und der Unterstrich (_).	CONST STRING
`expression`	Ausdruck, für den der symbolische Name steht; *ohne* Hochkommata. Alle Zeichen sind erlaubt.	CONST STRING

Bemerkungen

Stellen Sie diese Anweisung an den Beginn eines Quelltextes. Die Ersetzung wird im Quelltext ab der darauf folgenden Zeile durchgeführt.

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[1]. 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 4.2.3 auf Seite 110 sowie Informationen zu FIFO-Feldern unter dem Stichwort FIFO auf Seite 221.

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
    {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: `Float32`, `Float64`, `Float` (bis T11), `Long` für Felder zusätzlich: `String`
`dim1`, `dim2`	Feldgröße: Anzahl (1…2^31) der Feldelemente vom Typ `<var_type>`.

CONST

LONG

`<mem_type>` Speicherbereich, in dem die Variablen abgelegt werden:

`Cacheable`: Hauptspeicher CM, der Daten an den Cache liefern kann; nur für DATA-Felder mit T12/T12.1.

`Uncacheable`: Hauptspeicher UM, der keine Daten an den Cache liefern kann; nur für DATA-Felder mit T12/T12.1.

Bis T11:

`DRAM_Extern`: externer Datenspeicher (Default für Felder)

`DM_Local`:interner Datenspeicher (Default für Variablen)

Nur T11:

`EM_Local`: 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.

Beim Prozessor T12/T12.1 legt der Compiler den Speicherbereich anhand der Feldgröße fest (siehe Tabelle); dies gilt auch, wenn ein ungültiger Speicherbereich angegeben ist, siehe Warnung 22. Nur für globale Felder **Data_x** können Sie den Speicherbereich Cacheable / Uncacheable explizit deklarieren, siehe auch Speicherbereiche (T12/T12.1).

Datentyp	CM	UM
Data_x As Long / Float32	< 250.000	≥ 250.000
Data_x As Float64	< 125.000	≥ 125.000
Data_x As String	< 1.000.000	≥ 1.000.000
array As Long / Float32	< 2.499	≥ 2.499
array As Float64	< 1.249	≥ 1.249
array As String	< 9.999	≥ 9.999
lokale Variablen	immer	–

⚠ Wenn Sie die Include-Datei EEPROM_Support.Inc verwenden, dürfen Sie die Felder **Data_199** und **Data_200** nicht deklarieren oder für eigene Zwecke in *ADbasic*-Prozessen einsetzen. Die Felder **Data_199** und **Data_200** werden in der Include-Datei EEPROM_Support.Inc zum Datenaustausch mit dem ADwin Ethernet Interface deklariert und verwendet.
Beachten Sie bei FIFO-Feldern Besonderheiten bei der Feldgröße dim in Verbindung mit T11 (siehe Seite 221).

String-Variablen sind Felder vom Typ STRING (siehe „Strings“ auf Seite 129). Sie können nicht als FIFO deklariert werden.

Beim Prozessor T10 dürfen Strings nicht mit `At DM_Local` im internen Datenspeicher deklariert werden.

Hinweise zum Umgang mit den verschiedenen Speicherbereichen `<mem_type>` finden Sie über die unten angegebenen Querverweise.

Siehe auch

Data_n, Event:, FIFO, Finish:, Init:, LowInit:, "" String, „Variablen und Felder" auf Seite 108, „2-dimensionale Felder" auf Seite 126, „Variablen und Felder im Datenspeicher" auf Seite 121, „Speicherbereiche (T12/T12.1)" auf Seite 124, „Speicherbereiche (T9…T11)" auf Seite 122, „Datentypen" auf Seite 110

Beispiel

```
Rem Dimensioniere var1 als Long-Variable
Dim var1 As Long

Rem Dimensioniere das Feld array1 mit
Rem 1000 Long-Elementen
Dim array1[1000] As Long

Rem Dimensioniere das globale Feld Data_15 mit
Rem 1003 Long-Elementen als Fifo
Dim Data_15[1003] 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
  …                    'Anweisungsblock
Until (condition)
```

Parameter

condition	Boolesche Abbruchbedingung mit den Operatoren <, >, =, And und Or.	LOGIC

Siehe auch

< = > (Vergleich), And, Or, For … To … {Step … } Next, SelectCase

Bemerkungen

Sie können Do…Until-Schleifen beliebig tief verschachteln.

Vermeiden Sie Schleifen mit langer Ausführungszeit in hochprioren Prozessen, weil diese nicht unterbrochen werden können.

Beispiel

```
Dim count As Long
Dim Data_1[103] As Long As FIFO

Init:
  count = 1

Event:
  Do                        'Schleife beginnen
    Data_1 = ADC(1,4)       'Messwert auslesen
    Inc count               'Zählvariable erhöhen
  Until (count > 100)       '100 Messungen
                            durchgeführt?
```

End

End beendet einen Prozess im Abschnitt **Event:**.

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, ProcessN_Running, Restart_Process, Start_Process, Start_Process_Delayed, Stop_Process, Event:

Beispiel

```
Event:
  If (ADC(1) > 3000) Then 'Messen und Vergleichen
    End   'Prozess beenden, aber Finish: noch ausführen
  EndIf

Finish:
  Set_Digout(1)          'Dig. Ausgang 1 setzen
```

#End_Debug_Mode_Disable

#End_Debug_Mode_Disable hebt die Unterbrechung des Debug-Modus wieder auf.

Syntax

#End_Debug_Mode_Disable

Parameter

- / -

Bemerkungen

#End_Debug_Mode_Disable ist eine Präprozessor-Anweisung und hat nur eine Funktion, wenn der Prozess mit der Option Debug mode kompiliert wurde (siehe Seite 69) und der Debug-Modus mit **#Begin_Debug_Mode_Disable** unterbrochen wurde. Der Debug-Modus dient dazu, Laufzeitfehler zu erkennen, siehe Kapitel 5.3.1 auf Seite 152.

Der Debug-Modus wird mit **#Begin_Debug_Mode_Disable** unterbrochen.

Sie können Unterbrechungen des Debug-Modus nicht verschachteln. Achten Sie dabei besonders auf Unterbrechungen in Makros oder Bibliotheken.

Siehe auch

#Begin_Debug_Mode_Disable, #Include

Beispiel

```
Event:
  #Begin_Debug_Mode_Disable
  Rem unkritischer Quellcode, der nicht überwacht wird
  Rem …
  #End_Debug_Mode_Disable
  Rem kritischer Quellcode mit Überwachung auf
  Rem Laufzeitfehler
  Rem …
```

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 **Event:** 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 4.1.1 auf Seite 105.

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 wird durch einen zyklischen Timer-Event oder durch einen externen Event ausgelöst. Näheres ist in Kapitel 6 „Prozesse im Betriebssystem“ beschrieben.

Bei dem Prozessortyp T11 kann jeder Programmabschnitt separat einem bestimmten Speicherbereich zugeordnet werden (siehe Kapitel 4.3.2 „Speicherbereiche (T9…T11)“). Der große, aber langsamere Speicherbereich DRAM_Extern sollte nur für zeitunkritische Programmabschnitte genutzt werden; meistens sind das die Abschnitte **LowInit:**, **Init:**, **Finish:**.

Beim Prozessormodul Pro-CPU T11 kann der Speicherbereich DRAM_Extern erst ab Rev. E04 festgelegt werden.

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 Befehl `Exit` beendet die Ausführung des Prozesses und des Programmabschnitts sofort, weitere Programmzeilen im Abschnitt werden nicht mehr ausgeführt. Auch der Abschnitt **`Finish`** wird nicht mehr ausgeführt.

Verwenden Sie im Abschnitt **`Event:`** den Befehl `End`.

Siehe auch

End, ProcessN_Running, Reset_Event, Restart_Process, Start_Process, Start_Process_Delayed, Stop_Process, LowInit:, Init:, Finish:

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 1,3 µs, beim T10 bis zu 0,7 µs, beim T11 bis zu 0,31 µs.

Die Eulersche Zahl e hat ungefähr den Wert 2,7182818.

Siehe auch

LN, Log, Round, Sqrt

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[m] As <arr_type> As FIFO
```

Parameter

`Data_n`	Name des deklarierten **`Data`**-Felds (**`n`**: 1…200).
`<arr_type>`	Festgelegter Datentyp: `Float32`, `Float64`, `Float` (bis T11), `Long`
`m`	Feldgröße (1…2^31): Anzahl der Elemente vom Typ `arr_type` im Feld. CONST LONG Beim Prozessortyp T11 ist der Wertebereich für `m` nur in 4er-Schritten einstellbar: $m = 4 \times a + 3;\ a \geq 0$

Bemerkungen

Sie können nur **`Data`**-Felder als FIFO-Ringspeicher verwenden. Dadurch kann ein FIFO-Feld nicht mehr gleichzeitig als „normales“ Feld verwendet werden. Beachten Sie die Hinweise in Kapitel 4.3.5 auf Seite 127.

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 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 `m` 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 1003 Long-Elementen als Fifo
Dim Data_20[1003] 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 Programmablauf ist das Initialisieren der FIFO-Zeiger sinnvoll, wenn Sie alle im Feld gesammelten Daten (z. B. wegen eines Messfehlers) verwerfen wollen.

Beachten Sie: Bei einem *ADsim*-Modell werden die Zeiger eines FIFO-Felds nur einmal initialisiert, wenn der Prozess das erste Mal geladen und gestartet wird. Eine weitere Initialisierung ist mit **FIFO_Clear** möglich (oder mit einem *ADwin*-Treiber).

Siehe auch

FIFO, FIFO_Empty, FIFO_Full

Beispiel

```
Dim Data_1[20003] 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.

Beachten Sie beim Prozessor T11 die Dimensionierung in 4er-Schritten (siehe Seite 221).

Siehe auch

FIFO, FIFO_Clear, FIFO_Full

Beispiel

```
Dim Data_1[20003] 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…m). Beim Prozessortyp T11 ist der Wertebereich für m nur in Schritten einstellbar: m = 4 × a + 3; a ≥ 0	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.

Beachten Sie beim Prozessor T11 die Dimensionierung in 4er-Schritten (siehe Seite 221).

Siehe auch

FIFO, FIFO_Clear, FIFO_Empty

Beispiel

```
Dim Data_1[20003] 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
    Rem 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 4.1.1 auf Seite 105.

Der Programmabschnitt **Finish:** wird einmalig durchlaufen, sobald der Prozess gestoppt wird.

Wenn der letzte Befehl im Abschnitt **Finish:** abgearbeitet ist, vergeht noch eine bestimmte Zeit, bis der Prozessstatus „gestoppt“ erreicht ist.

Bei dem Prozessortyp T11 kann jeder Programmabschnitt separat einem bestimmten Speicherbereich zugeordnet werden (siehe Kapitel 4.3.2 „Speicherbereiche (T9…T11)“). Der große, aber langsamere Speicherbereich DRAM_Extern sollte nur für zeitunkritische Programmabschnitte genutzt werden; meistens sind das die Abschnitte **LowInit:**, **Init:**, **Finish:**.

Beim Prozessormodul Pro-CPU T11 kann der Speicherbereich DRAM_Extern erst ab Rev. E04 festgelegt werden.

Siehe auch

Dim, LowInit:, Init:, Event:, ProcessN_Running

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, .LIC für T12,
                          '.1.LIC für T12.1

FloToStr(val, string1[])
```

Parameter

val	Zu wandelnder Wert.	FLOAT
string1[]	Erstellte Zeichenfolge im Format: {-}#.######E{-}##	ARRAY STRING

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, Strings

Beispiel

```
Import String.LI9
  'T9: .LI9 / T10: .LIA / T11: .LIB / T12: .LIC / T12.1: .1.LIC

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 = StrLen[text]  'String-Länge = 13
  REM bis T11: die String-Länge steht auch in text[1]
  REM ab T12 ist der Zugriff auf text[1] nicht mehr erlaubt
  'Par_1 = text[1]
  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 T11: **Flo40ToStr** konvertiert eine Fließkomma-Zahl (float) von 40 Bit in eine Zeichenfolge (String).

Syntax

```
Import String.LIB        '.LIB für T11
Flo40ToStr(val, string1[])
```

Parameter

val	Zu wandelnder Wert	FLOAT
string1[]	Erstellte Zeichenfolge im Format: {-}#.########E{-}##	ARRAY STRING

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, Strings

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 = StrLen[text]  'String-Länge = 15
  REM bis T11: die String-Länge steht auch in text[1]
  REM ab T12 ist der Zugriff auf text[1] nicht mehr erlaubt
  'Par_1 = text[1]
  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 (≥1) der Laufvariablen; Vorgabewert: 1	LONG

Bemerkungen

Nur bei Prozessoren bis T11: 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
  Rem (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

`macro_name`	Name der Funktion und Rückgabewert, Datentyp `<var_type>`	
`val_1`, `val_2`	Namen der Übergabeparameter; für Felder ist die Syntax mit Dimensionsklammern erforderlich: `array[]` oder **`Data_n`**`[]`.	FLOAT LONG STRING
`var`	Lokale Variable oder lokales Feld mit dem Datentyp `var_type`. Die lokale Variable wird nur im Makro verwendet.	FLOAT LONG STRING
`<var_type>`	Datentyp der Funktion bzw. des Rückgabewerts: `Float32`, `Float64`, `Float` (bis T11), `Long`; nicht aber `String`	

Bemerkungen

Allgemeine Informationen über Makros finden Sie in Kapitel 4.5.1 auf Seite 138.

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.[1]
  Nach der Abarbeitung der Funktion bleiben die Werte lokaler Variablen erhalten, stehen also beim nächsten Funktionsaufruf wieder zur Verfügung.
  Eine lokale Variable kann den gleichen Namen haben wie eine Variable, die außerhalb der Funktion (z.B. auch in anderen Funktionen) definiert wurde.
- dem Funktionsnamen einen Wert zuweisen müssen, 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 Aufruf und Definition unterschiedlich:

- Funktions-Aufruf *ohne* Dimensions-Klammern:
  `ret_val = name(array_pass)`
- Funktions-Definition *mit* Dimensions-Klammern:
  `Function name(array_def[])`

1. Für fortgeschrittene Nutzer: Lokale Variablen werden im Heap gespeichert, nicht auf dem Stack.

Werte werden an Feldelemente (eines Felds als Übergabeparameter) zugewiesen wie gewöhnlich:

```
array_def[2] = value
```

Wenn Sie einem Übergabeparameter `x` in der Funktion einen Wert zuweisen, darf beim Funktionsaufruf für `x` nur eine Variable oder ein einzelnes Feld-Element angegeben werden, aber keine Konstante. Auf diese Weise können Übergabeparameter auch einen Rückgabewert enthalten.

Übergabeparameter in symbolischen Namen (**#Define**) führen zu Compiler-Problemen. Nutzen Sie deswegen Definitionen, die ohne Übergabeparameter auskommen.

```
Rem Ungeeignet: Übergabeparameter 'array_def' im Define
#Define pointer1 array_def[2]
Rem Geeignete Alternative
#Define pointer2 2
Function name(array_def[]) As Long
  If (pointer1 > 0) Then 'Zeile erzeugt Compiler-Fehler
  If (array_def[pointer2] > 0) Then 'so geht's
  REM ...
EndFunction
```

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 1

```
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:

```
Event:
  x = average(x1, x2, x3)
  DAC(1,average(x1, x2, x3))
```

Beispiel 2

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):

```
  Dim my_array[3] As Long
  Dim Data_1[3] As Long
  Dim x As Long
Event:
  x = average_array(my_array)
  DAC(1,average_array(Data_1))
```

Beim Aufruf können Sie ein globales Feld (wie `Data_1`) oder ein lokales Feld (wie `my_array`) angeben. Tragen Sie nur den Feldnamen ein, ohne Elementnummer und eckige Klammern.

Prozessor	Priorität	
	Hoch	Niedrig
T9	25 ns	100 µs
T10	25 ns	50 µs
T11	$3{,}\overline{3}$ ns	$3{,}\overline{3}$ ns = $0{,}00\overline{3}$ µs
T12	1,0 ns	1,0 ns = = 0,001 µs

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 `<`, `<=`, `>`, `>=`, `=`, `<>`, `And` und `Or`. Wenn die Bedingung „Wahr“ ist, werden die Anweisungen nach `Then` ausgeführt.	LOGIC
`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 <SYSPAR> = value Then
  …                    'Anweisungsblock
{#Else                 'Der else-Teil ist optional
  …                    'Anweisungsblock}
#EndIf
```

Parameter

LOGIC

`<SYSPAR>` = `value`

Boolesche Bedingung (ohne Klammern und Hochkommas) mit dem Gleich-Operator =.

Wenn die Bedingung erfüllt ist, werden die auf **Then** folgenden Anweisungen ausgeführt.

Die Systemparameter `<SYSPAR>` und die zugehörigen Werte `value` sind in der Tabelle unten aufgeführt.

<SYSPAR>	value	Bedeutung
ADWIN_SYSTEM	**ADWIN_CARD** **ADWIN_L16** **ADWIN_X** **ADWIN_GOLD** **ADWIN_GOLDII** **ADWIN_PRO** **ADWIN_PROII**	Einstellung `System` im Fenster `Compiler Options`. Der Wert **ADWIN_PRO** gilt sowohl für *ADwin-Pro I* als auch für *ADwin-Pro II*.
PROCESSOR	**T9** **T10** **T11** **T12** **T121**	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`.

Die bedingte Kompilierung funktioniert nicht für die Definition von `#Define` und von Makros (`Sub`, `Function`).

Beim Systemparameter `PROCESSOR` können mehrere Prozessortypen gleichzeitig angegeben werden (durch Komma getrennt):
`#If Processor = T11, T12, T121 Then`

Bei einem Kommandozeilen-Aufruf des Compilers (siehe Seite A-9) beziehen sich die Systemparameter `ADWIN_SYSTEM` und `PROCESSOR` auf die beim Aufruf übergebenen Parameter `/Sx` und `/Px`.

Siehe auch

< = > (Vergleich), If … Then … {Else … } EndIf

Beispiel

```
Rem niederpriores Processdelay auf 800µs setzen
#If Processor = T12 Then
  Rem T12: 800µs = 800000 x 1ns
  Processdelay = 800000
#Else
#If Processor = T11 Then
  Rem T11: 800µs = 240000 x 3,3ns
  Processdelay = 240000
#Else
#If Processor = T10 Then
  Rem T10: 800µs = 16 x 50µs
  Processdelay = 16
#Else
  Rem T9: 800µs = 8 x 100µs (auch andere CPUs)
  Processdelay = 8
#EndIf
#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 *ohne* Hochkommas. Die Dateiendung ist `.LI9` für T9, `.LIA` für T10, `.LIB` für T11, `.LIC` für T12, `.1.LIC` für T12.1.	CONST STRING
`path`	Vollständiger Pfad mit Laufwerk oder relativer Pfad der Library-Datei; *ohne* Hochkommas	CONST STRING

Bemerkungen

Allgemeine Informationen über Bibliotheken finden Sie im Abschnitt Kapitel 4.5.3 auf Seite 139.

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.

Wenn Sie keinen Pfadnamen angeben, wird die Datei nur im Standard-Verzeichnis (siehe Menü Options `Directories`, Seite 65) gesucht. Verwenden Sie bei der Pfadangabe den Backslash "\", um Verzeichnisnamen voneinander zu trennen.

Das Basisverzeichnis für relative Pfadangaben ist – wenn der Quelltext Teil einer Projektdatei ist – das Verzeichnis der Projektdatei, anderenfalls das Verzeichnis der Quelltextdatei.

Die folgenden Bibliotheken sind im Lieferumfang von *ADbasic* enthalten:

`String.liX`	Befehle für String-Operationen.

`FFT.liX`	Befehle für die schnelle Fourier-Transformation.
`math.liX`	Mathematik-Befehle.

Siehe auch

#Include, Lib_Function … Lib_EndFunction, Lib_Sub … Lib_EndSub

Beispiel

```
Rem String-Library für den Prozessor T9 importieren
Import String.LI9
  'T9: .LI9 / T10: .LIA / T11: .LIB / T12: .LIC / T12.1: .1.LIC
Rem Benutzerdefinierte Library für T10 importieren
Import C:\MyFiles\ADwinLibs\dig2volt.LIA
```

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**(var) führt zum gleichen Ergebnis wie die Programmzeile: var = var + 1. Allerdings 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
  Rem Nach 1000 Messungen das Programm beenden
  If (index > 1000) Then End
```

#Include

#Include bindet den vollständigen Inhalt einer Include-Datei in den Quelltext ein.

