ADbasic

Real-Time Development Tool for ADwin Systems

ADbasic Version 6.00 Feb. 2017

License Key:

For any questions, please don't hesitate to contact us:

Hotline: +49 6251 96320

Fax: +49 6251 568 19

E-Mail: info@ADwin.de

Internet www.ADwin.de



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



Table of contents

Table of contents	III
Conventions	2
1 Introduction	3
2 News in <i>ADbasic</i> 6	7
3.1 Basic Steps. 3.1.1 Starting the Development Environment 3.1.2 Check or change TiCoBasic licenses 3.1.3 Loading the ADwin Operating System 3.1.4 Basic Elements of the Development Environment 3.2 Creating source code 3.2.1 Calling online help 3.2.2 Context menu in source code window 3.2.3 Editor bar 3.3 Compiling source code 3.4 Formatting source code 3.4.1 Syntax highlighting 3.4.2 Smart formatting 3.4.3 Indenting text lines	9 9 10 11 16 16 18 19 20 21 23 23
3.4.4 Positioning comments	
3.4.6 Documenting self-defined instructions and variables	
3.4.7 Defining a foldable text range 3.4.8 Folding text ranges 3.5 Searching and replacing	. 26 . 27
3.5.1 Finding text quickly	. 29
3.5.2 Finding and replacing text	
Examples – Finding Text	
Examples – Replacing Text	
3.5.3 Regular expression	
3.5.4 Marking control blocks	
3.5.5 Using bookmarks	
3.5.6 Jumping to a program line	
3.5.7 Jumping to declaration of instruction or variable	. 38

Table of contents



3.5.8 Switching between jump targets	39
3.6 Writing programs with ease	
3.6.1 Autocomplete for instruction or variable	
3.6.2 Inserting code snippets	41
3.6.3 Displaying instruction parameters	42
3.6.4 Displaying declaration of instruction or variable	43
3.6.5 Displaying declarations of a file	43
3.6.6 Displaying used global variables and arrays	44
3.6.7 Adding file and folder shortcuts	
3.7 Finding Programming Errors	45
3.8 Managing Projects	
3.9 Menus	49
3.9.1 File Menu	50
3.9.2 Edit Menu	51
3.9.3 View Menu	51
3.9.4 Build Menu	52
3.9.5 Options Menu	54
Compiler Options dialog box	54
Process Options dialog box	56
Settings dialog box	60
Project Settings dialog box	64
3.9.6 Debug Menu	
Timing Analyzer Option	
Debug mode Option	
Option Stack Test	
Stack Test Analysis	
Create Call tree Info File	70
Show Call tree	
3.9.7 Tools Menu	71
3.9.8 Window Menu	72
3.9.9 Help Menu	
3.10 Windows	
3.10.1 Toolbox	73
3.10.2 Project Window	74
3.10.3 Parameter Window	
3.10.4 Process Window	78
3.10.5 Source code window	
3.10.6 Status Bar	
3.10.7 Call Graph Window	
3.10.8 Window Stack Test Analysis	
0.10.0 William Clack Test Maryolo	



3.11 Info range	86
3.11.1 Info window	86
3.11.2 To-Do List	88
3.11.3 Timing Analyzer Window	88
3.11.4 Global Variables Window	
3.11.5 Declarations Window	
3.12 ADtools	95
4 Programming Processes	. 97
4.1 Program Design	
4.1.1 The Program Sections	99
4.1.2 User defined instructions and variables	99
4.2 Variables and Arrays	. 101
4.2.1 Overview	
4.2.2 Data Structures	
4.2.3 Data Types	. 102
4.2.4 Entering Numerical Values	. 105
4.2.5 Global Variables (Parameters)	. 106
4.2.6 Global Arrays	. 107
4.2.7 System Variables	. 108
4.2.8 Local Variables and Arrays	. 109
4.3 Variables and Arrays – Details	. 110
4.3.1 Variables and Arrays in the Data Memory	
4.3.2 Memory Areas (T9T11)	
4.3.3 Memory Areas (T12)	. 113
4.3.4 2-dimensional Arrays	
4.3.5 Data Structure FIFO	
4.3.6 Strings	. 117
Normal Assignment	
Character Assignment via Escape Sequence	. 119
String Assignments that are NOT Recommended	
4.4 Expressions	. 121
4.4.1 Evaluation of Operators	
4.4.2 Type Conversion	
4.5 Selection structures, Loops and Modules	
4.5.1 Subroutine and Function Macros	
4.5.2 Include-Files	
4.5.3 Libraries	

Table of contents



5 Optimizing Processes	. 129
5.1 Measuring the Processing Time	129
5.1.1 Annotations for T12	130
5.2 Useful Information	131
5.2.1 Accessing Hardware Addresses	131
5.2.2 Constants instead of Variables	132
5.2.3 Faster Measurement Function	132
5.2.4 Setting Waiting Times Exactly	132
5.2.5 Using Waiting Times	135
5.2.6 Optimization with Processor T11	136
5.3 Debugging and Analysis	137
5.3.1 Finding Run-time Errors (Debug Mode)	137
5.3.2 Checking the Timing Characteristics (Timing Mode) .	137
Checking Number and Priority of Processes	138
Optimal Timing Characteristics of Processes	
5.3.3 Analyzing program structure	
5.3.4 Optimizing memory intensive procedures	141
6 Processes in the ADwin System	
6.1 Process Management	
6.1.1 Types of Processes	
6.1.2 Processes with High-Priority	
6.1.3 Processes with Low-Priority	
6.1.4 Communication Process	
6.1.5 Memory fragmentation	
6.2 Time Characteristics of Processes	
6.2.1 Processdelay	
6.2.2 Precise Timing of Process Cycles	
6.2.3 Low-Priority Processes with T11 and T12	
6.2.4 Workload of the <i>ADwin</i> system	
6.2.5 Different Operating Modes in the Operating System .	
6.3 Communication	
6.3.1 Data Exchange between Processes	153
6.3.2 Communication between PC and ADwin System	
6.3.3 The Device Number	
6.3.4 Communication with Development Environments	155



7 Instruction Reference	157
8 How to Solve Problems?	
Appendix	A-1
A.1 Short-Cuts in TiCoBasic	
A.2 ASCII-Character Set	
A.3 License Agreement	A-5
A.4 Command Line Calling	
A.5 Obsolete Program Parts	A-16
A.6 List of Debug Error messages	A-20
A.7 Index	A-22
A.8 Instructions in this manual	A-41





Dear Reader,

ADbasic 6 is the programming tool for your ADwin system that allows you to create special measurement, open-loop, or closed-loop control application. The purpose of this manual is to: introduce you to the basics of programming real-time processes for the ADwin system; and act as a reference manual.

It is presumed that you already have installed the *ADwin* system; if not please refer to the manual ADwin-Installation.

The development environment *ADbasic* 6 provides the familiar comfort for easy handling and editing and furthermore supports the new processor T12 (see also "News in ADbasic 6" on page 7).

The instruction reference contains the processor's calculation commands. Any instruction for access to input, outputs or interfaces are described in the appropriate manual of *ADwin* hardware.

First-time users of *ADbasic* are recommended to read chapters 1 and 4, in order to get easily into the subject. This manual assumes that the user has some programming experience with Basic or any other language. An introduction to the programming of *ADwin* systems and example programs can be found in our "*ADbasic* Tutorial and Programming Examples" manual.

chapter 3 describes the handling of the development environment and is recommended for all users.

If you have any suggestions on how to improve our documentation, do not hesitate to contact us. Your inputs will be greatly appreciated and will help us provide a system, which everyone can easily understand and operate.

We wish you great success upon programming.

For further questions, please, call our support hot-line (see address in the manual's cover page).



Conventions

In this manual, the following typographical conventions and icons are used:



This "attention" icon is located next to paragraphs with important information for correct function and error-free operation.



A note provides topics of interest and advice for an efficient operation.



The "information" icon refers to additional information in the manual or other sources (documentation, data sheets, literature etc.).



The light bulb icon denotes examples showing practicable solutions.

The Courier font-type is used for text displayed on screen, e.g in windows or menus, or input via the keyboard. The names of menus and submenus are shown similarly: Menu > submenu.

File names and path names are additionally emphasized as follows <path\xx.ext>.

Source code elements such as **Instructions**, variables, comments and any other text are displayed like the development environment editor does.

Key names are set in square brackets and in small capitals such as [RETURN] or [CTRL].

The bits of a data word (here 16-bit) are numbered through as follows:

Bit no.	15	14	13		01	00
Value of the bit	2 ¹⁵	2 ¹⁴	2 ¹³		2 ¹ =2	2 ⁰ =1
Name	MSB	-	-	-	-	LSB

Numbers not indicated in decimal notation have an identifying letter added, e.g. for the number 17:

Hexadecimal notation: 11h

Binary notation: 10001b



1 Introduction

The *ADwin* system is responsible for all time-critical tasks in fast dynamic test stands and industrial production facilities. For this task, the *ADwin* system is programmed with the *ADbasic* development tool.

To hit the target of an immediate and efficient start of programming, first of all we would like to explain the concept of the *ADwin* system.

Concept of the ADwin system

All *ADwin* systems have a central processing unit (CPU), which executes all time-critical tasks such as: measurement data acquisition, open-loop and closed-loop control, or online processing of measurement data in real-time. Analog and digital inputs and outputs as well as add-ons like counters and bus systems are connected to the test stand. Ethernet or USB set up the communication with a computer.

The processor of the *ADwin* system is programmed with the real-time development tool *ADbasic*, which enables easy construction of time critical real-time processes. *ADbasic* is an integrated development environment under Windows with capabilities of online debugging. The familiar BASIC command syntax has been expanded with more functions, which are used for accessing the inputs and outputs, controlling real-time processes, and preparing the data exchange with the computer. chapter 4 explains the design of *ADbasic* programs.

An ADbasic program with only a few lines can:



- Acquire measurement parameters up to sampling rates of 800kHz
- Develop fast digital controllers with sampling rates of up to 400kHz
- Simultaneously generate and measure analog signals, e.g. for dynamic measurement of a test stand characteristic

A user-defined hierarchy is responsible for the interaction and timing of the processes when several processes are needed for a complex algorithm. chapter 6 details the running of processes in the operating system.





Developing processes in ADbasic

Source code generated using the extended BASIC syntax of the *ADbasic* environment programs the hardware of your *ADwin* system enabling the implementation of tasks into processes. chapter 4 describes how to build programs.

Executable binary code, generated from the source code using the integrated compiler, is transferred to the *ADwin* system and tested. *ADbasic* is also a tool, which aids in process monitoring, error detection, and program optimization (see chapter 3).



Controlling an ADwin system from the PC

ADbasic is no longer needed once the real-time processes are running properly.

A user interface running on the computer transfers the generated binary code to the system, starts, controls, and stops the processes, and controls and monitors the processes and process data of the *ADwin* system.

Although the *ADwin* system operates independently of the computer, global variables and arrays are accessed through the user interface, without delaying time-critical processes.

A clear separation between real-time processes in the *ADwin* system and the user interface on the computer guarantees a high operating reliability and a good timing.

Under Windows, a DLL or ActiveX-interface enable access to the *ADwin* system from several programs simultaneously.

Based on this, drivers for .NET as well as for many development environments are available, which help in creating a user interface, e.g. Delphi, Visual-Basic, C#.NET, Visual-C++. Optionally, measurement packages such as Kallisté, TestPoint, LabVIEW, Diadem, HP-VEE, Intouch, and Matlab can be used.

Finally, there are also drivers for the platforms Linux, MacIntosh, and Java.

As an alternative to the program language *ADbasic*, you can create real-time processes for *ADwin* systems from Simulink[®]: using the program package *ADsim* you can have Simulink[®] models run on *ADwin* hardware. Then, model values and *ADwin* variables can be observed



and changed from the PC, either with the user interface *ADsimDesk* or with the *ADwin* drivers for many development environments.





2 News in ADbasic 6

The new feature of the real-time development environment *ADbasic 6* is the support of processor T12. All other functionality of the user interface refers to version 5.

Below you find the special features corresponding to processor T12:

- Technical data of T12:
 - XILINX ZYNQ™ with dual-core ARM Cortex-A9 Clock cycle 1 GHz
 - 1 Gigabyte memory for code and data
 - 1 Gigabit Ethernet interface
- Access to ADwin hardware: The processor module Pro II-T12 can only access modules via the Pro II bus, but not Pro I-modules.
- Memory: The memory management does no more distinguish program memory and data memory (see chapter 4.3.3). Instead, a "Cache"—a very fast buffer memory—accelerates the access to program and data.
 - Using the declaration Uncacheable, you can have the processor to not access selected arrays via Cache. This way, the Cache remains available for other data, especially if declared as Cacheable.
- Floating-point values: The processor can process floating-point values with 32-bit and 64-bit precision. Therefore, ADbasic provides the new data types Float32 and Float64 (see Data Types).
 - The previously used data type Float will be automatically replaced by Float64 by the compiler. Thus, you can use given programs without changes, and you benefit from the advantage of a higher computing precision than with T11 (40 bit).
 - Integer values (Long) are still processed with 32-bit precision.
- Strings: Each character requires only one Byte in memory; with previous processors, 4 bytes were required.
 You may no more access element 1 of a string (previously the element contained the string length). As before, you can still get the string length using StrLen.
- For-loops: The processor T12 checks the condition of a For-loop, before it processes the loop the first time, see For ... To ... {Step ...}
 Next.



Up to processor T11, the For-loop was executed at least once, even if the loop start value was greater than the end value.

 Time measuring: If you want to measure the timing of a program, please note that measuring of selected program parts depends much more on the surrounding conditions than with former processors.

It is still very easy to measure the length of a process cycle (Event section). You can also determine average values for the length of a program part; for exact values, you require a good understanding of how instructions are executed and of processor architecture to consider potential error influences.

You find more in chapter "Measuring the Processing Time", section Annotations for T12.

Observe Workload: Pay close attention to keep the processor workload (Busy in the Status Bar) well-spaced from 100%. Thus, you can make sure that each single process cycle is being processed with correct timing.

Different to previous processors the T12 can still run after temporary overload (see Workload of the ADwin system). However, this will only be useful as an exception, if your process can accept lost process cycles. You can then monitor the number of lost process cycles in the Process Window.

 Time-controlled Processes: The timing of several processes is now identical to the timing of a single process. The change refers to the behavior during processor overload only. See more under Single Time-Controlled Process.

For a single time-controlled process and an externally controlled process, the timing behavior remains unchanged.



3 Using the Development Environment

Processes for the *ADwin* systems are programmed quickly and easily with the *ADbasic* development environment. The *ADbasic* compiler works with an enlarged BASIC syntax and generates binary files, which may be executed and transferred to the *ADwin* system even without the development environment.

3.1 Basic Steps

3.1.1 Starting the Development Environment

To start the ADbasic development environment, do as follows:

1. Start the development environment by selecting Programs ADwin ADbasic from the Windows start menu.

The first start may last a few seconds until the environment shows up, since the Windows package .Net Framework is started, too.

The environment will appear with the Windows-specific elements such as windows, menu bar and tool bar.

 Upon first start-up, you will be prompted to enter the License key. The License key is to be found on the cover sheet of this ADbasic manual.

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

Without valid License key, ADbasic will operate in demo mode. In this mode, the development environment only works for demonstration, test or evaluation purposes. For example, you cannot create binary files.

Find more information about the *ADbasic* license in chapter 3.1.2 on page 10.

3. Set the *ADwin* system and processor in the menu Options\Compiler.



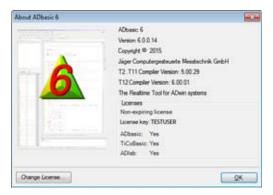
The development environment saves the settings so that upon a new start of *ADbasic* they will not need to be entered again, unless a different *ADwin* device is used.

3.1.2 Check or change ADbasic licenses

In order to check or change the ADbasic license key, do as follows:

1. Select the menu entry Help About.

The window About ADbasic opens, which displays the version of the development environment and the current Licenses (list of available licenses see below).



In order to enter or change the license key, click the button Change License.

The dialog window License key opens.

3. Enter your license key.

The License key is to be



found on the cover sheet of this *ADbasic* manual.

In ADbasic, the following licenses are available:

No license (demo mode)

Without valid License key, *ADbasic* will operate in demo mode. In this mode, the development environment works only



for demonstration, test or evaluation purposes. For example, you cannot create binary files.

Evaluation license (expiring by date)

The license enables all functions of the development environment for a fixed period. Afterwards, *ADbasic* will run in demo mode again (see above).

Non-expiring license of the Licensee

The following licenses can be enabled:

- · ADbasic 6, works with all ADwin processors
- ADbasic 5, works with ADwin processors up to version T11
- ADbasic 3.0, works with ADwin processors up to version T9
- ADbasic 2.0, works with ADwin processors up to version T8
- TiCoBasic
- ADIab (Matlab driver for ADwin)

The *TiCoBasic* and *ADlab* licenses can be combined with one of the *ADbasic* licenses.

The license conditions for *ADbasic* are described in the License Agreement (see annex, page A-5).

3.1.3 Loading the ADwin Operating System

The *ADwin* operating system is loaded to your *ADwin* system by clicking **B** (= boot).

The booting process must be repeated each time the *ADwin* system is powered up, after a power failure, or when the computer recognizes a communication error, which has interrupted the communication with the system.

The contents of the program and data memories on the ADwin system will be lost and all global parameters set to the value 0 when the operating system is booted.



An appropriate operating system for each processor type is needed and can be found in the corresponding file ADwin*.btl, (* stands for the processor type). The development environment uses the information from the Options \ Compiler menu setting to determine, which of the files to use during the boot process.



The files ADwin*.btl are saved during installation in the directory <C:\ADwin> (standard installation).

3.1.4 Basic Elements of the Development Environment

The development environment consists of several bars and windows (see fig. 2); the window dimensions may be individually adjusted.

Online help for a window or the currently marked key word is called with the key [F1]. The button opens the help index.



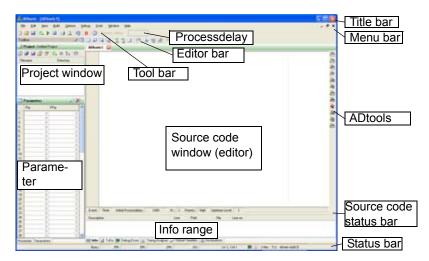


Fig. 2 – Elements of the *ADbasic* development environment



The functions of the development environment are called using:

- The tool bar and the editor bar (see fig. 3).
- The project bar in the Project Window
- The context menus of the windows (right mouse button).
- The menu bar.
- The Short-Cuts in TiCoBasic (see annex).
- The ADtools bar.
 You can add own shortcuts here (see chapter 3.6.7 on page 44).

While using a function, the function's description is shown to the left in the status bar.

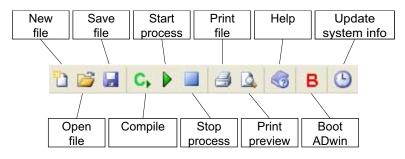


Fig. 3 – The tool bar

An instruction is selected when a menu entry is clicked with the left mouse button, or when the keys [ALT] + [FIRST LETTER] of the corresponding menu, are pressed. Some instructions have short-cuts (see Appendix A.1), which are displayed in the menus.

Each process is edited in its own source code window. Several windows may be opened at a time; the sizes of the windows can be individually adjusted. More information about the relevant source code window is displayed at various other locations:

- The title bar shows the names of the open source code window.
- The source code status bar displays the process options that have been set.



A right-click on the bar opens the Process Options dialog box.

- The global parameters used in the source code project are high-lighted in the Parameter Window (see chapter 3.10.3, page 76)
 by clicking Scan Global Variables; see Displaying used global variables and arrays on page 44.
- The info range at the bottom displays information in several windows:
 - Info window: The compiler's error messages (highlighted red) and warnings (see chapter 3.11.1 on page 86).
 - To-Do List: A simple To-Do list from comment lines (see chapter 3.11.2 on page 88).
 - Search results from a search in all files of a project (see chapter 3.5.2 on page 30).
 - Debug information if the debug mode is enabled (see Debug mode Option, page 67).
 - Timing Analyzer Window: Timing of running processes on the ADwin system (see chapter 3.11.3 on page 88).
 - Global Variables Window: A list of used global variables and arrays (see chapter 3.11.4 on page 92).
 - Declarations Window: A list of all displays all declarations, related to the source code file (see chapter 3.11.5 on page 93).

Please note: Editing in the source code window is supported by several tools (see Creating source code on page 16).

The Project Window shows the name of an opened project and the corresponding files; without project, the window remains empty.



Some data of the *ADwin* system are continuously read and displayed (only when PC communication to the *ADwin* system is established):

- Processdelay (process cycle time) of the process, which has the number as the currently edited source code. Displayed at the right side of the toolbar.
- The values of the global variables in the Parameter Window; a change to one of these values will immediately be transferred to the ADwin system.
- The status of running processes in the Process Window (page 78).
- Memory usage information in the Status Bar (see chapter 3.10.6 on page 80).



3.2 Creating source code

Open a new window for each process source code (using File) New or [CTRL]+[N]).

You can change the order of source code window tabs: press the [ALT] key, do a mouse click on the tab, and drag the tab to the new position.

If you use several files for your task, we recommend managing the files in a project file (see page 46: Managing Projects).

Editor and *ADbasic* compiler do not bother about upper or lower case letters. However, in the examples throughout this manual-for the purpose of better reading-a consistent notation is used.

Calling online help (see below) is a good idea when you need a guide for editing or programming.

The source code editor provides several useful tools. Call the tools via Context menu in source code window (page 18) or via Editor bar (page 19):

Numerical values may be entered into source code in decimal, hexadecimal, binary, and exponential notation (see also chapter 4.2.4 "Entering Numerical Values").

Find more editor functions here:

- Formatting source code, page 21
- Searching and replacing, page 29
- Writing programs with ease, page 40

3.2.1 Calling online help

The Help Menu (page 72) enables to call selected help pages, e.g. table of contents or sorted instruction lists.

Using [F1] opens a help page according to the currently opened dialog box or according to the instruction at cursor position.



If the cursor is set upon an invalid instruction, the help index shows up. Reasons may be:

- The text is not an instruction but a user-defined declaration:
 Variable / array, symbolic name, macro (Sub, Function). For a user define, a help page cannot be provided.
- The instruction is misspelled, e.g. Digin_Wrod instead of Digin_Word. After being corrected, the instruction will be highlighted correctly.
- The (user-defined) include or library file is missing where the instruction is defined. Please insert the appropriate line at the start of the source code.

The help window displays several register cards to the left and the called help page to the right. The register cards lead to the table of Contents, the Index, the Search window and to sorted Commands lists.

You can display several help pages side by side:

- Do a right mouse click on a link and select the entry Open in new window in the context menu. Then, the link opens in a new tab (sorry, Open in new tab is not available).
- Alternatively: Using the topright button
 Add new tab, you open any number of (initially empty) tabs. If you click any link, the link target will be opened in the leftmost empty tab. If all tabs are filled, the link target will be displayed in the active window.

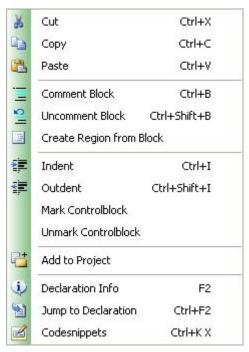
You can magnify many vector graphics in the online help (e.g. pinouts) to discover details by enlarging the help window with the frame handle.

Search inside the help accepts only the following characters: 0-9, A-Z, - (minus) and umlauts; search is not case sensitive. Use SPACE as delimiter for several words. With several words given, only those pages are found, which contain all of the words. Searching for whole words is optional.



3.2.2 Context menu in source code window

Various help functions are available from the context menu by rightclicking in the source code window.



The following functions use the cursor position or the active selection:

- Cut: Cut selection and copy into the clipboard.
- Copy: Copy selection into the clipboard.
- Paste: Delete selection and insert text from the clipboard.
- Comment Block, Uncomment Block: Changing lines into comment, page 23.
- Create Region from Block: Defining a foldable text range, page 26.
- Indent, Outdent: Indenting text lines, page 23.



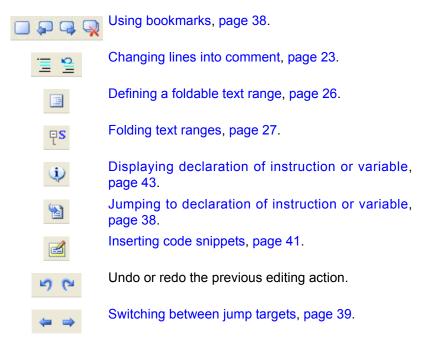
- Mark Control block, Unmark Control block: Marking control blocks, page 37.
- Declaration Info: Displaying declaration of instruction or variable, page 43.
- Jump to Declaration: Jumping to declaration of instruction or variable, page 38.

These functions are available without marking:

- Add to Project: Add a file to the project.
- Declaration Info: Displaying declarations of a file, page 43.
- Code snippets: Inserting code snippets, page 41.

3.2.3 Editor bar

The editor bar provides editor tools for use in the source code window.





3.3 Compiling source code

With *ADbasic*, you compile a source code into a binary file. The binary file can be transferred to the *ADwin* hardware and run as a process.

You can compile (and build) as follows.

- Test a source code while developing:
 - Compile and run a single source code file,

 Build Compile or button in the tool bar.
 - Compile and run selected project source code files, Button , in the project bar.
- Save tested source code as binary file:
 - Compile single source code and save as binary file, Build Make Bin File.
 - Compile selected project files and save as binary files,
 Button in the project bar.
 - Compile all project files and save as binary files,
 Button in the project bar.
- Run command lines before and after compiling.

Command lines are defined in the Project Settings dialog box, Tabs Prebuild / Postbuild (page 66).

You can use created binary files for different purposes:

- Transfer and run several binary files of a project from the environment ADbasic to the ADwin hardware: button in the project bar.
- Transfer, run, and control binary files from an external program to the *ADwin* hardware, see Communication with Development Environments.

There are interfaces for usual development environments (e.g. Java, Visual Basic, C#.NET) and numerous user interfaces (as DIAdem, MATLAB).

 Store binary files permanently in the bootloader of the ADwin hardware and run them automatically after power-up: see manual "ADwin-Bootlader".



In addition, you can create a binary file with library functions from a source code, see Libraries. Library functions can be imported and used in other source codes.

3.4 Formatting source code

Source code can be formatted (mostly automatically) to clearly show the program structure:

- Syntax highlighting, page 21
- Smart formatting, page 23
- Indenting text lines, page 23
- Positioning comments, page 23
- Changing lines into comment, page 23
- Documenting self-defined instructions and variables, page 24
- Defining a foldable text range, page 26
- Folding text ranges, page 27

Find more editor functions in the sections:

- Creating source code, page 16
- Searching and replacing, page 29
- Writing programs with ease, page 40

3.4.1 Syntax highlighting

Once a command line is written, the editor will automatically change the color of the instruction words, variable names and array names, while indenting the lines to give a clear structure.

The editor divides the character strings you have entered, into several groups of syntax elements being displayed differently. The color design may be changed under Options Settings, Editor – Syntax Colors (see page 61); the window also shows an overview of syntax groups.

Syntax highlighting requires an active option Parse Declarations under Editor – General (see page 60).



3.4.2 Smart formatting

Once a command line is written, the editor will automatically correct the number of spaces, thus giving the line a clear structure. This way e.g. operators like "=" or keywords like "If" will have a space to left and right.

If you like to format manually you have to switch off smart format under Editor – General, Smart format (see page 60).

3.4.3 Indenting text lines

Once a command line is written, the editor will automatically indent the lines to give a clear structure. Manual indenting is not available in combination with automatic indenting.

If you like to indent manually you have to switch off automatic indentation under Editor – General, AutoIndent. Afterwards, indents may be set with [TAB] or [SPACE]. Several marked lines may be indented or outdented by selecting Indent or Outdent in the source code context menu (right mouse click).

The menu entry Options Settings, Editor – General, Tabsize be used to set the number of spaces for one indent.

3.4.4 Positioning comments

If you add a comment with 'at the end of a line, the editor puts the start of the comment to a specified position. Thus, readability of comments will be enhanced.

If using double comment chars "you can position a comment manually even though automatic positioning is enabled.

You can enable or disable automatic positioning under Editor – General, Align comments as well as set the comment start position (see page 60).

3.4.5 Changing lines into comment

Marked lines may be changed into comment lines in one action by selecting the menu entry Comment Block from the source code context menu (right mouse click). The editor will then insert a comment char ' at every of the marked lines so the compiler will skip these lines.



In the same way, Uncomment Block will delete a comment char at the start of the lines.

3.4.6 Documenting self-defined instructions and variables

You can document self-defined instructions (Sub, Function), variables, arrays and symbolic names (#Define), so that the description text is shown in tooltips and with autocomplete while programming. Find more in the sections "Autocomplete for instruction or variable" (page 40), "Displaying instruction parameters" (page 42) and "Displaying declaration of instruction or variable" (page 43).

Example

```
The following examples show how to organize the description text:
```

```
''' This starts the description of the function my func;
'''+the first description part refers to the function's
'''+task. All lines beginning with 3 comment chars and a
'''+plus form a description part.
''' val1: a one-liner for the first passed parameter.
'''The second passed parameter val2. Obviously the line
'''+may start without the parameter name but clear design
'''+should use it anyway.
'''These lines start a new part, which cannot be related
'''+to passed parameter and therefore is not used.
Function my func(val1, val2) As Long
 my func = (val1 + val2) / 2
EndFunction
'''Now for the description of a variable.
'''Multi-line descriptions do not require the plus char
'''+but it does no harm either.
Dim var As Float
'''Description of a symbolic name
#Define max Par 1
```

This is how a tooltip is displayed while typing the function:



```
27 Event:
28 max = my_func(

29 Function my_func(value1, value2) As Float
value1: a one-liner for the first passed parameter.
```

Please note the rules how to create own descriptions:

- You can document the following self-defined elements
 - Symbolic names (#Define)
 - Variables and arrays (Dim)
 - Macros (Sub, Function)
 - Libraries (Lib Sub, Lib Function)
- Insert all description text into comment lines, which start with 3 comment chars ''.'. A line with less than 3 comment chars stops the description range.
- Description lines must be placed directly above the definition line containing one of the keywords given above. If done correctly, the description range is displayed as foldable text range (see page 26).

If there is an additional (empty) code line between description and definition lines the description range will be completely disabled.

- With macros and libraries, you create a description part for the instruction and one for each of the passed parameters.
- A description part can be single-line or multiline:
 - The first line of the description part starts with 3 comment chars ''', each following line of the part requires an additional plus like ''''.
 - The number of lines in a description part is not limited.
 - The description text of a passed parameter starts with the parameter's name directly followed by a colon. Alternatively, the name can be omitted.

Display of the description texts (e.g. in tooltips) is only available, if the option Parse Declarations under Editor - General (see page 60) is active.



3.4.7 Defining a foldable text range

You can define any source code lines as foldable range. These ranges are marked by a gray line to the left of the line, with a minus sign in the first line of the range. You fold the range with a click on the minus sign in the first line.

The editor also recognizes control structures like conditions or loops, program sections, macros and library modules as foldable text ranges.

You define an own foldable range as follows, using the keywords <Region> and <EndRegion>:

- Mark the source code lines.
 - Please make sure to completely enclose control structures within the range or exclude them. Including (or overlapping) another self-defined foldable range is not allowed.
- In the context menu of the source code window, select the menu entry Create Region from Block.
 - A user dialog opens.
- Enter a short description of the foldable range under Region description and confirm with OK.
- The marked range is now enclosed by the keywords <Region>
 and <EndRegion> and can be folded and unfolded.
 - Alternatively, you can enter the region keywords manually.



```
12
   □ '<REGION> #Defines and Dims
    #Define x PAR 10
13
     #Define x1 PAR 11
14
     #Define x2 PAR 12
15
16
     #Define x3 PAR 13
17
18
     Dim x x1, x2, x3 As Float
19
     '<ENDREGION>
20
```

Using the button Toggle Folding [75] all foldable text ranges may be folded or unfolded at once.

Foldable text ranges can be recognized only, if the option Parse Declarations under Editor - General (see page 60) is active.

3.4.8 Folding text ranges

The editor recognizes control structures like conditions or loops, program sections, macros and library modules as foldable text ranges. You can also define your own foldable text ranges (see above). These ranges are marked by a gray line to the left of the line start, with a minus sign in the first line of the range.

You fold a range with click on the minus sign in the first line; in the example below you would click left of Function sumsquare.



```
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
      EndFunction
9
10
11
      Dim x, x1, x2, x3 As Long
12
13
   ⊞ Function sumsquare(w1,w2,w3) As Long...
4
10
11
      Dim x, x1, x2, x3 As Long
```

Using the button Toggle Folding [75] all foldable text ranges may be folded or unfolded at once.

Foldable text ranges can be recognized only, if the option Parse Declarations under Editor - General (see page 60) is active.



3.5 Searching and replacing

Find, mark or replace any part of source code with these functions:

- Finding text quickly, page 29
- Finding and replacing text, page 30
- Regular expression, page 35
- Marking control blocks, page 37
- Using bookmarks, page 38
- Jumping to a program line, page 38
- Jumping to declaration of instruction or variable, page 38
- Switching between jump targets, page 39

There are more editor functions:

- Creating source code, page 16
- Formatting source code, page 21
- Writing programs with ease, page 40

3.5.1 Finding text quickly

You can find text quickly using the short-cut [CTRL]-[F3]. There is also the short-cut [CTRL]-[SHIFT]-[F3] to start a quick find backward.

Find uses the marked text or—if no text is marked—the word at cursor position. The following find options are fixed:

- Uppercase and lowercase letters are of no importance.
- Find text also as part of a word.
- Folded text areas are searched.
- All open documents are searched.

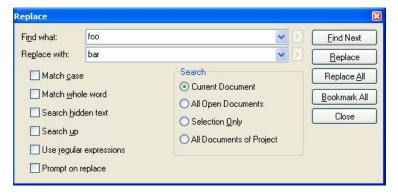
Using quick find, you cannot use regular expressions nor can you create bookmarks.



3.5.2 Finding and replacing text

You can find any occurrence of a combination of any characters, including uppercase and lowercase characters, whole words, parts of words, or regular expressions (see Regular expression on page 35).

Select the menu entry Edit Find to search or Edit
 Replace to replace. A dialog box opens, which remains on the screen until you close it.



- 2. In the Find what box, type in the search string, or choose a previous string from the drop-down list.
- Replace only: Type the replacement expression in the Replace With box, or choose a previous string from the dropdown list.
- 4. Set the scope of the search.

Option	Description
Match case	Option active: Find text having the given pattern of uppercase and lowercase letters.
	Option inactive: Uppercase and lowercase letters are of no importance.
Match whole word	Option active: Find occurrences of the text as whole words.
	Option inactive: Find text also as part of a word.



Option	Description				
Search hid- den text	The option refers to Folding text ranges (see page 27).				
	Option active: Folded text areas are searched.				
	Option inactive: Folded text areas are skipped.				
Search up	Option active: Search in direction to start of file.				
	Option inactive: Search in direction to end of file.				
Use regular expres-sions	Specify that the search string is a Regular expression (see page 35).				
Prompt on	Option valid with Replace All only.				
replace	Option active: Each occurrence opens a dialog box to control replacing.				
	Option inactive: All occurrences are replaced without query.				

5. Set the search range.

Option	Description
Current Document	Start search in the current source code at cursor position.
	If text is selected, the cursor is positioned behind the selection.
All open Documents	All open documents are searched, starting with the current source code.
Selection	Only the selected range is searched.
only	If no selection is given, search starts at cursor position.



Option	Description
	All files of the project ^a are searched, not regarding whether the current source code is also part of the project. Cannot be used for replace.
	The results are shown in the Find window in the Info range. Double click a result to jump to the appropriate code line (or use the arrow buttons). Alternatively, use the arrow buttons in the Find window to jump to the previous/next result.

- a. Files of the Other Files section are excluded from the search in any case.
- 6. Start the action with one of the buttons.
 - Find Next: If the search string is found, the screen scrolls so you can see the text in context.
 - Replace: Replace the current selection and select the next occurrence.
 - Replace All: Replace all occurrences of the search text, in the specified scope.
 - Bookmark All: Place a bookmark on each line containing the search string.
- 7. Close the dialog by clicking the Close button, or continue editing as normal.

With the option All Documents of Project, the dialog closes automatically. Search results are shown in the Find Window in the info range below.



Notes

- The menu entry Edit Find Next finds the next occurrence of the search string using the current search options, even if the Find dialog box is closed.
- The action Replace replaces selected text only, when the selection fits to the search string.
- Beware of replacing a pattern that is matched with a regular expression that can optionally match nothing, such as ".+" or "a*". In these degenerate cases, the editor can go into a loop, until the line becomes too long.
- Hint: If you want to use regular expressions for a great number of replacements in one or even all open documents, you should use Find Next and Replace to make sure you have spelled the replacement string correctly, before replacing the rest with Replace All.

Examples – Finding Text

Examples for finding text with Regular expressions.

Find all spaces or tabs at the end of a line:

The search string finds one or more spaces or tabs, being followed by the end of the line.

- Find everything on a line:

The search string finds the beginning of a line, followed by one or more of any characters, up to the end of the line.

- Find \$12.34: \\$12\.34

Note that . and \$ have been escaped using the backslash \setminus to hide their regular expression meanings.

Find a string, which is valid as variable name in ADbasic:



The search string finds a word starting with a alphabetic character, followed by zero, one or more underscores or alphanumeric characters.

Find an innermost bracketed expression:

The search string finds a left bracket, followed by zero or more characters excluding left and right brackets, followed by a right bracket.

Find a repeated expression:

$$([0-9]+)-1$$

The search string in braces (...) finds one or more digits; the braces define the tagged expression. It is followed by a hyphen, followed by the string matched by the tagged expression. So this regular expression will find 14-14 and 08-08, but not 08-15.

Examples – Replacing Text

Examples for replacing text with Regular expressions.

Find two numeric strings separated by one or more spaces:

```
([0-9]+)+([0-9]+)
```

and swap them around, using a colon to separate them: \$2:\$1

To change simultaneously:

from X100000 to X100.000

from Y100123 to Y100.123

from Z600 to Z.600

Search: ([XYZ]) ([0-9]*) ([0-9] [0-9] [0-9])

Replace by: \$1\$2.\$3



3.5.3 Regular expression

A regular expression is a search string that uses so called meta characters to match patterns of text. Meta characters are valid with the Find command only, not with the Replace command.

To use a regular expression for search/replace, check the option Use regular expressions in the dialog box. With active option, the buttons > to the right of the input fields are enabled, where you can select meta chars.

The syntax of regular expressions is defined in the .NET-Framework 2.0. a more A detailed description be found on the Internet at the address http://msdn2.microsoft.com (search for "regular expressions").

Meta char:	Meaning:
	Any single character.
	Example: Ma.s matches Mats, Mars and Mads, but not Mas.
[]	Any one of the characters
	given explicitely in brackets, or
	any of a range of characters separated by a hyphen (-).
	Examples: h[aeiou] [a-z]d matches: hard, head, hand and hold; [A-Za-z] matches any single letter. The regular expression x [0-9] matches x 0, x 1,, x 9.
[^]	Any characters except for those after the caret ^.
	Example: h [^uo] t matches hat and hit, but not hot or hut.
^	The start of a line (column 1).
	Example: The search string <code>^start</code> matches <code>start</code> only, when it is the first word on a line.



Meta char:	Meaning:						
\$	The end of a line (not the line break characters). Use this for restricting matches to characters at the end of a line, but not \n .						
	Example: end\$ only matches end when it is the last word on a line.						
\b	The start of a word.						
\B	The end of a word.						
\n	A new line character, for matching expressions that spaline boundaries.						
	A $\setminus n$ cannot be followed by operators *, + or {}. Do not use this for constraining matches to the end of a line. It is much more efficient to use "\$".						
()	Expression in braces is stored as pattern in internal registers. The register content may be re-used in the search or replacement string.						
	Up to 9 patterns can be stored, numbered according to their order in the regular expression. The corresponding replacement expression is $x = 19$ in the search string for $x = 19$.						
	Example: If the search string $([a-z]+)$ $([a-z]+)$ matches guide user, \$2 \$1 would replace it with user guide.						
*	Matches zero, one or more of the preceding characters or expressions.						
	Example: ha*d matches hd, had and haad.						
?	Matches zero or one of the preceding characters or expressions.						
	Example: ha?d matches hd and had, but not haad.						
+	Matches one or more of the preceding characters or expressions.						
	Example: ha+d matches had and haad, but not hd.						



Meta char:	Meaning:
	Matches either the expression to its left or its right.
	Example: had haad matches had, or haad.
\	"Escapes" the special meaning of the above expressions, so that they can be matched as literal characters. Hence, to match a literal backslash you must use \\.
	Example: ^a matches an a at the start of a line, but \^a matches the string ^a.

3.5.4 Marking control blocks

The lines of a control block may be highlighted altogether, e.g. to optically check nested structures. To do so, place the cursor on the keyword of a control block and select Mark Control block from the source code context menu (right mouse click).

Only one control block can be highlighted at a time.

The highlighting is removed using Unmark Control block (context menu). The cursor position does not matter in this case.

The following control block can be highlighted:

```
- Program sections Init:, LowInit:, Event:, Finish:
```

```
- Do ... Until
```

- For ... Next

- If ... EndIf

SelectCase ... EndSelect

- Function ... EndFunction

- Sub ... EndSub

- Lib_Function ... Lib_EndFunction

- Lib_Sub ... Lib_EndSub

All control structures are also foldable text ranges (see Folding text ranges on page 27).



3.5.5 Using bookmarks

Bookmarks mark selected source code lines. You can jump to bookmarked lines.

You can use these actions:

Set a Bookmark

Either bookmark a line either with the Toggle Bookmark button from the editor bar or click Bookmark All in the Replace dialog box.

Use Toggle Bookmark to remove single bookmarks.

Go to Next Bookmark

Select the Next Bookmark button from the editor bar.

Go to Previous Bookmark

Select the Previous Bookmark button from the editor bar.

Remove all Bookmarks

Select the Delete all Bookmark button from the editor bar.

Use Toggle Bookmark to remove single bookmarks.

Bookmarks are saved together with the source code file.

3.5.6 Jumping to a program line

You can jump to a program line in the source code with a double click on the line number in the status bar or by selecting GoToLine in the Edit menu. A dialog box opens, where you enter the number of the desired program line.

To show source code line numbers, the option show linenumbers under Editor – General (see page 60) must be enabled.

3.5.7 Jumping to declaration of instruction or variable

From a variable name, you can directly jump the variable's declaration. This is true for all self-declared names: local variables, arrays, instructions (Sub, Function) and symbolic names (#Define).

To jump to a declaration, you place the cursor on the self-declared name and then either select Jump to Declaration from the context



menu (right mouse click), or click the Jump to Declaration button in the editor bar.

A jump to declaration is only available, when the option Parse Declarations under Editor – General (see page 60) is active.

Of course, the jump is not available for instructions of standard include files as well as for global variables Par / FPar.

3.5.8 Switching between jump targets

You can switch between already used jump targets with a click on the arrows Jump backward / Jump forward in the Editor bar.



The development environment automatically creates a history of the used jump targets. The following jump actions are collected in the jump target history:

- Jumping to declaration of instruction or variable
- Jumping to a program line
- Double click in one of the following windows of the Info range:
 - Info window (compiler errors and warnings)
 - To-Do List
 - Global Variables Window
 - · Declarations Window
- Double click in the call list of the Call Graph Window
- Finding and replacing text

Used bookmarks are not collected in the jump target history, see Using bookmarks.

If—from the middle of the jump target history—you jump to a new target, the previous jump targets remain in the history and the new target becomes the last jump target in the history.

When you open a project the history is deleted.



3.6 Writing programs with ease

Be at ease while programming using the following functions:

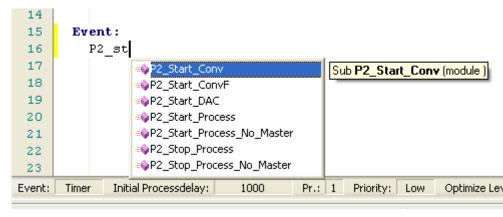
- Autocomplete for instruction or variable, page 40
- Inserting code snippets, page 41
- Displaying declaration of instruction or variable, page 43
- Displaying declarations of a file, page 43
- Displaying used global variables and arrays, page 44

Find more editor functions here:

- Creating source code, page 16
- Formatting source code, page 21
- Searching and replacing, page 29

3.6.1 Autocomplete for instruction or variable

You can use autocomplete to type keywords, instruction and variable names and even code snippets: Type some of the name's first characters and press [CTRL-SPACE].



Using autocomplete, you don't have to type instructions or variables completely.



Do as follows:

1. Write the first letters of the word and press CTRL-SPACE.

A drop-down list opens, the entries of which fit to complete the previous letters.

If you use autocomplete behind a space character, the list will contain all available keywords.

2. Select the desired list entry with mouse or arrow keys.

After a moment, an annotation to the selected list entry is displayed to the right:

- the declaration of the instruction or variable
- the string "Reserved Keyword"
- the complete code snippet (see below).
- 3. If you continue typing a name, the drop-down list is not updated automatically. Press [CTRL-SPACE] again for a list update.
- 4. To insert the selected string you simply type a brace open (best for an instruction) or a space.

Else, you could also use the [RETURN] key or type any other non-alphanumeric char.

Autocomplete is only available, when the option Parse Declarations under Editor – General (see page 60) is active.

3.6.2 Inserting code snippets

The editor provides the use of pre-defined code snippets, given in a collection. According to its definition, a code snippet can expand to some characters, some lines or a complete program listing.

To insert a code snippet at cursor position, do one of the following:

 Enter the first letters of a code snippet keyword, e.g. Sele for a SelectCase structure, select the code snippet of from the list, and press CTRL-SPACE (see also Autocomplete for instruction or variable).





 Use Codesnippets from the context menu or from the editor bar.

A drop-down list with folders opens, which each contain several code snippets (or more folders).

Navigate through the folders via mouse or via keyboard. The following keys be used:

- · Arrow up/down: Select list entry
- · Return: Insert selected code snippet or open folder.
- Backspace: Return to previous folder level.

After you have selected a code snippet, the appropriate keyboard shortcut is displayed to the right.

Insert the shortcut of a code snippet, followed by [TAB].

To display a list of code snippets and short-cuts, open <codesnippets.xml> in the folder $C:\ADwin\ADbasic\Common\$ with a browser.

3.6.3 Displaying instruction parameters

The passed parameters of an instruction are displayed automatically with a description, as soon as you type in the opening brace after the instruction's name. While you type in the parameter expressions, the appropriate passed parameters is displayed bold in the tooltip.

```
27
28
29
30
31
Function my_func(value1, value2) As Float
value1: a one-liner for the first passed parameter.
```

The tooltip vanishes as soon as the cursor is placed outside the braes around the parameters. You can re-activate the tooltip if you retype the opening brace. Alternatively, you can call the function <code>Declaration</code>



Info from the context menu or the editor bar to display the complete declaration of the instruction.

Self-defined instructions and their parameters can be documented with description texts being displayed in the same way, see "Documenting self-defined instructions and variables" on page 24.

The display of instruction parameters is only available, when the option Parse Declarations under Editor - General (see page 60) is active.

3.6.4 Displaying declaration of instruction or variable

From an instruction, a variable name, or any declared keyword, you can display its declaration and notes as tooltip, when you

- move the mouse over the keyword.
 - The declaration is displayed only, when the option Display quick info on mouse over under Editor - General (see page 60) is active.
- set the cursor on the keyword and press [F2].
- set the cursor on the keyword and select Declaration Info in the editor bar or in the context menu.