Syntax

```
#Include {path}filename
```

Parameter

filename	Name der einzubindenden Datei (mit Endung „.Inc“) ohne Hochkommas	CONST STRING
path	Vollständiger Pfad mit Laufwerk oder relativer Pfad.	CONST STRING

Bemerkungen

Allgemeine Informationen über Include-Dateien finden Sie in Kapitel 4.5.2 auf Seite 139.

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.

Wenn Sie keinen Pfadnamen angeben, wird die Datei nur im Standard-Verzeichnis (siehe Menü Options Directories, Seite 65) gesucht. Verwenden Sie bei der Pfadangabe den Backslash "\", um Verzeichnisnamen voneinander zu trennen.

Das Basisverzeichnis für relative Pfadangaben ist – wenn der Quelltext Teil einer Projekts ist – das Verzeichnis der Projektdatei, anderenfalls das Verzeichnis der Quelltextdatei.

Um eine der im Lieferumfang von *ADbasic* enthaltenen Include-Dateien – sie enthalten Befehle zur Ansteuerung der Hardware-I/Os – einzubinden, geben Sie die ersten Zeichen des Befehls **#Include** ein, drücken [CTRL]+[SPACE] und wählen die passende Include-Datei aus der Liste. Alternativ verwenden Sie einen der Textbausteine aus der Gruppe „Hardware“.

Zur Syntaxhervorhebung von Befehlen aus Include-Dateien siehe Include-Dateien.

Folgende Include-Dateien gehören zum Lieferumfang von *ADbasic* und enthalten die Befehle zur Ansteuerung der Hardware-I/Os:

`ADwL16.inc`	*ADwin-light-16*; Textbaustein einfügen mit IL[TAB].
`ADwin-X.inc`	*ADwin-X-A20*; Textbaustein einfügen mit IX[TAB].
`ADwGCnt.inc` `ADwGCAN.inc`	*ADwin-Gold*: Befehle für die Erweiterungen *CO1* und *CAN*; Textbaustein einfügen mit IG[TAB].
`ADwinGoldII.inc`	*ADwin-Gold II*: Alle Befehle; Textbaustein einfügen mit IG2[TAB].
`ADwinPro_All.inc`	*ADwin-Pro*, *ADwin-Pro II*: Alle Befehle; Textbaustein einfügen mit IP[TAB].
`Math.inc`	Mathematik-Befehle.
`ADwin_Utils.inc`	Ab T12: Zusatzbefehle, z.B. variabler Zugriff auf DATA-Felder; Textbaustein einfügen mit IU[TAB].

Siehe auch

#Define, Import, Function … EndFunction, Sub … EndSub

Beispiel

```
Rem Datei im angegebenen Verzeichnis suchen
#Include C:\Test\demofunc.Inc

Rem Datei im Standard-Verzeichnis suchen
#Include demofunc.Inc

Rem Relative Pfadangabe.
Rem Der Pfad ist relativ zum Verzeichnis der Projektdatei,
Rem wenn der Quelltext zu einem Projekt gehört.
Rem Gehört der Quelltext nicht zu einem Projekt, ist
Rem der Pfad relativ zum Verzeichnis der Quelltextdatei.
#Include .\demofunc.Inc
```

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 4.1.1 auf Seite 105.

Der Programmabschnitt **Init:** wird einmal durchlaufen, sobald der Prozess gestartet wird und der Programmabschnitt **LowInit:** (falls vorhanden) beendet ist. Die Zeit zwischen dem letzten Befehl im Abschnitt **Init:** und dem Beginn des Abschnitts **Event:** beträgt 1…2 × **Processdelay**.

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 4.3.2 „Speicherbereiche (T9…T11)“). Der große, aber langsamere Speicherbereich DRAM_Extern sollte nur für zeitunkritische Programmabschnitte genutzt werden; meistens sind das die Abschnitte LowInit:, Init:, Finish:.

Beim Prozessormodul Pro-CPU T11 kann der Speicherbereich DRAM_Extern erst ab Rev. E04 festgelegt werden.

Siehe auch

Dim, LowInit:, Event:, Finish:, Processdelay

Beispiel

```
Dim val_1 As Float

Init:
  val_1 = -5.3
```

IO_Sleep

IO_Sleep lässt Befehle für Ein- und Ausgänge für eine bestimmte Zeit warten.

Syntax

```
IO_Sleep(val)
```

Parameter

val	Anzahl (12, 14 … 715827879 ≈ 2^{31} / 3) der zu wartenden Zeiteinheiten in 10ns. Nur gerade Zahlen sind möglich. Eine ungerade Anzahl von Zeiteinheiten wird automatisch um 1 verringert.	LONG

Bemerkungen

Alternativ gibt es die Anweisung **CPU_Sleep** (siehe auch Kapitel 5.2.4 „Wartezeit genau einstellen“).

Die Anweisung **IO_Sleep** wird eingesetzt, wenn zwischen 2 Zugriffen auf Ein-/Ausgänge eine definierte Zeit gewartet werden muss. Die Gesamtzeit ist die Summe aus der Ausführungszeit des I/O-Zugriffs und der Wartezeit durch **IO_Sleep**.

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

Bei hochprioren Prozessen können ungeeignete Argumente zu einem Abbruch der Kommunikation führen:

- Stellen Sie sicher, dass das Argument immer einen Wert größer 12 hat, weil sonst extrem lange Wartezeiten entstehen können.
- Verwenden Sie sehr große Werte mit Bedacht, weil dann die Kommunikation mit dem PC sehr lange unterbrochen ist (Gefahr eines Timeout).

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.
- Nur Gold II: 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

Beispiel

```
#Include ADwinGoldII.inc

Event:
  Rem Mux 1: Kanal und Verstärkung setzen
  Set_Mux1(11010b)
  Rem 2 µs (=200*10ns) warten = max. MUX-Einschwingzeit
  IO_Sleep(200)
  Rem AD-Wandlung ADC1 starten
  Start_Conv(1)
  Rem …
```

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>}
  …                'Anweisungsblock
  lib_name = …
Lib_EndFunction
```

Syntax der Übergabeparameter <lib_par>:
<by_type> var_name As <var_type> {At <mem_type>}

Parameter

lib_name	Name der Library-Funktion und des Rückgabewerts; Datentyp <fct_type>.
<fct_type>	Datentyp: Float32, Float64, Float (bis T11), Long.
var_name	Name eines Übergabeparameters innerhalb der Library-Funktion; für Felder ist die Syntax mit Dimensionsklammern erforderlich: array[] oder **Data_n**[].
<by_type>	Methode zur Übergabe der Parameter: ByRef: Zeiger auf Variable oder Feld übergeben. ByVal: Wert übergeben.
<var_type>	Datentyp: Float32, Float64, Float (bis T11), Long, String.
<mem_type>	Sinnvoll nur für Prozessor T10: Speicher, in dem die Variablen abgelegt werden; nur gemeinsam mit Feldern verfügbar: DRAM_Extern: externer Datenspeicher. DM_Local:interner Datenspeicher.
var	Lokale Variable oder lokales Feld mit dem Datentyp var_type. Die lokale Variable wird nur in der Library verwendet.

Bemerkungen

Allgemeine Informationen über Library-Dateien finden Sie in Kapitel 4.5.3 auf Seite 139.

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 eindimensionale Felder behandelt (siehe auch Kapitel 4.3.4 auf Seite 126).
- dem Funktionsnamen einen Wert zuweisen, damit dieser zum Rückgabewert an die aufrufende Stelle wird.

Innerhalb einer Library-Funktion ist *nicht* erlaubt:
- Prozess-Abschnitte wie **`LowInit:`**, **`Init:`**, **`Event:`**, oder **`Finish:`** definieren.
  Library-Funktion oder -Unterprogramm aus der gleichen Library-Datei aufrufen (auch rekursiv).
  Gegebenenfalls müssen Sie die aufzurufende Funktion in eine neue Library-Datei auslagern und von dort importieren.
- Anweisung `SelectCase` benutzen.
- Symbolische Namen mit **`#Define`** deklarieren.

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.

Prüfen Sie beim Prozessor T12/T12.1 im Zusammenhang mit Libraries, ob die Größe des Stapelspeichers ausreicht. Eine ungenügende Stack-Größe kann zu Datenverlust oder Programmabbruch führen Näheres finden Sie im Fenster Stack Test Analysis (Seite 87).

Siehe auch

Lib_Sub … Lib_EndSub, Import, Function … EndFunction, Sub … EndSub

Beispiel

```
Rem ---------- Mittelwertbildung ----------
Rem Binärdatei speichern als MEAN.LI9
Rem (Endung je nach Prozessor)
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
```

Library-Aufruf

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 'siehe oben
#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

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**[].
<by_type>	Methode zur Übergabe eines Parameters: ByRef: Zeiger auf Variable oder Feld übergeben. ByVal: Wert übergeben.
<var_type>	Datentyp: Float, Long, String.
<mem_type>	Sinnvoll nur für Prozessor T10: Speicher, in dem die Variablen abgelegt werden; nur gemeinsam mit Feldern verfügbar: DRAM_Extern: externer Datenspeicher. DM_Local:interner Datenspeicher.
var	Lokale Variable oder lokales Feld mit dem Datentyp var_type. Die lokale Variable wird nur in der Library verwendet.

Bemerkungen

Allgemeine Informationen über Bibliotheken (Libraries) finden Sie in Kapitel 4.5.3 auf Seite 139.

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 überge-

ben, diese werden aber innerhalb der Funktion wie eindimensionale Felder behandelt (siehe auch Kapitel 4.3.4 auf Seite 126).

Innerhalb eines Library-Unterprogramms ist *nicht* erlaubt:

- Prozess-Abschnitte wie **`LowInit:`**, **`Init:`**, **`Event:`**, oder **`Finish:`** definieren.
- 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` benutzen.
- Symbolische Namen mit **`#Define`** deklarieren.

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.

Prüfen Sie beim Prozessor T12/T12.1 im Zusammenhang mit Libraries, ob die Größe des Stapelspeichers ausreicht. Eine ungenügende

Stack-Größe kann zu Datenverlust oder Programmabbruch führen. Näheres finden Sie im Fenster Stack Test Analysis (Seite 87).

Siehe auch

Lib_Function … Lib_EndFunction, Import, Function … EndFunction, Sub … EndSub

Beispiel

```
Rem Binärdatei speichern als DIG2VOLT.LI9
Rem (Endung je nach Prozessor)
Rem Messwertumrechnung von Digit(0…65535)
Rem 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
```

Library-Aufruf

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 Sample
#Define gain 1          'Verstärkungsfaktor PGA
#Define samples Data_1 'Speicher für Messwerte
#Define scaled Data_2 'Speicher konvertierte Messwerte
#Define length 1000     'Feldlänge
Dim samples[length] As Long'Quell-Feld
Dim scaled[length] As Long'Ziel-Feld
Dim i As Long           'Zählvariable

Init:
  i = 1                 'Startwert Zählvariable
  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

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

val	Argument	FLOAT
ret_val	natürlicher Logarithmus des Arguments	FLOAT

Bemerkungen

Die Ausführungszeit des Befehls beträgt beim T9 bis zu 1,45 µs, beim T10 bis zu 0,7 µs, beim T11 bis zu 0,37 µs.

Siehe auch

Log, Exp, Round, Sqrt

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, .LIC für T12,
                          '.1.LIC für T12.1

LngToStr(value, string1[])
```

Parameter

value	Wert, der gewandelt werden soll	LONG
string1[]	Ergebnis-Zeichenfolge im Format: {-}##########	ARRAY STRING

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 4.3.6 auf Seite 129.

Siehe auch

"" String, + (String-Addition), Asc, Chr, FloToStr, Flo40ToStr, StrComp, StrLeft, StrLen, StrMid, StrRight, ValF, ValI, Strings

Beispiel

```
Import String.LI9
  'T9: .LI9 / T10: .LIA / T11: .LIB / T12: .LIC / T12.1: .1.LIC
Dim digits[11] As String 'Ergebnis-String
Dim a As Long

Init:
  a = -1234567890

Event:
  LngToStr(a,digits)     'zum String konvertieren
  Par_1 = StrLen[digits] 'String-Länge = 11
  REM bis T11: die String-Länge steht auch in digits[1]
  REM ab T12 ist der Zugriff auf digits[1] nicht mehr erlaubt
  'Par_1 = digits[1]
  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, Round, Sqrt

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 4.1.1 auf Seite 105.

Der Abschnitt **LowInit:** darf nur in Prozessen mit hoher Priorität verwendet werden.

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 seinen niedriger Priorität) unterbrochen werden kann.

Bei dem Prozessortyp T11 kann jeder Programmabschnitt separat einem bestimmten Speicherbereich zugeordnet werden (siehe Kapitel 4.3.2 „Speicherbereiche (T9…T11)“). Der große, aber langsamere Speicherbereich DRAM_Extern sollte nur für zeitunkritische Programmabschnitte genutzt werden; meistens sind das die Abschnitte **LowInit:**, **Init:**, **Finish:**.

Beim Prozessormodul Pro-CPU T11 kann der Speicherbereich DRAM_Extern erst ab Rev. E04 festgelegt werden.

Siehe auch

Dim, Init:, Event:, Finish:

Beispiel

```
Dim val_1 As Float

LowInit:
  val_1 = -5.3
```

Parameter

Registernummer

Max_Float

Max_Float liefert den größeren von 2 Float-Werten.

Syntax

```
ret_val = Max_Float(val1, val2)
```

Parameter

val_1	Vergleichswert 1	FLOAT
val_2	Vergleichswert 2	FLOAT
ret_val	Der größere der beiden Vergleichswerte.	FLOAT

Bemerkungen

- / -

Siehe auch

AbsF, Min_Float, Max_Long, Min_Long, ValF

Beispiel

```
Event:
  FPar_10 = Max_Float(FPar_1,FPar_2)
```

Min_Float

Min_Float liefert den kleineren von 2 Float-Werten.

Syntax

```
ret_val = Min_Float(val1, val2)
```

Parameter

val_1	Vergleichswert 1	FLOAT
val_2	Vergleichswert 2	FLOAT
ret_val	Der kleinere der beiden Vergleichswerte.	FLOAT

Bemerkungen

- / -

Siehe auch

AbsF, Max_Float, Max_Long, Min_Long, ValF

Beispiel

```
Event:
  FPar_10 = Min_Float(FPar_1,FPar_2)
```

Max_Long

Max_Long liefert den größeren von 2 ganzzahligen Werten.

Syntax

```
ret_val = Max_Long(val1, val2)
```

Parameter

val_1	Vergleichswert 1	LONG
val_2	Vergleichswert 2	LONG
ret_val	Der größere der beiden Vergleichswerte.	LONG

Bemerkungen

- / -

Siehe auch

AbsI, Min_Float, Max_Float, Min_Long, ValI

Beispiel

```
Event:
  Par_10 = Max_Long(Par_1,Par_2)
```

Min_Long

Min_Long liefert den kleineren von 2 ganzzahligen Werten.

Syntax

```
ret_val = Min_Long(val1, val2)
```

Parameter

val_1	Vergleichswert 1	LONG
val_2	Vergleichswert 2	LONG
ret_val	Der kleinere der beiden Vergleichswerte.	LONG

Bemerkungen

- / -

Siehe auch

AbsI, Max_Long, Min_Float, Max_Float, ValI

Beispiel

```
Event:
  Par_10 = Min_Long(Par_1,Par_2)
```

MemCpy

Ab Prozessor T11: **MemCpy** kopiert eine bestimmte Anzahl von Feldelementen aus einem Quellfeld in ein Zielfeld.

Syntax

```
MemCpy(source[i1], dest[i2], count)
```

Parameter

source[]	Name des Quellfelds.	ARRAY LONG FLOAT STRING
i1	Index (≥1) des ersten kopierten Feldelements.	LONG
dest[]	Name des Zielfelds; der Datentyp muss der gleiche sein wie beim Quellfeld.	ARRAY LONG FLOAT STRING
i2	Index (≥1) des ersten Feldelements, das beschrieben wird.	LONG
count	Anzahl (≥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 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.

Beim Prozessor T11 kann der Zugriff auf zu große oder zu kleine Elementindexe des Zielfelds mit dem Debug-Modus überwacht werden (siehe Option Debug mode auf Seite 69). 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	25,0 ns
T10	12,5 ns
T11	$3,\overline{3}$ ns
T12	1,0 ns
T12.1	1,5 ns

Sie können mit dieser Anweisung erforderliche Wartezeiten überbrükken (beispielsweise nach **Set_Mux**), wenn es keine sinnvolle andere Verwendung der Prozessorzeit gibt. Für den Prozessor T11 beachten Sie bitte Kapitel 6.2.2 auf Seite 165: „Zeitlich exakter Aufruf von Prozesszyklen“.

Siehe auch

CPU_Sleep, P1_Sleep, P2_Sleep, Sleep

Not

Not invertiert die Bits eines Werts.

Syntax

```
ret_val = Not(val)
```

Parameter

val	zu invertierender Wert (kein logischer Ausdruck)	LONG
ret_val	invertiertes Argument	LONG

Bemerkungen

Verwenden Sie die Funktion möglichst nur mit ganzzahligen Werten (Typ Long).
Fließkomma-Werte (Typ Float) werden vor der Invertierung in ganzzahlige Werte konvertiert: Die Nachkommastellen werden abgeschnitten und ggf. der Wertebereich angepasst, so dass das Berechnungsergebnis beeinflusst wird.

Not arbeitet nur bitweise. Daher können Sie mit **Not** 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
Rem -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	LONG
`expr1`, `expr2`	Boolescher Ausdruck mit dem Wert „wahr“ oder „falsch“	LOGIC

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 bitweisen Verknüpfung ganzzahliger Werte sind keine Klammern erforderlich.

Siehe auch

And, If … Then … {Else … } EndIf, Not, XOr

Beispiel

```
Dim val_1, val_2, val3 As Long
Dim x, val4 As Long

Init:
  Rem Bitweise Operator
  val_1 = 0100b
  val_2 = 0110b
  val3 = val_1 Or val_2 'Ergebnis: val3 = 0110b

  Rem Boolescher Operator
  x = 15

Event:
  If ((x < 3) Or (x > 9)) Then
    val4 = 1
  Else
    val4 = 0
  EndIf                    'Ergebnis: val4 = 1
```

P1_Sleep

Ab T11: **P1_Sleep** lässt den Bus des Pro I-Systems für eine bestimmte Zeit warten.

Syntax

P1_Sleep(val)

Parameter

val	Anzahl der zu wartenden Zeiteinheiten in 10ns: bei Konstanten: 7…715827879 (≈ 2^{30} / 1,5) bei Variablen: 9…715827879	LONG

Bemerkungen

Alternativ gibt es die Anweisungen **CPU_Sleep** und **P2_Sleep** (siehe auch Kapitel 5.2.4 „Wartezeit genau einstellen“). Verwenden Sie bei Prozessoren bis T10 die Anweisung **Sleep**.

Beim Prozessor T12 hat **P1_Sleep** nur eine Wirkung auf die TTL-Signaleingänge Dig I/O 0 und Dig I/O 1.

Die Anweisung **P1_Sleep** wird eingesetzt, wenn zwischen 2 Zugriffen auf Module am Pro I-Bus (oder die TTL-Signaleingänge Dig I/O 0 und Dig I/O 1) 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 keine Werte, die kleiner sind als die angegebenen Mindestwerte.

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

```
#Include ADwinPro_All.inc

Event:
  Set_Mux(1,0)          'Multiplexer des AIn-Moduls 1 setzen
  P1_Sleep(250)         '2.5 µs (=250*10ns) warten
                        '= Einschwingzeit des MUX
  Start_Conv(1)         'Konvertierung starten
  Rem …
```

P2_Sleep

Nur T11 und T12: **P2_Sleep** lässt den Bus des Pro II-Systems für eine bestimmte Zeit warten.

Syntax

P2_Sleep(val)

Parameter

val	Gerade Anzahl (14…715827878 ≈ 2^{30} / 1,5) der zu wartenden Zeiteinheiten in 10 ns. Eine ungerade Anzahl ist nicht erlaubt.	LONG

Bemerkungen

Alternativ gibt es die Anweisungen **CPU_Sleep** und **P1_Sleep** (siehe auch Kapitel 5.2.4 „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 **Processdelay** eingestellte Zykluszeit.

⚠ 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.

⚠ 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.
- nur T11: 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

```
#Include ADwinPro_All.inc

Event:
  P2_Set_Mux(1,0)        'Multiplexer setzen
  P2_Sleep(210)          '2.1 µs (=210*10ns) warten
                         '= Einschwingzeit des MUX
  P2_Start_Conv(1)       'Konvertierung starten
  Rem …
```

Peek

Peek liest den Inhalt einer bestimmten Speicherstelle auf dem *ADwin*-System.

Syntax

```
ret_val = Peek(addr)
```

Parameter

addr	Adresse der auszulesenden Speicherstelle.	LONG
ret_val	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 **20400030h**, das ist bei *ADwin-Gold* das Datenregister des ADC1 und enthält den gewandelten Analogwert.