The function is available for all keywords, which belong to the language or are self-declared: local and global variables, arrays, instructions (Sub, Function) and symbolic names (#Define).

The display of declarations is only available, when the option Parse Declarations under Editor - General (see page 60) is active.

3.6.5 Displaying declarations of a file

To display all declarations, include and library files referring to a source file, set the Declarations Window to the foreground (see page 93). Declarations of other source code files will not be displayed—even if combined within a project.

The display of declarations is only available, when the option Parse Declarations under Editor - General (see page 60) is active.



3.6.6 Displaying used global variables and arrays

You can display global variables and arrays being used in the active source code and in the appropriate project (if present) by a click on the Scan Global Variables button in the Parameter Window (see also page 76).

This results in two displays:

- The Global Variables Window displays all used global variables and arrays.
- In the Parameter Window, the used global variables (not the arrays) are highlighted.

The highlighting uses three colors, according to the use of parameters:

- Green: Parameter is used in the active source 2 1336227 code only.
- Red: Parameter is used both in the active source code, and in another source code of the project, too.
- Blue: Parameter is used in an inactive source code of the project, and not in the active source code.

 5B12H

 5B12H

Using the Clear Scan button \geqslant , both displays are cleared.

If you change the source code, the displays are not updated automatically. To do so, click the Scan Global Variables button again.

3.6.7 Adding file and folder shortcuts

In the *ADtools* bar, you can add shortcuts to own files, executable programs, or folders.

Simply do a right click on the ADtools bar and select the entry Add file shortcut or Add folder shortcut. Now you can click on the inserted shortcut icon to open the file from within ADbasic.





With a double click on the link, the file or folder is activated; the computer operating system then starts the action, which is related to the file type.

Removing a shortcut is guite as easy: Just right click on the desired shortcut icon and select the menu entry Remove <file/folder> to confirm.

3.7 **Finding Programming Errors**

The development environment provides several tools to find programming errors. The tools are described in chapter 5.3 "Debugging and Analysis" (page 137):

- Finding Run-time Errors (Debug Mode), page 137
- Checking the Timing Characteristics (Timing Mode), page 137
- Analyzing program structure, page 140
- Optimizing memory intensive procedures, page 141



3.8 Managing Projects

One project can manage many process source codes, include files, library files, and files of other type, for instance when programming an application with several processes. The files are displayed in the Project Window, see page 74. Only one project can be open at a time.

The project file also saves the display parameters of the development environment: window position, size, open project files. Furthermore, you can (see Project Settings dialog box) save program settings with the project as well as command line calls before and after compiling (Prebuild/Postbuild). With opening a project, all saved settings will be rearranged.

A project allows the following actions.

- Managing a project
 - Create a new project: File New Project.
 - Open a saved project: File Open Project.
 - Save a project: File Save Project / Save Project As.
 Only project settings and display parameters are saved, but not the source files of the project.
 - Close the project: File Close Project.
 Closing the project will also close all the files of the project.
 - Change project settings: Options ▶ Project Settings or button Project Settings in the project bar; see Project Settings dialog box.
- Adding files to a project
 - Add a file to the project: File Add to Project or Add to Project in the context menu of the source code.
 - Add all open files to the project:
 Add Open Files To Project in the context menu of the
 Project Window, see page 74.
 - Add a file of other type to the project:
 Add other File to Project in the context menu of the Project Window, see page 74.

 The file is displayed under project category Other Files.

We recommend saving project files in the same folder as the project (*.abp) or a subfolder to it. Thus, if the folder is copied,



the new project includes all files and the files can be used independently from the previous project.

- With adding a file to a project, the file path is saved relative to the project folder. Only if the file is located on a different drive, an absolute file path is saved. Managing files of a project
 - Open a file of the project as active source code: double click the file in the Process Window, see page 78.
 - Remove selected files from project:
 DEL key or Remove from Project in the context menu of the Project Window, see page 74.
 - Save all files of the project at once: Save all Files of Project in the context menu of the Project Window, see page 74.
 - Compile all files of project at once:
 Build Make all Bin Files of Project.
- Compiling project source code
 - Compile selected project files:
 - button Compile selected Files ; in the project bar. The menu entry Build > Compile does not refer to the project but to the current source code only.
 - Create binary files from selected project files: button Make
 Bin Files from selected Files in the project bar.
 The menu entry Build Make Bin File does not refer to
 the project but to the current source code only.
 - Compile all files of project at once:
 - Build Make all Bin Files of Project or button in the project bar.
 - Transfer selected binary files of the project to the *ADwin* hardware: button Load selected Bin Files into ADwin system in the project bar.

If the option Autostart in the Compiler Options dialog box is enabled, binary files are automatically transferred to the *ADwin* hardware and started.

 Searching through all files of a project, including not yet opened files (except files of project category Other Files).



To do so, enable the All Documents of Project option in the find window (see chapter 3.5.2 "Finding and replacing text"). The option is not available for replacing.

- Displaying used global variables and arrays of a project (see page 44).
- Opening the Windows Explorer with the path of the selected file, using Open Path in Explorer Window from the project window context menu

Project-related capabilities can be accessed via project window context menu (right mouse click, see "Project Window" on page 74), via project bar in the project window, or in the menu File (chapter 3.9.1).



3.9 Menus

The menu bar contains these menus:

-	File:	Manage files and projects	(page 50)
_	Edit:	Edit source codes	(page 51)
_	View:	Show windows and bars	(page 51)
_	Build:	Tool for generating executable programs	(page 52)
_	Options:	Program settings	(page 54)
_	Debug:	Tools for error detection	(page 67)
_	Tools:	Various help functions	(page 71)
_	Window:	Arrange source code windows	(page 72)
_	Help:	Help, version and license information	(page 72)



3.9.1 File Menu

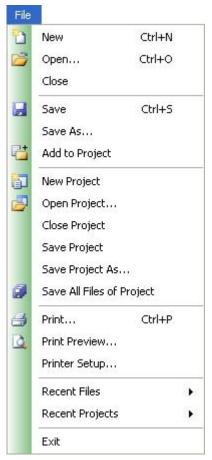
The File menu contains instructions for managing files and projects.

Files can be opened, created, saved, or closed. Multiple source code windows may be open simultaneously; no more than ten processes may be loaded to the *ADwin* system at a time.

Projects can also be opened, saved and created in the same way as files, with the exception that no more than one project can be open at a time. More instructions are available in the project window (see chapter 3.10.2).

The print functions can also be found in the menu.

Under Recent Files and Recent Projects, a list of previously opened files and projects is displayed.



Source code files are being saved in a proprietary format. In order to use files with *ADbasic 4*, one can do a Save as to convert the file into the ADbasic4 Bas-File type.

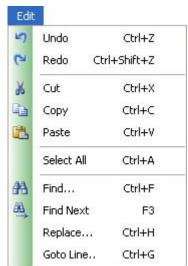


3.9.2 Edit Menu

The menu Edit contains the edit functions, in accordance with the standard Windows conventions.

Moreover, the menu offers functions for searching (Find, Find Next) and replacing (Replace); see Finding and replacing text on page 30.

Unforeseen errors may occur when inserting characters or program lines from other programs with "Cut and Paste" into the source code, and therefore is not recommended.

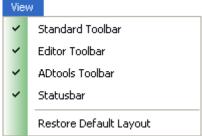


3.9.3 View Menu

In the View menu, you may open or close

- the tool bar
- the editor bar
- the ADtools bar
- the status bar.

You find further information about the process window in chapter 3.10.4 on page 78, about the toolbar see fig. 3.



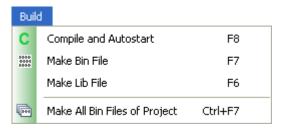
With Restore Default Layout, the default layout, which was active at the initial start of the *ADbasic* program, can be restored with a single mouse-click. This refers also to the Toolbox settings (page 73).



3.9.4 Build Menu

With the Build menu, the active source code can be compiled into

- a process using Compile.
- a binary file using Make Bin File.
- a library using Make Lib File.
- all files of the project to binary files using Make all Bin Files of Project.



Please note: Before compiling, all changed source code, library- and include files are saved automatically (AutoSave).



A change of file may occur by automatic indenting of text lines (see chapter 3.4.3 on page 23), for example when opening a previously unformatted file.

Compile is the most comprehensive instruction: It compiles the source code, transfers the generated binary file as process to the ADwin system and starts the process.

The process is only started automatically if the Autostart option, in the Options\Compiler menu, is set to Yes. Otherwise, the process can be started with the button > in the toolbar or in the process window (see page 78).

If the compiler detects errors or critical sequences in the source code, it is shown in the Info window. A double click highlights the appropriate line in red.

Make Bin File is only available for licensed ADbasic users. It compiles the active source code into a binary file and saves it automat-



ically. The file is stored in the directory of the source code file, but with the extension . Txn. The x denotes the processor type and n the process number (see Options Menu, Process Options dialog box).



A binary file with the extension <*.TC3> can be transferred to an *ADwin* system equipped with a T12 processor, which administers it as process 3. Binary files can be transferred to the *ADwin* system from development environments such as C or Visual Basic (chapter 6.3.4 on page 155).

Make Lib File is available for licensed *ADbasic* users only. It compiles the active source code—the file must be saved as file type ${\tt LibFile}$ —into a binary file and automatically saves it as library file. The library is stored in the same directory and with the same name as the source code file, but with the file extension . ${\tt LIx}$ (where ${\tt x}$ denotes the processor type.)

Afterwards the library can be included into other source codes that use their functions and subroutines (see chapter 4.5.1 on page 125).



Please note with processor T12: Each library consists of three files *.lic, *.o, and *.h. All three files must be located in the same folder to make the library run correctly.

Make All Bin Files of Project is available for licensed *ADbasic* users only. The function refers to both Make Lib File and Make Bin File: The function compiles all source code files of the project and creates both library files and binary files.

Please note: Files are compiled in the order as they are displayed in the project window. If you require a different order with library files you have to compile the files individually using Make Lib File.



3.9.5 Options Menu

In the Options menu, a number of options can be set, which will have an immediate effect. For each menu item, a dialog box opens where the settings are entered.



Compiler Options dialog box

The settings in this dialog box are used in every source code compilation. In particular, the information refers to the *ADwin* system where the compiled source codes are to be executed as process.

To compile source codes for different *ADwin* systems, the parameters need to be set for each system in the dialog box.





Fig. 4 - The Compiler Options dialog box

- System: Select the ADwin system.
- Processor: Select the system's processor type.

The abbreviations correspond to the following full names:

Abbreviation	T12	T11	T10	T9	T8	T5	T4	T2
Full name		ADSP TS101S	_	_	T805	T450	T400	T225

Fig. 5 – Processor Names

Device No.: Select the device number to access the ADwin system.



The device number is set using the program <ADconfig.exe>. The default setting is 150 Hex.

 Do not access the Device: If inactive, a binary file will be automatically transferred to the hardware after compilation.
 Thus, the ADwin hardware must be connected before compilation.

With active option, a source code can be compiled, even if the ADwin hardware is not connected to the PC.

 Load standard processes: With active option, the standard processes 11, 12 and 15 (see chapter 6.1.1 on page 144) are loaded into the *ADwin* system during boot process. With inactive option, the loading of processes 11 and 12 is suppressed.

This setting is only available for *ADwin-Gold* and *ADwin-light-16*.

- Autostart: Active option causes the binary file, generated and transferred to the ADwin system during compilation, to be immediately started. With inactive option, the process requires to be started by clicking the button in the toolbar or in the process window.
- Remember Device No.: Active option saves the last used
 Device No. (see above) on closing ADbasic; the next start-up will automatically use the saved number.

Inactive option skips saving the device number. Thus, *ADbasic* starts up with the formerly (when Yes was set) saved device number NONE.

Process Options dialog box

This dialog box contains the compiler options for the currently opened source code window; the properties of the process, which is to be compiled from the opened source code and transferred to the *ADwin* system.

This applies to library files as well, where only the option Optimize can be set.



Each process must be configured separately by opening the dialog box for each source code window, unless using the default settings. To open this window quickly, do a double click on the source code's status bar.

The dialog box for T4, T5, or T8 processors differs slightly from the standard dialog box; the dialog box is described in the Appendix A.4.1.

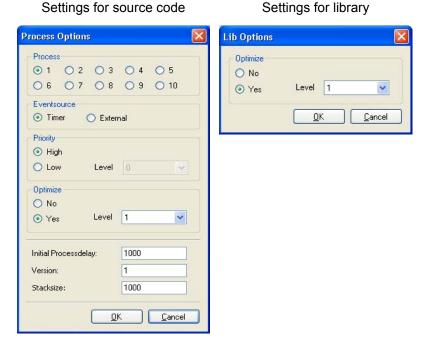


Fig. 6 - The Process Options dialog box



Process: Process number

The number, under which the transferred process is started on the system.

If there is more than one process to be run, each process must have its own process number.

- Eventsource: Sets the event source signal, which initiates the Event: section of the process.
 - Timer
 sets the internal counter as event signal. The system
 variable Processdelay determines the delay, in which the
 counter creates an event signal.
 - External
 sets the (external) signal the event input of the ADwin
 system as event signal, for instance a sensor impulse. In this
 case, the Priority option must be set to High anyway.
 How you can use an external event input in an ADwin-Pro
 system, is explained in the ADwin-Pro software
 documentation under EventEnable.
- Priority: The priority of the process.

Set the priority the process will be run with in the *ADwin* hardware. For more information, see chapter 6.1.1 "Types of Processes".

Level (-10...+10) defines the priority within processes with low priority, so that a process with a higher Level can interrupt those with a lower level, but not vice versa. A higher number represents a higher level.

Optimize: Status and level of compiler optimization.

Compiler optimization, which may be used optionally, can reduce the execution time of the process by up to 20 percent. A higher setting under Level will lead to shorter execution times. With processor T12, there are additional settings fast and size, which enable optimization for speed or smaller memory size.



Under certain circumstances, a process causing unexpected compiler or run-time errors can be solved by setting a lower optimization level.

As long as the debug mode is activated, optimization is automatically deactivated.

- Initial Processdelay: The initial Processdelay (cycle time), with which the process is to be started.
 As an alternative, you can set the variable Processdelay in the source code.
- Version: An integer value for differentiating between several versions of a process.
- Stacksize: Processor T12 only. The program's stack size in Bytes.

With libraries, it can be required to set a greater value. The number of local variables and arrays determines the required program stack size. An insufficient stack size can lead to data loss or program abortion. Using the Option Stack Test (page 69) you can check if the stack size is sufficient.

 Number of Loops: This setting is only available for processors T2...T8.



Settings dialog box

The Settings dialog box has several sheets, which are activated via tree diagram in the left pane:

- Editor
 - Editor General
 - Editor Syntax Colors
 - Editor Print Settings
- General
- Language
- Directories
- Help
- ADtools

Editor – General

Parse and format: The editor can format the source code automatically, e.g. indent and do syntax highlighting. To do so, the editor must parse all source codes continuously. The found information is the base for more comfortable functions like Autocomplete for instruction or variable and Displaying declarations of a file.

Please note: Continuous parsing of source codes may cause a loss of editor speed on slow PCs.



Parse Declarations: The editor continuously parses source codes. Some comfortable functions depend on this function.

Autoindent: Source code is indented automatically. Indent positions are set via Tabsize. See also "Indenting text lines" on page 23.

Indent ADbasic sections: Program sections are indented by one tab more.

Smart format: Format lines automatically, see "Smart formatting" on page 23.



Align comments at specified position: Any comment after source code is automatically set to the specified Position.

Please note: While using double comment chars ' you can position a comment "manually" using spaces or tabs.

Display quick info on mouse over: When moving the mouse over a keyword or number, a tooltip displays related information.

Tabsize: Setting, how much spaces make one tab indent. Indenting is always done with spaces.

Font size: Font size (in points) of the source code.

"#If Processor = ..." graying: A conditional block of instructions is displayed gray, if the given processor setting does not meet the condition with system parameter Processor; see #If ... Then ... {#Else ... } #EndIf on page 242. Other conditions are not considered.

Show line numbers: Line numbers are displayed in the gutter left of the source code. See also "Jumping to a program line" on page 38.

Column mark, visible: A gray line is displayed at the given Position. The line enables easy line breaking at the desired position, e.g. in order to avoid long lines for print.

Editor – Syntax Colors

The editor highlights the syntax elements with different colors; see also chapter 3.4.1 "Syntax highlighting" on page 21; complete syntax highlighting requires an active option Parse Declarations under Editor — General.

You may set the highlighting individually for each syntax element (definition see list below):

- Color: Text color.
- Bold: Font style bold.
- Italic: Font style italic.

The example text above shows how source code be formatted.



Set to Default deletes all individual changes and resets default settings.

You can change the source code font size under Editor – General, Font size.

The editor distinguishes the following syntax elements:

- ADbasic-Syntax (System related):
 - ADbasic sections: **Keywords Init:**, **LowInit:**, **Event:** and **Finish:** for program sections.
 - Compiler Directives: Pre-compiler instructions like #Define, starting with a #.
 - Reserved Keywords: Basic instructions as Dim in ADbasic.
 - Global Variables: Global variables Par_1 ... Par_80, FPar 1 ... FPar 80 and Data 1 ... Data 200.
 - External Keywords: ADbasic instructions for access to inputs/outputs like P2_ADC. Most of these instructions are declared in the delivered standard include or library files.
 - Symbols: Operators as braces, + or =.
- User related:
 - Defined Names: Symbolic names like myName, declared with #Define.
 - Local Variables: Variables like myVar declared with Dim.
 - Sub Names: Names (like mySub) of user-defined modules, declared with Sub or Lib Sub.
 - Function Names: Names (like myFunction) of userdefined modules, declared with Function or Lib Function.
- Other:
 - Numbers: Numbers in decimal (15), hexadecimal (0Fh) and binary notation (1111b).
 - Strings: Strings in "double quotes".
- Comments: Comments after Rem or quote '.
- Standard Text: All elements, which do not belong to other groups, e.g. invalid instructions like Eixt (instead of Exit).



Editor – Print Settings

The settings refer to printing of source code.

Header refers to the printed header line.

Print Header: A header line is printed on top of each page.

Header text: The text of the header line.

Layout determines the elements of the screen display are to be printed.

Color: With inactive option, the printout is black and white.

Syntax Highlight: Syntax highlighting is printed.

Line numbers: Line numbers are printed at the left.

Font size: Sets the font size of the output.

General

Set whether floating-point numbers (FPar) in the Parameter Window are displayed with 7 digits or 14 digits mantissa.

Language

The language, in which the error messages of the compiler is displayed. Options are either Deutsch (german) or English.

Directories

Set the directories where the operating system and the compiler search for *ADbasic* files:

 BTL-Directory: The directory, in which the development environment searches for the system files <*.btl>, which are



transferred to the *ADwin* system during the boot process (see chapter 3.1.3).

- Include-Directory: The directory, in which the compiler searches for include files <*.inc>, which can be included into the source code using #Include instruction (without path).
- Lib-Directory: The directory, in which the compiler searches for library files <*.lib>, which can be included into the source code using Import instruction (without path).
- Default working directory: The directory, in which the development environment searches searches for files, if a source code file or a project is opened.

We recommend not changing default directories for BTL, Include and Library. To include library and include files from other directories, type the full or relative path name with the instruction.

Help

Using the option stay on top, the help windows will always be the top-most window. The help window may then mask the development environment *ADbasic*.

ADtools

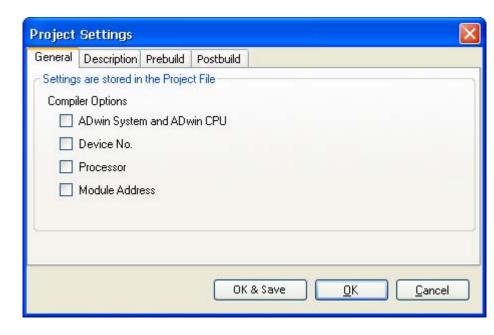
The *ADtools* (description see chapter 3.12) can be started from the ADtools bar. If the appropriate option is active, the tool is displayed in the bar.

Project Settings dialog box

The settings of the dialog window refer to the current project. Settings are stored in the project file. A saved project setting will be restored with loading the project file; previous settings will then be lost.

- Tab General
- Tab Description
- Tabs Prebuild / Postbuild





Tab General

The project settings under the General tab determine, which program settings are saved with the project file:

- Dialog window Compiler Options
 - ADwin System and Processor
 - · Device No.
- Debug menu
 - · Debug Mode
 - Timing Analyzer
 - Stack Test (T12 only)

A saved project setting will be restored with loading the project file; previous settings will then be lost.

Tab Description

You can create a Description and save it with the project file. The description has no function in *ADbasic*.



Tabs Prebuild / Postbuild

You can enter command line calls, which are processed automatically before building binary files (Prebuild) or afterwards (Postbuild). The settings are stored in the project file.

Each compiler run (see below, \$BUILDACTION) triggers the execution of the command line calls; with menu entries in the Build menu this is true, if the the source code file to be compiled is part of the project.

The command lines are consecutively handed over to the PC operating system, the project path is used as working directory. Then, *ADbasic* waits for successful execution of the comand line. If execution fails and crashes, you have terminate execution via the task manager.

In a command line, you can use the following variables:

- SPROJECTNAME: Project file name (without path and file extension).
- \$PROJECTPATH: Path of project directory (without file name).
- \$PROJECT: Projekt file name (without path).
- \$BUILDACTION: String indicating the triggering instruction (see above).
 - C: Menu entry Build > Compile
 - MB: Menu entry Build Make Bin File
 - ML: Menu entry Build ▶ Make Lib File
 - CS: Button Compile Selected Files (project bar)
 - MAB: Button Make all Bin Files of Project (project bar)
 - MSB: Button Make Bin Files from Selected Files (project bar)

Any path or file name should be surrounded by quotation marks in the event of contained space characters.

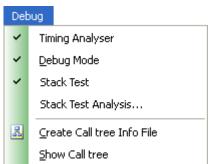
If a line starts with # (hash) it is rated as comment line and ignored.



3.9.6 Debug Menu

The Debug menu offers option settings and tools, which help in debugging and optimizing a given source code.

Please note that option settings will only be active after the next compilation.



Timing Analyzer Option

When the Timing Analyzer compiler option is activated, additional information about the timing characteristics of this process are available after compiling a source code (for display of information, see the Timing Analyzer Window).

The setting of this compiler option is displayed in the Status Bar, the setting of a running process in the Process Window.

This option needs approximately 60 clock cycles (when using a T9, T10, or T11 processor) per event and process additionally and therefore slightly affects the timing characteristics. We recommend activating the option only to compile one or only some processes; deactivate the option afterwards. These option settings of the processes are not saved when quitting *ADbasic*.

Debug mode Option

The Debug mode compiler option, when activated, includes additional security queries into the process during the compilation of a source code (see also chapter 5.3.1 on page 137).

The setting of this compiler option is displayed in the Status Bar, the setting of a running process in the Process Window.

Activation of this option increases program execution time as well as the demand for memory. As a rule of thumb, this increase has a dimension of approximately 20%, whereas greater values are also possible.



Therefore, this option should only be used during program development.

Using #Begin_/#End_Debug_Mode_Disable you can temporarily disable the debug mode.

The window <code>Debug Errors</code> (see Info range) opens when a run-time error occurs in the *ADwin* system the first time. If an error type occurs the first time the window is automatically set to front.

Error messages are not reset upon restart of the process, but upon reloading the process (compile and run).



Fig. 7 – The Debug Errors Window

The operating system corrects run-time errors in a way to obtain a stable state of operation; this may nevertheless cause unexpected program results. Certain run-time errors on Pro II modules will stop the process.

The following table shows, which errors are displayed and which corrections are made. The complete list of debug error messages—including those where no corrections are made—are to be found in the annex on page A-20.

Run-time error	Correction
Division by zero	The result of a float division is replaced by +3.40282E+38, the result of a long division is replaced by +2147483647.