```
Rem Speicherstellen eines ADwin-Gold auslesen
val = Peek(20400030h)
```

Länge und Datentyp eines Felds **Data_n** lesen.

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, Poke, Read_Timer

Beispiel

```
Rem Speicherstellen eines ADwin-Gold verändern
Rem DAC-Register 1 beschreiben: 3072
Rem (=+5V im ±10V Bereich)
Poke(20400050h, 3072)
Rem Ausgabe auf allen DAC starten
Poke(20400010h, 011b)
Rem Setze Ausgänge DIO18…DIO21 auf High
Poke(204000C0h, 111100b)
```

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 (≥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. Mit **Calc_Processdelay** berechnen Sie das Processdelay zu einer Frequenz.

Prozessor	Priorität	
	Hoch	Niedrig
T9	25 ns	100 µs
T10	25 ns	50 µs
T11	$3{,}\overline{3}$ ns	$3{,}\overline{3}$ ns = $0{,}00\overline{3}$ µs
T12	1,0 ns	1,0 ns
T12.1	1,5 ns	1,5 ns

Wählen Sie bei hochprioren Prozessen eine ausreichend großes Processdelay, um eine Überlastung des *ADwin*-Systems zu vermeiden (siehe auch Kapitel 6.1.4 auf Seite 162). Als Faustregel sollte die Aus-

lastung 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 250 ms 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.

Eine Änderung des **`Processdelay`** 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.

Siehe auch

Read_Timer, Kapitel 6.2.1 „Processdelay“

Beispiel

```
Init:
  Rem Zykluszeit einstellen
  Processdelay = 40000
  Rem Für T9 und T10, hohe Priorität: 1 ms
  Rem Für T11, hohe/niedrige: 0.133 ms

  Rem Zykluszeit 10000 Hertz einstellen
  Processdelay= Calc_Processdelay(10000)
```

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
  Rem Für T9 und T10, hohe Priorität: 89,5 Min.
  Rem Für T11, hohe+niedrige Priorität: 12 Min.
  If (Par_1 = 100) Then
    Par_1 = 0
    Rem Programm ausführen
    …
  EndIf
```

Process_Error

`Process_Error` gibt den (in der Regel) zuletzt aufgetretenen Fehler des aktuellen Prozesses zurück.

Syntax

```
ret_val = Process_Error
```

Parameter

`ret_val` Nummer des zuletzt aufgetretenen Fehlers im Prozess. `LONG`

Einige Fehlernummern:

0: kein Fehler
1: Division durch Null
2: Wurzel aus negativer Zahl
10: Zugriff auf eine zu große Elementnummer eines globalen Felds.
11: Zugriff auf eine zu kleine Elementnummer (≤ 0) eines globalen Felds.
12: Zugriff auf eine zu große Elementnummer eines lokalen Felds.
13: Zugriff auf eine zu kleine Elementnummer (≤ 0) eines lokalen Felds.
20: Nur Transputer: Timing-Fehler, d.h. Angabe eines zu kleinen Processdelay.
30: FIFO-Index ist kein FIFO.

Bemerkungen

Der Rückgabewert der Systemvariablen ist nur definiert, wenn der Debug-Modus eingeschaltet ist (siehe Option Debug mode, Seite 69). Die Variable kann nur gelesen werden.

Siehe auch

ProcessN_Running, Start_Process, Stop_Process

Beispiel

```
Event:
  Par_10 = Sqrt(Par_12)
  Rem zuletzt aufgetretene Fehlernummer im Prozess lesen
  Par_2 = Process_Error
```

ProcessN_Running

Die Systemvariable **ProcessN_Running** gibt den aktuellen Status eines bestimmten Prozesses zurück.

Syntax

```
ret_val = ProcessN_Running
```

Parameter

n	Prozessnummer n (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 = Process2_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

Der Zählerstand kann nur gelesen werden.

Es gibt im *ADwin*-System 2 Zähler (32 Bit), die in unterschiedlichen Zeiteinheiten zählen:

Prozesspriorität	T9	T10	T11	T12	T12.1
hoch	25 ns	25 ns	$3,\overline{3}$ ns	1,0 ns	1,5 ns
niedrig	100 µs	50 µs	$3,\overline{3}$ ns	1,0 ns	1,5 ns

Sie können eine Zeitdifferenz aus der Differenz von 2 Zählerständen ermitteln. Die Zeitdifferenz kann mit **Calc_TicksToNs** in Nanosekunden umgerechnet werden. Beachten Sie dabei bitte, dass ein gelesener Zählerstand nach einer bestimmten Zeit wieder erneut erreicht wird. Dieser Zählerüberlauf hängt von den oben angegeben Zeiteinheiten ab:

Prozesspriorität	T9	T10	T11	T12	T12.1
hoch	107,4 s	107,4 s	14,3 s	4,3 s	6,4 s
niedrig	119,3 h	59,7 h	14,3 s	4,3 s	6,4 s

Beachten Sie beim Prozessor T12/T12.1, dass die Zeitmessung innerhalb eines Programmabschnitts besonderen Regeln unterliegt. Näheres ist unter „Besonderheiten beim T12/T12.1“ beschrieben.

Siehe auch

Processdelay, Read_Timer_Sync

Beispiel

```
Dim timervalue As Long
Event:
  timervalue = Read_Timer()
```

Siehe auch Beispiel seconds_timer.bas im Ordner C:\ADwin\ADbasic\samples_ADwin

Read_Timer_Sync

Nur ab Prozessor T12: **Read_Timer_Sync** gibt den aktuellen Zählerstand des *ADwin*-Systems zurück, nachdem alle Hardware- und Speicher-Zugriffe abgeschlossen sind.

Syntax

```
#Include ADwinPro_All.inc 'bzw. ADwin-X.inc
ret_val = Read_Timer_Sync()
```

Parameter

ret_val	Aktueller Zählerstand in Einheiten von 1 ns.	LONG

Bemerkungen

Der Zählerstand kann nur gelesen werden.

Sie können eine Zeitdifferenz aus der Differenz von 2 Zählerständen ermitteln. Beachten Sie dabei bitte, dass ein gelesener Zählerstand nach etwa 4,3 s wieder erneut erreicht wird (Zählerüberlauf).

Im Unterschied zu **Read_Timer** beendet **Read_Timer_Sync** alle bis dahin noch aktiven Lese- und Schreibzugriffe auf *ADwin*-Hardware und auf den Speicher, bevor der Zählerstand gelesen wird.

Beachten Sie, dass die Zeitmessung innerhalb eines Programmabschnitts besonderen Regeln unterliegt. Näheres ist unter „Bearbeitungszeit messen“ beschrieben, insbesondere unter „Besonderheiten beim T12/T12.1“.

Siehe auch

Processdelay, Read_Timer

Beispiel

```
Dim timervalue As Long
Event:
  timervalue = Read_Timer_Sync()
```

Siehe auch Beispiel seconds_timer.bas im Ordner C:\ADwin\ADbasic\samples_ADwin

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	*ADbasic*-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 dem Zeichen `'` 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.

Sie können die Position eines Kommentars hinter einer Anweisung festlegen; mehr dazu finden Sie unter Kommentare positionieren (Seite 22).

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 169.

Siehe auch

End, Exit, ProcessN_Running, Start_Process, Stop_Process

Beispiel

```
Init:
  Rem Initialisierung
  Rem ...
  Reset_Event 'Bisherige Event-Signale löschen

Event:
  Rem Hauptprogramm startet, sobald ein
  Rem Event-Signal auftritt
  Rem ...
```

Restart_Process

Nur ab 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:
  Rem …

Finish:
  Rem …
  If (cond = 2) Then
    Rem Wenn Bedingung erfüllt, Prozess erneut starten
    Restart_Process
  EndIf
```

Round

Round gibt die nächstliegende ganze Zahl zu einem Wert vom Typ Float zurück.

Syntax

```
ret_val = Round(value)
```

Parameter

value	Variable oder Berechnungsausdruck.	FLOAT
ret_val	Gerundeter, ganzzahliger Wert.	LONG

Bemerkungen

Verwenden Sie den Befehl Round, bevor Sie einen Fließkomma-Wert einer Variablen vom Datentyp Long zuweisen. Anderenfalls werden die Nachkommastellen des Fließkomma-Werts abgeschnitten. Dies ist insbesondere bei Fließkomma-Werten wichtig, die knapp unterhalb eines ganzzahligen Werts liegen, z.B. bei dem Wert 0.99999.

Die Rundung wird nach der Norm IEEE 754 durchgeführt, also zum nächstgelegenen ganzzahligen Wert. Liegt der Fließkomma-Wert in der Mitte zwischen zwei ganzen Zahlen, ist das Ergebnis die nächstliegende gerade Zahl.

Siehe auch

Kapitel 4.4.2 „Typkonvertierung“, AbsF

Beispiel

```
Init:
  Par_2 = Round(1.5)     'result = 2
  Par_3 = Round(2.5)     'result = 2

Event:
  Inc(FPar_1)
  Par_1 = Round(FPar_1 / 12.2E05)
```

SelectCase

Die Kontrollstruktur `SelectCase` 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; ab T12: alle Long-Werte erlaubt), bei dem der nachfolgende Anweisungsblock ausgeführt wird.	CONST LONG

Bemerkungen

Bis T11 gilt: In einer Library-Funktion oder einem Library-Unterprogramm sowie im Abschnitt **`Lowinit:`** 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.

Bis T11 gilt: Die Struktur verwendet intern eine Sprungtabelle im Datenspeicher (DM), deren Speicherbedarf sich nach der größten angegeben `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] $\times$ 4

Beispielsweise wäre der Speicherbedarf bei `Case 200`:
(200 + 1) $\times$ 4 = 804 Bytes; der max. Bedarf beträgt genau 1 KiB.

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?
      Rem digitale Eingänge einlesen
      Par_2 = Digin_Word()
    CaseElse              'Par_1: sonstige Werte
      Rem Wert von Par_10 an digitale Ausgänge ausgeben
      Digout_Word(Par_10)
  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 (-π…π)	FLOAT
ret_val	Sinus des Winkels (-1…1)	FLOAT

Bemerkungen

Wenn Sie für den Winkel Werte außerhalb von -π…+π 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, ArcTan2, Round

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 definierte Zeit warten.

Syntax

Sleep(val)

Parameter

val	Anzahl (≥1) der zu wartenden Zeiteinheiten in 100 ns.	LONG

Bemerkungen

Bei Prozessoren ab T11 muss **Sleep** ersetzt werden durch einen der Befehle **CPU_Sleep**, **P1_Sleep** oder **P2_Sleep** (siehe auch Kapitel 5.2.4 „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
  Rem …
```

Sqrt

Sqrt gibt die quadratische Wurzel eines Werts zurück.

Syntax

```
ret_val = Sqrt(val)
```

Parameter

val	Argument (≥0).	FLOAT
ret_val	Quadratwurzel des Arguments.	FLOAT

Bemerkungen

Die Ausführungszeit des Befehls beträgt beim T9 bis zu 0,9µs, beim T10 bis zu 0,45µs, beim T11 bis zu 0,26µs.

Für ungültige Argumente ist der Rückgabewert undefiniert.

Siehe auch

Exp, LN, Log, Round

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:**.

Start_Process übergibt den Start-Befehl zwar sofort ans Betriebssystem, wartet aber nicht, bis der Prozess tatsächlich gestartet ist. Falls erforderlich, können Sie mit **ProcessN_Running** abfragen, ob der gestartete Prozess bereits läuft. Die Zeit bis zum Start des Prozesses lässt sich nicht genauer vorherbestimmen.
Wird aus einem niederprioren ein hochpriorer Prozess gestartet, kann der neu gestartete Prozess den niederprioren Prozess unterbrechen.

Siehe auch

End, Exit, Restart_Process, Start_Process_Delayed, Stop_Process, ProcessN_Running

Beispiel

```
Event:
  If (ADC(1) > 3072) Then 'Schwellwert überschritten?
    Rem Wenn Prozess 2 nicht mehr aktiv ist: neu starten
    If (Process2_Running = 0) Then
     Start_Process(2)    'Messprozess 2 starten
     Do                  'Prozessbeginn abwarten
     Until(Process2_Running = 1)
    EndIf
    End
  EndIf
```

Start_Process_Delayed

Ab Prozessor T11: **Start_Process_Delayed** startet einen bestimmten Prozess (Abschnitt **Event:**) mit einer definierten Verzögerung.

Syntax

```
Start_Process_Delayed(processno, delay)
```

Parameter

processno	Nummer (1…10) des zu startenden Prozesses.	LONG
delay	Verzögerungszeit (>30) in Taktzyklen des Zählers. Beim T11 dauert 1 Taktzyklus 3,3 ns, beim T12 1,0 ns, beim T12.1 1,50 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 hochprioren Programmabschnitt heraus kann der Startzeitpunkt nur eingehalten werden, wenn die Verzögerungszeit `delay` größer ist als die restliche Bearbeitungszeit

des Abschnitts.
Alle im Programmabschnitt noch folgenden Zeilen werden 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, dann Prozess 2 um 100
    Rem Zyklen 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 ProcessN_Running), bei niederprioren 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, ProcessN_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 doppelten Hochkommas “ “ angegeben.

Syntax

```
Import String.LI*        '.LI9 für T9, .LIA für T10,
                         '.LIB für T11, .LIC für T12,
                          '.1.LIC für T12.1

Dim text[length] As String

text = "ADwin"
```

Parameter

`text[]`	Name der Text-Variablen	ARRAY STRING
`length`	Länge der Text-Variablen	CONST LONG

Bemerkungen

Dimensionieren Sie Text-Variablen mit `Dim` … `As String` (siehe Seite 210). Setzen Sie eine Zeichenfolge, die sie der Variablen zuweisen möchten, in Hochkommas.

Näheres zu Textvariablen und dem Aufbau von Strings finden Sie unter „Strings“ auf Seite 129.

Sie können Zeichenfolgen mit den unten aufgeführten Anweisungen bearbeiten. Außerdem können Sie Zeichenfolgen mit dem „+“-Operator aneinander hängen.

Beachten Sie beim Prozessor T12/T12.1, dass Sie nicht auf das Feldelement 1 eines Strings (Stringlänge) zugreifen können. Verwenden Sie **`StrLen`**, um die Stringlänge zu bestimmen.

Siehe auch

+ (String-Addition), Dim, Asc, Chr, FloToStr, Flo40ToStr, LngToStr, StrComp, StrLeft, StrLen, StrMid, StrRight, ValF, ValI, ASCII-Zeichensatz, Strings

Beispiel

```
Import String.LI9
  'T9: .LI9 / T10: .LIA / T11: .LIB / T12: .LIC / T12.1: .1.LIC

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 = StrLen[chars] 'String-Länge = 3
  REM bis T11: die String-Länge steht auch in chars[1]
  REM ab T12 ist der Zugriff auf chars[1] nicht mehr erlaubt
  'Par_1 = chars[1]
  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-Ende

  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, .LIC für T12,
                          '.1.LIC für T12.1

ret_val = StrComp(string1[], string2[])
```

Parameter

string1[], string2[]	Zeichenkette	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, Strings

Beispiel

```
Import String.LI9
  'T9: .LI9 / T10: .LIA / T11: .LIB / T12: .LIC / T12.1: .1.LIC

Dim text1[7], text2[7], text3[8] As String

Init:
  text1 = "ADbasic"      'ADbasic richtig
  text2 = "ADbasci"      'ADbasic falsch
  text3 = "ADbasica"     'noch einmal verkehrt

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, .LIC für T12,
                          '.1.LIC für T12.1

StrLeft(string1[], length, string2[])
```

Parameter

string1[]	Zeichenkette, aus der kopiert wird	ARRAY STRING
length	Anzahl der zu kopierenden Zeichen	LONG
string2[]	Zeichenkette, in die kopiert wird	ARRAY STRING

Siehe auch

"" String, + (String-Addition), Asc, Chr, FloToStr, Flo40ToStr, LngToStr, StrComp, StrLen, StrMid, StrRight, ValF, ValI, Strings

Beispiel

```
Import String.LI9
  'T9: .LI9 / T10: .LIA / T11: .LIB / T12: .LIC / T12.1: .1.LIC

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 = StrLen[text]  'String-Länge = 13
  REM bis T11: die String-Länge steht auch in text[1]
  REM ab T12 ist der Zugriff auf text[1] nicht mehr erlaubt
  'Par_1 = text[1]
  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, .LIC für T12,
                          '.1.LIC für T12.1

ret_val = StrLen(string[])
```

Parameter

string[]	Zeichenkette, deren Länge ermittelt wird	ARRAY STRING
ret_val	Anzahl der Zeichen in der Zeichenfolge	LONG

Bemerkungen

Beachten Sie beim Prozessor T12, dass Sie zum Bestimmen der Stringlänge nicht auf das Feldelement 1 eines Strings zugreifen dürfen (möglich bis T11). Verwenden Sie stattdessen **StrLen**.

Siehe auch

"" String, + (String-Addition), Asc, Chr, FloToStr, Flo40ToStr, LngToStr, StrComp, StrLeft, StrMid, StrRight, ValF, ValI, Strings

Beispiel

```
Import String.LI9
  'T9: .LI9 / T10: .LIA / T11: .LIB / T12: .LIC / T12.1: .1.LIC
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, .LIC für T12,
                          '.1.LIC für T12.1

StrMid(string1[], start, length, string2[])
```

Parameter

string1[]	Zeichenkette, aus der kopiert wird	ARRAY STRING
start	Position des ersten Zeichens, das kopiert wird	LONG
length	Anzahl der zu kopierenden Zeichen	LONG
string2[]	Zeichenkette, in die kopiert wird	ARRAY STRING

Siehe auch

"" String, + (String-Addition), Asc, Chr, FloToStr, Flo40ToStr, LngToStr, StrComp, StrLeft, StrLen, StrRight, ValF, ValI, Strings

Beispiel

```
Import String.LI9
  'T9: .LI9 / T10: .LIA / T11: .LIB / T12: .LIC / T12.1: .1.LIC

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 = StrLen[text]  'String-Länge = 18
  REM bis T11: die String-Länge steht auch in text[1]
  REM ab T12 ist der Zugriff auf text[1] nicht mehr erlaubt
  'Par_1 = text[1]
  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, .LIC für T12,
                          '.1.LIC für T12.1

StrRight(string1[], length, string2[])
```

Parameter

string1[]	Zeichenkette, aus der kopiert wird	ARRAY STRING
length	Anzahl der zu kopierenden Zeichen	LONG
string2[]	Zeichenkette, in die kopiert wird	ARRAY STRING

Siehe auch

"" String, + (String-Addition), Asc, Chr, FloToStr, Flo40ToStr, LngToStr, StrComp, StrLeft, StrLen, StrMid, ValF, ValI, Strings

Beispiel

```
Import String.LI9
  'T9: .LI9 / T10: .LIA / T11: .LIB / T12: .LIC / T12.1: .1.LIC

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 = StrLen[text]  'String-Länge = 13
  REM bis T11: die String-Länge steht auch in text[1]
  REM ab T12 ist der Zugriff auf text[1] nicht mehr erlaubt
  'Par_1 = text[1]
  Par_2 = text2[2]       'ASCII-Zeichen 41h = "A"
  Par_3 = text2[3]       'ASCII-Zeichen 44h = "D"
  Par_4 = text2[4]       'ASCII-Zeichen 77h = "w"
  Par_5 = text2[5]       'ASCII-Zeichen 69h = "i"
  Par_6 = text2[6]       'ASCII-Zeichen 6Eh = "n"
  Par_7 = text2[7]       'ASCII-Zeichen 2Dh = "-"
  Par_8 = text2[8]       'ASCII-Zeichen 53h = "S"
  Par_9 = text2[9]       'ASCII-Zeichen 79h = "y"
  Par_10 = text2[10]     'ASCII-Zeichen 73h = "s"
  Par_11 = text2[11]     'ASCII-Zeichen 74h = "t"
  Par_12 = text2[12]     'ASCII-Zeichen 65h = "e"
  Par_13 = text2[13]     'ASCII-Zeichen 6Dh = "m"
  Par_14 = text2[14]     '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 4.5.1 auf Seite 138.

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 ein Feld (kein Feldelement) als Übergabeparameter eines Unterprogramms verwendet wird, ist die Syntax für Aufruf und Definition unterschiedlich:

- Unterprogramm-Aufruf *ohne* Dimensions-Klammern:
  `subname(array_pass)`
- Unterprogramm-Definition *mit* Dimensions-Klammern:
  `Sub subname(array_def[]) …`

Werte werden an Feldelemente (eines Felds als Übergabeparameter) zugewiesen wie gewöhnlich:
`array_def[2] = value`

Wenn Sie einem Übergabeparameter `x` im Unterprogramm einen Wert zuweisen, darf beim Unterprogramm-Aufruf für `x` keine Konstante angegeben werden, sondern nur eine Variable oder ein einzelnes Feld-Element. Auf diese Weise können Übergabeparameter auch einen Rückgabewert enthalten.

Siehe auch

#Include, Function … EndFunction, Lib_Sub … Lib_EndSub, Lib_Function … Lib_EndFunction

Beispiel 1

```
Sub Fast_Dac1(val_1)
Rem Gibt val_1 auf dem analogen Ausgang 1 eines
Rem 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:

```
Event:
  Fast_Dac1(NeuerWert)
```

Beispiel 2

Das gleiche Unterprogramm mit einem Feld als Übergabe-Parameter:

```
Sub Fast_Dac1(array[])
  Rem Gibt Element 3 des Felds array auf dem
  Rem analogen Ausgang eines ADwin-Gold aus
  Rem Wert ins Ausgangsregister schreiben:
  Poke(20400050h, (array[3]))
  Poke(20400010h, 11011b) 'Wandlung starten
EndSub
```

Der Aufruf dieses Unterprogramms erfolgt wieder in ähnlicher Weise (allerdings *ohne* die Dimensionsklammern):

```
Event:
  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

angle	Winkel im Bogenmaß (-π/2…π/2)	FLOAT
ret_val	Tangens des Winkels (-1…1)	FLOAT

Bemerkungen

Wenn Sie für den Winkel Werte außerhalb von -π/2…+π/2 verwenden, nimmt der Berechnungsfehler mit wachsendem Wert zu.

Die Ausführungszeit des Befehls beträgt beim T9 bis zu 1,33 µs, beim T10 bis zu 0,67 µs, beim T11 bis zu 0,31 µs.

Siehe auch

Sin, Cos, ArcSin, ArcCos, ArcTan, ArcTan2, Round

Beispiel

```
Dim val_1, val_2 As Float

Event:
  val_1 = 5.3
  val_2 = Tan(val_1)     'Ergebnis: val_2 = -1.50…
```

User_Version

Nur ab T12: **User_Version** gibt die Versionsnummer des Programms zurück.

Syntax

```
ret_val = User_Version
```

Parameter

- / -

Bemerkungen

User_Version ist eine Systemkonstante und kann nur gelesen, aber nicht verändert werden.

Sie stellen die Versionsnummer im Dialogfenster „Process Options“ (Seite 61) ein.

ValF

ValF konvertiert eine Zeichenfolge (String) in eine Fließkomma-Zahl (float).

Syntax

```
Import String.LI*        '.LI9 für T9, .LIA für T10,
                         '.LIB für T11, .LIC für T12,
                          '.1.LIC für T12.1

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 und T12/T12.1 bis zu 10 Zeichen. Wenn Sie mehr Ziffern angeben, gehen die jeweils letzten Zeichen der Mantisse verloren. Als Dezimalzeichen sind sowohl der Punkt als auch das Komma zulässig.

Beachten Sie den Wertebereich für Float-Werte in Kapitel 4.2.3 auf Seite 110. 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, Strings

Beispiel

```
Import String.LI9
  'T9: .LI9 / T10: .LIA / T11: .LIB / T12: .LIC / T12.1: .1.LIC

Dim text[20] As String

Init:
  text="-27.14159E-10" 'Zu konvertierender String
  Par_1 = StrLen[text]  'String-Länge = 13
  REM bis T11: die String-Länge steht auch in text[1]
  REM ab T12 ist der Zugriff auf text[1] nicht mehr erlaubt
  'Par_1 = text[1]
  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
```

Vall

ValI konvertiert eine Zeichenfolge (String) in eine ganze Zahl (Long).

Syntax

```
Import String.LI*        '.LI9 für T9, .LIA für T10,
                         '.LIB für T11, .LIC für T12,
                          '.1.LIC für T12.1

ret_val = ValI(string[])
```

Parameter

string[]	Zu wandelnde Zeichenfolge im Format: Vorzeichen: + (optional) oder - Vorkommastellen: max. 10 Ziffern {+} vvvvvvvvvv -	ARRAY STRING
ret_val	Erzeugter Long-Wert	LONG

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, Strings

Beispiel

```
Import String.LI9
  'T9: .LI9 / T10: .LIA / T11: .LIB / T12: .LIC / T12.1: .1.LIC

Dim text[20] As String

Init:
  text="-1234567890" 'Zu konvertierender String
  Par_1 = StrLen[text]  'String-Länge = 11
  REM bis T11: die String-Länge steht auch in text[1]
  REM ab T12 ist der Zugriff auf text[1] nicht mehr erlaubt
  'Par_1 = text[1]
  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
  Rem Ergebnis: value = (4 XOr 6) = 0010b = 2
```

7.3 Mathematik-Befehle

Die Bibliothek `Math.li?` enthält zusätzliche Mathematik-Befehle, die über den Befehlssatz des *ADbasic*-Compilers hinaus gehen.

Die Befehle sind verfügbar für Prozessoren ab dem Typ T9.

Mathematik-Befehle

Name	Funktion	Seite
`Mod`	Mod berechnet den ganzzahligen Rest bei einer Division von zwei ganzen Zahlen.	334

Mod

.Hypertext[OnlineHilfe].newlink G3_Mod;

Mod berechnet den ganzzahligen Rest bei einer Division von zwei ganzen Zahlen.

Syntax

```
Import Math.li9        '.LI9 für T9, .LIA für T10,
                        '.LIB für T11, .LIC für T12

ret_val = Mod(x_param, y_param)
```

Parameter

x_param	Dividend.	LONG
y_param	Divisor.	LONG
ret_val	Rest aus der Division von x_param / y_param.	LONG

Bemerkungen

Die Division mit Rest arbeitet nach der symmetrischen Variante, bei der der Quotient durch Abschneiden der Nachkommastellen berechnet wird. Mit dieser Definition wird der Quotient zur Null hin gerundet und der Rest hat das gleiche Vorzeichen wie der Dividend.

Der ganzzahlige Rest bei einer Division durch Null ist gleich dem Dividend: **Mod**(x, 0) = x.

Die Ausführungszeit für die Modulo-Funktion beträgt beim T9 bis zu 3,5 µs, beim T10 bis zu 1,67 µs und beim T11 bis zu 0,44 µs (hohe Priorität).

Siehe auch

/ (Division), Absl

Beispiel

```
Import Math.li9
Par_1 = Mod(17, 3)      'Par_1 = 2
Par_2 = Mod(-9, 5)      'Par_2 = -4
Par_3 = Mod(72, Par_2)  'Par_3 = 0
```

7.4 FFT-Library

Die FFT-Library enthält *ADbasic*-Befehle für die schnelle Fourier-Transformation (Fast Fourier Transformation).

Die Library ist verfügbar für Prozessoren ab dem Typ T9.

Hinweise zur Anwendung der Library

Bei Prozessoren bis T11 können Sie die Rechenzeit deutlich verringern, wenn Sie die Felder im internen Speicher (`At DM_Local`) deklarieren. So dauert die Berechnung einer FFT mit 1024 Werten mit Prozessor T9 etwa 23 ms anstatt 35 ms.

Bei Prozessoren ab T12 kann die Rechenzeit verringert werden, wenn vorwiegend der Cache-Speicher verwendet wird, siehe Kapitel 4.3.3 „Speicherbereiche (T12/T12.1)“. Zur Verkürzung der Rechenzeit ist es grundsätzlich sinnvoll, Programmcode und Daten möglichst klein zu halten; dazu kann auch beitragen, eine FFT mit Werten vom Datentyp `Float32` anstelle `Float64` zu berechnen.

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.

Der Speicherbedarf (in Byte) für die Berechnung einer FFT ist etwa das 20fache der Punktanzahl der Quelldaten. Für 1024 Punkte werden also etwa 20 kB Speicherplatz benötigt.

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	Seite
FFT, FFT_32	FFT führt eine komplexe Fast-Fourier-Transformation mit komplexen Eingangs- und Ausgangsdaten durch.	338
FFT_Scale, FFT_Scale_32	FFT_Scale normiert das Ergebnis einer FFT-Berechnung auf die Größenordnung der einzelnen Signalkomponenten der Quelldaten.	340
FFT_Mag, FFT_Mag_32	FFT_Mag gibt die Absolutbeträge komplexer Werte zurück.	342
FFT_Phase, FFT_Phase_32	FFT_Phase gibt die Phasenlagen komplexer Werte zurück.	344
FFT_Mag_Scale, FFT_Mag_Scale_32	FFT_Mag_Scale gibt die skalierten Absolutbeträge komplexer Werte zurück.	346
FFT_Init, FFT_Init_32	FFT_Init initialisiert 2 Hilfsfelder für die Berechnung von Fast-Fourier-Transformationen.	347
FFT_Calc, FFT_Calc_32	FFT_Calc berechnet eine Fast-Fourier-Transformation nach vorheriger Initialisierung.	348

Name	Funktion	Seite
FFT_Calc_DM	FFT_Calc_DM berechnet eine Fast-Fourier-Transformation nach vorheriger Initialisierung und ist für den Prozessor T10 optimiert.	350
FFT_Calc_DX	FFT_Calc_DX berechnet eine Fast-Fourier-Transformation nach vorheriger Initialisierung und ist für den Prozessor T10 optimiert.	352

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, *.LIC für T12

FFT(real[], img[], z_real[], z_img[], array1[],
    array2[], count)
```

Parameter

real[]	Realteile der Quelldaten.	FLOAT ARRAY
img[]	Imaginärteile der Quelldaten.	FLOAT ARRAY
z_real[]	Ergebnis: Realteile (Index 1…count / 2) der transformierten Daten. Feldgröße: 4 × count.	FLOAT ARRAY
z_img[]	Ergebnis: Imaginärteile (Index 1…count / 2) der transformierten Daten. Feldgröße: 4 × count.	FLOAT ARRAY
array1[], array2[]	Felder für interne Berechnungen. Feldgröße: 4 × count.	FLOAT ARRAY
count	Punktanzahl (≥ 2) der Quelldaten. Die Anzahl der Punkte muss eine Potenz von 2 sein.	LONG

Bemerkungen

Die Fourier-Transformation liefert korrekte Ergebnisse, wenn die Frequenzkomponenten f_i der Quelldaten innerhalb des folgenden Bereichs – bezogen auf die Abtastfrequenz f_{Abtast} – liegen:

$$0 \le f_i < \frac{f_{Abtast}}{2}$$

Die transformierten Daten, die komplexen Amplituden des Frequenzspektrums, befinden sich nur in den Elementen 1…count / 2 der Fel-

der `z_real` und `z_img`. Die restlichen Feldelemente (bis 4 × `count`) 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 `z_real` und `z_img` zugeordnet ist (Normierung der Frequenzachse). Dabei ist t_{total} die gesamte Abtastzeit, in der die Quelldaten abgetastet wurden.
Im Beispiel unten ist die Abtastzeit t_{total} = 0,1 s; dem Element [1024] ist daher die Frequenz (1024-1) / 0,1 s = 10230 Hz zugeordnet.

Elementindex	$[1]$	$[2]$	$\cdots$	$[i]$	$\cdots$	$[count/2]$
Frequenz [Hz]	0	$\frac{1}{t_{total}}$	$\cdots$	$\frac{i-1}{t_{total}}$	$\cdots$	$\frac{count/2-1}{t_{total}}$

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 0,1 s) und berechnet daraus eine FFT. Wenn beispielsweise ein Sinussignal von 1000 Hz 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, *.LIC für T12

FFT_Scale(unscaled[], scaled[], count)
```

Parameter

unscaled[]	Nicht normierte Daten einer FFT-Berechnung.	FLOAT ARRAY
scaled[]	Ergebnis: Normierte Daten.	FLOAT ARRAY
count	Anzahl der Daten.	LONG

Bemerkungen

Die Anweisung arbeitet nach der Formel:

$$\text{scaled[i]} = \begin{cases} i \neq 1: & \text{scaled[i]} = \text{unscaled[i]}/(\text{count}) \\ i = 1: & \text{scaled[i]} = \text{unscaled[i]}/(\text{count} \cdot 2) \end{cases}$$

Wenn Sie **FFT_Scale** direkt auf die Ergebnisfelder der Anweisung **FFT** anwenden, muss count = count / 2 sein (count: 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.