Run-time error	Correction		
SQRT from negative number	The square root's result is replaced by the value 0.		
Data index too large / <1 Array index too large / <1	The array access is not executed.		
Access to local or global array elements, which are not declared, with indices that are too large or too small.			
Fifo index is no fifo	Instruction is not executed:		
The array with the given index is not declared as FIFO or not declared at all.	FIFO_Clear, FIFO_Full, FIFO_Empty.		
Address of Pro II module is	The process is stopped.		

>15 or < 1

Run-time error	Correction
Data index too large / <1 Array index too large / <1	The array access is not executed.
Access to local or global array elements, which are not declared, with indices that are too large or too small.	

For each process, only one error is shown (in most cases the error, which occurred last), even if the process has generated more run-time errors.



Please note: Using the MemCpy instruction only the access to the destination array will be controlled and corrected; an access to undeclared elements of the source array will not be detected.

Option Stack Test

With processor T12, you can set the program stack size with the parameter Stacksize in the Process Options dialog box. The Option Stack Test enables you to check if the stack size is sufficient for the



process. You start the stack test analysis in the Window Stack Test Analysis (page 70).

The Stack Test compiler option, when activated, includes additional code into the process during the compilation of a source code (see also chapter 5.3.1 on page 137).

Activation of this option increases program execution time (see Stack Test Analysis ...) as well as the demand for memory. Therefore, this option should only be used during program development.

The setting of this compiler option is displayed in the Status Bar, the setting of a running process in the Process Window.

Stack Test Analysis ...

The menu entry Debug / Stack Test Analysis (with processor T12 only) opens the Window Stack Test Analysis (see page 84).

Inside the window you can start the stack test and display the results. The stack test comprises all running processes on the *ADwin* hardware, which have been compiled with the Stack Test option.

Create Call tree Info File

The menu entry Create Call tree Info File creates or updates a call graph and opens the Call Graph Window (see page 82).

The call graph is a basic program analysis result for better understanding of more complex programs (see Analyzing program structure, page 140) and of memory usage (see Optimizing memory intensive procedures, page 141).

Show Call tree

The menu entry Debug / Show Call tree displays a previously created call graph in the Call Graph Window (see page 82).



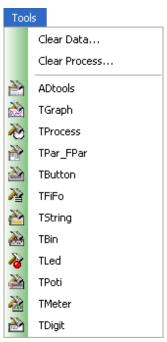
3.9.7 Tools Menu

The Tools menu option calls utility programs.

The Clear Data menu option clears the memory of the *ADwin* system, which is used by a specified **DATA** array. This is the counterpart to the Dim instruction. All data of the array will be lost.

In the dialog box, type the data array index to be cleared, e.g. 3 for Data_3 and confirm.

The Clear Process menu option deletes a specified process from the memory. Please note that a process can only be deleted after being stopped.



The menu entries ADtools and following start a tools each. Find a short description in chapter 3.12 on page 95.

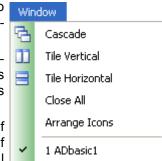


3.9.8 Window Menu

From the Window menu it is possible to switch between different source code windows and arrange them on the monitor.

The Arrange Icons menu reorders minimized source code windows, which is useful after the screen resolution has changed.

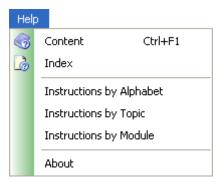
At the bottom of the menu, there is a list of open source codes; by clicking one of these menu items that source code will become the active window. The active source code is checked; in the example at right it is ADbasic1.bas.



3.9.9 Help Menu

The Help menu calls the online help of the development environment:

- Content: Table of contents
- Index: Index directory
- Instructions by:
 Sorted lists of instructions.
 - by Alphabet
 - by Topic
 - by Module (ADwin-Pro only)



The instruction lists refer to that *ADwin* system, which is set in the Compiler Options dialog box (page 54).

Alternatively, you may use the obutton in the toolbar. With the [F1] key, help is opened for a dialog box or for the selected keyword.

The About menu entry opens a window that displays the version of the development environment and the License key. The license key



can be entered or changed by pressing the Change License button (see also page 9).

Without entering a valid License key, *ADbasic* runs in demo mode. In demo mode, the use is only allowed for demonstration, test or evaluation purposes.

3.10 Windows

There is separate information about the Info range on page 86.

3.10.1 Toolbox

The Toolbox is the window range of the environment to the left, where Project Window, Parameter Window, and Process Window are displayed.

The toolbox divides into an upper and lower display region, where to the windows can be assigned freely. A hidden window is drawn to the front with a click on its tab.

To assign a window to the upper or lower region, do as follows:

- Do a right mouse click to the head bar of the window to open the context menu.
- Select whether to dock the window at top or bottom.



 You may dock all windows to the same region. Thus, only one window can be in front at a time.

The standard setting can be reset via the menu entry View Restore default layout.

The toolbox can be displayed as movable window or be completely hidden via the buttons in the head.



3.10.2 Project Window

The project window shows an opened project and the source code and include files added.

The project window is located in the Toolbox (see page 73).

In the project window, the following actions may be executed:

Add all open files to the project:
 Select Add Open Files to Project from the project window context menu.



Additionally you can add a single file to the project:

Select Add to Project from the source code context menu.

- Add a file of other type to the project:
 Select Add Other File to Project from the project window context menu.
- Delete a source code file from the project:
 Highlight the file in the project window, then
 - press the [DEL] key or
 - select Remove from Project from the context menu.
- Open a source code file and make it the active source code:
 - · Double-click the file or
 - Highlight the file in the project window, then select Open from the context menu (right mouse button).
- Save all open source code files of the project:
 Select Save All Files of Project from the context menu.
- Compile all files selected in the project window:
 Select the button Compile selected Files of Project c, from the project bar.
- Create binary files of all files selected in the project window:
 Select the button Make Bin Files from selected Files from the project bar.
- Create binary files of all project files:
 Select the button Make Bin Files of Project from the project bar.
- Load binary files of all files selected in the project window into the ADwin system:
 Select the button Load selected Bin Files into ADwin system from the project bar.
- Open the Windows Explorer with the path of the selected file: Select Open Path in Explorer Window from the context menu.



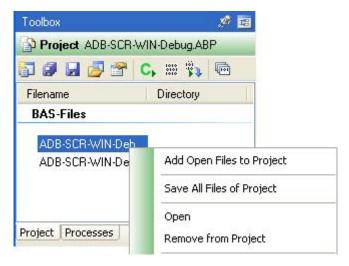


Fig. 8 – The Project Window with the Context Menu

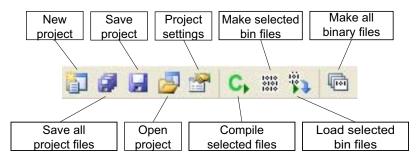


Fig. 9 – The project bar

3.10.3 Parameter Window

The parameter window displays a table showing the values of the global parameters Par_1...Par_80 and FPar_1...FPar_80. With the scroll bar at the right, you can scroll through the parameters.

The parameter window is located in the Toolbox (see page 73).

When the communication between the computer and *ADwin* system is active (icon Enable Cyclic Update () in the toolbar), the fields in



the table are enabled and appear with a white background color, and display the values of the global parameters. The values are continuously read out from the system. Fields are disabled and appear with a gray background color when the communication is inactive.

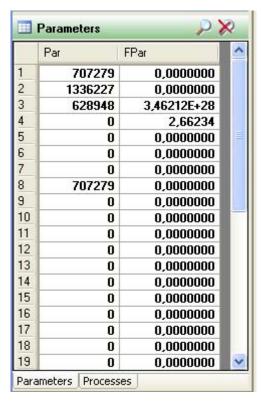


Fig. 10 - The parameter window

To change the display of a parameter's value (Par_1...Par_80) between decimal and hexadecimal notation (see Par_5 in fig. 10), do a mouse click on the number of the variable (left of the table field). A click on the column header changes the display of all parameters Par_1...Par_80 at once.



For use of the Scan Global Variables button, see "Displaying used global variables and arrays" on page 44.

3.10.4 Process Window

The process window shows information about the processes 1...10 on the *ADwin* system, when the communication between the computer and the system is active (icon) in the toolbar). Otherwise the fields are gray.

The process window is located in the Toolbox (see page 73). Open the process window with a click on the tab Processes.

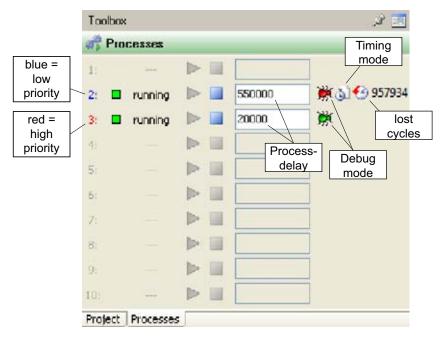


Fig. 11 – The Process Window

For each process, the following information is displayed:

Process priority



The color of the process number indicates the priority:

- red = high priority
- blue = low priority

The time units and meaning of the process delay are explained in chapter 6.2.1 "Processdelay", page 148.

- Process status
 - running: process is running.
 - stopped: process was stopped.
 - ---: process does not exist.

A process can be stopped with the Stop button and started again with the Start button. The buttons of the toolbar have the same function, but they refer to the process related to the active source code.

Process delay (process cycle time)

The process delay for the active source code is displayed in the toolbar, too.

To change the cycle time, type a value into the input field. As soon as the cursor leaves the input field the value is transferred to the *ADwin* system. Please note to not overload the system by small values.

Process runs in debug mode

The icon is displayed if the process runs in debug mode. Find more about debug mode under Debug mode Option.

The compiler setting debug mode is displayed in the Status Bar.

Process runs in timing mode

The icon is displayed if the process runs in debug mode. Find more about debug mode under Timing Analyzer Option (page 67). Timing information is displayed in the Timing Analyzer Window.

The compiler setting timing mode is displayed in the Status Bar.

T12 only: Process runs with stack test



The icon is displayed if the process runs with Stack Test mode. You find more under Option Stack Test and Window Stack Test Analysis.

The compiler setting stack test mode is displayed in the Status Bar.

T12 only: Process loses process cycles ⁶

The number determines how many process cycles have been lost since process start. Lost process cycles (lost events) show that the process timing has errors, which you have to remove. At least for some time there is a processor overload (see Workload of the ADwin system).

To avoid processor overload keep the Busy display in the Status Bar always clearly below 100%.

3.10.5 Source code window

The source code window is the main window of the development environment, where you edit the source codes of *ADbasic* programs.

The multitude of functions in the source code window is described in chapter 3. Important sections are the following:

- Basic Elements of the Development Environment, page 12.
- Creating source code, page 16.
- Context menu in source code window, page 18.

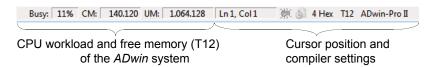
3.10.6 Status Bar

The status bar is located at the bottom of the *ADbasic* program window.



Last ADbasic CPU workload and free memory Cursor position and action of the ADwin system (up to T11) compiler settings





- Left side: Information about the last ADbasic action.
- Middle: The current CPU workload and free memory of the *ADwin* system. This information is displayed, if the communication between the computer and *ADwin* system is active.
- Right: The current cursor position in the source code window (line and column); further compiler settings: debug mode, timing mode, stack test (T12 only), device no., processor, ADwin hardware.

A double click on the compiler settings opens the Compiler Options dialog box (page 54).



The displayed information about the CPU/memory usage:

- Busy: the processor workload in percent, calculated as:

CPU time / (CPU time + idle time).

PM: free program memory in bytes.

EM: T11 only: free extra memory in bytes.
 DM: free internal data memory in bytes.

DX / SX: free external data memory in bytes.

CM: T12 only: memory range, which can deliver data

to the cache.

UM: T12 only: memory range, which cannot deliver

data to the cache.

3.10.7 Call Graph Window

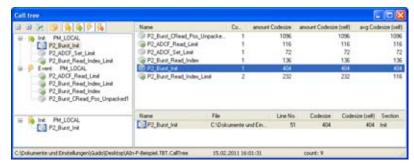
The Call Graph window displays all procedures (Sub , Function , Lib_Sub , Lib_Function) being called in an ADbasic program and their code sizes. The call graph is static and created or updated with the menu entry Debug / Create Call tree Info File.

The menu entry Debug / Show Call tree displays a previously created call graph.

The call graph is a basic program analysis result for better understanding of more complex programs (see Analyzing program structure, page 140) and of memory usage (see Optimizing memory intensive procedures, page 141).

If a procedure call in the source code is not displayed, the reason is often having the call inside a conditional program branch, which is not run in any case. This happens if a condition (If ... EndIf, #If ... #EndIf, SelectCase ... EndSelect) uses a constant value and results in any case to "False".





The call graph window has 4 panes:

- Call tree (top left): A tree where each node represents the call of a procedure in the program source code. Sub-nodes are (nested) calls within a procedure.
- Root path of the currently selected node (bottom left).
- List of called procedures (top right): A list of all called procedures given with
 - Name: Name of the procedure.
 - Count: Number of calls of a procedure.
 - Total code size: Code size, to which the calls of the procedure sum up to, including calls from inside the procedure.
 - Code size: Like Total code size but excluding calls from inside the procedure.
 - avg code size: Average code size of the procedure, i.e. Code size divided by Count.
- Call list of selected procedure (bottom right): A list of calls of the currently selected procedure given with
 - Name: Name of the procedure.



- File: File name, in which the procedure is called.
- Line no.: Line number in the file, where the procedure is called in.
- Total code size: Code size, to which the calls of the procedure sum up to, including calls from inside the procedure.
- Code size: Like Total code size but excluding sub calls from inside the procedure.
- Section: Program section where the procedure is called in.

Do a double click on a row of the call list to jump to the source code line where the procedure is being called.

Via the bar above the call tree pane 🗀 🗷 🧐 🐕 🧗 👼 you can restrain, which tree nodes be displayed in the panes:

- Collapse/expand (call tree pane only):
 - The minus button collapses all nodes leaving only the sections visible.

 - With the Autocollapse option being set, only the path to the selected node is expanded while other nodes are collapsed.
- Library procedures : The library option enables library subs and functions to be displayed.
- Sections: The section options enable/disable the nodes of the appropriate program section.
 The sections are LowInit , Init , Event , and Finish .

3.10.8 Window Stack Test Analysis

The Stack Test Analysis window (processor T12 only) enables you to check if the set stack sizes (Stack size, see Process Options dialog box) are sufficient for the running processes on the *ADwin* hardware. An insufficient stack size can lead to overwriting of other memory ranges and therefore cause data loss or program abortion.



The stack size is especially important if you use libraries (Lib_Sub ... Lib_EndSub, Lib_Function ... Lib_EndFunction). The number and size of local variables and arrays in the library determine the required program stack size of a process. Nested library calls may also increase the required stack size.

You start the stack test with the Start button. The stack test determines for each running process—which was compiled with the Option Stack Test (page 69)—the number of bytes being used in the stack since the process was started. The used percentage of the stack is also displayed. The stack test may take some seconds.

To close the window again click the OK button.

Please note that the stack test cannot evaluate the influence of the program section **Finish** to the stack.

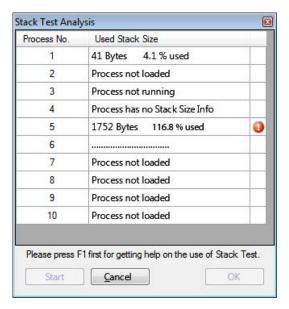


Fig. 12 - Window Stack Test Analysis

In the column to the right, a warning symbol is displayed if the used percentage of the stack size is >90% an error symbol if the percentage is 100% or more (see fig. 12, process 5). In this case, you have to set



a greater value for the stack size of the process. You open the process source code, set the Stack size in the Process Options dialog box anew and compile the process again. We recommend repeating the stack test afterwards.

Please note that the stack is stored in the CM memory and should therefore not be oversized, see Memory Areas (T12).

The symbol (see fig. 12, process 6) appears if the stack test waits to be started. The stack test starts as soon as the program section Event reaches its end the very next time. A delay may happen e.g. if an external process waits for an external event signal, or if a low priority process takes a long time to be processed.

Alternatively, you can stop the stack test with the Cancel button.

3.11 Info range

The info range is located at the bottom of the main window and encloses the following windows:

- Info window (see page 86)
- To-Do List (see page 88)
- The window Debug Errors (see page 67)
- Timing Analyzer Window (see page 88)
- Global Variables Window (see page 92)
- Declarations Window (see page 93)

3.11.1 Info window

In the info window, the compiler messages concerning the current source code are displayed:

- Error messages (colored red)
- Warnings
- Status message after compilation

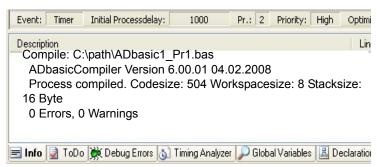
The window is part of the Info range (see above).

Warnings and error messages are displayed with the place of occurrence (line, file name and path). A double click turns the appropriate



code line to red and the cursor jumps to the line.

The (successful) status message after compiling looks like this:



Processors up to T11 only: The values be used as hints about the required memory

- Codesize: Size of the created binary file in bytes; the file will be stored in the program memory (PM) as process.
 - With processor T11, the binary code size is given separately for PM, EM, and DX; see Init:, LowInit:, Event:, Finish:.
- Workspacesize: Required memory size in bytes in the local data memory (DM).

The DM memory space is required for:

- · Local variables and arrays
- Internal purposes (8 byte)
- Global arrays Data_x
 For internal purposes, each array requires 40 byte, even if the data are stored in external data memory.
 If the array is declared At DM_Local, each array element requires 4 byte (data type Long) or 8 byte (data type Float).
- Stacksize: Internal stack size (DM) being used for libraries.

The memory size required in the external data memory (DX) will not be displayed.

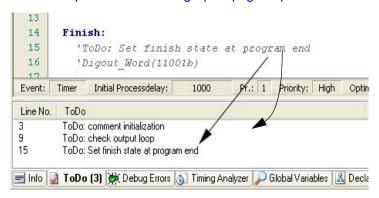


3.11.2 To-Do List

The ${\tt ToDo}$ window serves as a simple To-Do list: lines from the current source code are shown where the text " ${\tt ToDo}$:" is contained as a comment. By use of such commenting lines not yet completed tasks can be flagged in the source code and clearly arranged in the ${\tt ToDo}$ window.

If a task is completed, just delete the comment line.

The window is part of the Info range (see page 86).



A double click on a To-Do entry positions the cursor in the appropriate line of the source code.

3.11.3 Timing Analyzer Window

The window Timing Analyzer displays 7 parameters describing the timing characteristics of the processes 1...10 since the moment of the previous start. More detailed information can be found in chapter 5.3.2 "Checking the Timing Characteristics (Timing Mode)".

The timing mode is enabled before compiling for each process separately in the Debug menu with the Timing Analyzer Option (page 67).

The Timing Analyzer window is part of the Info range (see page 86).

All timing information is given in clock cycles of the processor (units see fig. 19 on page 148).



The parameters can only be used with high-priority processes. In an externally controlled process, the values in the lines 4-6 are not useful and are displayed as 0 (zero).

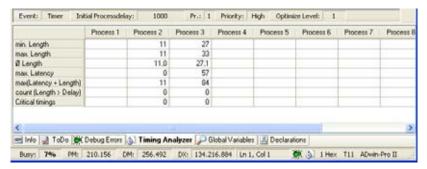
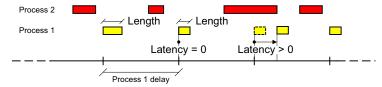


Fig. 13 - The Timing Analyzer window

All duration values are counted in clock cycles of 25ns. Length describes the time a process cycle needs (section Event:); this processing time can also be determined as described in chapter 5.1 "Measuring the Processing Time". Latency is the time between an event signal (external or generated by internal timer) and the start of the process cycle, shown in the picture below for the time-controlled Process 1.



The parameters in the window have the following meaning:

- min. Length: The minimum time measured for a process cycle
- max. Length: The maximum time measured for a process cycle
- Ø Length: Average time of a process cycle.
 The average is calculated as mean value from the previous length values:



 \emptyset Length = $0.999 \cdot \emptyset$ Length + $0.001 \cdot$ Length

After start of a process it takes 7000 cycles until the average time reaches a valid value.

This parameter shows with min. Length and max. Length how long and regular the processing time is for a process cycle. Varying processing times will arise e.g. when large quantities of data are only evaluated after a longer time period or if conditions (If, Case) contain program sections with very different processing times (loops).

 max. Latency: The maximum measured latency of a process cycle; only available for timer-controlled processes.

A latency emerges from the occurrence of an event signal while a high-priority process is running. This happens when the processing time of a process cycle exceeds its Processdelay. With 2 or more high-priority processes, every now and then process cycles do start time-delayed, except their Processdelays are integer multiples of each other.

The sum of all delays should always average 0; this corresponds to keeping an average frequency. Moreover, the parameter is important for processes whose process cycles must run at a precisely pre-defined period in time.

 max. (Latency+Length): The maximum sum of the latency and the processing time of a process cycle; only available for timer-controlled processes.

To get optimal timing characteristics, this parameter value should be lower than the value of the Processdelay; if you can fulfill this condition, the process does not cause latencies for its process cycles (but nevertheless can do for other process cycles).

 count (Length > Delay): A value indicating how often the processing time of a process cycle has exceeded the Processdelay; only available for time-controlled processes. This value should preferably be zero.



The higher the value, the more frequently the process has caused a latency for its own process cycles (and perhaps for other processes too). The operating system is continuously trying to make up this delay. The amount of exceeded values gives no information about the loss of event signals.

 Critical timings: describes how often a condition is fulfilled, which could signify a lost event signal. The value should definitely be zero.

This parameter has a different meaning depending on the type and amount of processes (see chapter 6.2.5 "Different Operating Modes in the Operating System", page 152).

Event signals can be lost under the following circumstances:

- In a single time-controlled high-priority process (also in combination with the externally controlled process)
- In the externally controlled process (also in combination with one or more time-controlled processes).

In several time-controlled processes, event signals cannot be lost; the following condition will nevertheless be counted. Here the parameter must be interpreted as a poor timing characteristic, which should be improved in any case.

Loosing event signals means that (since the last start of the process) fewer process cycles have been executed than event signals occurred, probably the amount fewer being indicated. Lost event signals cannot be compensated by the operating system.

A loss of an event signal is equated to the fulfilment of the condition:

- In time-controlled processes: max.latency+length > 2 x Processdelay
- In externally controlled processes:
 When processing the section Event: has just been finished, a new external event signal is already waiting. Any more event signals having arrived during this processing time will be lost.



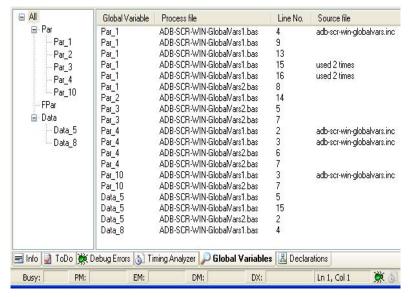
Sometimes it happens that, despite a true condition, no event is lost. Thus, you play it safe reducing the amount of true conditions as far as possible.

3.11.4 Global Variables Window

The window Global Variables displays, which global variables (Par_1 ... Par_80, FPar_1 ... FPar_80) and arrays (Data_1 ... Data_200) are used in a source code or a project.

To start or update the display click the button Scan Global Variables in the Parameter Window (see Displaying used global variables and arrays, page 44).

The window is part of the Info range (see page 86).



The marked element of the tree view (on the left) determines, which variables are displayed in the window to the right. After a click on Par the list will display only Par 1...Par 80 (as far as used).



The window columns can be alphabetically sorted with a click on the column header.

- Global Variable: Process file: name of the main file of the scanned process.
- Line no.: line number where the variable is called or used.
- Source file: name of the source file, where the variable is called or used. A blank means that the line refers to the main process file (see column Process file).