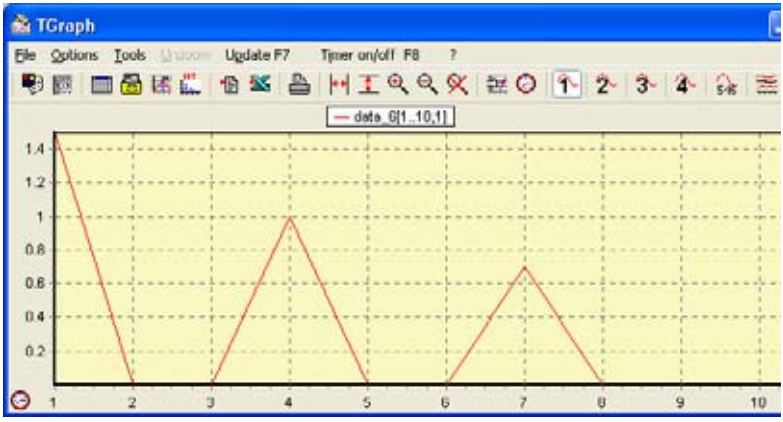
Das Quellsignal entsteht aus:

- 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, erzeugt mit `Tgraph.exe`) 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, *.LIC für T12

FFT_Mag(real[], img[], magnitude[], count)
```

Parameter

real[]	Realteile der komplexen Werte.	FLOAT ARRAY
img[]	Imaginärteile der komplexen Werte.	FLOAT ARRAY
magnitude[]	Ergebnis: Absolutbeträge der komplexen Werte.	FLOAT ARRAY
count	Anzahl der komplexen Werte.	LONG

Bemerkungen

Der Betrag eines komplexen Werts wird errechnet mit der Formel:

$$\text{magnitude}[i] = \sqrt{\text{real}[i]^2 + \text{img}[i]^2}$$

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 count = count / 2 sein (count: 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 0,1 s), berechnet daraus eine FFT und die Absolutbeträge. Wenn beispielsweise ein Sinussignal von 1500 Hz 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, *.LIC für T12

FFT_Phase(real[], img[], phase[], count)
```

Parameter

real[]	Realteile der komplexen Werte.	FLOAT ARRAY
img[]	Imaginärteile der komplexen Werte.	FLOAT ARRAY
phase[]	Ergebnis: Phasenlagen der komplexen Werte.	FLOAT ARRAY
count	Anzahl der komplexen Werte.	LONG

Bemerkungen

Die Phasenberechnung arbeitet nach folgender Formel:

$$\text{phase}[i] = \begin{cases} \text{real}[i] \neq 0: & \text{phase}[i] = \text{atan}(\text{img}[i]/\text{real}[i]) \\ \text{real}[i] = 0: & \text{phase}[i] = \pi/2 \cdot \text{sgn}(\text{img}[i]) \end{cases}$$

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 count = count / 2 sein (count: 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 π/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 p/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, *.LIC für T12

FFT_Mag_Scale(real[], img[], mag_scal[], count)
```

Parameter

real[]	Realteile der komplexen Werte.	FLOAT ARRAY
img[]	Imaginärteile der komplexen Werte.	FLOAT ARRAY
mag_scal[]	Ergebnis: Normierte Absolutbeträge der komplexen Werte.	FLOAT ARRAY
count	Anzahl der komplexen Werte.	LONG

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 count = count / 2 sein.

Siehe auch

FFT, FFT_Mag, FFT_Scale

Beispiel

Das Beispielprogramm (für alle *ADwin*-Systeme)
C:\ADwin\ADbasic\lib\FFT_doc+demo\FFT_scale_demo_opt.bas

entspricht dem Beispiel FFT_scale_demo.bas (siehe Seite 340FFT_Scale), 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, *.LIC für T12

FFT_Init(array1[], array2[], count)
```

Parameter

array1[], array2[]	Ergebnis: Hilfswerte für interne Berechnungen. Feldgröße: 4 × count.	FLOAT ARRAY
count	Punktanzahl (≥ 2) der Quelldaten. Die Anzahl der Punkte muss eine Potenz von 2 sein.	LONG

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 count 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 (für alle *ADwin*-Systeme) im Ordner C:\ADwin\ADbasic\lib\FFT_doc+demo:
FFT_scale_demo_opt.bas

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, *.LIC für T12

FFT_Calc(real[], img[], z_real[], z_img[],
     array1[], array2[], count)
```

Parameter

real[]	Realteile der Quelldaten.	FLOAT ARRAY
img[]	Imaginärteile der Quelldaten.	FLOAT ARRAY
z_real[]	Ergebnis: Realteile (Index 1…count / 2) der transformierten Daten. Feldgröße: 4 × count.	FLOAT ARRAY
z_img[]	Ergebnis: Imaginärteile (Index 1…count / 2) der transformierten Daten. Feldgröße: 4 × count.	FLOAT ARRAY
array1[], array2[]	Felder für interne Berechnungen. Feldgröße: 4 × count.	FLOAT ARRAY
count	Punktanzahl (≥ 2) der Quelldaten. Die Anzahl der Punkte muss eine Potenz von 2 sein.	LONG

Bemerkungen

Die Anweisung ist nur sinnvoll, wenn vorher **FFT_Init** aufgerufen wurde.

☞ Wenn Sie mehrere FFT mit der *gleichen* Anzahl count 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 (für alle *ADwin*-Systeme) im Ordner C:\ADwin\ADbasic\lib\FFT_doc+demo:
FFT_scale_demo_opt.bas

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(real[], img[], z_real[], z_img[],
     arr1[], arr2[], count)
```

Parameter

real[]	Realteile der Quelldaten. Das Feld muss At DM_Local deklariert werden.	FLOAT ARRAY
img[]	Imaginärteile der Quelldaten. Das Feld muss At DM_Local deklariert werden.	FLOAT ARRAY
z_real[]	Ergebnis: Realteile (Index 1…count / 2) der transformierten Daten. Das Feld muss At DM_Local mit der Größe: 4 × count deklariert werden.	FLOAT ARRAY
z_img[]	Ergebnis: Imaginärteile (Index 1…count / 2) der transformierten Daten. Das Feld muss At DM_Local mit der Größe: 4 × count deklariert werden.	FLOAT ARRAY
array1[], array2[]	Felder für interne Berechnungen. Die Felder müssen At DM_Local mit der Größe: 4 × count deklariert werden.	FLOAT ARRAY
count	Punktanzahl (≥ 2) der Quelldaten. Die Anzahl der Punkte muss eine Potenz von 2 sein.	LONG

Bemerkungen

Die Anweisung ist nur sinnvoll, wenn vorher **FFT_Init** aufgerufen wurde.

FFT_Calc_DM hat die gleiche Funktion wie **FFT_Calc**, berechnet eine FFT aber etwas schneller. Der Befehl **FFT_Calc_DM** kann nur auf dem Prozessor T10 sinnvoll eingesetzt werden.

FFT_Calc_DM darf nur verwendet werden, wenn alle übergebenen 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 (für alle *ADwin*-Systeme) im Ordner C:\ADwin\ADbasic\lib\FFT_doc+demo>:
FFT_scale_demo_opt.bas

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(real[], img[], z_real[], z_img[],
     array1[], array2[], count)
```

Parameter

real[]	Realteile der Quelldaten. Das Feld sollte At DRAM_Extern deklariert werden.	FLOAT ARRAY
img[]	Imaginärteile der Quelldaten. Das Feld sollte At DRAM_Extern deklariert werden.	FLOAT ARRAY
z_real[]	Ergebnis: Realteile (Index 1…count / 2) der transformierten Daten. Das Feld sollte At DRAM_Extern mit der Größe 4 × count deklariert sein.	FLOAT ARRAY
z_img[]	Ergebnis: Imaginärteile (Index 1…count / 2) der transformierten Daten. Das Feld sollte At DRAM_Extern mit der Größe 4 × count deklariert sein.	FLOAT ARRAY
array1[], array2[]	Felder für interne Berechnungen. Die Felder müssen At DRAM_Extern mit der Größe: 4 × count deklariert werden.	FLOAT ARRAY
count	Punktanzahl (≥ 2) der Quelldaten. Die Anzahl der Punkte muss eine Potenz von 2 sein.	LONG

Bemerkungen

Die Anweisung ist nur sinnvoll, wenn vorher **FFT_Init** aufgerufen wurde.

FFT_Calc_DX hat die gleiche Funktion wie **FFT_Calc**, berechnet eine FFT aber etwas schneller. Der Befehl **FFT_Calc_DX** kann nur auf dem Prozessor T10 sinnvoll eingesetzt werden.

FFT_Calc_DX darf nur verwendet werden, wenn alle übergebenen Felder im externen Speicher deklariert sind.

Mit dem Prozessor T10 dauert die Berechnung einer FFT mit 1024 Werten nur noch etwa 49 ms anstatt 53 ms 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 (für *ADwin*-Systeme außer Pro II) im Ordner C:\ADwin\ADbasic\lib\FFT_doc+demo:
FFT_scale_demo_opt_DX.bas

FFT_32

FFT_32 führt eine komplexe Fast-Fourier-Transformation mit komplexen Eingangs- und Ausgangsdaten durch.

Syntax

```
Import FFT.LI*        '.LIC für T12,
                      '.1.LIC für T12.1

FFT_32(real[], img[], z_real[], z_img[], array1[],
    array2[], count)
```

Parameter

real[]	Realteile der Quelldaten.	FLOAT32 ARRAY
img[]	Imaginärteile der Quelldaten.	FLOAT32 ARRAY
z_real[]	Ergebnis: Realteile (Index 1…count / 2) der transformierten Daten. Feldgröße: 4 × count.	FLOAT32 ARRAY
z_img[]	Ergebnis: Imaginärteile (Index 1…count / 2) der transformierten Daten. Feldgröße: 4 × count.	FLOAT32 ARRAY
array1[], array2[]	Felder für interne Berechnungen. Feldgröße: 4 × count.	FLOAT32 ARRAY
count	Punktanzahl (≥ 2) der Quelldaten. Die Anzahl der Punkte muss eine Potenz von 2 sein.	LONG

Bemerkungen

Die Fourier-Transformation liefert korrekte Ergebnisse, wenn die Frequenzkomponenten f_i der Quelldaten innerhalb des folgenden Bereichs – bezogen auf die Abtastfrequenz f_{Abtast} – liegen:

$$0 \le f_i < \frac{f_{Abtast}}{2}$$

Die transformierten Daten, die komplexen Amplituden des Frequenzspektrums, befinden sich nur in den Elementen 1…`count`/2 der Felder `z_real` und `z_img`. Die restlichen Feldelemente (bis 4 × `count`) 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_32`** normieren.

Die folgende Tabelle zeigt, wie das berechnete Frequenzspektrum den Elementen der Felder `z_real` und `z_img` zugeordnet ist (Normierung der Frequenzachse). Dabei ist t_{total} die gesamte Abtastzeit, in der die Quelldaten abgetastet wurden.
Im Beispiel unten ist die Abtastzeit t_{total} = 0,1 s; dem Element [1024] ist daher die Frequenz (1024-1) / 0,1 s = 10230 Hz zugeordnet.

Elementindex	[1]	[2]	…	[i]	…	[count/2]
Frequenz [Hz]	0	$\frac{1}{t_{total}}$	…	$\frac{i-1}{t_{total}}$	…	$\frac{count/2-1}{t_{total}}$

Wenn Sie mehrere FFT mit der *gleichen* Anzahl an Quelldaten berechnen, kann die Berechnungszeit verringert werden: Rufen Sie anstelle von **`FFT_32`** zuerst die Anweisung **`FFT_Init_32`** und anschließend mehrfach **`FFT_Calc_32`** auf.

Siehe auch

`FFT_Mag_32`, `FFT_Scale_32`, `FFT_Phase_32`, `FFT_Mag_Scale_32`, `FFT_Init_32`, `FFT_Calc_32`

Beispiel

siehe FFT

FFT_Scale_32

FFT_Scale_32 normiert das Ergebnis einer FFT-Berechnung auf die Größenordnung der einzelnen Signalkomponenten der Quelldaten.

Syntax

```
Import FFT.LI*        '.LIC für T12,
                          '.1.LIC für T12.1

FFT_Scale_32(unscaled[], scaled[], count)
```

Parameter

unscaled[]	Nicht normierte Daten einer FFT-Berechnung.	FLOAT32 ARRAY
scaled[]	Ergebnis: Normierte Daten.	FLOAT32 ARRAY
count	Anzahl der Daten.	LONG

Bemerkungen

Die Anweisung arbeitet nach der Formel:

$$\text{scaled}[i] = \begin{cases} i \neq 1: & \text{scaled}[i] = \text{unscaled}[i]/(\text{count}) \\ i = 1: & \text{scaled}[i] = \text{unscaled}[i]/(\text{count} \cdot 2) \end{cases}$$

Wenn Sie **FFT_Scale_32** direkt auf die Ergebnisfelder der Anweisung **FFT_32** anwenden, muss count = count / 2 sein (count: Parameter von **FFT_32**).

FFT_Scale_32 normiert das Ergebnis einer FFT-Berechnung auf die Größenordnung der einzelnen Signalkomponenten der Originaldaten. Dagegen normiert **FFT_Scale_32** nicht die Frequenzachse des Spektrums (siehe Erläuterungen hierzu unter **FFT_32**).

Siehe auch

FFT_32, FFT_Mag_32, FFT_Phase_32, FFT_Mag_Scale_32

Beispiel

siehe FFT_Scale

FFT_Mag_32

FFT_Mag_32 gibt die Absolutbeträge komplexer Werte zurück.

Syntax

```
Import FFT.LI*        '.LIC für T12,
                          '.1.LIC für T12.1

FFT_Mag_32(real[], img[], magnitude[], count)
```

Parameter

real[]	Realteile der komplexen Werte.	FLOAT32 ARRAY
img[]	Imaginärteile der komplexen Werte.	FLOAT32 ARRAY
magnitude[]	Ergebnis: Absolutbeträge der komplexen Werte.	FLOAT32 ARRAY
count	Anzahl der komplexen Werte.	LONG

Bemerkungen

Der Betrag eines komplexen Werts wird errechnet mit der Formel:

$$\text{magnitude}[i] = \sqrt{\text{real}[i]^2 + \text{img}[i]^2}$$

Die Anweisung **FFT_32** berechnet die Amplituden des Frequenzspektrums als komplexe Werte. Die Anweisungen **FFT_Mag_32** und **FFT_Phase_32** rechnen die komplexen Amplituden in Betrag und Phase um.

Wenn Sie **FFT_Mag_32** direkt auf die Ergebnisfelder der Anweisung **FFT_32** anwenden, muss count = count / 2 sein (count: Parameter von **FFT_32**).

Siehe auch

FFT_32, FFT_Phase_32, FFT_Mag_Scale_32

Beispiel

siehe FFT_Mag

FFT_Phase_32

FFT_Phase_32 gibt die Phasenlagen komplexer Werte zurück.

Syntax

```
Import FFT.LI*        '.LIC für T12,
                          '.1.LIC für T12.1

FFT_Phase_32(real[], img[], phase[], count)
```

Parameter

real[]	Realteile der komplexen Werte.	FLOAT32 ARRAY
img[]	Imaginärteile der komplexen Werte.	FLOAT32 ARRAY
phase[]	Ergebnis: Phasenlagen der komplexen Werte.	FLOAT32 ARRAY
count	Anzahl der komplexen Werte.	LONG

Bemerkungen

Die Phasenberechnung arbeitet nach folgender Formel:

$$\text{phase}[i] = \begin{cases} \text{real}[i] \neq 0: & \text{phase}[i] = \text{atan}(\text{img}[i]/\text{real}[i]) \\ \text{real}[i] = 0: & \text{phase}[i] = \pi/2 \cdot \text{sgn}(\text{img}[i]) \end{cases}$$

Die Anweisung **FFT_32** berechnet die Amplituden des Frequenzspektrums als komplexe Werte. Die Anweisungen **FFT_Mag_32** und **FFT_Phase_32** rechnen die komplexen Amplituden in Betrag und Phase um.

Wenn Sie **FFT_Phase_32** direkt auf die Ergebnisfelder der Anweisung **FFT_32** anwenden, muss count = count / 2 sein (count: Parameter von **FFT_32**).

Siehe auch

FFT_32, FFT_Mag_32, FFT_Mag_Scale_32

Beispiel

siehe FFT_Phase

FFT_Mag_Scale_32

FFT_Mag_Scale_32 gibt die skalierten Absolutbeträge komplexer Werte zurück.

Syntax

```
Import FFT.LI*        '.LIC für T12,
                          '.1.LIC für T12.1

FFT_Mag_Scale_32(real[], img[], mag_scal[], count)
```

Parameter

real[]	Realteile der komplexen Werte.	FLOAT32 ARRAY
img[]	Imaginärteile der komplexen Werte.	FLOAT32 ARRAY
mag_scal[]	Ergebnis: Normierte Absolutbeträge der komplexen Werte.	FLOAT32 ARRAY
count	Anzahl der komplexen Werte.	LONG

Bemerkungen

Die Anweisung **FFT_Mag_Scale_32** liefert das gleiche Ergebnis wie der Aufruf von **FFT_Mag_32** und **FFT_Scale_32**, arbeitet aber schneller.

Wenn Sie **FFT_Mag_Scale_32** direkt auf die Ergebnisfelder der Anweisung **FFT_32** anwenden, muss count = count / 2 sein.

Siehe auch

FFT_32, FFT_Mag_32, FFT_Scale_32

Beispiel

siehe FFT_Mag_Scale

FFT_Init_32

FFT_Init_32 initialisiert 2 Hilfsfelder für die Berechnung von Fast-Fourier-Transformationen.

Syntax

```
Import FFT.LI*        '.LIC für T12,
                          '.1.LIC für T12.1

FFT_Init_32(array1[], array2[], count)
```

Parameter

array1[], array2[]	Ergebnis: Hilfswerte für interne Berechnungen. Feldgröße: 4 × count.	FLOAT32 ARRAY
count	Punktanzahl (≥ 2) der Quelldaten. Die Anzahl der Punkte muss eine Potenz von 2 sein.	LONG

Bemerkungen

Die Anweisung **FFT_Init_32** ist nur notwendig und sinnvoll, wenn im Anschluss eine der Anweisungen **FFT_Calc_32**, **FFT_Calc_32_DM** oder **FFT_Calc_32_DX** aufgerufen wird.

Wenn Sie mehrere FFT mit der *gleichen* Anzahl count an Quelldaten berechnen, kann die Berechnungszeit verringert werden: Rufen Sie anstelle von **FFT_32** zuerst die Anweisung **FFT_Init_32** und anschließend mehrfach **FFT_Calc_32** auf.

Siehe auch

FFT_32, FFT_Calc_32

Beispiel

siehe FFT_Init

FFT_Calc_32

FFT_Calc_32 berechnet eine Fast-Fourier-Transformation nach vorheriger Initialisierung.

Syntax

```
Import FFT.LI*        '.LIC für T12,
                          '.1.LIC für T12.1

FFT_Calc_32(real[], img[], z_real[], z_img[],
     array1[], array2[], count)
```

Parameter

real[]	Realteile der Quelldaten.	FLOAT32 ARRAY
img[]	Imaginärteile der Quelldaten.	FLOAT32 ARRAY
z_real[]	Ergebnis: Realteile (Index 1…count / 2) der transformierten Daten. Feldgröße: 4 × count.	FLOAT32 ARRAY
z_img[]	Ergebnis: Imaginärteile (Index 1…count / 2) der transformierten Daten. Feldgröße: 4 × count.	FLOAT32 ARRAY
array1[], array2[]	Felder für interne Berechnungen. Feldgröße: 4 × count.	FLOAT32 ARRAY
count	Punktanzahl (≥ 2) der Quelldaten. Die Anzahl der Punkte muss eine Potenz von 2 sein.	LONG

Bemerkungen

Die Anweisung ist nur sinnvoll, wenn vorher **FFT_Init_32** aufgerufen wurde.

Wenn Sie mehrere FFT mit der *gleichen* Anzahl `count` an Quelldaten berechnen, kann die Berechnungszeit verringert werden: Rufen Sie anstelle von **`FFT_32`** zuerst die Anweisung **`FFT_Init_32`** und anschließend mehrfach **`FFT_Calc_32`** auf.

Siehe auch

`FFT_32`, `FFT_Init_32`

Beispiel

siehe FFT_Calc

8 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 die Einstellungen im Menü „`Options\Compiler`“ 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

Um die Tastaturkürzel für Textbausteine zu sehen, öffnen Sie die Datei `ADbasicCS.xml` in `C:\ADwin\ADbasic\Common\` mit einem Browser.

Tastenkürzel	Funktion	Menüaufruf
F1	Hilfethema für den markierten Befehl aufrufen.	
Ctrl-F1	Inhaltsverzeichnis der Hilfe aufrufen.	`Help` ▶ `Content`
F2	Deklaration eines Befehls oder Variablen anzeigen.	
Ctrl-F2	Zur Deklaration eines Befehls oder Variablen springen.	
F3	Text nochmals vorwärts suchen.	`Edit` ▶ `Find Next`
Shift-F3	Text nochmals rückwärts suchen.	
Ctrl-F3	Text an Cursor-Position vorwärts suchen.	
Ctrl-Shift-F3	Text an Cursor-Position rückwärts suchen.	
Ctrl-F5	*ADwin*-System booten.	
F6	Bibliothek erzeugen.	`Build` ▶ `Make Lib File`
F7	Binärdatei erzeugen.	`Build` ▶ `Make Bin File`
Ctrl-F7	Alle Binärdateien des Projekts erzeugen.	`Build` ▶ `Make All Bin Files`
F8	Quelltext kompilieren.	`Build` ▶ `Compile`
Ctrl-F8	Prozess starten.	
F9	Prozess stoppen.	
Ctrl-Space	Befehle und Variablen automatisch vervollständigen.	
Ctrl-Shift-Space	Befehlsparameter anzeigen.	

Tastenkürzel	Funktion	Menüaufruf
Ctrl-a	Alles markieren.	`Edit ▶ Select All`
Ctrl-b	Zeilen in Kommentar ändern.	Source context menu: `Comment Block`
Ctrl-Shift-b	Kommentarzeichen aus markierten Zeilen löschen.	Source context menu: `Uncomment Block`
Ctrl-c	Kopieren.	`Edit ▶ Copy`
Ctrl-F	Text suchen.	`Edit ▶ Find`
Ctrl-g	Zu einer Zeile springen.	
Ctrl-h	Text ersetzen.	`Edit ▶ Replace`
Ctrl-i	Zeilen nach rechts einrücken	Source context menu: Indent
Ctrl-Shift-i	Zeilen nach links rücken	Source context menu: Outdent
Ctrl-n	Neue Quelltextdatei.	`File ▶ New`
Ctrl-o	Quelltextdatei öffnen.	`File ▶ Open`
Ctrl-p	Quelltextdatei drucken.	`File ▶ Print`
Ctrl-r	Verwendete Parameter markieren.	Parameter window: Icon
Ctrl-s	Quelltextdatei speichern.	`File ▶ Save`
Ctrl-u	Kleinschreibung.	
Ctrl-Shift-u	Großschreibung.	
Ctrl-v	Einfügen.	`Edit ▶ Paste`
Ctrl-x	Ausschneiden.	`Edit ▶ Cut`
Ctrl-z	Änderung zurücknehmen.	`Edit ▶ Undo`
Ctrl-Shift-z	Änderung wieder herstellen.	`Edit ▶ Redo`
Ctrl-k + k	Textmarke ein- oder ausschalten.	
Ctrl-k + n	Zur nächsten Textmarke springen.	
Ctrl-k + p	Zur vorigen Textmarke springen.	
Ctrl-k + x	Einen Textbaustein aus einer Liste einfügen.	

Legende:

A-B: Tasten A und B gleichzeitig drücken.

A+B: Tasten A und B nacheinander drücken, erst A, dann B.

A.2 ASCII-Zeichensatz

NUL 00h 0	SOH 01h 1	STX 02h 2	ETX 03h 3	EOT 04h 4	ENQ 05h 5	ACK 06h 6	BEL 07h 7
BS[1] 08h 8	TAB[2] 09h 9	LF[3] 0Ah 10	VT 0Bh 11	FF 0Ch 12	CR[4] 0Dh 13	SO 0Eh 14	SI 0Fh 15
DLE 10h 16	DC1 11h 17	DC2 12h 18	DC3 13h 19	DC4 14h 20	NAK 15h 21	SYN 16h 22	ETB 17h 23
CAN 18h 24	EM 19h 25	SUB 1Ah 26	ESC 1Bh 27	FS 1Ch 28	GS 1Dh 29	RS 1Eh 30	US 1Fh 31
SPC[5] 20h 32	! 21h 33	" 22h 34	# 23h 35	$ 24h 36	% 25h 37	& 26h 38	' 27h 39
(28h 40	) 29h 41	* 2Ah 42	+ 2Bh 43	, 2Ch 44	- 2Dh 45	. 2Eh 46	/ 2Fh 47
0 30h 48	1 31h 49	2 32h 50	3 33h 51	4 34h 52	5 35h 53	6 36h 54	7 37h 55
8 38h 56	9 39h 57	: 3Ah 58	; 3Bh 59	< 3Ch 60	= 3Dh 61	> 3Eh 62	? 3Fh 63
@ 40h 64	A 41h 65	B 42h 66	C 43h 67	D 44h 68	E 45h 69	F 46h 70	G 47h 71
H 48h 72	I 49h 73	J 4Ah 74	K 4Bh 75	L 4Ch 76	M 4Dh 77	N 4Eh 78	O 4Fh 79
P 50h 80	Q 51h 81	R 52h 82	S 53h 83	T 54h 84	U 55h 85	V 56h 86	W 57h 87
X 58h 88	Y 59h 89	Z 5Ah 90	[5Bh 91	\ 5Ch 92	] 5Dh 93	^ 5Eh 94	_ 5Fh 95
` 60h 96	a 61h 97	b 62h 98	c 63h 99	d 64h 100	e 65h 101	f 66h 102	g 67h 103
h 68h 104	i 69h 105	j 6Ah 106	k 6Bh 107	l 6Ch 108	m 6Dh 109	n 6Eh 110	o 6Fh 111
p 70h 112	q 71h 113	r 72h 114	s 73h 115	t 74h 116	u 75h 117	v 76h 118	w 77h 119
x 78h 120	y 79h 121	z 7Ah 122	{ 7Bh 123	| 7Ch 124	} 7Dh 125	~ 7Eh 126	☒ 7Fh 127

[1] Backspace, [2] Tabulator, [3] Linefeed,
[4] Carriage Return, [5] Space

A.3 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, zurückzuentwikkeln, 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.4 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 Kommandozeilen-Aufruf hat sich im Vergleich zu *ADbasic 4* geändert, d.h. Sie müssen die Syntax vorhandener Aufrufe überprüfen.

Beachten Sie die allgemeinen Hinweise zur Kommandozeile unter Windows auf Seite A-9.

A.4.1 Syntax

Es gibt Kommandozeilen-Aufrufe zum Erzeugen einer Binärdatei (Hauptoption `/M`) und zum Erzeugen einer Bibliothek (Hauptoption `/L`).

Über Schalter werden die Optionen für den Compiler eingestellt. Wenn ein Schalter nicht angegeben ist, wird die jeweilige Voreinstellung (Default) verwendet; wir empfehlen aber dennoch, alle Schalter anzugeben, um Unklarheiten zu vermeiden[1].

Alternativ können Sie die Optionen für einen Aufruf in eine Datei, das `makefile`, schreiben und den Compiler mit der Hauptoption `/MAKE` aufrufen.

Schließlich gibt es die Hauptoptionen `/H` zum Aufruf eines kurzen Hilfetexts und `/VER` zur Anzeige der Compiler-Version.

Der Aufruf wird in einer einzelnen Zeile geschrieben. Achten Sie auf Großschreibung der Schalter.

1. Beispielsweise bleibt ein Aufruf mit allen Schaltern korrekt, auch wenn sich eine Default-Einstellung ändern sollte.

Syntax

```
ADbasicCompiler /M src.bas
[/A"dest"] [/IP"path"] [/LP"path"] [/Lx] [/Sx] [/Px]
[/ET | /EE] [/PNx] [/PH | /PL | /PLx] [/PDx] [/Ox]
[/Vx]
```

```
ADbasicCompiler /L src.bas
[/A"dest"] [/IP"path"] /LP"path"] [/Lx] [/Sx] [/Px]
[/Ox]
```

```
ADbasicCompiler /MAKE"makefile"
```

```
ADbasicCompiler /H
```

```
ADbasicCompiler /VER
```

Achtung: Für den Prozessortyp T12 (Option `/P12`) muss anstelle von `ADbasicCompiler` der Aufruf `ADbasic_C` geschrieben werden.

Optionale Angaben stehen in eckigen Klammern `[]`. Das Zeichen | trennt Schalter, die sich gegenseitig ausschließen.

Dateinamen können ohne, mit relativem oder mit absolutem Pfadnamen angegeben werden. Das Basisverzeichnis für die beiden ersten Angaben ist das Arbeitsverzeichnis, aus dem heraus die Kommandozeile aufgerufen wird.

Hauptschalter

/M	Binärdatei mit der Endung `.Txn` erzeugen.	
	x	Prozessortyp; siehe Schalter `/Px`.
	n	Prozessnummer; siehe Schalter `/PNx`.
/L	Library-Datei (Bibliothek) mit der Endung `.LIx` erzeugen.	
	x	Prozessortyp; siehe Schalter `/Px`.
/MAKE	Hauptschalter, Dateiname und weitere Schalter für einen Aufruf aus dem `makefile` entnehmen.	
	Der Text im `makefile` kann über mehrere Zeilen verteilt geschrieben werden. Schalter außerhalb des `makefile` sind nicht erlaubt.	
/H	Kurzen Hilfetext anzeigen.	
/VER	Versionsnummer des Compilers anzeigen.	

Schalter

src.bas	Dateiname des zu kompilierenden Quelltextes; mit Dateiendung `.bas` angeben. Compiler-Warnungen werden in die Datei `src.wrn` geschrieben, Fehlermeldungen in die Datei `src.err`.
/A"dest"	[Pfad und] Name der zu erzeugenden Datei <`dest`> *ohne* Dateiendung. Voreinstellung ist der Dateiname `src`. Die Dateiendung `.Txn` (Binärdatei) oder `.LIx` (Library-Datei) wird automatisch angehängt.
/IP"path"	Verzeichnis, in dem nach Include-Dateien gesucht wird. Die Einstellung überschreibt den Standard-Pfad von *ADbasic* und sollte mit Bedacht eingesetzt werden.
/LP"path"	Verzeichnis, in dem nach Library-Dateien gesucht wird. Die Einstellung überschreibt den Standard-Pfad von *ADbasic* und sollte mit Bedacht eingesetzt werden.
/Lx	Sprache, in der Fehlermeldungen und Warnungen ausgegeben werden. /LE Englisch. Default. /LG Deutsch
/Sx	Hardware einstellen, für die die Datei kompiliert wird: /SC Karten /SL Light-16 /SG Gold; Default. /SGII Gold II /SP Pro /SPII Pro II /SX X-A20

/Px	Prozessortyp, für den die Datei kompiliert wird: /P2 Prozessor T2 /P4 Prozessor T4 /P5 Prozessor T5 /P8 Prozessor T8 /P9 Prozessor T9; Default /P10 Prozessor T10 /P11 Prozessor T11 /P12 Prozessor T12 /P121 Prozessor T12.1
/ET	Zeitgesteuerten Prozess erzeugen, siehe Kapitel 6 auf Seite 159. Default. Schließt `/EE` aus.
/EE	Extern gesteuerten Prozess erzeugen, siehe Kapitel 6 auf Seite 159. Schließt `/ET` aus.
/PNx	Nummer `x` (1…10) des Prozesses. Default: 1.
/PH	Prozess mit hoher Priorität erzeugen. Default-Einstellung. Siehe auch Kapitel 6.1.2 auf Seite 161.
/PL	Prozess mit niedriger Priorität und Prioritätsstufe 1 erzeugen (nur für zeitgesteuerten Prozess). Siehe auch Kapitel 6.1.3 auf Seite 161.
/PLx	Prozess mit niedriger Priorität und Prioritätsstufe `x` (-10…10) erzeugen.
/PDx	Zykluszeit (Processdelay) des Prozesses auf `x` einstellen. Default: 1000, für T11: 3000. Siehe auch Kapitel 6.2.1 auf Seite 164.
/PSx	nur mit Optionen `/P12` oder `/P121`: Größe des Programm-Stapelspeichers (stack size) auf `x` einstellen, Default: 1000. Siehe auch Dialogfenster „Process Options“.
/Ox	Optimierungsstufe `x` für die Kompilierung einstellen, siehe auch Dialogfenster „Process Options“ (Seite 59).

	/O0	Optimierungsstufe 0 (=keine Optimierung)
	/O1	Optimierungsstufe 1 (Default)
	/O2	Optimierungsstufe 2
	/O3	Optimierungsstufe 3 (nur mit Option `/P12`)
	/Os	Geringe Speichergröße (nur mit Option `/P12`)
	/Of	Hohe Geschwindigkeit (nur mit Option `/P12`)
/Vx	Version `x` des Prozesses einstellen, siehe Dialogfenster „Process Options“ (Seite 59). Default: 1.	

A.4.2 Bemerkungen

Die Reihenfolge der Schalter ist beliebig. Bei den Schaltern wird Groß-/Kleinschreibung unterschieden, bei Dateinamen nicht.

Wenn der Schalter `/A` nicht verwendet wird, wird die erzeugte Binärdatei oder Library-Datei in dem Verzeichnis gespeichert, in dem sich der Quelltext befindet.

Treten während der Kompilierung Warnungen oder Fehler auf, werden sie in die Dateien `src.wrn` und `src.err` geschrieben. Die Fehlermeldungen sind identisch mit denjenigen, die *ADbasic* im Infofenster (siehe Kapitel 3.12.1) ausgeben würde.
Die Dateien werden in dem Verzeichnis gespeichert, wo sich der Quelltext `src.bas` befindet. Wenn Sie den Schalter `/A` verwenden, werden die Dateien in dem Verzeichnis gespeichert, wo die Binär- oder Library-Datei erzeugt wird.

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.4.3 Beispiele

```
C:\ADwin\ADbasic\ADbasiccompiler.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, werden alle Standardeinstellungen verwendet:

- erzeugte Datei im Verzeichnis der Quelldatei speichern
- englische Warnungen und Fehlermeldungen.
- Hardware: *ADwin-Gold*.
- Prozessor: T9.
- Optimierungsstufe 1.

Wenn Sie sich beim Aufruf im Verzeichnis `C:\ADwin\ADbasic` befinden, können Sie obige Zeile kürzer schreiben:

```
ADbasiccompiler.exe /L Z:\Myfiles\test.bas
```

Die kürzeste Aufruf-Variante ergibt sich, wenn auch der Quelltext im Verzeichnis `C:\ADwin\ADbasic` liegt:

```
ADbasiccompiler /L test.bas
```

Wir empfehlen in jedem Fall – speziell für eine Automatisierung des Aufrufs – die vollständige Variante:

```
ADbasiccompiler /L test.bas /A"test" /LE /SG /P9 /O1
```

```
ADbasicCompiler /L Z:\Myfiles\string.bas /SP /O1
```

Der Aufruf kompiliert den Quelltext `string.bas` mit Optimierungs-Level 1 in eine Library-Datei für ein *Pro*-System mit Prozessor T9. Der Prozess ist zeitgesteuert, hat die Nummer 1 und hohe Priorität.
Der gleiche Aufruf, nur für den Prozessor T10, sieht so aus:

```
ADbasicCompiler /L Z:\Myfiles\string.bas /P10 /SP /O1
```

```
ADbasicCompiler /M
C:\ADwin\ADbasic\samples_ADwin\bas_dmo6f.bas /LE /SG /P9
/ET /PN3 /PH /O1
```

Kompiliert die Demo-Datei `bas_dmo6f.bas` in eine Binär-Datei für ein Gold-System mitProzessor T9; der Prozess ist zeitgesteuert, hat die Nummer 3 und hohe Priorität.

```
ADbasic_C /M bas_dmo6 /LE /SPII /P12 /EE /PN5 /PH /Os
```

Kompiliert die Demo-Datei `bas_dmo6.bas` in eine Binär-Datei für Pro II mit dem Prozessor T12, optimiert für geringe Speichergröße; der Prozess ist zeitgesteuert, hat die Nummer 5 und hohe Priorität.

```
ADbasicCompiler /M
C:\ADwin\ADbasic\samples_ADwin\bas_dmo6 /LE /SL /P8 /EE
/PN2 /PL /O0
```

Kompiliert die Demo-Datei `bas_dmo6.bas` in eine Binär-Datei für eine Light-16-Karte mit dem Prozessor T8, Optimierung ist deaktiviert; der Prozess ist zeitgesteuert, hat die Nummer 2 und niedrige Priorität.

```
ADbasicCompiler /M C:\user\my_file.bas /LE /SC /P4 /EE
/PN5 /PL /A"your_file" /O1
```

Die Anweisung kompiliert die Datei `my_file.bas` für eine *ADwin*-Karte mit Prozessor T4; der Prozess ist extern gesteuert, hat die Nummer 5 und niedrige Priorität. Die erzeugte Binärdatei hat den Namen `your_file.T45` und befindet sich im gleichen Verzeichnis wie der Quelltext: `C:\user`.

```
ADbasicCompiler /M C:\user\my_file.bas /LE /SG /P9
/A"Y:\somewhere\your_file" /ET /PN3 /PH /O1
```

Die Binärdatei heißt nun `your_file.T93` und befindet sich im Verzeichnis `Y:\somewhere`; der Prozess ist zeitgesteuert, hat die Nummer 3 und hohe Priorität.

A.4.4 Kommandozeile unter Windows

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 und eignen sich z. B. für die Automatisierung von Abläufen.

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.

A.4.5 Fehlermeldungen

Beim Kommandozeilen-Aufruf können folgende Fehlermeldungen auftreten:

Nr.	Meldung	Hinweis
5000	Fatal Error: MFC initialization failed	Die Initialisierung des Microsoft-Programmpakets MFC ist fehlgeschlagen.
5001	Error Parsing Command line	Fehler beim Lesen der Kommandozeile.
5002	Invalid License	Lizenzschlüssel wurde nicht eingegeben oder der vorhandene Lizenzschlüssel ist ungültig für diesen Kommandozeilen-Aufruf.
5003	Wrong License key	Der Lizenzschlüssel im Kommandozeilen-Aufruf ist ungültig.[a]
5004	Translate mode is missing	Weder Schalter `/M` noch `/L` ist angegeben.
5005	Translate mode is wrong	Ungültiger Schalter ist angegeben.
5006	Error opening Makefile	Fehler beim Öffnen des `makefile`.
5007	Only 3 processes are allowed	Bei *TiCo*-Prozessor sind maximal 3 Prozesse erlaubt.
5008	LP path is missing	Library-Pfad fehlt.
5009	IP path is missing	Include-Pfad fehlt.
5010	Invalid Option: <options>	Ungültige Verwendung von Optionen.

a. Derzeit kann kein Lizenzschlüssel in der Kommandozeile angegeben werden.

A.5 Obsolete Programmteile

Die Entwicklungsumgebung enthält aus Gründen der Kompatibilität auch Einstellmöglichkeiten für *ADwin*-Systeme mit Transputer-Prozessoren (T4, T5, T8).

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 unten gezeigte Dialogfenster geöffnet.

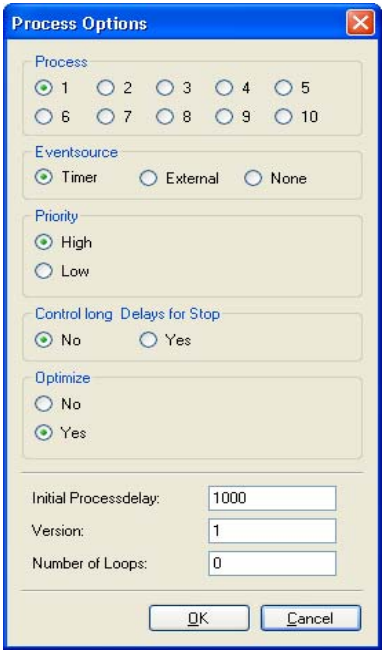


Abb. 1 – Dialogfenster „`Process Options`“ für Prozessoren T4…T8

– „`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 er im System laufen soll. Weitere Informationen zu diesem Thema finden Sie in Kapitel 6.1 „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 Liste der Debug-Fehlermeldungen

Debug-Fehlermeldungen können angezeigt werden, wenn in *ADbasic* die Option `Debug mode` aktiv ist; siehe Option Debug mode, Seite 69.

Fehlermeldung
`Division durch Null`
`Wurzel aus negativer Zahl`

`Data n: Index ist zu groß` / `Data n: Index ist kleiner als 1`
`Array index ist zu groß` / `Array index ist kleiner als 1`
Zugriff auf nicht deklarierte Elemente eines lokalen oder globalen Felds, d.h. auf eine zu große oder zu kleine Elementnummer.
Der Zusatz `(inc)` in der Fehlermeldung heißt, dass auf einen Befehl einer System-Include-Datei im o.g. Sinn fehlerhaft zugegriffen wurde.