If you change the source code the window is not updated automatically. To do so, use the button Scan Global Variables in the parameter window.

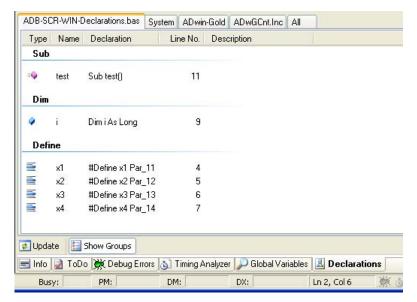
3.11.5 Declarations Window

The Declarations window displays all declarations related to the current source code file and the linked include and library files. For update of the display, click the Update button.

Declarations of other source code files will not be displayed—even if combined within a project.

The window is part of the Info range (see page 86).





The declarations are displayed sorted under tabs, representing the declaration sources:

- [file] .bas: Declarations within the source file: local variables, arrays, instructions (Sub, Function) and symbolic names (#Define).
- System: System variables and instructions being implemented in ADbasic, if they fit to the current compiler settings.
 - Global variables PAR and FPAR are not displayed here. Please note the Global Variables Window (page 92) and the function "Displaying used global variables and arrays" (page 44).
- ADwin-Gold, ADwin-light-16: Instructions for hardware access, which are implemented in ADbasic and fit to the current compiler settings.
- [file].inc: Variables and instructions being declared in this include file. Such tabs only show up if there are #Include lines in the source code file.



- [file] .lib: Variables and instructions being declared in this library file. Such tabs only show up if there are Import lines in the source code file.
- All: All valid declarations of the above sources.

The window columns can be sorted with a click on the column header. With active option Show Groups, declarations are grouped by type.

If you change the source code the window is not updated automatically. To do so, use the Update button.

The display of declarations is only available, when the option Parse Declarations under Editor – General (see page 60) is active.

3.12 ADtools

ADtools is a collection of simple utility programs, with which you can display and change the global variables (Par, FPar) and arrays (Data) of ADwin systems. These programs aid the development of processes for the ADwin system by: displaying the status or values, changing them with practical tools, displaying simple measurement sequences in a graph.

Start one of the ADtools simply from the vertical bar at the right.

Please note: You can add shortcuts to own files or folders to the bar (see Adding file and folder shortcuts, page 44).

Each *ADtool* is its own independent Windows program; each can be started several times, allowing for comprehensive views of parameters of interest on the computer monitor. Once an appropriate screen layout is selected, the whole configuration may be saved and used later.



The following ADtools are available:

	22
١	123

TDigit

Global variable and array values can be displayed and adjusted.



TGraph

Global array contents can be displayed in a graph.



TButton

Button control for booting the ADwin system, loading, starting or stopping a process, or setting a parameter value.



Displays the value of a variable by a simulated LED. The LED can be off, on, blinking slowly or flickering rapidly depending on the value. An audible alarm can also be set with this tool.



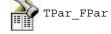
Global variable and array values can be viewed as an analog dial.



Global variable and array values can be adjusted with a potentiometer-style control.



Start/stop, adjust timing, and display information about the processes loaded on the *ADwin* system.



All or selected global variables can be displayed or entered.



TFIFO

Save FIFO array data into a file.



Up to five PAR variables can be displayed in binary (as DIL switch) and in hexadecimal notation, and adjusted.



TString

Save and/or load a configuration to/from several ADtools.



ADtools

saves and loads a user-defined configuration of several *ADtools*.

All further information about the help programs can be found in the online help of the used *ADtools* program.



4 Programming Processes

This chapter provides information about how to build and structure an *ADbasic* program and which variables can be used.

By help of the Using the Development Environment, the completed *ADbasic* program is compiled and transferred to the ADwin processor. There, the program will be executed as an independently running process.

4.1 Program Design

An *ADbasic* program is an ASCII text file created with the editor of the development environment, using an extended Basic syntax. The compiler translates this source code into an executable process for a specific *ADwin* system.

The source code consists of any number of command lines; each containing one instruction or assignment (exception see : Colon). One line may contain up to 2048 (ASCII-) characters; exception see #Include.

ADbasic accepts instructions and variable names in lowercase and uppercase letters (for more clarity all examples use unique spelling).

A program consists of up to 4 sections, which take on different tasks when executed on the *ADwin* system. fig. 14 outlines the ideal steps for an *ADbasic* program.



Each program must at a minimum, have an Event: section.

Optionally functions and subroutines can be defined, as well as libraries and "include"-files to be included.



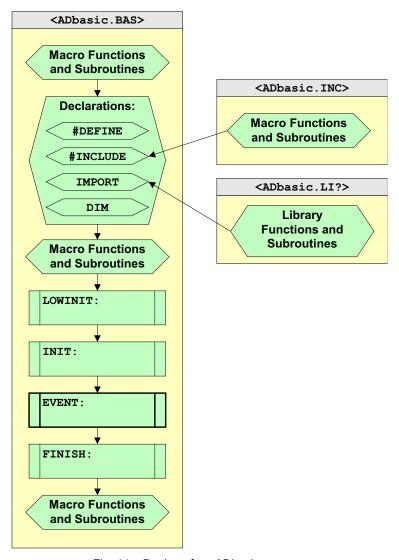


Fig. 14 – Design of an *ADbasic* program



4.1.1 The Program Sections

Each of the program sections (see fig. 14) start with a keyword, as described below.

LowInit: can only be used within high-priority processes.

When the process starts, this section is executed only once and is used for initialization, for instance of variables or data I/O lines. It is always executed prior to the execution of the Init: section (if there is one) and at low-priority, level 1.

This section is ideal for extensive initialization sequences, because it can be interrupted, due to its low-priority.



Init: is similar to the LowInit: section, as it is executed only once
at the start of the process. However, it will be executed with the priority
that has been assigned for the process (menu item Options / Process).

This section cannot be interrupted when configured as high-priority and should therefore be rather short.



- Event: is the main program section, which is (characteristically) called in regular time intervals until it is stopped. This section is triggered by a cyclic timer event or an external event, depending on the configuration.
- Finish: is executed only once after a process has been stopped; it is, therefore, the counterpart to the initialization. This section is always executed at low-priority, level 1.



The LowInit:, Init: and Finish: sections are optional, while the Event: section is not and must be included in your program.

4.1.2 User defined instructions and variables

You can document self-defined instructions etc. (see page 24) so the inserted description text is automatically shown as tooltip while programming.

The Declarations Window (see page 93) displays an overview of all defined instructions and variables.

All user-defined names (except symbolic names) must begin with a letter and may only consist of alphanumeric characters (a-z, A-Z, or 0-9) or an underscore ("_")¹. Special characters like German umlauts (Ä, Ö, Ü) are not allowed

Please note: Names of local arrays must not begin with the string Data_, otherwise using the array will cause a compiler error.



and there is no case sensitivity. The name length is only limited by the maximum line length (2048 characters).

Symbolic names

The instruction **#Define** defines symbolic names (see page 193). Group all of these definitions at the beginning of the file and before the start of the program sections.

Symbolic names are often used to give a name to constants, global variables and global arrays, but also to expressions.

Arrays and Local Variables



In an *ADbasic* program, the local variables and all arrays must be declared with <code>Dim</code> before they can be used (see page 195). The global variables <code>Par_n</code> and <code>FPar_n</code> are already pre-defined and do not need to be declared. Variables and arrays have no defined contents after being declared, therefore they should be initialized.

Within the process, all variables and arrays are available in all program sections. The global variables and arrays may also be accessed from other processes and from the PC, in order to exchange data.

Global variables and arrays being used in a program can be displayed in the Global Variables Window (page 92).

Macros

A macro function Function ... EndFunction or subroutine Sub ... EndSub call inserts the macro into the program text where it is being used (see also chapter 4.5.1 on page 125). However, the macro definition cannot be done within the program sections (see fig. 14 on page 98).

The group of macros and libraries is referred to as procedures.

Libraries

Libraries must be included before the program sections that use them. Library functions Lib_Function ... Lib_EndFunction and subroutines Lib_Sub ... Lib_EndSub, when used more than once within a program, require less memory than similar macro functions or subroutines described above (see also chapter 4.5.3 on page 126).

The group of macros and libraries is referred to as procedures.



4.2 Variables and Arrays

4.2.1 Overview

Da	ata structure	Name	Data type	Notes		
Global variables and arrays						
	Variable (Scalar)	Par_1Par_80	Long	Pre-defined, not declarable,		
		FPar_1FPar_80	Float, Float64			
	System variable	Processdelay	Long	memory area DM (up		
		Process_Error	Long	to T11)		
		Prozessn_Running	Long			
	One- or two- dimensional array (vector)	Data_1[][] Data_200[][]	Long, Float32/64, Float, String, FIFO	Name Data_ not changeable, only declaration of array number and dimension.		
Local variables and arrays						
	Variable (Scalar)	selectable	Long, Float32/64, Float	Must be declared.		
	One-dimensional array (vector)	selectable	Long, Float32/64, Float, String	Must be declared.		

Variables are normally stored in the internal memory DM and arrays in the external memory DX (memory map, see chapter 4.3.1), if not determined explicitly. With processor T12, variables are stored in the memory area Cacheable; for arrays the array size is decisive, see Memory Areas (T12).

All data types have a length of 32-bit, only Float64 has 64-bit length.

4.2.2 Data Structures

In ADbasic, there are two main types of data structures:

Variables (scalars)
 Each variable can store one value only.

VAR



Arrays, one- or two-dimensional.

ARRAY

An array consists of any user-defined number of array elements, each storing one value.

One-dimensional global arrays Data_n may also be used as FIFO (a ring buffer, which works according to the principle: First in, first out, see chapter 4.3.5 on page 115).

The maximum number of variables and array size are limited only by the memory size of the *ADwin* system.

The compiler differentiates

 Global Variables (Parameters) variables and Global Arrays (see chapter 4.2.5 and chapter 4.2.6):

All processes as well as computer applications can access global variables, for instance to exchange data.

System variables are global variables (see page 108).

Local Variables and Arrays (see page 109):

Local variables are available only in the process, function, or subroutine where they have been declared.

Variables and arrays are declared with the <code>Dim</code> instruction; this determines the data type, as well as the necessary memory place, and allocates it to the variable name.

For easier programming, global variables Par_1 ... Par_80 and FPar_1 ... FPar_80 are already pre-defined; thus, global variables do not have to (and cannot) be declared.

The compiler recognizes the declaration of global arrays by the names Data_n, where "Data_" is a fixed text and "n" is the array index number (1...200) specified.



After declaration, variables and array elements have an undefined value and thus should be initialized with a useful value (e.g. zero). Exception: After power-up of the *ADwin* system, the global variables are automatically initialized with zero.

4.2.3 Data Types

A data type must be indicated when declaring variables and arrays.

The compiler processes the following data types:

LONG: 32-bit integer values with the ranges:



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

FLOAT32, FLOAT64 Floating-point values with 32 bit / 64 bit since processor T12.

Accuracy	Value range		
32 bit	$-3.402823 \cdot 10^{+38} \dots -1.175494 \cdot 10^{-38}$		
32 DIL	+1.175494 · 10 ⁻³⁸ +3.402823 · 10 ⁺³⁸		
64 bit	-1.797693134862315·10 ⁺³⁰⁸ -2.2250738585072014·10 ⁻³⁰⁸		
	+2.2250738585072014·10 ⁻³⁰⁸ +1.797693134862315·10 ⁺³⁰⁸		

The value ranges is not equivalent to the IEEE floating-point format.

The data types Float32 / Float64 replace the data type Float (see below). For reasons of compatibility, the compiler still accepts Float, but replaces it by Float64.

FLOAT until T11: Floating-point values, the accuracy depends on the processor:

Processor	Accuracy	Value range
T9, T10	32 bit	-3.402823 · 10 ⁺³⁸ 1.175494 · 10 ⁻³⁸ +1.175494 · 10 ⁻³⁸ +3.402823 · 10 ⁺³⁸
T11	40 bit	$-3.402823668 \cdot 10^{+38} \dots -1.175494351 \cdot 10^{-38} +1.175494351 \cdot 10^{-38} \dots +3.402823669 \cdot 10^{+38}$

The value range is not equivalent to the IEEE floating-point format.

Accuracy of 40 bit with processor T11 is solely restricted to:



- · Calculations inside the ADwin system.
- · Evaluation of constants by the compiler.

The 40-bit accuracy may not be used or displayed on the PC since data will only be transmitted—for reasons of speed—as 32-bit values between PC and *ADwin* system.

In memory, a 40-bit float variable allocates 64 bit.

 STRING: ASCII character strings, in which each character is stored as a single array element (for details see chapter 4.3.6 on page 117). A



single character corresponds to an integer 8-bit value in the range 0 \dots 255.

Unicode is not supported.

The obsolete data types Short and Integer—used with processors before T9—were replaced by data type Long. For reasons of compatibility, the compiler accepts these data types furthermore but automatically replaces them by Long.



When combining integer and floating-point values, a type conversion will occur. Under certain circumstances, this may cause other calculation results than expected. More about this is found in section "Type Conversion" on page 122.

With 32-Bit floating-point values until processor T11, bit patterns of invalid values in the *ADwin* hardware are converted during transfer to the PC into different values, see following table. Numbers inside the valid value range (normalized numbers) stay unchanged.

With processor T12, the IEEE denominations as #INF are displayed.

IEEE denomination	Bit pattern area	Value or display on the PC
+0	00000000h	0
Positive denormalized	0000001h	0
numbers	007FFFFFh	
Positive normalized	00800000h	+1,175494 · 10-38
numbers	7F7FFFFFh	+3,402823 · 10+38
+∞ (Infinity, #INF)	7F800000h	3.402823E+38
Signaling Not a number	7F800001h	3.402823E+38
(SNaN)	7FBFFFFFh	
Quite Not a number	7FC00000h	3.402823E+38
(QNaN)	7FFFFFFFh	
-0	80000000h	0
Negative denormalized	80000001h	0
numbers	807FFFFFh	
Negative normalized	80800000h	-1,175494 · 10-38
numbers	FF7FFFFFh	-3,402823 · 10+38



IEEE denomination	Bit pattern area	Value or display on the PC
-∞ (Infinity, #INF)	FF800000h	3.402823E+38
Signaling Not a number (SNAN)	FF800001h FFBFFFFFh	3.402823E+38
Indeterminate	FFC00000h	3.402823E+38
Quite Not a number (QNAN)	FFC00001h FFFFFFFFh	3.402823E+38

The next section illustrates, in which notation a numeral value can be entered.

4.2.4 Entering Numerical Values

You can use 4 different notations in order to enter numerical values. The following examples assign the (decimal) value 930 to a variable $\bf x$.

For floating-point values, the dot "." is used as decimal separator (English notation).

1. Decimal notation:

$$x = 930$$
 LONG | $x = 930.0$ FLOAT |

Please note the difference: The number 930 has the Long data type, while the number 930.0 has the Float data type. This is important when you use both data types in one expression (see chapter 4.4.2).



2. Exponential notation:

Here 9.3E2 stands for 9.3×10^2 , where "E" is followed by the exponent to the basis of 10 (max. 2 decimal places).

3. Binary notation:

```
x = 111010001b LONG
```

4. Hexadecimal notation (an h is added):



```
x = 3A2h LONG
```

If the hexadecimal value begins with a letter (A-F), a leading zero (0) must be added: Instead of "F6h" the value must be written "0F6h", otherwise the compiler takes the value as the name of a local variable.

With a given numerical value in source code, you can switch between decimal, binary, and hexadecimal notation using [CTRL-SPACE].

4.2.5 Global Variables (Parameters)

All running processes and the computer can access global variables and arrays; therefore, they are ideal for data exchange between the processes or between the processes and the computer (see also chapter 6.3.1 "Data Exchange between Processes"). 80 integer variables, 80 floating-point variables as well as up to 200 arrays of the Long or Float data type are available. All variables and array elements have a length of 32-bit. With processor T12, the global float variables have a length of 64-bit (Float64).

System Variables, also globally available, are described on page 108.

With processor T12, global variables are slower than Local Variables and Arrays. We recommend using global variables only if data exchange with other processes or the PC is required.

Global variables can be used anywhere in a program without being declared. Since the variables have an undefined value at program start they should be initialized with a useful value (e.g. zero). Exception: After power-up of the *ADwin* system the global variables are automatically initialized with zero.

A list of global variables and arrays being used in the program can be displayed in the Global Variables Window (see page 92).

The global variables are also termed parameters and have the names:

- Par_1, Par_2, ... Par_80 with the Long data type for 32-bit integer values.
- FPar_1, FPar_2, ... FPar_80 with the Float64 data type for floating-point values. For processors until T11, the data type is Float.



Example



Contrary to other variables, global variables, Par_n and FPar_n, must not be declared because they are pre-defined and are already known to the compiler.



4.2.6 Global Arrays

Global arrays enable the exchange of data between the processes on the ADwin system or the computer (see also chapter 6.3.1 "Data Exchange between Processes"). Up to 200 arrays of the Long or Float data type are available.

Since size and data type are selectable, global arrays must be declared at the beginning of a program and preferably be initialized, too (Else the array elements have undefined values).



Global arrays can be declared as standard arrays, 2-dimensional Arrays (page 114), or as Data Structure FIFO (page 115).

A list of global arrays and variables being used in the program can be displayed in the Global Variables Window (see page 92).

With processor T12, global arrays are slower than Local Variables and Arrays. We recommend using global arrays only if data exchange with other processes or the PC is required.

The compiler recognizes the declaration of global variables by their names Data n, where "Data" is a fixed text and "n" is the array number (1...200). The names for **DATA** arrays are:

```
Data 1, Data 2, ... Data 200.
```

Other array numbers are not allowed. However, the declaration of nonsequential array numbers is permissible, for instance Data 5 without Data 1 ... Data 4 is allowed. In your program, the compiler differentiates the arrays by their numbers.

Example



```
REM Declare the array 5 with 20000 elements of the type Long.
Dim Data 5[20000] As Long
REM Declare the array 3 with 7x5 elements of the type Float.
Dim Data 3[7][5] As Float
```

There is more information about 2-dimensional arrays in chapter 4.3.4 on page 114.

The maximum size of the array depends on the memory size. For instance on an ADwin system with 16MiB memory, an array of up to 4 million elements of the Long type may be declared. For processor T12, please note the automatic assignment of memory areas by means of array size, see Memory Areas (T12).

After the array has been declared, each individual element can be accessed. The first element of an array has the index 1.





Do *not* assign a value to the element 0 of an array, for instance with Data 1[0] = ...



Examples

```
Rem The value of the 200th element from array 5 is assigned Rem to the global integer variable Par_1.

Par_1 = Data_5[200]

Rem In this program line, the 345th element from the array Rem Data_5 gets the value 4000.

Data_5[345] = 4000

Rem This instruction assigns the value 300.1 to the 1st element Rem of the 2 dimensional array Data_3.

Data 3[1][1] = 300.1
```

A variable can be used as an index number of an array element:

```
'Here, too, as in the example above, the value 4000 is 'assigned to the 345th element of the array Data_5.

number1 = 345

Data 5 [number1] = 4000
```



However, a variable cannot be used as number of an *array*. The following instruction results in an error message of the *ADbasic* compiler:

The compiler determines <code>Data_num</code> to be the name of a local array, which (probably) has not been declared and is therefore not available. Instead, use the notation <code>Data_2</code>. Note the different syntax highlighting of the variables.

4.2.7 System Variables

In order to get information about the status of the *ADwin* system, the following system variables are available. These are global variables that can be accessed by all processes and by the computer. More information can be found in the description of the instructions.

```
Prozessn Running
```

Returns the status of the process n (with n = 1...10): the process is running, just being stopped or already stopped (see page 269). The variable can only be read.



Process Error

Returns the number of the previous error of process n, if debug mode is active (with n = 1...16, see page 268). The variable can only be read.

Processdelay

The nominal time interval, in which time-controlled processes are called by the counter, is the processdelay (cycle time). With the system variable **Processdelay** (see also page 265), you query and set this time, measured in clock cycles of the counter.

You read and write into the variable Processdelay in the sections Init: and Event: only. But writing into the variable is only allowed once per section, because otherwise the status of the ADwin system may become instable.

Please note that the workload of the processor is at least less than 90 percent, and must not exceed 100 percent.



User Version

T12 only: Using the constant User_Version (see also page 305) you can query the version number of a program. You set the version number in the Process Options dialog box (page 56).

4.2.8 Local Variables and Arrays

All local variables and arrays, needed for a process must be declared before the start of the first section of the *ADbasic* program and preferably be initialized, too (else the variables have undefined values).



Variable names can consist of any alphanumeric characters (a-z, A-Z, or 0-9) or an underscore ("_")¹. Special characters like German umlauts (Ä, Ö, Ü) are not allowed and there is no case sensitivity. The length of variable names is only limited by the maximum line length (2048 characters).

Variables (scalars) can be defined as either integer values (type Long) or floating-point values (type Float), and each are 32 bits long.

With processor T12, local variables and arrays work faster than global variables and arrays (see chapter 4.2.5 and 4.2.6). Please note also the automatic assignment of memory areas by means of array size, see Memory Areas (T12).

Please note: Names of local arrays must not begin with the string Data_, otherwise using the array will cause a compiler error.





Example

```
Rem Define the variable 'value' with data type Long
Dim value As Long
Rem Define the variables value1 and value2 with
Rem data type Float
Dim value1, value2 As Float
```

Variables may also be declared as a one-dimensional array, allowing the user to generate and/or process an array of variables. The number of elements to dimension in an array is put into square brackets after the array name.



Example

```
Rem Define an array with the length 100, with the name
Rem 'value', and the data type Float
Dim value[100] As Float
```



The first element of an array has the index 1, in the example: value [1]. The element index 0 must not be accessed at all.

4.3 Variables and Arrays – Details

4.3.1 Variables and Arrays in the Data Memory

This section applies to processors T9 ... T11 only. For processor T12, please see chapter 4.3.3.

The user can explicitly determine, in which memory area (internal or external) to store arrays and local variables (see below). This allocation is made, in the source code, when the variable is declared using the Dim statement using the additions At DM Local or At DRAM Extern. With processor T11, an additional memory area is available via At EM Local.

Without the use of these allocation statements, all variables are stored in the internal memory DM and all arrays in the external memory DX.

We recommend using internal memory for variables and (small) arrays for fast access. The slower, external memory is more suitable for arrays, due to its size.

The fig. 15 shows examples of declarations, in order to store variables and arrays in the different memory areas.



Variable /	Memory area	Source code declaration
Array		
Local	Internal (DM)	Dim var As <vartype></vartype>
Variable		or
		Dim var As At DM_Local
	Addit. (EM)	Dim var As At EM_Local
	External (DX)	Dim var As At DRAM_Extern
Array	Internal (DM)	Dim array[5] As At DM_Local
	Addit. (EM)	Dim array[5] As At EM_Local
(global/ local)	External (DX)	Dimarray[5] As
		or
		Dim array [5] As At DRAM_Extern

Fig. 15 – Allocation of the Memory Area with Declarations

The global variables Par_1...Par_80 and FPar_1...FPar_80 are predefined in the internal memory (DM), therefore they cannot be re-declared in the external memory (DX).



4.3.2 Memory Areas (T9...T11)

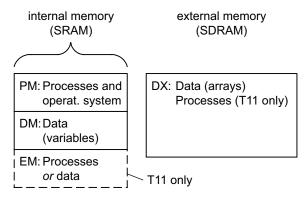
This section applies to processors T9 ... T11 only. For processor T12, please see chapter 4.3.3.

The processor of the *ADwin* system uses a fast internal memory (SRAM) and a huge external memory (SDRAM).

Half of internal memory is available as program memory PM and as data memory DM. Processor T11 has an additional internal memory EM, which may be used either as program or as data memory.

memory.





- Program memory (PM):
 Program memory occupies half of the internal SRAM and contains the operating system and processes.
- Internal data memory (DM)
 The internal data memory occupies half of the internal SRAM for storing the global and local variables.
- Additional memory (EM)
 Additional internal memory EM is available with processor T11 only.
 Additional memory can be used either as data memory or as program
- External data memory (DX)
 The external data memory covers the external SDRAM and stores the global and local arrays.