`FIFO-Index ist kein FIFO` Das Feld mit dieser Nummer ist nicht als FIFO oder gar nicht deklariert.
`Adresse des Pro II Moduls ist >15 oder <1`
`P2_Burst_xxx`[a]`: "startadr" ist nicht durch 4 teilbar`
`P2_Burst_xxx`[a]`: Anzahl der Werte ist nicht durch 4 teilbar`
`P2_Burst_Init: Anzahl der Werte ist nicht durch 4 / 8 teilbar`
`P2_Burst_Read_Unpacked1: Anzahl der Werte ist nicht durch 8 teilbar`
`P2_Burst_Read_Unpacked2: Anzahl der Werte ist nicht durch 4 teilbar`
`P2_Burst_Read_Unpacked8: Anzahl der Werte ist nicht durch 2 teilbar`
`P2_Burst_Read: Anzahl der Werte ist kleiner als 1 / als 4`

Fehlermeldung
`P2_Burst_(C)Read_xxx: Die bei P2_Burst_Init eingestellte Kanalzahl passt nicht zu diesem Lesebefehl.` Verwenden Sie einen Lesebefehl, der die Daten für die eingestellte Anzahl von Kanälen liest.
`P2_GetData/SetData_Long: TiCo DATA existiert nicht`
`P2_GetData/SetData_Long: TiCo DATA hat falschen Datentyp`
`P2_GetData/SetData_Long: TiCo DATA-Index zu groß`
`P2_GetData/SetData_Long: TiCo DATA-Index < 1`
`P2_Digout_FIFO_Write: Zeitstempel-Differenz < 2`
P2_Get/Set_TiCo_Ringbuffer: TiCo Ringbuffer ist deklariert als READ und WRITE
`P2_TDRV_Init: Kein TiCo vorhanden, kein TiCo Prg. geladen oder ungültige Daten im TiCo Speicher`

`Media_Read / Media_Write:`
`start_block + count_blocks128 > num_blocks`
`start_block < 0`
Zugriff auf ungültige Bereiche des Speichermediums, d.h. auf eine zu große oder zu kleine Blocknummer.

`P2_Seq_Read:` `Eine ungerade Anzahl zu lesender Messwerte ist nicht erlaubt`

a. Gilt für **P2_Burst_Init**, **P2_Burst_Read**, **P2_Burst_Write**

A.7 Index

Symbole

- · 178
· 183
#Begin_Debug_Mode_Disable · 196
#Define · 208
#Else · 241
#End_Debug_Mode_Disable · 216
#EndIf · 241
#If · 241
#Include · 246
* · 179
+ · 175
+ (String) · 176
.NET · 172
/ · 180
: · 184
< = > · 186
= · 185
^ · 181
’ (Rem) · 292
150h, siehe Device No
2-dimensionale Felder · 126

A

Abbruch, siehe Prozess beenden
AbsF · 187
AbsI · 188
Absolutwert
  Fließkomma-Zahlen · 187
  Ganze Zahlen · 188
ActiveX · 172
  aus Entwicklungsumgebung nutzen · 172
  Kommunikation zum ADwin-System · 171
ADbasic
  ADsim · 100
  Demo-Modus · 10
  Lizenzvertrag · 5
  starten · 9
  Verzeichnisse einstellen · 65
ADbasic 5:Umstellungen · 5
ADbasic, Lizenzschlüssel · 10
ADbasicCompiler · 9
Add Open Files to Project · 76
Add to Project
  Kontextmenü · 18
  Projektfenster · 76
Addition
  Strings · 176
  Werte · 175
ADsim
  Ablauf mit T12 · 100
  Echtzeitprozesse · 4
  Kommunikation mit Simulink-Modell · 171
  Lizenzschlüssel · 10
  Menü · 55
ADtools
  Leiste einstellen · 66
ADtools · 98
ADWIN_CARD · 241
ADWIN_GOLD · 241
ADWIN_GOLDII · 241
ADWIN_L16 · 241
ADWIN_PRO · 241
ADWIN_PROII · 241
ADWIN_SYSTEM · 241
ADwin32.dll · 171
ADwin64.dll · 171
ADwin-Bootloader (T12.1) · 44
Align comments · 62
Analyse
  Allgemein · 152
  Laufzeitfehler · 152
  Zeitverhalten · 153

And · 189
Anhalten, siehe Prozess beenden
Anmerkungen, siehe Kommentar
Anzahl von Prozessen · 154
Anzeige
  aktuelle Informationen · 15
  Auslastung T12: CPU, CM, UM · 84
  Auslastung: CPU, PM, EM, DM, DX · 84
  Befehlsparameter · 41
  Prozessoptionen · 13
  ToDo-Liste · 91
Anzeigegenauigkeit, Fließkommazahlen · 65
Arcus-Cosinus: ArcCos · 191
Arcus-Sinus: ArcSin · 192
Arcus-Tangens: ArcTan · 193
Arcus-Tangens: ArcTan2 · 194
Arithmetische Funktionen
  - · 178
  * · 179
  + · 175
  / · 180
  ^ · 181
  Dec · 207
  Exp · 220
  Inc · 245
  LN · 262
  Log · 265
  Sqrt · 305
Array-Index (lokal) zu groß / <1, siehe Laufzeitfehler, erkennen
Arrays, siehe Felder
(Dim) As · 210
Asc · 195
ASCII-Zeichensatz · 4
(Dim …) At · 210
Aufrufgraph · 85
  anzeigen · 72
  erzeugen / aktualisieren · 72
Auslastung
  100%, wegen Speicherfragmentierung · 162
  Anzeige · 84
  Definition · 167
  Einfluss der Prozess-Anzahl · 154
auswerten
  Operatoren · 134
Autoindent · 62
Automatisch
  Einrücken · 22
  Formatieren · 22
  vervollständigen, Befehle und Variablen · 39
Automatische Typkonvertierung · 136
AutoSave · 54
Autostart · 54

B

Backslash im String, Escape-Sequenz · 133
Basis e · 220
Bearbeitungszeit messen · 143
Bedienoberfläche · 12
Bedingter Sprung
  If … Then · 239
  SelectCase · 296
Befehl
  Automatisch vervollständigen · 39
  Deklaration anzeigen · 42
  selbst dokumentieren · 23
  Übergabeparameter anzeigen · 41
  zur Deklaration springen · 37
Befehlsreferenz · 173
Befehls-Separator (:) · 184

Befehlszeile
Groß-/Kleinschreibung · 103
max. Zeilenlänge · 103
Begin_Debug_Mode_Disable · 196
benutzerdefinierte Befehle / Variablen
dokumentieren · 23
Übersicht · 106
Berechnungsausdrücke · 134
auswerten · 134
getrennte Auswertung · 137
symbolische Namen · 106
Typkonvertierung · 135
Betriebssystem
laden, siehe Booten
Standardverzeichnis · 12
Verzeichnis einstellen · 65
Bibliothek
Allgemeines · 139
Ein-/ausfalten · 27
Einbinden · 243
erlaubte Zeichen im Namen · 106
Erzeugen
aus ADbasic · 55
erzeugen · 54
aus Kommandozeile · 9
Funktion · 252
kontra Makro · 141
Position im Programm · 107
Rekursion verboten · 140
Unterprogramm · 257
Verzeichnis einstellen · 65
Binärdatei
erzeugen · 54
aus ADbasic · 55
aus Kommandozeile · 9
nutzen in Entwicklungsumgebung · 172
siehe auch Bibliothek
Binäre Schreibweise · 113
Bit umschalten / maskieren · 114
Bitmaske · 114
Bitmuster
mit ~ arbeiten · 114
von Fließkomma-Zahlen · 112
Bits verschieben
nach links · 299
nach rechts · 300
Bookmark · 36
Booten · 11
Bootloader
aktivieren, T12.1 · 46
Menüeintrag · 73
programmieren, T12.1 · 44
Prozesse rücklesen, T12.1 · 46
Bootloader (bis T12) · 159
Bootloader, T12.1 · 44
BTL-Datei: Verzeichnis einstellen · 65
Busy-Anzeige · 84

C

C#.NET, C++ · 172
Cacheable, Speicher · 124
Call Tree · 85
call tree
anzeigen · 72
Aufrufgraph erzeugen / aktualisieren · 72
Carriage Return im String, Escape-Sequenz · 133
Case, CCase, CaseElse (Select-Case …) · 296
Cast_Float32ToLong · 199
Cast_FloatToLong · 197
Cast_LongToFloat · 198
Cast_LongToFloat32 · 200
Chr · 201
Clear Parameter Scan · 42
CM
Dim · 210
Speicherbereich

Code size · 89
code snippets · 40
Column mark · 62
Comment Block · 23
Compiler
Aufruf · 54
AutoSave · 54
Compilermeldung, Fehler / Status · 89
Fehlermeldung · 54
Kommandozeilen-Aufruf · 9
Optionen einstellen · 56
Prebuild, Postbuild · 67
Compiler-Anweisungen
#Begin_Debug_Mode_Disable · 196
#Define · 208
#End_Debug_Mode_Disable · 216
#If … Then · 241
#Include · 246
Überblick · 183
Control block
im Kontextmenü · 18
markieren · 36
Cosinus: Cos · 202
CPU_Sleep · 203
Cursor-Position · 84

D

Data_n
dimensionieren · 210
Globale Felder · 117
Globale Felder, 2-dimensional · 126
Übersicht · 205
Data-Index (global) zu groß / <1, siehe Laufzeitfehler, erkennen
Dateiname
Bibliothek · 55
Binärdatei · 55
Datenaustausch
mit dem PC · 171
mit der Entwicklungsumgebung · 172
mit Simulink-Modell · 171
zwischen Prozessen · 170
Datenspeicher
siehe auch Speicher
2-Dim. Felder im ~ · 126
intern (DM) · 123
Übersicht, intern, extern · 122
Zusatzbedarf durch
Debug-Modus · 152
Timing-Modus · 155
Datenstrukturen
FIFO · 127
Globale Felder · 117
Speicherfragmentierung · 162
Globale Felder, 2-dimensional · 126
Globale Variablen · 116
Lokale Variablen und Felder · 120
Übersicht · 109
Datentypen
String · 129
Typkonvertierung · 135
Übersicht · 110
Datenverlust
beim Booten · 11
FIFO · 128
Datenwort: Nummerierung von Bits · 2

Debug
#Begin_Debug_Mode_Disable · 196
#End_Debug_Mode_Disable · 216
Allgemein · 152
Debug-Modus anwenden · 152
Menü · 69
Timing-Fenster · 91
Timing-Modus aktivieren · 69
Timing-Modus anwenden · 153
Debug errors · 69
Debug mode · 69
Dec · 207
Declarations · 96
DEFINE, siehe #Define
Dekadischer Logarithmus · 265
Deklaration
alle anzeigen · 42, 96
anzeigen · 42
siehe Dimensionierung
zur ~ springen · 37
Dekrementieren · 207
Delphi · 172
Demo-Modus · 10
Deutsch · 65
Device No
Definition · 172
einstellen · 58
Dezimale Schreibweise · 113
Dezimaltrennzeichen · 113
Dim · 210
Dimensionierung
Befehl Dim · 210
Position im Programm · 106
Speicherbereich · 121
Directory siehe Verzeichnis
Division
durch 2 · 300
einfache · 180
ganzzahliger Rest · 334
Division durch Null, siehe Laufzeitfehler, erkennen
DM, siehe Datenspeicher
DM_Local (Dim) · 210
Do … Until · 214
dokumentieren, Befehle / Variablen · 23
DRAM_Extern
Dim · 210
Event · 217
Finish · 227
Init · 248
LowInit · 266
siehe Externer Speicher · 123
Druckeinstellungen · 64
DX, siehe Externer Speicher

E

Editor
General · 62
Print Settings · 64
Syntax Colors · 63
Editor-Leiste · 19
e-Funktion: Exp · 220
eigene Befehle / Variablen dokumentieren · 23
Ein-/ausfalten, Textbereich · 27
Einbinden
Include-Datei · 246
Library · 243
Einrücken
Kommentar · 22
Programmabschnitte · 62
Textzeilen · 22
Else (If …) · 239
EM, siehe Zusatzspeicher

EM_Local
Dim · 210
Event · 217
Finish · 227
Init · 248
LowInit · 266
End · 215
End_Debug_Mode_Disable, siehe #End_Debug_Mode_Disable
EndFunction · 235
EndIf (If …) · 239
ENDREGION · 26
EndSelect (SelectCase …) · 296
EndSub · 323
English · 65
Entwicklungsumgebung
Leisten und Fenster · 12
starten · 9
Tastaturkürzel · 1
erlaubte Zeichen im Namen
Makros, Libraries · 106
Ersetzen
Beispiele · 33
Reguläre Ausdrücke · 34
Text · 28
Escape-Sequenz · 132
Event
externes Signal: Aufruf · 159
externes Signal: löschen · 293
Signalquelle einstellen · 60
verlorenes Signal
ein Prozess zeitgesteuert · 168
extern gesteuerter Prozess · 169
mehrere Prozesse zeitgesteuert · 169
prüfen · 93
T12, mehrere Prozesse zeitgesteuert · 168
Zeitdifferenz messen · 143
Event: · 217
Exit · 219
Exklusiv-ODER-Verknüpfung · 332
Exponential-Funktion Exp · 220
Exponential-Schreibweise · 113
Externer Speicher
DX · 123
SDRAM · 122
SX · 123
T12/T12.1 · 124
externes Event-Signal · 159
Extremwert
Maximum Fließkomma-Zahlen · 268
Maximum Ganze Zahlen · 270
Minimum Fließkomma-Zahlen · 269
Minimum Ganze Zahlen · 271

F

F1: Hilfe aufrufen · 17
Faltbaren Textbereich definieren · 26
Falten, Textbereiche · 27
Farbe einstellen · 63
Farbige Darstellung · 21
fast, Optimierungslevel · 60
Fehler
durch Cut&Paste erzeugt · 53
geringere Optimierung einstellen · 60
Fehlermeldung
des Compilers · 89
Laufzeitfehler · 69
Time out · 162

Felder
2-dimensional · 126
Data_n · 205
erlaubte Zeichen im Namen · 120
FIFO · 221
globale · 117
erstes Element · 118
verwendete anzeigen · 42
initialisieren · 106
kopieren · 272
lokale · 120
erstes Element · 121
Speicherbereich festlegen · 121
Übersicht · 108
Fenster
Call Tree · 85
Compiler Options · 56
Debug errors · 69
Declarations · 96
Global Variables · 95
Info-Bereich · 89
Info-Fenster · 89
Parameter · 79
Process Options · 59
Project Settings · 66
Beschreibung · 67
General · 67
Prebuild, Postbuild · 67
Projekt · 76
Quelltext · 83
Quelltext-Statusleiste · 13
Stack Test Analysis · 87
Statusleiste · 84
Timing Analyzer · 91
ToDo-Liste · 91
Toolbox · 76
Übersicht · 12
Quelltext-Informationen · 13
FFT
FFT · 338
FFT_32 · 354
FFT_Calc · 348
FFT_Calc_32 · 364
FFT_Calc_DM · 350
FFT_Calc_DX · 352
FFT_Init · 347
FFT_Init_32 · 363
FFT_Mag · 342
FFT_Mag_32 · 358
FFT_Mag_Scale · 346
FFT_Mag_Scale_32 · 362
FFT_Phase · 344
FFT_Phase_32 · 360
FFT_Scale · 340
FFT_Scale_32 · 356
FIFO
Aufbau der Datenstruktur · 127
belegte Elemente abfragen · 226
Datenverlust · 128
dimensionieren · 210
Elementanzahl prüfen · 129
freie Elemente abfragen · 225
initialisieren · 223
Übersicht · 221
Zugriff mehrfach · 128
FIFO_Clear · 223
FIFO_Empty · 225
FIFO_Full · 226
Finden · 28
Beispiele · 32
Deklaration von
Befehl/Variable · 37
Reguläre Ausdrücke · 34
Finish: · 227

Fließkomma-Zahlen
Anzeigegenauigkeit im Parameterfenster · 65
Bitmuster · 112
Dezimale Schreibweise · 113
Exponential-Schreibweise · 113
Flo40ToStr · 231
FloToStr · 229
Font
Farbe einstellen · 63
Größe einstellen · 62
Font size · 62
For … Next · 233
formatieren, Text im Editor · 22
Fourier transformation, siehe FFT
FPar
globale Variablen · 116
Fragmentierung, Speicher · 162
Function · 235
Funktion
Allgemeines zu Bibliotheken · 139
Allgemeines zu Makros · 138
Bibliothek (Lib_Function) · 252
dokumentieren · 23
erlaubte Zeichen im Namen · 106
Makro · 235
Position im Programm · 107

G

Ganze Zahlen
Binäre Schreibweise · 113
Hexadezimale Schreibweise · 113
Typkonvertierung · 135
ganzzahliger Rest bei Division · 334
Gerätenummer, Definition · 172
gleich = · 186
Global Variables, Fenster · 95
Globaldelay · 284
globale Felder, siehe Felder, globale
globale Variablen, siehe Variablen, globale
Goto Line · 37
Groß-/Kleinschreibung · 16
größer als >, >= · 186
Gutter · 62

H

Halt, siehe Prozess beenden
Hardware-Zugriff
ADwin-Hardware · 105
Lesen · 282
Schreiben · 283
Header · 64
hervorheben, Syntax · 21
Hexadezimale Schreibweise · 113
highlighting, Syntax · 21
Hilfe aufrufen · 17
Hilfsprogramme (*ADtools*) · 98

I

If · 239
siehe auch #If · 241
Import · 243
Inc · 245
Include · 246
Include-Datei
Allgemeines · 139
einbinden · 246
Rekursion verboten · 139
Syntax hervorheben · 139
Verzeichnis einstellen · 65
Indent
manuell / automatisch · 22
Option ADbasic sections · 62
#INF (Fließkomma-Zahlen) · 112
Info-Bereich · 89
Info-Fenster · 89

Init: · 248
Initialisierung · 105
  Booten · 11
  umfangreiche · 105
Inkrementieren · 245
Installation,
  Standardverzeichnis · 12
INTEGER · 111
Interner Speicher
  Datenspeicher (DM) · 123
  Gesamt (SRAM) · 122
  Programm (PM) · 123
  T12/T12.1 · 124
  Zusatzspeicher (EM) · 123
IO_Sleep · 250

J

Java · 172
Jump forward/backward · 38
Jump to Declaration · 37

K

kleiner als <, <= · 186
Kommandozeilen-Aufruf · 9
Kommentar
  positionieren · 22
  Syntax · 292
  Zeilen in ~ ändern · 23
Kommunikation
  mit dem PC · 171
  mit der
    Entwicklungsumgebung · 172
  mit Simulink-Modell · 171
  Prozess im ADwin-System · 162
  Time out · 162
  zwischen Prozessen · 170
kompilieren siehe Compiler
Konstante · 106
Kontextmenü · 18
  Projektfenster · 76
Kontrollstrukturen · 138
Konvertierung, Datentyp · 135