On T11, external memory can store processes of up to one megabyte size.

Data in the internal memory (DM) can be accessed faster than data in the external memory (DX) by approximately a factor of five. The access speed to the memory areas PM, EM, and DM in internal SRAM is equal.

Memory size (SRAM, SDRAM) is an ordering option and cannot be upgraded. The size of memory areas is the only limiting factor to the size of the processes and for the number of declared variables and arrays (indirectly to the size of source files, too). In the status line of the development environment, the amount of available memory of PM, DM, EM, and DX is displayed in bytes.

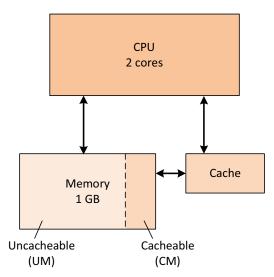


4.3.3 Memory Areas (T12)

This section applies to processor T12 only. For processors T9 ... T11, please see chapter 4.3.1 and 4.3.2.

The processor T12 provides an external main memory with a size of 1 Gigabyte. Program code and data are not stored in different areas but together in the main memory.

The CPU access to main memory is quite slow. To accelerate data access the CPU uses a fast internal memory area, the cache with a size of 512kB. The CPU first fills the cache with the values and instructions, which are required next from the main memory. As needed, the CPU loads additionally required data and code into the cache; the point of reload time cannot be predicted in *ADbasic*.



The part of main memory, which may deliver information to the cache, is called "cacheable" (CM), the greater rest of main memory is called "uncacheable" (UM). Parts of CM and UM are used by the *ADbasic* operating system after booting, so about 16MB CM and 1087MB UM remain free for use.

The CM holds variables, small arrays and program code from the *ADbasic* source code; big arrays are stored in the UM area. Normally, the automatic assignment of memory areas is pretty fine.

Arrays, which shall be stored in CM, must not exceed a specified number of elements, referring to data type:



Data type	local array	DATA_x
Long, Float32	< 2.499	< 250.000
Float64	< 1.249	< 125.000
String	< 9.999	< 1.000.000

If the memory size e.g. in CM is too small for program code and data you receive an error message in *ADbasic* when the process is loaded to the *ADwin* system. It may then be reasonable to declare the memory area of <code>Data_x</code> arrays on your own, for example to declare rarely used <code>Data_x</code> arrays <code>At Uncacheable</code> (UM) and frequently used <code>Data_x</code> arrays <code>At Cacheable</code> (CM). Please note that manual memory declaration indirectly influences the memory assignment of other arrays.

You cannot select the memory area of local arrays.

Program code and data are assigned to CM and UM as follows:

Data block	CM	UM
ADbasic operating system	Х	-
Program code, local variables, local arrays (<10.000 Byte)	х	-
Local arrays >=10.000 Byte	-	х
DATA_x At Cacheable	Х	ı
DATA_x At Uncacheable	-	х
DATA_x <= 1.000.000 Byte DATA_x > 1.000.000 Byte	x -	_ x

4.3.4 2-dimensional Arrays

Global arrays Data_n may be declared with 1 or 2 dimensions. The basic array features are described in chapter 4.2.6 "Global Arrays".



2-dimensional notation may simplify a problem's solution (compared to 1-dimensional arrays). At the same time it will slow down data access and require additional program memory.

The loss of access speed and the need of additional memory will increase with each access to the 2-dimensional arrays by the program.

The following cases require to access the data of a 2-dimensional array as if it were declared 1-dimensional:

 On the PC, if the data of a 2D-array is transferred to or from an ADwin system.



The other way round, data of a 1D-array on the PC may be transferred to an *ADwin* system, even though the destination array is declared 2-dimensional in *ADbasic*.

 Inside of a library module (Lib_Sub, Lib_Function), which receives a 2D-array as an argument.

With this kind of data access, the order of data in the memory becomes important. As an example a 2D-array shall be declared as

```
Dim Data 1[3][2] As Float
```

The 3×2 array elements will be stored sequentially in the data memory. The following table shows, which element index be used for the 1D-access to the example array.

array index 2D	[1][1]	[1][2]	[2][1]	[2][2]	[3][1]	[3][2]
array index 1D	[1]	[2]	[3]	[4]	[5]	[6]
memory address	n	n+1	n+2	n+3	n+4	n+5

Thus, an element Data_1[3][1] used in the main program had to be accessed e.g. in a library module as fifth element of the passed array:

Please note: This kind of access is permissible only in the two cases mentioned above. In any other case, the 2-dimensional notation is needed.

Generally, this is the mapping of 2D-elements to 1D-elements:



$$DATA_n[i][j] \equiv DATA_n[j \cdot (i-1) + j]$$

where s is the 2^{nd} dimension of n in the declaration. In the example above, there is s=2.

4.3.5 Data Structure FIFO

For applications requiring a large quantity of data to be transferred continuously, we recommend using a Data_n global array with the FIFO data structure: a "First In, First Out" ring buffer.

The basic features of global arrays are described in chapter 4.2.6 "Global Arrays".





The data structure Ringbuffer of the TiCo processor is quite different from a FIFO. TiCo ringbuffer is described in the TiCoBasic manual.

In a ring buffer, data is handled in a special way; like a queue where data is appended to the end of the queue and retrieved from the beginning of the queue. Unlike a "normal" array, data in the array is not accessed by its element number, but by the first or the last element of the array (via a data pointer). Consequently, data elements are read out in the same order as they were written into the array (= First In, First Out).

Global FIFO arrays Data n must be one-dimensional; possible data types are Long or Float.



Example

```
Dim Data 5[1003] As Long As FIFO
```

This instruction declares the global array with the number 5 as FIFO ring buffer with 1,003 elements of the type Long. Please note the special size of a FIFO with the T11 processor (see FIFO).



Please note: A FIFO array cannot be accessed as "normal" array in the source code

Since a FIFO array has a finite number of elements (which is declared), the chain of used and unused array elements form a ring, the ring buffer. The data pointers to the first and last used array element are managed automatically when a new value is assigned to the array or when a value is read out.

After the declaration of a FIFO array the pointer should be initialized with the FIFO Clear instruction.



From the ring structure of the FIFO array it is possible for the head of the data chain to "overtake" the data end. This can only occur when data is written faster into the FIFO than it is being read out. Subsequently, the earlier stored data will be overwritten and lost.

A certain FIFO array can be accessed by indicating its array name (with the corresponding array number).



Example

```
Dim Data 5[1003] As Long As FIFO
Data 5 = 95
                        'Writes the value 95 into the
                        'Data 5 array which is declared as FIFO
Par 7 = Data 5
                        'Reads a value from the FIFO and
                        'stores it in the global variable
                        'Par 7
```

To ensure that the FIFO is not full, the FIFO Empty function should be used



before writing into it. Similarly, the FIFO_Full function should be used to check if there are values, which have not yet been read, before reading from the FIFO.

Example

```
Dim free,used,value1 As Long
Dim Data_1[1003] As Long As FIFO

REM Are there still elements, which are not empty?
free = FIFO_Empty(1)
If (free > 0) Then
   Data_1 = value1
EndIf

REM Are there still elements, which haven't been read?
used = FIFO_Full(1)
If (used > 0) Then
   Par_7 = Data_1
EndIf
#Define free PAR 1
```

4.3.6 Strings

Control characters and texts from other process monitoring devices can be transferred, converted and processed by the *ADwin* system e.g. via an RS-232 interface.

The following instructions are available for string processing:

Asc	Get ASCII number of a character
Chr	Get character from an ASCII number
FloToStr	Convert a float value into a string
LngToStr	Convert a long value into a string
StrComp	Compare 2 strings to be equal
StrLeft	Get left-aligned substring from a string
StrLen	Get length of a string
StrMid	Get substring from a string
StrRight	Get right-aligned substring from a string
ValF	Convert a string into a float value
ValI	Convert a string into a long value
+ String Addition	Operator to concatenate strings





For most string instructions, the library file <STRING.LI*> must be imported (where * indicates the processor type: 9 for T9, A for T10, B for T11, C for T12). The library file is found in the library directory (default: <C:\ADwin\ADbasic\LIB>) after the installation.

A string variable has a structure similar to an array, where each array element contains one character. The dimensioning of a string for 5 characters is as follows:

```
Import String.LI9
Dim text[5] As String
```

This dimensioning reserves an array for the string in the memory, which is structured as follows:

text[1]	Length of the string in characters (5); not available with T12
text[2]	Character 1 of the string
text[3]	Character 2 of the string
text[4]	Character 3 of the string
text[5]	Character 4 of the string
text[6]	Character 5 of the string
text[7]	The end of string character, terminating zero (00h)

Each element requires 4 bytes of memory, or 1 byte with T12. The first element text[1] and the last element text[7] of the string are automatically reserved by the *ADbasic* compiler.

Note: Do not use element number 0 in ADbasic, here text[0]. With processor T12, do not use element numbers 0 and 1, here text[0] and text[1].

After dimensioning the elements are not initialized. Values must be assigned to a string before the string can be read from or processed.

Normal Assignment

Values are assigned to string variables by placing the string's actual text into quotation marks (") and setting it equal to the string variable. *ADbasic* stores the corresponding ASCII numbers for each character in the memory (see ASCII table in the Appendix).



Example

```
text = "HELLO"
```



Element Index	Memory Contents	Meaning
text[1]	05h	Length of the string in characters (5); not available with T12
text[2]	48h	ASCII value for "H"
text[3]	45h	ASCII value for "E"
text[4]	4Ch	ASCII value for "L"
text[5]	4Ch	ASCII value for "L"
text[6]	4Fh	ASCII value for "o"
text[7]	00h	End-of-string character

Only characters with the ASCII values between 20h...7Fh (displayable characters in the normal ASCII character set), should be assigned using quotation marks, except the following characters, which are assigned using the escape sequence:

single quote ('): \x27

– double quote ("): \x22

– backslash (\): \x5C

Character Assignment via Escape Sequence

The escape sequence is used to include ASCII values or control characters into a string. Each escape sequence transfers a single ASCII value to the *ADbasic* compiler, which stores it in memory without any changes.

The escape sequence is indicated as part of a string inside quotation marks with the notation \xhh, where hh is the ASCII value to be transferred, written in hexadecimal notation. Each escape sequence must have exactly 4 characters.

Example

 $text = "\x48\x45\x4C\x4C\x4F"$



The memory contents is the same as the one given in the previous example.

The escape sequence is necessary for assigning characters that are not displayed (such as line feed, carriage return, etc.). The range of values using the escape sequence is from <code>00h</code> to <code>FFh</code>.



In addition to the notation \xhh, there are also special escape sequences for frequently used (control) characters:

Sequence	ASCII Value	Meaning
\\	5C	Backslash (\)
\t	09	Tab (TAB)
\n	0A	Line Feed (LF)
\r	0D	Carriage Return (CR)

It is also possible to combine the notations described earlier when assigning values to a string variable.



Example

 $text = "HE\x4C\x4CO"$

The memory content is the same as the one given in the previous examples.



The end-of-string character should not be inserted into a string (example: text = "HE\x00LLO"). The *ADbasic* compiler will properly assign each character to the string, but errors will most likely occur when the string is processed further on.

String Assignments that are NOT Recommended

Unfortunately, it is possible to insert characters with ASCII values <code>00h...1Fh</code> or <code>80h...0FFh</code> on various ways, for instance typing [?] or the German characters [ß] and [Ö], or using "copy and paste", or the key sequence [ALT]+number. We explicitly do recommend using Character Assignment via Escape Sequence!

The compiler is able to process such characters. However, these characters may either have no unique ASCII value (because they are country-specific), or they may cause unwanted actions (carriage return, etc.) and program errors.



We recommend inserting any control or special characters into a string only by using the escape sequence.



4.4 Expressions

An expression is the term, which you assign to a variable, pass to an instruction as argument, or use as a condition. It consists of an arbitrary combination of:

- Simple data: constant, variable or array element.
- Operators being applied to arguments, which are expressions themselves.
- Instructions from the ADbasic instruction set or user-defined instructions

4.4.1 Evaluation of Operators

For the evaluation of an expression, it is important to understand the order, in which the operators are used. The operators are divided into categories, which are resolved according to priorities: A category of higher priority is processed before a category of lower priority (see fig. 16).

Please note that automatic Type Conversion may in some cases influence the evaluation of an expression (see page 122), too.

Operator	Category
н н	Delimiter of character strings
ADbasic keyword	Instruction, function, variable, etc.
=	Assignment
()	Parentheses
-	Negation of a constant
^	Power
* /	Multiplication / Division operators
+ -	Arithmetic operators
And Or XOr	Binary operators
< > =	Comparison operators
And Or	Boolean operators

Fig. 16 – Priories of operator categories (highest priority on top)

Example

```
var = Par 1 + Par 2 * Par 1^3 / 4
```





```
corresponds to var = Par 1 + (Par 2 * (Par 1^3) /4)
```

If 2 or more operators, appearing in the same line, have the same priority (or if there are the same operators), the compiler processes them in the order they appear, from left to right.



Using a negative sign with variables, may return unexpected results, in some cases, and can be avoided by using parentheses.



Example

```
var = 1/-x 'not recommended

var = 1 / (-x) 'correct: negative inverse value
```

4.4.2 Type Conversion

In *ADbasic*, variables can (after dimensioning) generally be used without paying attention to their data types (Long or Float, see also chapter 4.2.3 "Data Types"). If necessary the data of the Long type will automatically be converted into the Float type.

The following description also applies to processor T12, but data type Float should then be replaced by Float64.



Do not mix up this conversion with the instructions Cast_FloatToLong or Cast_LongToFloat, which do quite a different job (see page 182).

Consider the following special features:

Cut off decimal places



If a floating-point value is assigned to an integer variable, then the decimal places are cut off and will be lost.

With processor T11, this effect shows up on some divisions of float values and provides unexpected results:

```
FPar_1 = 6
FPar_2 = 3
Par_1 = FPar_1 / FPar_2 'unexpected result: Par_1=1
Par 2 = Round(FPar 1 / FPar 2) 'correct result: Par 2=2
```

Therefore, it is useful—especially with T11—to use the instruction Round (page 276) before assigning the value.

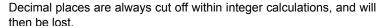
Converting all Integers to Floats



If an expression contains a floating-point value, *all* integer values are automatically converted *before* the expression is evaluated. This applies if an integer expression

- · is assigned to a floating-point variable or
- serves as argument for an ADbasic instruction, expecting a floating-point value.

Example





Here the floating-point variable FPar_1 and the floating-point value 4.0 demand the conversion of all integer values.

Prevent integers from Conversion

Even using parentheses does not prevent the automatic conversion into Float. To absolutely make calculations in Long, an individual program line must be used.

Ŵ

Example

```
Par_1 = 2
Par_2 = 5
Rem a conversion is made here:
FPar_3 = (Par_2 / Par_1) + 0.2'FPar_3 = 2.7
Rem ... but not here:
Par_9 = Par_2 / Par_1 'Par_9 = 2 (cut off)
FPar 4 = Par 9 + 0.2 'Result: FPar 1 = 2.2
```

Conversion of Arguments

The following expressions are always evaluated separately (and will be converted, if necessary, as described above):

- Each individual parameter for an instruction.
 Additionally a cut off may occur according to the parameter's data type (data type see instruction's description).
- Each argument passed to a function or subroutine.
- Each individual part of a condition within a Boolean expression in an If...Then or Do...Until where the condition can be linked with And or Or.





Example

```
Par_1 = 2
FPar_2 = 5.5

Rem Both conditions are true, Par_1 is not converted
Rem into Float, therefore Par_3 = 1.

If ((Par_1 / 4 * 3 = 0) And (FPar_2 * 1.1 > 5.5)) Then
Par_3 = 1
EndIf

Rem The condition with Float does not influence the
Rem Long calculation, therefore Par_3 = 0.

If (FPar 2 * 1.1 > 5.5) Then Par 3 = Par 1 / 4 * 3
```

4.5 Selection structures, Loops and Modules

When writing extensive programs, *ADbasic* provides the following structure elements:

- Control structures to help shorten large sections.
 - Loops for sections being frequently repeated:

```
Do ... Until or For ... Next.
```

· Structures for case-by-case decisions:

```
If ... EndIf, #If ... #EndIf Or SelectCase ... EndSelect.
```

- Subroutine and Function Macros to define frequently used program sections as
 - Subroutine macros with Sub ... EndSub
 - Function macros with Function ... EndFunction
- Libraries of compiled subroutines and functions, which can be included into a user's source code with Import:
 - Library subroutines with Lib Sub ... Lib EndSub
 - Library functions with Lib Function ... Lib EndFunction
- Collections of source code sections and program modules in Include-Files, which can be included into a user's source code using #Include_filename.Inc

```
More information and examples of instructions can be found in chapter 7 "Instruction Reference".
```



4.5.1 Subroutine and Function Macros

The syntax of subroutine and function macros is simple, only requiring the terms Sub ... EndSub and Function ... EndFunction around the relevant program sections, like parentheses. Contrary to subroutines, functions return a value.

Source code is more clearly structured with subroutines and functions. These subroutines and functions define macros, whose complete instruction block is inserted (prior to compilation) into the place of the source code, where it is called.

Please note: upon each subroutine or function call, the generated binary file is increasing in size. You can use library functions or subroutines as an alternative (see below).

You will find more information about the structure of macro modules in the instruction reference (page 217: Function ... EndFunction; page 301: Sub ... EndSub).

4.5.2 Include-Files

Source code sections can be collected and stored in an "include" file. Such files (as well as the source code they contain), can very easily be included into a source code file with the **#Include** instruction.

The content of an include file is based on the same rules as normal source code files. However, in most cases include files contain only subroutine and function macros.

When an include file is generated, the source code is entered in the same way as a "normal" ADbasic file but saved using the File / Save as menu option with the Include file *.inc file type.

Depending on the include file's source, attention must be paid to the position, at which the file is included into another source code file, to maintain a working program structure. If the include-file contains function and subroutine macros, it must be included before the Init: section or after the Finish: section. You can also include an include-file into source codes of library files and other include-files (nested include).

Include files installed with *ADbasic* contain only subroutine and function macros, defining instructions for hardware access. Thus, the appropriate position for these files to be included is the beginning of the source code (see page 98).





The following include files are delivered with *ADbasic* and contain the instructions to access the hardware I/Os. You can insert the files easily as text snippet:

ADwL16.inc *ADwin-light-16*;

insert text snippet with IL[TAB].

ADwGCnt.inc ADwin-Gold: Instructions for CO1 and

ADwGCAN.inc CAN add-on;

insert text snippet with IG[TAB].

ADwinGoldII.inc ADwin-Gold II;

insert text snippet with IG2[TAB].

ADwinPro_All.inc ADwin-Pro, ADwin-Pro II;

insert text snippet with IP[TAB].

4.5.3 Libraries

In a library, compiled library subroutines and functions (modules) can be assembled. With the Import instruction, the modules of a library can be included into a process where they will be called.

The library modules are similar to the subroutine and function macros. They are created in a source code file using the Lib_Sub ... Lib_EndSub and/or Lib_Function ... Lib_EndFunction instructions. The library file is then compiled using the Build / Make lib file menu option.



Please note with processor T12: Each library consists of three files $\star.lic$, $\star.o$, and $\star.h$. All three files must be located in the same folder to make the library run correctly.

If you compile a source code which imports a library, the binary file contains only library modules being called in the source code. Calling library modules multiple times does not increase the size of the binary file. Compared to macro functions and subroutines, library modules require less memory when they are called more than once. However, additional execution time is needed for calling them (compare to chapter 4.5.1 "Subroutine and Function Macros").



Please note that a library module cannot call a library module within the same library file. We recommend using macro functions and subroutines instead. Alternatively, additional libraries may also be used.

When nesting libraries (including a library within another library), the source code calling the libraries must include all levels (see fig. 17), otherwise an error message will be returned by the compiler.



Recursive calls of library functions or subroutines are not allowed.



If you use libraries with processor T12, check if the set stack size is sufficient. An unsufficient stack size can lead to data loss or program abortion. You find more under Window Stack Test Analysis (page 84).



You will find more information about the structure of the library modules in the instruction reference (page 234: Lib_Function ... Lib_EndFunction; page 239: Lib_Sub ... Lib_EndSub).

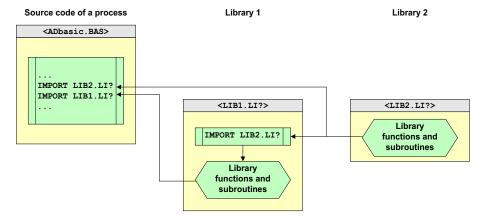


Fig. 17 - Interlaced Libraries

The following libraries are delivered with *ADbasic*.

String.li* Instructions for string operations.

Math.li* Special mathematics instructions.

FFT.li* Instructions for FFT transformations.

Macros versus Libraries

What should you do, if you can use a macro as well as a library to solve a task? In this case, you have to weigh execution speed against program size.

A macro will be expanded in the code as often as it is being called in the source code. If your program calls a macro function 100 times, there will be 100 copies of the macro in the resulting program code. On the contrary, the code of a library function will always exist once only. Aiming for a smaller program size, the library function would be the better choice.

If a program calls a library function, there is some processing expense transferring the program execution to the library and later returning to the calling program code. If "calling" a macro, there is no such processing expense since



the code is part of the program. Regarding higher execution speed, the macro does it better.

Weighing program size against execution speed is normally irrelevant for beginning programmers. A closer examination of this issue will only gain importance with huge or time-critical applications.



5 Optimizing Processes

The *ADwin* system is designed to quickly and precisely execute control and measurement tasks. Depending on the requirements it may be necessary to optimize your *ADbasic* program for a faster processing time.

The following pages illustrate steps for optimizing a program. Many factors determine the optimization process, which needs to be considered with each individual case. Please refer to the "*ADbasic* Tutorial and Programming Examples" manual to find more examples for optimizing processes.

5.1 Measuring the Processing Time

For optimization, it is important to measure the processing time of a process cycle or of a program section. This can be done using the internal counters of the *ADwin* system.

The processor of the *ADwin* system has two internal counters, one for high-priority processes and another for low-priority processes, each incrementing in different clock rates (see fig. 19 on page 148). The current counter value can be read using the <code>Read_Timer</code> instruction; the counter corresponding to the running process's priority will automatically be read out.

After power-up, both counters are set to the value 0 (zero), then continually incremented in fixed clock pulses.



-χ̈́-

The processing time of the program is measured as a time difference. In the following example, the processing time of a time-critical program section (minus an offset) is stored in the global variable Par 1.

To obtain the offset run the both <code>Read_Timer</code> lines in succession—without any program lines between them—and calculate the difference of these values. The offset is to calculate only once for the surveyed program.

Example



If Par_1 in the example above equals 37, the time-critical section of the high-priority process requires 37×25 ns = 925 ns.

It is also possible to measure the time difference between two external events, in an event-driven process. In the following example, the measurement is stored in the global variable Par 1.



Example

```
Dim oldtime, time As Long
Init:
   oldtime = Read_Timer()
Event:
   time = Read_Timer()
   Par_1 = time - oldtime
   oldtime = time
```

5.1.1 Annotations for T12

Regarding processor T12, measuring the timing of selected program lines depends a lot on the surrounding conditions. This refers especially to the effects described below. You don't have to make changes if you measure the time difference between the event signals.

The notes of the previous section are still valid.

 The compiler may strongly optimize calculations in the ADbasic source code, in order to achieve a fast processing. Thus, the compiler may (among others) summarize calculation steps or arrange independent calculations in different order.

As a result, a time measurement with two **Read_Timer** lines may enclose other instructions than given in the source code. This effect decreases with the length of the enclosed program segment.

On the other hand: Accesses to the *ADwin* hardware like **ADC** or **DAC** are always processed in the order that you have set in the source code. Program lines with global variables **Par**, **FPar**, **Data** and with **Read Timer** are excluded from optimization to a large extent.