L

Language · 65
latency (Timing-Fenster) · 92
Laufzeitfehler
  Anzeige · 69
  erkennen · 152
  siehe auch Debug-Modus
length (Timing-Fenster) · 92
Lib_EndFunction · 252
Lib_EndSub · 257
Lib_Function · 252
Lib_Sub · 257
Library
  dokumentieren · 23
  Funktion · 252
  Import · 243
  siehe auch Bibliothek
  Unterprogramm · 257
Lib-Verzeichnis einstellen · 65
License key eingeben · 10
Line Feed im String, Escape-
  Sequenz · 133
Line numbers · 62
Lizenzschlüssel eingeben · 10
Lizenzvertrag · 5
LN · 262
LngToStr · 263
Log · 265
Logarithmus
  dekadischer · 265
  natürlicher · 262
Logische Funktionen
  And · 189
  Not · 275
  Or · 276
  Shift_Left · 299
  Shift_Right · 300
  XOr · 332

Long
Typkonvertierung · 135
lost events (T12) · 167
LowInit: · 266

M

Make Bin File, Make Lib File · 54
Makro
Allgemeines · 138
dokumentieren · 23
Ein-/ausfalten · 27
erlaubte Zeichen im Namen · 106
Funktion · 235
kontra Bibliothek · 141
Position im Programm · 107
Mark Controlblock · 36
maskieren, Bitmuster · 114
Matlab · 172
Lizenzschlüssel · 10
Matrix, 2-dimensional · 126
Max_Float · 268
Max_Long · 270
Maximum
Fließkomma-Zahlen · 268
Ganze Zahlen · 270
MemCpy · 272
Menü
ADsim · 55
auswählen · 13
Build · 54
Debug · 69
Edit · 53
File · 52
Help · 74
Leiste · 51
Options · 56
Tools · 73
View · 53
Window · 74
Menüleiste · 51
Messwertverlauf darstellen · 98
Metazeichen · 34
Min_Float · 269
Min_Long · 271
Minimum
Fließkomma-Zahlen · 269
Ganze Zahlen · 271
Mod · 334
Mouse over · 62
Multiplikation
einfache · 179
mit 2 · 299

N

Nachkommastellen
abschneiden · 135
Namen
Felder und lokale Variablen · 120
Makros, Libraries · 106
Natürlicher Logarithmus · 262
negatives Vorzeichen · 135
Neues in ADbasic 6 · 5
Next (For …) · 233
NICHT · 275
niederpriore Prozesse beim
T11 · 166
NOP · 274
Not · 275
Nummerieren, Zeilen · 62

O

ODER-Verknüpfung · 276
Operatoren
And · 189
auswerten · 134
negatives Vorzeichen · 135
Or · 276
Priorität · 134
XOr · 332

Optimales Zeitverhalten
ein Prozess · 155
mehrere Prozesse · 153
Optimierung
Allgemein · 143
Bearbeitungszeit messen · 143
Konstanten statt Variablen · 146
Polynom schneller berechnen · 182
Registerzugriff · 146
schneller messen · 147
T11 Speicherzugriff · 151
Wartezeit einstellen · 147
Wartezeit nutzen · 150
Optionen einstellen
ADtools · 66
Allgemein (Settings) · 62
Compiler · 56
Drucken · 64
Editor · 62
General · 65
Projekt · 66
allgemein · 67
Beschreibung · 67
Prebuild, Postbuild · 67
Prozess · 59
Sprache · 65
strukturierte Befehlsdarstellung · 63
Verzeichnisse · 65
Or · 276
Ordner siehe Verzeichnis
Outdent · 22

P

P1_Sleep · 278
P2_Sleep · 280
Par_n, globale Variablen · 116
Parameter Scan · 42
Parameter, siehe Variablen, globale
Parameterfenster · 79
Anzeigegenauigkeit FPar · 65
Parse and format · 62
Parse declarations · 62
Peek · 282
PM, siehe Programmspeicher
PM_Local
Event · 217
Finish · 227
Init · 248
LowInit · 266
Poke · 283
Polynom, schneller berechnen · 182
Positionieren, Kommentar · 22
Postbuild · 67
Potenz · 181
in Polynom ersetzen · 182
zur Basis e · 220
Präprozessor-Anweisungen
#Begin_Debug_Mode_Disable · 196
#Define · 208
#End_Debug_Mode_Disable · 216
#If … Then · 241
#Include · 246
Überblick · 183
Prebuild · 67
Print layout · 64
Priorität
niederpriore Prozesse mit T11 · 166
Operatoren · 134
Prozess, siehe Prozess, Priorität
von Prozessen prüfen · 154
Probleme, langsamer Editor · 62
Process_Error · 287
Process_Running · 288

Processdelay
Syntax · 284
Zeitverhalten · 164
Processor · 241
Programm verbessern, siehe Optimierung
Programmabschnitte
Event: · 105
Finish: · 105
Init: · 105
LowInit: · 105
Übersicht · 105
programmieren
ADwin-Bootloader (T12.1) · 44
Programmspeicher
PM · 123
T12/T12.1 · 124
Zusatzbedarf durch
Debug-Modus · 152
Timing-Modus · 155
Programmstruktur
Übersicht · 138
Bibliothek
Lib_Function · 252
Lib_Sub · 257
Übersicht · 139
Ein-/ausfalten · 27
Include-Datei · 139
Kommentar Rem · 292
Makros
Funktion Function · 235
Übersicht · 138
Unterprogramm Sub · 323
Schleife
Do … Until · 214
For … Next · 233
Sprung
If … Then · 239
SelectCase · 296
Projekt
Alle Quelltextdateien übersetzen · 55
Optionen
einstellen · 66
allgemein · 67
Beschreibung · 67
Prebuild, Postbuild · 67
Projektfenster · 76
Projektverwaltung
Allgemeines · 47
Fenster · 76
verwendete Variablen anzeigen · 42
Prozess
Anzahl · 160
Bearbeitungszeit · 165
beenden
andere · 310
sich selbst (in Event:) · 215
sich selbst (in Init:, LowInit:,

Finish:) · 219
Betriebszustände beim Zeitverhalten · 168
Fehler auslesen · 287
Kommunikation zwischen ~en · 170
Kommunikationsprozess · 162
mehrere · 165
neu laden · 162
Nummer · 160
Optimales Zeitverhalten
ein Prozess · 155
mehrere Prozesse · 153
optimieren, siehe Optimierung
Optionen
einstellen · 59
Optionen, Anzeige · 13
Priorität
Hoch · 161
Kommunikation · 162
Niedrig · 161
Niedrig mit T11 · 166
Übersicht · 160
prüfen auf Anzahl, Priorität · 154
Speichernutzung · 162
Standardprozesse 11, 12 · 161
starten
anderen Prozess · 306
gleichen Prozess erneut · 294
verzögert · 308
Status ermitteln · 288
Zeitverhalten · 164
Zyklus, siehe Prozesszyklus
Prozessor, siehe Processor · 241
Prozess-Steuerung
End · 215
Exit · 219
Process_Error · 287
ProcessN_Running · 288
Reset_Event · 293
Restart_Process · 294
Start_Process · 306
Start_Process_Delayed · 308
Stop_Process · 310
Prozesszyklus
Aufruf
mit Event-Signal · 159
Zeiteinheit · 164
zeitlich exakter Aufruf · 165
Punkt vor Strich, siehe Operatoren

Q

QNaN (Fließkomma-Zahlen) · 112
Quadratwurzel · 305
Quelltext
Editor-Fenster · 83
erstellen · 16
farbig darstellen · 21
formatieren · 21
im Projekt verwenden · 76
Reiter verschieben · 16
ToDo-Liste · 91
übersetzen · 20
Quelltext-Statusleiste · 13
Quick info · 62

R

Read_Timer · 289
Read_Timer_Sync · 291

Rechenzeit sparen
Konstanten statt Variablen · 146
Registerzugriff · 146
schneller messen · 147
Wartezeit einstellen · 147
Wartezeit nutzen · 150
redo · 19
REGION · 26
Registerzugriff · 146
Reguläre Ausdrücke · 34
Reiter, Reihenfolge verändern · 16
Rekursion bei Bibliothek · 140
Rekursion bei Include-Datei · 139
Rem · 292
Reset_Event · 293
Rest, ganzzahliger bei Division · 334
Restart_Process · 294
Ringbuffer · 127
Ringspeicher · 127
Round · 295

S

Save All Files of Project · 76
(Bits) schieben
nach links · 299
nach rechts · 300
Schleife, Do … Until · 214
Schreibweise von Zahlen · 113
Schriftart einstellen · 63
SDRAM, siehe Externer Speicher, SDRAM
SelectCase · 296
Separator : · 184
Shift_Left · 299
Shift_Right · 300
SHORT · 111
Short-Cuts · 1
Show Declarations · 42
Show line numbers · 62
Simulink
Ablauf mit T12 · 100
Echtzeitprozesse · 4
Kommunikation mit Modell · 171
Sinus: Sin · 302
size, Optimierungslevel · 60
P1_Sleep · 278
P2_Sleep · 280
Sleep · 303
Smart format · 22
SNaN (Fließkomma-Zahlen) · 112
snippets · 40
Speicher
Auslastung · 84
Bedarf bestimmen · 89
Bereich festlegen · 121
Bereiche CM, UM · 124
Bereiche PM, DM, EM, DX · 122
Fragmentierung · 162
siehe auch Datenspeicher
String · 130
Zusatzbedarf durch
Bibliotheken · 140
Debug-Modus · 152
Makros · 138
Timing-Modus · 155
Speicher (T12/T12.1) · 124
Speicher (T9..T11) · 122
Springen
zur Deklaration von Befehl/Variable · 37
springen
anderes Sprungziel · 38
zu Zeile · 37
Sprung, bedingter
If … Then · 239
SelectCase · 296
Sprungziel, wechseln zwischen · 38
Sqrt · 305
SRAM, siehe Interner Speicher, Gesamt (SRAM)

Stack size
bis T11 · 89
Stack Test
Analysis · 87
anzeigen · 72
Stack test
Option einstellen · 71
Start_Process · 306
Start_Process_Delayed · 308
Starten, ADbasic · 9
Statusleiste · 84
Statusmeldung Compiler · 89
Step (For …) · 233
Steuerzeichen in Strings · 132
Stop_Process · 310
Stopp, siehe Prozess beenden
StrComp · 314
String · 312
Element 1 mit T12 · 130
Escape-Sequenz · 132
nicht empfohlene Zuweisung · 133
Steuerzeichen · 132
Variablenaufbau · 130
Werte normal zuweisen · 130
String-Anweisung
Addition · 176
ASCII-Wert in Zeichen · 201
Dimensionierung · 312
Float in String · 229
Float in String (40 Bit) · 231
Länge eines Strings · 318
Long in String · 263
String in Float · 328
String in Long · 330
Teilstring
linker · 316
mittlerer · 319
rechter · 321
Vergleichen · 314
Zeichen in ASCII-Wert · 195
StrLeft · 316
StrLen · 318
StrMid · 319
StrRight · 321
Strukturieren
Ein-/ausfalten · 27
Farbige Darstellung · 21
Programmabschnitte · 138
Zeilen einrücken · 22
Sub · 323
Subtraktion · 178
Suchen
Beispiele · 32
Deklaration von Befehl/Variable · 37
Reguläre Ausdrücke · 34
schnell · 28
Text · 28
SX, siehe Externer Speicher
symbolische Namen · 106
Syntax
Colors · 63
Farbige Darstellung · 21
Systemvariablen
Process_Error · 287
Processdelay · 284
ProcessN_Running · 288
Übersicht · 119

T

T10, Bedingung mit #If PROCESSOR · 241
T11
Bedingung mit #If PROCESSOR · 241
niederpriore Prozesse · 166
Wartezeit einstellen · 148
T12
besondere Eigenschaften · 5
Simulink-Modell auf ADwin-Hardware · 100

T12.1
Bootloader · 44
T12/T12.1
Bearbeitungszeit messen · 145
Bedingung mit #If PROCESSOR · 241
For … Next-Schleife · 233
Speicherbereiche · 124
String, Element 1 · 130
Versionsnummer abfragen · 327
Zähler-Taktzyklus · 164
Zeitverhalten, mehrere Prozesse zeitgesteuert · 168
T9, Bedingung mit #If PROCESSOR · 241
Tabsize · 62
Tabulator
im String, Escape-Sequenz · 133
Schrittweite einstellen · 62
Tangens: Tan · 326
Tastatur
Anzeige von Einstellungen · 84
Kürzel · 1
Terminierung, siehe Prozess beenden
Testpoint · 172
Text
formatieren · 22
schnell suchen · 28
suchen und ersetzen · 28
Textbausteine · 40
Textmarke · 36
Textbausteine einfügen · 40
Textbereich ein-/ausfalten · 27
Textbereich, als faltbar definieren · 26
Textmarke · 36
Then (If …) · 239
TiCoBasic
Demo-Modus · 10
TiCoBasic, Lizenzschlüssel · 10
Time out · 162
Timer, siehe Zähler
Timer-Event · 159
Timing Analyzer · 91
Timing, siehe Zeitverhalten
Timing-Modus
aktivieren · 69
anwenden · 153
Fenster · 91
zusätzliche Prozessorzeit · 155
To (For …) · 233
ToDo-Liste · 91
Toolbox · 76
Tools
Bootloader · 73
Transputer, Einstellungen · 17
Trigonometrische Funktionen
ArcCos · 191
ArcSin · 192
ArcTan · 193
ArcTan2 · 194
Cos · 202
Sin · 302
Tan · 326
Typkonvertierung
ASCII-Wert in Zeichen · 201
automatische · 135
Float in Long (nur Datentyp) · 197
Float in String · 229
Float in String (40 Bit) · 231
Float32 in Long (nur Datentyp) · 199
Long in Float (nur Datentyp) · 198
Long in Float32 (nur Datentyp) · 200
Long in String · 263
String in Float · 328
String in Long · 330

U

Übergabeparameter anzeigen · 41

Überlastung des Prozessors · 167
UM
Speicherbereich
umschalten, Bit · 114
umsteigen auf ADbasic 6 · 5
Uncacheable, Speicher · 124
Uncomment Block · 23
undo · 19
UND-Verknüpfung · 189
unendlich (Fließkomma-Zahlen) · 112
ungleich <> · 186
Unmark Controlblock · 36
Unterprogramm
Allgemeines zu Bibliotheken · 139
Allgemeines zu Makros · 138
Bibliothek (Lib_Sub) · 257
dokumentieren · 23
erlaubte Zeichen im Namen · 106
Makro (Sub) · 323
Position im Programm · 107
Until (Do …) · 214
User_Version · 327

V

ValF · 328
ValI · 330
Variablen
Automatisch vervollständigen · 39
Deklaration anzeigen · 42
Deklaration finden · 37
dokumentieren · 23
globale · 116
Anzeige · 79
kopieren, große Anzahl · 272
verwendete anzeigen · 42
Werte hexadezimal anzeigen · 80
initialisieren · 106
Initialisierung beim Booten · 11
lokale · 120
erlaubte Zeichen im Namen · 120
Länge des Namens · 120
Speicherbereich festlegen · 121
symbolische Namen · 106
Übersicht · 108
vordefinierte Namen · 108
siehe auch Systemvariablen
Vergleich
< = > · 186
Strings · 314
vergleiche Makros mit Bibliotheken · 141
Verlauf der Sprungziele · 38
verlorene Zyklen (T12) · 167
Versionsverwaltung
Versionsnummer abfragen · 327
Versionsnummer einstellen · 61
Verzeichnis bei Standardinstallation · 12
Verzeichnisse einstellen · 65
Visual Basic · 172

W

Wagenrücklauf im String, Escape-Sequenz · 133
Warten
CPU_Sleep: Prozessor ab T11 · 203
IO_Sleep: Gold II · 250
P1_Sleep: Pro I-Bus · 278
P2_Sleep: Pro II-Bus · 280
Prozessor: NOP · 274
Sleep: Prozessor bis T10 · 303
Wartezeit genau einstellen · 147
Werkzeuge (*ADtools*) · 98

Werkzeugleiste · 13
Wertebereich · 110
Workspace size · 89
Wurzel · 305
aus negativer Zahl, siehe Laufzeitfehler, erkennen

X

XOr · 332

Z

Zahlenwerte, Schreibweise · 113
Zähler
auslesen · 289
interner, Taktzyklus · 164
Zeichenkette, siehe String
Zeile
einrücken · 22
formatieren · 22
max. Länge · 103
nummerieren · 62
zu ~ springen · 37
Zeilenkommentar einrücken · 22
Zeilenvorschub im String, Escape-Sequenz · 133
Zeit
exakter Aufruf · 165
Zeitdifferenz messen · 143
Zykluszeit · 164
Zeitdifferenz messen · 143
Zeitoptimierung, siehe Optimierung
Zeitverhalten
Änderung durch
Debug-Modus · 152
Timing-Modus · 155
Betriebszustände
allgemein · 168
ein Prozess
zeitgesteuert · 168
extern gesteuerter
Prozess · 169
mehrere Prozesse
zeitgesteuert · 169
T12, mehrere Prozesse
zeitgesteuert · 168
Informationen abfragen · 155
optimales
ein Prozess · 155
mehrere Prozessen · 153
prüfen, optimieren · 153
Zugriff auf FIFO, mehrfach · 128
Zusatzspeicher (EM) · 123
zuweisen, Zahlen · 113
Zuweisung (=) · 185

Symbole

< = > (Vergleich) 186
+ (Addition) 175
+ (String-Addition) 176
- (Subtraktion) 178
* (Multiplikation) 179
/ (Division) 180
^ (Potenz) 181
= (Zuweisung) 185
: Doppelpunkt 184
" " (String) 312
#Begin_Debug_Mode_Disable 196
#Define 208
#End_Debug_Mode_Disable 216
#If … Then … {#Else …} #EndIf 241
#Include 246
#…, Compiler-Anweisung 183

A-B

AbsF 187
AbsI 188
And 189
ArcCos 191
ArcSin 192
ArcTan 193
ArcTan2 194
Asc 195

C

Cast_Float32ToLong 199
Cast_FloatToLong 197
Cast_LongToFloat 198
Cast_LongToFloat32 200
Chr 201
Cos 202
CPU_Sleep 203

D

Data_n 205
Dec 207
Dim 210
Do … Until 214

E-F

End 215
Event: 217
Exit 219
Exp 220
FIFO 221
FIFO_Clear 223
FIFO_Empty 225
FIFO_Full 226
Finish: 227
Flo40ToStr 231
FloToStr 229
For … To … {Step …} Next 233
Function … EndFunction 235

G-J

If … Then … {Else …} EndIf 239
Import 243
Inc 245
Init: 248
IO_Sleep 250

K-L

Lib_Function … Lib_EndFunction 252
Lib_Sub … Lib_EndSub 257
LN 262
LngToStr 263
Log 265
LowInit: 266

M-O

Max_Float 268
Max_Long 270
Min_Float 269
Min_Long 271
Mod 334
NOP 274
Not 275
Or 276

P

P1_Sleep 278
P2_Sleep 280
Peek 282
Poke 283
Processdelay 284
ProcessN_Running 288
Process_Error 287

R

Read_Timer 289
Read_Timer_Sync 291
Rem 292
Reset_Event 293
Restart_Process 294
Round 295

S

SelectCase 296
Shift_Left 299
Shift_Right 300
Sin 302
Sleep 303
Sqrt 305
Start_Process 306
Start_Process_Delayed 308
Stop_Process 310
" " (String) 312
StrComp 314
StrLeft 316
StrLen 318
StrMid 319
StrRight 321
Sub … EndSub 323

T-Z

Tan 326
User_Version 327
ValF 328
ValI 330
XOr 332