The processor architecture enables parallel processing of multiple instructions. The total processing time is reduced, but also the processing time of single instructions lose relevance.
 In addition, parallel processing sometimes makes it difficult to determine the processing time of a single instruction.



Therefore, you cannot calculate the processing time of a program segment by adding processing times of the appropriate single instructions. We recommend measuring the timing of larger program segments.

 If the processor accesses ADwin hardware or memory, a subsequent time measurement with Read_Timer may already be processed while the hardware access is still in action.

The reason is the buildup of the processor in several levels, which may work in parallel. As soon as the processor core passes a hardware access instruction to the next processor level, it starts processing the very next instruction in order to accelerate the total processing speed.

The less instructions are enclosed by time measurements the more you have to consider processing times of previous or following instructions. Using the instruction <code>Read_Timer_Sync</code> you can make sure that an access to <code>ADwin-Hardware</code> or the main memory is completed before the time is read.

Read Timer Sync takes more time than Read Timer.

- It makes a difference if you use global or local variables. Global variables Par, FPar, Data are slower to process since the processor has to synchronize the variable value with the cache or main memory with every single access; this is not required with local variables.
- The processor contains several buffers for data and instructions. The status of the buffers depend a lot on the previously processed instructions. Multiple measurements of the same program segment may therefore result in different processing times.

5.2 Useful Information

5.2.1 Accessing Hardware Addresses

Many of the *ADwin* system functions are managed by its control and data registers. These functions can quickly be executed by *directly* accessing the relevant registers with the **Peek** and **Poke** instructions. Here, "directly" means that the functions' addresses are not calculated in the process cycle, but passed as constant values: saving computing time for the calculation.

The addresses for control and data registers can be found in the relevant hardware manual.



5.2.2 Constants instead of Variables

A calculation is executed faster when the values are specified as constants and not as variables.



Example

```
FPar 1 = Sqrt(Par 2) 'with Par 2=17
FPar 1 = Sqrt(17)
```

For the first calculation, the value of the variable Par 2 must be determined during run-time. The root must then be calculated and assigned to Par 1. In the second calculation, the compiler already has determined the value. During run-time it will only be assigned.

5.2.3 Faster Measurement Function

With the ADC instruction, an analog-to-digital (A/D) conversion for a channel with a specified gain is carried out. In order to make its application easier, the instruction is kept rather simple and combines several sequences (see hardware manual for the ADwin system).

There are different situations resulting in a faster processing when using these individual sequences, compared to using the ADC instruction.

For instance, the ADC instruction does not consider that the ADwin-Gold-system has two ADCs, which are able to convert two different channels at the same time. This is illustrated in the following example:



Example

```
REM Example for Gold
REM Set both multiplexers of the ADC to the channel 1
Set Mux(000000b)
Rem wait for settling time
Start Conv(11b)
                       'Start conversion on both ADCs
Wait EOC(11b)
                       'Wait for end of conversion
Par 1 = ReadADC(1)
                       'Read ADC1
Par 2 = ReadADC(2)
                       'Read ADC2
```



The ADwin-light-16 system has only one ADC.

5.2.4 Setting Waiting Times Exactly

Using a waiting time, you can easily set an exact offset between 2 instructions, for example to bridge the multiplexer settling time between Set Mux and Start Conv.



The instruction for setting the waiting time depends on the processor type:

Processors T9 and T10:

The instruction **Sleep** sets the waiting time exactly: The processor stops for the pre-set time, causing the next instruction to be started with appropriate delay.

Waiting for the multiplexer settling time of 14 µs on a Pro I module would then work like this:

```
Set_Mux(2,00000b) 'Set Mux to channel 1

REM Here a calculation may be done, which e.g. takes

REM 8µs of the free processor time.

Sleep(60) 'wait remaining 6µs until 14µs

Start Conv(2) 'Start conversion
```

Processors T11, T12:

According to the addressed hardware there are several instructions:

Instruction	addressed hardware
CPU_Sleep	Processor (refers to Sleep)
IO_Sleep	Gold II only: all hardware
P2_Sleep	Pro II only: modules on the Pro II bus
P1_Sleep	Pro-T12: Digital I/O channels of the processor module Pro-T11: modules on the Pro I bus, but also Pro II bus and external DRAM

If the waiting time gaps a delay between I/O-instructions for Pro II modules, P2_Sleep is the right choice; for Pro I modules it is P1_Sleep, with Gold II it is IO_Sleep. The instruction CPU_Sleep makes sense only rarely.

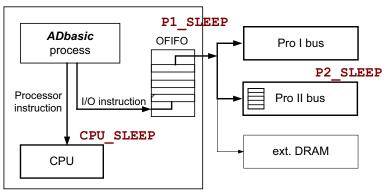
Waiting for the multiplexer settling time of 4 µs on a Pro II module would then work like this:

```
P2_Set_Mux(2,0100000010b)'Set Mux to channel 3
P2_Sleep(400) 'Make Pro II bus wait 4µs
P2_Start_Conv(2) 'Start conversion
REM The calculation follows but now; the T11/T12 processor will
REM process it automatically in parallel with the I/O
REM instructions.
```



Attention with T11: Within the calculation you should use variables from internal memory only. Otherwise the calculation may anyhow not be run until the I/O instructions are completely processed.

Processor T11



Why are there different instructions for the waiting time? The processor T11 runs processor instructions and I/O instructions¹ quasi-parallel (see sketch above). This is very fast, and also leads to parallel and thus separate timing, resulting in 3 instructions for the waiting time.

The quasi-parallel processing is enabled via a 5-level buffer OFIFO: The operating system passes an I/O instruction into the OFIFO (if there is enough space) and immediately starts processing the next instruction. The example above passes the instructions <code>Set_Mux</code>, <code>P1_Sleep</code> and <code>Start_Conv</code> into the OFIFO; the subsequent calculation is then run in the CPU, while e.g. the Pro I bus² is still waiting.



Please note: A calculation that is to be processed in parallel in the CPU, may only use variables from internal memory. The operating system regards each access to the external DRAM, the common memory area for arrays, as an I/O instruction that has to walk through the OFIFO buffer.

I/O instructions are those, which access external devices via the OFIFO buffer. External devices (as regards the CPU) are modules on the Pro I or Pro II bus and the external memory DX.

^{2.} More precisely, the instruction **P1_Sleep** makes the buffer OFIFO wait, not the Pro I bus.



5.2.5 Using Waiting Times

Some instructions require a certain waiting time after being called. This time can be used for other calculations.

The <u>Set_Mux</u> and <u>Start_Conv</u> instructions require waiting time for the settling of the multiplexer and the conversion of the ADCs. During this waiting time, the processor is not busy and could be used for other tasks.

More detailed information about the required waiting times for data conversion can be found in your hardware manual.

The next example is an extension of the previous example, showing how two measurements are executed across two separate ADCs. Compared to the ADC instruction, this enables execution of 4 times the number of measurements.

The key feature of the example is to carry out the individual steps in the conversion process not sequentially but rather in parallel. The time delay for multiplexer setting is carried out during the A/D conversion of the other channels. Both measurement processes are overlapped: The start of conversion (16 bit: time 5μ s) for the channels 1+2 is followed by setting the multiplexer (16 bit: time 6.5μ s) for the channels 3+4.

Example

REM Example for Gold Rev. B, 16 Bit (not suitable for T11)

```
-<u>Ö</u>-
```

```
Init:
 Set Mux(000000b)
                      'Set Mux for first measurement,
                       'channels 1+2
 Sleep(65)
                      'Wait 6.5 µs
Event:
 Start Conv(11b)
                     'Start conversion, channels 1+2
 Set Mux(001001b)
                     'Set Mux, channels 3+4
 Wait EOC(11b)
                     'Wait for end of conversion (1+2)
                     'Read ADC1, channel 1
 Par 1 = ReadADC(1)
 Par 2 = ReadADC(2)
                      'Read ADC2, channel 2
 Sleep(15)
                      'wait remaining 1.5 µs
 Start Conv(11b)
                       'Start conversion (channels 3+4)
                      'Set Mux, channels 1+2
 Set Mux(000000b)
                      'Wait for end of conversion (3+4)
 Wait EOC(11b)
 Par 3 = ReadADC(1)
                     'Read ADC1, channel 3
 Par_4 = ReadADC(2)
                      'Read ADC2, channel 4
```



The Init: section sets the multiplexer up for the first measurement so that the A/D is ready the first time the Event: section is executed.



It is very important that adequate delay for the multiplexer settling time and A/D conversions be provided, or incorrect measurements or A/D conversion failures may be obtained. There are some hints in chapter 5.2.4 "Setting Waiting Times Exactly".

5.2.6 Optimization with Processor T11

This section describes how to use the specific features of the T11 processor to speed up a process, especially by optimized memory access.



If nonetheless you reach the processor's limits, further optimizations are possible, but only in connection with your specific application. Please contact our support (see address inside the manual's cover page). **Using internal memory**

For time-critical sequences, use variables and arrays in the internal memory (EM or DM) as possible. While variables are declared automatically in the internal memory, arrays (both local and global) have to declared as follows:

```
Dim DataLocal[100] As Long At DM_Local Dim Data 5[2000] As Float At DM Local
```

Compared to internal memory the access of processor T11 to external memory slows down for 2 reasons. On the one hand the memory access is passed into the OFIFO buffer (see page 134) as I/O instruction, which can cause delays. On the other hand the administration of external memory is slower than of the internal memory.

Data in the internal memory (DM) can be accessed faster than data in the external memory (DX) by approximately a factor of five. The access speed to the memory areas PM, EM, and DM in internal SRAM is equal.

Accessing the external memory

For the access to the external memory, try to use—as far as possible in the program—data blocks, and don't access single values. If using block-wise data transfer the processor enables an accelerated access, so e.g. transferring a block of 20 values quicker than 3 single values.

As an example, the block data transfer is quite useful, if a lot of measurement values are read in short time: At first the collected data packet is saved in quick internal memory. As soon as the measuring task reaches a non-critical stadium, the data are transferred as block into external memory using the instruction MemCpy, leaving the internal memory ready for the next collected data packet.



5.3 Debugging and Analysis

Debug and timing modes are the *ADbasic* hands-on tools for debugging and program analysis. The modes are activated via the "Debug" menu (see page 40) and add their helping features to those programs, which are compiled with active mode.

The Call graph is a tool of static analysis, which enables you to analyze program structure und optimize memory assignment.

Please note: Activating of the modes produces additional program code. Thus, the program will need a longer processing time as well as additional memory—at times at considerable rate. We therefore recommend using these tools for developing and testing of programs only.



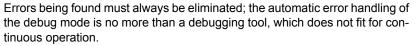
5.3.1 Finding Run-time Errors (Debug Mode)

The debug mode is a helping tool to find the following run-time errors in *ADbasic* programs:

- Division by zero
- Square root from a negative value
- Access to too large / too small element numbers of an array

Without debug mode, these run-time errors are simply ignored, i.e. though the result of the program line is undefined it is nevertheless used for the following program. This may cause, depending on the program, an unwanted behavior, in worst case even the "crash" of the *ADwin* system.

The option "Debug mode" is activated from the "Debug" menu; do then compile the source code to be checked. On occurrence of a run-time error it is automatically displayed in the "Debug Errors" windows (see Debug mode Option, page 67). As well, the run-time error is being handled to maintain a stable mode of operation.





Details about activating and display of run-time errors are shown in section "Debug mode Option" on page 67.

5.3.2 Checking the Timing Characteristics (Timing Mode)

The *ADwin* system is designed in such a manner that an arriving event signal for a high-priority process (externally generated or by an internal counter) immediately starts the relevant process cycle. Processes with such "good"



timing characteristics are deterministic and execute their tasks exactly at a predetermined period of time.

To check timing characteristics of processes requires some effort, especially when changes are to be made later, to obtain good timing characteristics. This effort is worth its price, when required higher frequencies or additional tasks put the processor workload to its limit. Another example are process cycles not start as exactly as predetermined according to the measurement task.

In the timing mode, information is generated, which can be used to check selected high-priority processes if they have "good" timing characteristics. For these processes, 7 parameters are calculated, which are displayed in the Timing Analyzer Window.

Processes have good timing characteristics when the following situations *do not* (or rarely) occur:

- An event signal does not start a process cycle immediately, but a certain (not exactly defined) time later.
- An event signal does not start a process cycle at all, but gets "lost". Even several lost event-signals are possible.

In the first case, the operating system tries to make up the delay by using available idle times in the workload of the processor, until all process cycles again start at the pre-defined period of time. In the latter case, the operating system cannot make up the delay: Event signals and therefore process cycles are really lost (see chapter 6.2.5 "Different Operating Modes in the Operating System").

An optimal timing characteristic, especially of the high priority processes, is obtained in 2 steps by:

- 1. Checking Number and Priority of Processes
- 2. Optimal Timing Characteristics of Processes (Use Timing Mode)

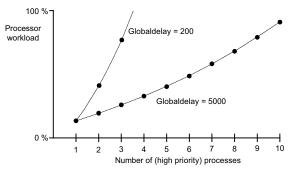
Checking Number and Priority of Processes

In a high-priority process, only time-critical tasks should be processed, all other tasks in one or more low-priority processes (or even processed on the PC).

If possible use only one single high-priority process. Several processes can very often be merged to a single process; if the Processdelay is identical, we highly recommend this. It's worth the effort—especially with a shorter Processdelay of the processes—because the processor workload will be essentially



lower even if the same tasks are executed. The graphic below illustrates this more clearly:



With several high-priority, time-controlled processes, process cycles cannot be prevented from starting time-delayed (except their Processdelay values are integer multiples of each other).

Optimal Timing Characteristics of Processes

A high-priority process has an optimal timing characteristic under the following conditions:

- All process cycles of the process have an almost equal processing time.
- The processing time of the process cycle is as short as possible.
- The Processdelay of the process is longer than the longest processing time of all process cycles.

Nevertheless, the processor workload for high-priority processes must leave enough processor time available for the tasks of low-priority and communication processes.

To get more information about the timing characteristics of interesting processes proceed as follows:

- 1. Activate the timing option with Debug ▶ Enable timing analyzer.
- 2. Compile (and start) the *ADbasic* source code.

For each source code, which you compile with active timing option, information about timing characteristics are generated automatically. We



recommend viewing only a small number of processes at once, so that the timing characteristics will not be influenced too much (see below).

- 3. Disable the Debug ▶ Enable timing analyzer option again, so that other processes being compiled do not unnecessarily generate timing information.
- 4. Open the Timing Information window via the Debug ▶ Show timing information menu item.



Note that the timing characteristics on the *ADwin* system depend on the number and type of the processes, thus causing accordingly different parameters. One reason for this fact is the process management of the operating system (see chapter 6.2.5 "Different Operating Modes in the Operating System").



The evaluation of the information is made during run-time and needs approx. 60 clock cycles additionally (when using a T9, T10 or T11 processor) per process cycle and process. The parameters in the window are continuously updated and refer to the time passed since the last start of the processes. A short description of the parameters can be found under Timing Analyzer Window, page 88.

The (minor) change of timing characteristics by the timing mode itself cannot be avoided and exists even if no parameters are displayed. This may result under certain circumstances in further latencies, and is also reproduced in the corresponding parameters; in short processes with a short Processdelay, a processor workload of more than 100% can be reached sometimes, so that the communication to the PC is interrupted.

Please note that during compiling high-priority processes using the timing option, a low-priority process can be considerably delayed.

5.3.3 Analyzing program structure

For better understanding of huge and complex *ADbasic* programs, do a basic program analysis using the call graph. You can also use the call graph for Optimizing memory intensive procedures (page 141).

The Call Graph Window (see page 82) displays all procedures (Sub), Function , Lib_Sub , Lib_Function) being called in an ADbasic source code in a static call graph. The call graph is context-insensitive, i.e. the tree contains one node for each procedure. The handling of the call graph is described in chapter 3.10.7 "Call Graph Window".

One simple application of the call graph is analyzing the program structure of a given complex *ADbasic* project possibly consisting of several files including libraries. The call tree displays the nesting of procedure calls and enables you to easily recognize the calling order.



The call graph is suitable:

- to introduce yourself into the logic of an unknown program,
- to review the current state of your own program development,
- to monitor the steps to implement your programming concept in an overview,
- in teamwork, to transparently visualize the main program logic of different authors.

The graph can be used as a basis for further analyses, such as an analysis that tracks the flow of values between procedures.

The call graph always refers to a single process only.

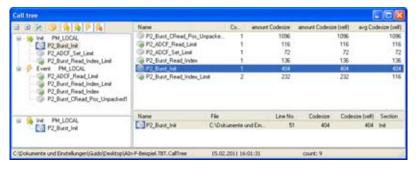
5.3.4 Optimizing memory intensive procedures

If you have a program exceeding the *ADwin* memory size, follow these steps to reduce the required memory size:

Use the call graph to find memory intensive procedures.
 Detect procedures with huge Total code size and macros (Sub, Function) with a high Count value in the upper right pane.

The Call Graph Window (see page 82) displays all procedures (Sub , Function , Lib_Sub , Lib_Function) being called in an *ADbasic* process in a static call graph together with their size in memory.

Find more about the handling of the call graph in chapter 3.10.7 "Call Graph Window".



 Reduce the number of macro calls (Sub, Function) with huge Code size since macros are expanded with every call. Count is the number of calls of a procedure.



With libraries, the number of calls does not matter; see also Macros versus Libraries on page 127.

- Reduce the code size of huge procedures, especially of often called macros.
- Re-compile the program and check the required memory size again.
 Since program sections can be defined in specified memory areas, this information is given in the panes to the left, behind the section names.

With processor T11, please make sure to use memory areas either for data or for program sections. This is also true for the additional memory EM. If you still mix both data and program sections in the EM memory, the information is processed (according to the program sequence) sequentially and therefore more slowly.



6 Processes in the ADwin System

An *ADwin* system has the capability to control complex test stands while rapidly executing measurements. Programs using one or more *ADbasic* processes are used to provide this capability. Within these processes you can specify how analog and digital data is processed within the *ADwin* system and how it is exchanged with external devices and PC.

After starting the process, the program¹ in the *ADwin* system is (characteristically) restarted and processed in regular time intervals. This calling of a process cycle is triggered by one of the following start signals, called events:

- 1. Timer Event: A pulse of the internal counter. You determine for each process separately, in which time interval (processdelay) a new event is triggered.
- External Event: An external signal, which arrives at the event input of the ADwin system. This could be for instance the pulse of an incremental encoder.

Only one of the 10 possible processes can be controlled by an external event, all other processes have to run time-controlled.

You define the exact function of a process in the *ADbasic* source code:

- The initialization in the sections LowInit: and/or Init:.
- The actual function of the process cycle in the central Event: section (event loop).
- The final processing in the Finish: section.

It is possible to control the processes from the computer that is the processes are started, stopped or their processdelays changed. You can do this with *ADbasic* as well as with other development environments such as C++ or Visual Basic.

With the bootloader option, it is also possible to have processes start automatically on power-up of the *ADwin* hardware. For programming the bootloader, see manual "*ADwin* bootloader".

^{1.} More precisely: the program section Event:.



6.1 Process Management

6.1.1 Types of Processes

Within the *ADwin* system several processes can run simultaneously. The operating system is responsible for calling the process cycles according to specified rules, and for their being processed by the CPU without blocking each other.

When referring to a "process" in this manual, we mean one of the processes 1...10 that you have programmed.

You assign a priority to each process and thus determine the interaction and timing of the processes. There are the priorities:

- Processes with High-Priority
- Processes with Low-Priority

Low-priority processes are further divided into the levels -10 (low) up to +10 (high).

The process priority is set via the menu Options \ Process Options.

Process	Function	Priority ^a
110	User-defined processes with functions and priorities you can freely define	low level <i>n</i> / high
11, 12	Predefined input / output processes	high
15	Process for controlling the flashing LED in ADwin-Pro and ADwin-Gold systems	low, level 1
Communica- tion	Communication between the <i>ADwin</i> system and the computer: Instruction and data exchange	medium

a. The meaning of the priorities is described in the following sections

Fig. 18 – Overview of all processes

The standard processes, processes 11 and 12, are only necessary when using the drivers for the Labview and Testpoint environments. These processes can be loaded during the boot process along with the operating system, either from a developer environment (for more details, see the *ADwin* developer manual), or from *ADbasic*. To do this, set the option Load Standard processes to Yes in the *ADbasic* menu Options / Compiler. If you are not using one of these applications you can stop the transfer of the standard processes during booting (setting No).



The communication process (see page 146) is part of the operating system. It receives commands of the computer and exchanges data between the *ADwin* system and computer only when the computer requests them.

If you transfer more than one process with the same process number to the system, only the last process transferred is executed, because the earlier transferred processes are overwritten.



6.1.2 Processes with High-Priority

Processes with "high" priority get preferential treatment from the operating system:

- The maximum latency from when a high priority process is called by an event to when execution of the process begins is 300ns.
- A high-priority process cycle cannot be interrupted and is always completely processed. During this time all process cycles with low-priority are blocked.

Neither another high-priority process cycle nor a stop instruction can interrupt a running, high-priority process cycle. In both cases, the system will complete the current high priority process cycle before proceeding.

In time-controlled high-priority processes, the cycle time (processdelay) can be set in intervals of 25 ns.

The software should be written so that time-critical measurement processes run with high-priority and all others run with low-priority, so that the processor can process the time-critical process cycles without any interference from other operations.



The sections LowInit: and Finish: of a process—if there are any—are always executed with low-priority, priority level 1, even if the process is set to run with high-priority.



6.1.3 Processes with Low-Priority

Process cycles with low-priority are immediately interrupted when a process cycle with a higher priority is called and will stay interrupted until that higher priority process cycle has finished.

Low-priority processes are further divided into the priority levels –10 (low) up to +10 (high). Process cycles with a low level can be interrupted by those with a higher level at any time. The processors T11 and T12 keep strictly to the priority levels for process management (see chapter 6.2.3 on page 150).



Low-priority processes of the same priority level participate in time slicing. Here the operating system apportions the computing time to the process cycles alternating and in equal time slices. One time slice takes 2ms (processor T9), 1ms (processors T10, T11), or 100 µs (processor T12) on average.

Low-priority processes must always be time-controlled. The cycle time (processdelay) can be set in discrete intervals; interval size see fig. 19 on page 148.

Processes with low-priority on principle do not influence the time characteristic of high-priority processes, but vice versa they surely do.

6.1.4 Communication Process

The communication process has a priority level between the priorities "high" and "low". Therefore, it can interrupt low-priority process cycles any time and can be interrupted by high-priority process cycles.

If the computer requests information from the *ADwin* system, the communication process must respond within 250 ms or a time-out will occur, the communication between the computer and the *ADwin* system may be interrupted. In this case, the message The ADwin system does not respond will be displayed and the system will have to be reinitialized by rebooting the *ADwin* system. The time-out is independent of the communications interface, either USB or Ethernet.

The cause of an interruption in the communication is that the communication process does not have enough processor time allocated to it. This can be caused by the following facts:

- the processdelay of the high-priority processes is too short or
- the processing time of a high-priority process cycle is too long.
 More about this subject can be found in chapter 6.3.2 on page 154.

6.1.5 Memory fragmentation

The operating system of the *ADwin* system cares for storing processes, arrays and variables at an adequate memory position and using them correctly. Therefore, the user normally has no problems with memory management, which thus would need no explanation.

Under certain circumstances, the error message "Not enough memory or memory access error. Please reboot the ADwin system." 1 occurs.

^{1.} With processors T9 and T10, there is no error message, but the *ADwin* system has a workload of 100%.



Often, the reason is an external memory fragmentation, which arises from processes or data arrays being loaded into memory multiple times and with increasing size; a typical action e.g. for the development of new processes. A simple solution is to boot the *ADwin* system and load the data anew.

A memory fragmentation is defined as free storage being dispersed between allocated regions. If now a new data block like a process or an array is loaded into memory—where it can only be stored as complete unit—it may happen that the data block does not fit into any of the free memory fragments. You receive the above error message and have to reorganize the memory in order to obtain free memory of sufficient size.

Booting and loading anew is useful here, since the data blocks are stored consequently without a gap and the free memory remains as a unit.

The result of memory fragmentation can be a memory, which cannot store any more data—regardless of being a process or a data array. According to the processor type an error message pops up or the *ADwin* system shows 100% workload. A simple solution is to boot the *ADwin* system and load the data anew.

Example: Two (quite large) processes are already loaded to memory. Process 1 is to be replaced by new code with increased size, but the data block does not fit into memory neither before nor behind process 2 and you receive the mentioned error message. After booting and loading in different sequence, process 1 can be loaded any time without the risk of memory fragmentation.



As an alternative, you may also delete process 2 manually and load both processes anew. The advantage is to retain the values of global variables and arrays; for a global array this is only true, if the array size remains unchanged. The difficulty in manual deletion, especially with increasing number of processes, is to keep the overview of the order, in which processes are stored in memory.

Alike with process memory, memory fragmentation may also occur in data memory multiple dimensioning of data arrays with changing size, e.g. during development of a process. If so, loading the process will release the allocated memory of the (newly dimensioned) arrays and for each array a new memory



range has to be found, leading to memory fragmentation. The simple solution is to boot the *ADwin* system and load the data anew, too.

Generally, global arrays may be deleted individually in *ADbasic* using Clear Data (see chapter 3.9.7 on page 71), in order to obtain free memory of sufficient size. But if a fragmentation occurs, most times you don't know the order, in which arrays are stored in data memory, so booting is normally to be preferred.

Please note: If global arrays are used in several processes, they have to be declared identically in each process. In this case, it is practical to save these declarations of global arrays into an include file and include the file into all of these processes (see also chapter 4.5.2 "Include-Files").

6.2 Time Characteristics of Processes

6.2.1 Processdelay

The time interval, in which time-controlled process cycles are called by the counter, which is the cycle time of the **Event**: section of the process. The cycle time is usually measured in clock cycles of the system clock and called *Processdelay*, (in earlier *ADbasic* versions: Globaldelay). The process delay of each process is specified by setting the value of the system variable **Processdelay** (see also page 265).

Also, the time interval between the end of the last instruction in the Init: section and the start of the Event: section is one *Processdelay*.

The time resolution of the system clock depends on the process priority and on the processor type:

Processor	Priority	
	High	Low
Т9	25ns	100 µs
T10	25ns	50 µs
T11	3. 3 ns	$3.\overline{3}$ ns = $0.00\overline{3}$ µs
T12	1ns	1ns

Fig. 19 – The time resolution of the system clock (units of the processdelay)

For instance, a processdelay with the value 1000 means that for a high-priority process on a processor T9 it is called in time intervals of $1,000 \times 25 \text{ ns} = 25,000 \text{ ns} = 25 \mu \text{s}$, while for a low-priority process in a time interval of



 $1,000 \times 100 \,\mu s = 100,000 \,\mu s = 100 \,ms$. You can specify this event interval in the program line:

Processdelay = 1000

The processing time of a process cycle must not, even under worst case circumstances, be higher than the cycle time, so that each process cycle can be called at the time specified (with Processdelay). Differences in the computing time may arise from different program sections, which are run conditionally (If, Case).

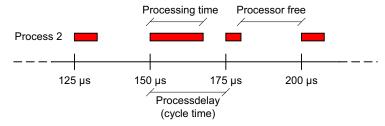


Fig. 20 – Processdelay and processing time in high-priority process cycles

Example

If an extensive calculation is executed only every, say 1,000 measurements, then the long processing time of this process cycle must be shorter than the cycle time. In order to obtain short process cycles, one alternative is to divide the calculations into small steps and to process a step in each process cycle. Thus, the process cycles have a consistent, short processing time.

6.2.2 Precise Timing of Process Cycles

If you have (as shown in fig. 20) only one high-priority process, it will be called and processed exactly in its time schedule.

Make sure that the processing time of a high-priority process cycle never exceeds its cycle time (in the example below: 25 µs). This process cycle cannot be interrupted, thus other process cycles can only be partially processed or not at all, for instance the important communication process.

With processors until T11: If there are several high-priority processes, the actually running process cycle can influence the time schedule of the remaining process cycles. In fig. 21 for instance, process 1 has to start with a delay when the processing of the active process 2 has finished.





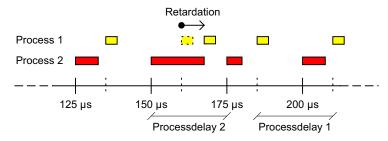


Fig. 21 – Delay of a high-priority process cycle



Keep the execution time of high-priority process cycles as short as possible. Have event loops, which require long processing time, or calculations, the result of which cannot be immediately be processed, always run in process cycles with low-priority.

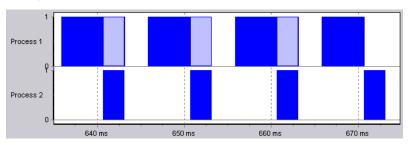
A low-priority process depends on the time characteristics of all other process cycles with the same or higher priority. Each interruption minimizes the time, a low-priority process cycle can use the computing power, and in the worst case it will not be called at all.

6.2.3 Low-Priority Processes with T11 and T12

The processors T11 and T12 manage low-priority processes strictly be their priority level -10...+10, see Level in the Process Options dialog box on page 56. In contrast, priority levels are of little importance with T9 or T10. Nevertheless, communication process and high-priority processes still take precedence over all low-priority processes.

The process management of low-priority processes is different for:

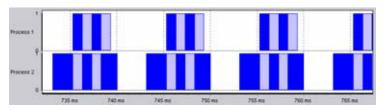
 Processes of different priority levels: All processes of lower priority level are interrupted, as soon as and as long as a process of higher priority level is processed.





In this case, process 2 is of higher priority level and therefore interrupts process 1 several times.

 Processes of equal priority levels: The processes take part in time slicing that is, within the priority level, the operating system portions out the processor's operating time to the process cycles alternating and in equal time slices (1 ms).



The example shows the changeover of the processes quite clearly. Please note the rule that a process—process 1 in this example—immediately receives a time slice upon the call of its process cycle.

There is a rare and special case, which annuls time slicing: A process receives a lot of processing time, if both it is frequently called and its process cycle takes shorter than one time slice. With each call, the process interrupts other processes of the same priority level and thus "steals" their processing time.

6.2.4 Workload of the ADwin system

The workload of the processor on the *ADwin* system is the ratio of the computing time used to the available computing time, indicated in percent.

You can monitor the workload of the processor in the status line display Busy within the development environment (see chapter 3.10.6). This value gives you an indication if the processor still has enough computing time available to complete all of the required activities.

The workload of the processor should exceed 90 percent only in exceptional cases and must not exceed 100 percent.

Please note for processor T11: Although a workload below 90% is displayed, an overload can exist, so that some process cycles might be processed with delay. In this case, the overload exists on the internal Pro I or Pro II bus, not in the processor, and can therefore not be displayed.

Please note for processor T12: Although a workload below 90% is displayed, an overload can exist. During overload—which is therefore to be strictly avoided—process cycles get lost. In this case, you revise your process or



adjust the Processdelay to reduce the workload.

In the Process Window (Lost events), you can read how much process cycles a process has lost already.

6.2.5 Different Operating Modes in the Operating System

The operating system differentiates between 2 operating modes for the timing characteristics in high-priority processes, depending on the fact if several time-controlled (high-priority) processes are active or only one. If an additional externally controlled process is running, is of no importance here. The externally controlled process is managed separately by the operating system and can therefore be seen as a third operating mode.

Only with T12: The operating system has only two operating modes, distinguished by high priority (timer-controlled, externally controlled) and low priority processes.

Single Time-Controlled Process

With processor T12, several timer-controlled processes are handled the same way as a single timer-controlled process.

With a single time-controlled process, the operating system uses hardware components to process the event signals of the internal counter. In this case, the operating system processes an incoming event signal very quickly.

The hardware components can buffer if an event signal has arrived, but not how many event signals have arrived. If an event signal has arrived, the operating system activates the next process cycle at the fixed period in time (Processdelay see chapter 6.2.1), unless a high-priority process cycle is just being processed. In this case, the operating system activates the next process cycle immediately after the currently running process cycle.



If a number of event signals arrive during a high-priority process cycle, only one single process cycle is called and not the number of arrived process cycles, respectively. As a consequence all but one of those event signals are lost. Therefore, we recommend setting the process cycles absolutely shorter than the cycle time (Processdelay) of the process.

Several Time-Controlled Processes

This section does not apply to the processor T12. Several timer-controlled processes are each handled the same way as a Single Time-Controlled Process (see previous section).

With several time-controlled processes, the operating system itself manages arriving event signals. This operating mode is working slower due to this man-



agement efforts, but the number of all arriving event signals are buffered for each process. Thus, it is ensured that for each event signal a process cycle is started, even if this happens later than the pre-defined instant of time.

Frequently the time schedules for starting the process cycles are the reason for the fact that event signals continuously occur during the processing of another process cycle. With other words, the Processdelay values are not integer multiples of each other. We recommend using only few processes; it is often possible to merge several processes to one single process (this results in a smaller processor workload, too).

Always keep in mind that the processor workload depends very much on the number of processes running. Thus, a task performed by 2 (or even more) processes will always take more workload than the same task within a single process. This is the more of importance the shorter a Processdelay is (see also chapter 5.3.2 on page 137).

Example: Processes 1 and 2 with a very short Processdelay running as a single process each generate 10% workload; both processes together have a workload of 55%.

Externally Controlled Process

The operating mode for the externally controlled processes is, independent of time-controlled processes, always the same. The operating system manages the external process as a single time-controlled process (see above) that is, arriving event-signals are processed very quickly, but event signals can also be lost.

An external event signal is a rather important information—in particular, because it cannot be predefined by the *ADwin* system—and must not get lost (finding lost events, see page 88). Therefore, note to have short process cycles in this process (in the section **Event:**).



6.3 Communication

6.3.1 Data Exchange between Processes

Data can be exchanged between different processes via global variables (Par_n, FPar_n) or global arrays (Data_n). Data can be exchanged with programs running on the PC using these variables and arrays as well.

If global arrays are used in several processes, they have to be declared identically in each process. In this case, it is practical to save these declarations of global arrays into an include file and include the file into all of these pro-





cesses (see also chapter 4.5.2 "Include-Files"). Global variables can be used by one process to control a process running simultaneously.



Example

Process 1 is a function generator and Process 2 is a controller. The function generator regularly writes the generated value into the global variable Par_10. At every event loop the controller process reads out the global variable Par_10 and uses its contents as set point of the control loop.

Thus, the function generator very easily controls the set point of the controller. All *local* variables and arrays of Process 1 are hidden from Process 2 (and vice versa). Take into account that the timing characteristics of both processes must be considered.

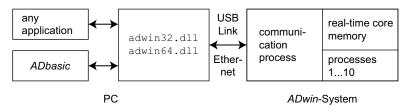
6.3.2 Communication between PC and ADwin System

From PC applications and development environments, you can control the processes on the *ADwin* system, as well as request data from or send data to the system. An *ADwin* system cannot communicate with the computer on its own, but instead responds to requests coming from the computer.

All data exchange is made via global variables (Par_n, FPar_n) or global arrays (Data_n). This refers also to the Data Exchange between Processes (see above).

The communication to the *ADwin* system is managed under Windows with the ADwin32.dll (dynamic-link library). In the *ADwin* system, the communication process is responsible for this task (page 146).

If you are working with the ActiveX interface, the latter is responsible for the communication with the *ADwin* system. Internally the ActiveX interface transfers or gets the data via the ADwin32.dll.





The ADwin32/64.dll has the following tasks:

- Communication with the connected ADwin system via the specified communication interface: USB, Ethernet (TCP/IP).
- Recognizing and handling of communication errors.
- Blocking several computer applications if they want to access the same system at the same time.

With the blocking mechanism, several applications can simultaneously access one or more *ADwin* systems independent of each other.

If a computer application starts the communication to a system, it transfers a device number in addition to the specified instruction. The ADwin32.dll uses this "Device Number" to differentiate between the various *ADwin* systems and assign the corresponding configurations.

The communication functions in the same way also real-time processes, which were created from a Simulink[®] model using *ADsim*. Find more details in the *ADsim* manual.

6.3.3 The Device Number

Each *ADwin* system connected to a computer is accessed via a unique device number (unique to the PC).

You set the device number with the program ADconfig.

In *ADconfig*, you link a Device Number with the communication parameters, which define how a system can be accessed (USB, Ethernet). This is the information the <code>ADwin32.dll</code> needs in order to being able to communicate with the system.

6.3.4 Communication with Development Environments

You access the *ADwin* system from the PC with the help of a user interface. You may generate this user interface with one of the conventional development environments such as ActiveX, Java, Visual Basic, C++, Delphi, or C#.NET, or you may use a ready-made user interface such as Kallisté, TestPoint, DIAdem, or MATLAB.

For each of these, an appropriate driver software is provided, which enables you to access the *ADwin* system. If you have a special request, please contact us. We can also provide turnkey measurement data evaluation programs.

Under Windows, a DLL or ActiveX interface can establish the communication with the system simultaneously from several programs (see also "Communication between PC and ADwin System" on page 154). The special instructions





for your user interface are described more detailed in the relevant *ADwin* developer software.

From your user interface you can

- transfer compiled programs (binary files) into the ADwin system. Compile the program in ADbasic with Build Make Bin File (see chapter 3.9.4 on page 52).
- start, control and stop processes in the ADwin system.
- request data from the ADwin system or send data to the system.

Although the *ADwin* system works independently, you can access global variables and arrays from the user interface any time, without delaying time-critical processes. This way all processes can quickly exchange data with the computer (or with each other).



7 Instruction Reference

Below, the available *ADbasic* instructions for *ADwin* processors are listed. Find instructions for inputs/outputs in the hardware manual.

The instructions are listed in alphabetical order. In the annex, there are instruction overviews sorted by *ADwin* system and by alphabet.

In chapter 7.3 and chapter 7.2, the *ADbasic* instructions are listed for the use of the FFT Library as well as Mathematics Instructions.

7.1 Instruction Syntax

Please note:

- Any expressions can be used as arguments.
- Some arguments require a specified data structure, which are labelled as follows:
 - constant numbers such as 35 or 3.14159, and expressions without variables.

 Character constants (strings) are enclosed in
 - quotes such as "this text".

 VAR variable or array element.
 - array, also identified in the command syntax by its brackets [] after the array name.
 - FIFO fifo array (Data n declared as fifo).
- The expected data type is given for each argument and for a function's return value:
 - LONG | integer number
 - FLOAT floating-point number

With processor T12, FLOAT stands for data type Float64.

STRING character string

LOGIC logic expression in a condition



- If the argument has a different data type than expected, you will get a type conversion of the argument (chapter 4.4.2 on page 122).
- Some instructions can only be used, when a specific library or include file is included. Under Syntax, the relevant include-instruction is indicated (place this command line at the beginning of the source code).

We assume that the necessary library or include file is located in the directory, which is set under the Options Settings menu, Directory item, (see also the instructions #Include or Import).

7.2 Instructions for L16, Gold, Pro

The instructions in this section are valid for the processors of all *ADwin* systems.



+ Addition

The "+" operator adds two values (see also "+ String Addition").

Syntax

Parameters

arameters		
val_1	Addend 1.	FLOAT
		LONG
val_2	Addend 2.	FLOAT
		LONG

Notes

Please note that combining different variable types with the "+" operator will cause a type conversion. During conversion from the type Long into the type Float rounding differences can occur, which influence the result.

See also

```
- Subtraction, * Multiplication, / Division, ^ Power
```



+ String Addition

The "+" operator concatenates two strings (see also "+ Addition").

Syntax

```
Import String.LI*
  'T9: .LI9 / T10: .LIA / T11: .LIB / T12: .LIC
str[] = val_1[] + val_2[]
```

Parameters

val_1[]	Destination string.	
val_1[]	Character string1.	
val_2[]	Character string 2.	

ARRAY

STRING

ARRAY STRING

ARRAY

STRING

Notes

If you concatenate two strings and assign them to another string, the size of the destination string must be declared greater or equal to the sum of the sizes of the input strings.

See also

String "", Asc, Chr, FloToStr, Flo40ToStr, LngToStr, StrComp, StrLeft, StrLen, StrMid, StrRight, ValF, ValI





- Subtraction

The "-" operator subtracts one value from another.

Syntax

Parameters

val_1	Minuend.		
-------	----------	--	--

val 2 Subtrahend.

FLOAT |
FLOAT |
LONG

Notes

Please note that combining different variable types with the "-" operator will cause a type conversion. During conversion from the type Long into the type Float rounding differences can occur, which influence the result.

If you use "-" as a sign of a variable (unary operator), you may in some cases get unexpected results, which can be avoided by using brackets (see also chapter 4.4.1 on page 121).

See also

```
+ Addition, * Multiplication, / Division, ^ Power
```



* Multiplication

The "*" operator multiplies two values.

Syntax

Parameters

val 1	Multiplicator 1.	FLOAT
va1_1	maniphodica 1.	LONG
val 2	Multiplicator 2.	FLOAT
V41_2		LONG

Notes

Please note that combining different variable types with the "*"operator will cause a type conversion. During conversion from the type Long into the type Float rounding differences can occur, which influence the result.

See also

```
+ Addition, - Subtraction, / Division, ^ Power
```



LONG

/ Division

The "/" operator divides one value by another.

Syntax

```
val = val 1 / val 2
```

Parameters

val_1	Dividend.	FLOAT
		LONG
val_2	Divisor.	FLOAT

Notes

Please note that combining different variable types with the "/"operator will cause a type conversion (see chapter 4.4.2 on page 122). During conversion from the type Long into the type Float rounding differences can occur, which influence the result.

If the divisor is a variable with a negative sign, you should use braces to ensure you get the expected result (see also chapter 4.4.1 "Evaluation of Operators" on page 121).

See also

```
+ Addition, - Subtraction, * Multiplication, ^ Power, Mod
```



^ Power

The "^" operator calculates the value of a number raised to a power.

Syntax

```
val = val 1 ^ val 2
```

Parameters

a. aoto. o		
val_1	Basis.	FLOAT
		LONG
val_2	Exponent.	FLOAT
		LONG

Notes

Please note that combining different variable types with the power operator will cause a type conversion. During conversion from the type Long into the type Float rounding differences can occur, which influence the result.

If basis and exponent are variables with even value (but not constants), the power is nevertheless calculated using Float arithmetic. Large results therefore show the typical Float inaccuracy with large numbers.

Example:

If the basis and/or the exponent are a variable with a negative sign, you should use braces to ensure the sign will be considered upon exponentiation (see also chapter 4.4.1 "Evaluation of Operators" on page 121). This is not necessary with constants.

Polynoms are calculated quicker, if you reduce powers by factoring out receiving a multiplication.

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

y = a + x*(b + x*(c + x*(d + x*e))) 'quicker version
```









See also

+ Addition, - Subtraction, * Multiplication, / Division, Exp, LN, Log, Round, Sqrt



#..., Compiler Statement

An *ADbasic* instruction beginning with the "#" sign instructs the compiler to treat the following source code differently before creating binary code.

The following compiler statements are available:

#Define	Definition of symbolic constants: Search and replace character strings in the source code with other character strings.
#Include	Include a file: Insert a file (with source code) into the source code.
#If #EndIf	Conditional compilation: If the condition is true the corresponding code lines are compiled, otherwise deleted.
#Begin_ Debug_ Mode_ Disable	Interrupts the debug mode in a running process.
#End_ Debug_ Mode_ Disable	Cancels the debug mode interruption in a running process.



: Colon

The sign ": " separates more than one instruction within a single line.

Syntax

```
[Step 1] : [Step 2] {: [Step 3] ...}
```

Notes

[Step_n] refers to any program instruction as is otherwise indicated in one individual program line.

We recommend using this instruction only when it makes the source code more clearly structured.

```
Inc Par_1 : Inc Par_2
'Increase Par 1 and Par 2 in *one* line
```



=, Assignment

The operator "=" assigns the result of the expression on the right side of the operator to the variable or the array element on the left side of the operator.

Syntax

```
var = expr
```

Parameters

var	Variable or array.	VAR
		FLOAT
		LONG
		STRING
expr	Expression.	FLOAT
		LONG
		STRING

Notes

If the data format of the expression is not similar to the data format of the destination variable or the array, it is converted into the appropriate data format or the assignment is rejected as illegal. During the conversion rounding differences can occur, which influence the result.



< = > Comparison

The operators "<", "=" and ">" are used to compare two values. In *ADbasic*, these operators can only be found in conditional expressions.

Syntax

Parameters

val_1	Operand.
val_2	Operand.

LONG FLOAT LONG

Notes

The following comparisons are possible:

Operator	Meaning
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
=	equal to
< >	not equal to

See also

```
If ... Then ... {Else ...} EndIf, #If ... Then ... {#Else ... } #EndIf
```



```
Dim value As Long
Event:
  value = -5
  If (value < 0) Then value = 0
  Rem Result: value = 0</pre>
```



Abs

Since processor T11: Abs provides the absolute value of a Float or Long variable.

Syntax

```
ret_val = Abs(value)
```

Parameters

value	Argument.	FLOAT
ret_val	Absolute value of the argument.	LONG
		FLOAT
		LONG

Notes

The smallest negative integer value -2³¹ has no positive counterpart in *ADbasic*; the absolute value of is therefore undefined.

See also

```
Absl, AbsF, Round
```

Dim value As Float



AbsF

AbsF provides the absolute value of a Float variable.

Syntax

```
ret val = AbsF(value)
```

Parameters

value Argument. FLOAT

ret val Absolute value of the argument.

FLOAT

Notes

The execution time of the function takes 150ns with a T9, 75ns with a T10, and 17ns with a T11.

See also

Abs, Absl, Round



Absl

AbsI provides the absolute value of a long variable.

Syntax

```
ret val = AbsI(value)
```

Parameters

```
value Argument: -(2^{31}-1) \dots + 2^{31}-1. Long Long ret val Absolute value of the argument (0 \dots + 2^{-31}-1). Long
```

Notes

The execution time of the function takes 75ns with a T9, 50ns with a T10, and 17ns with a T11.

The smallest negative integer value -2^{31} has no positive counterpart in *ADbasic*; the absolute value of -2^{31} is therefore undefined.

See also

```
Abs, AbsF, Mod
```



And

The operator And combines two integer values bit by bit or two Boolean expressions as Boolean operator.

Syntax

```
var = val_1 And val_2
    'bitwise operator

If ((expr1) And (expr2)) Then
    'boolean operator
```

Parameters

```
val_1,Integer value.LONGval_2expr1, expr2Boolean operator with the value "true" or "false".LOGIC
```

Notes

With And, you can only combine expressions of the same type (integer or Boolean) with each other, mixing them is not possible.

You can use Boolean operators only in statements such as If ... Then ... Else or Do ... Until (variables cannot have Boolean values).

If you use several Boolean operators in one line, you have to put each operation into separate parentheses. This is not necessary for combining integer values.

See also

```
Not. Or. XOr
```





ArcCos

ArcCos provides the arc cosine of the argument.

Syntax

```
ret val = ArcCos(val)
```

Parameters

```
val Argument (-1 ... +1). FLOAT ret val Arc cosine of the argument in radians (0...\pi).
```

Notes

For val < -1, the value π (3.14159...) is returned, for val > 1 the value 0 (zero).

The execution time of the function takes $2.9\mu s$ with a T9, $1.45\mu s$ with a T10, and $0.68\mu s$ with a T11.

See also

```
Sin, Cos, Tan, ArcSin, ArcTan, Round
```

```
Dim val_1, val_2 As Float

Event:
  val_1 = 0.5
  val_2 = ArcCos(val_1)
  Rem Result: val_2 = 1.0472
```



ArcSin

ArcSin provides the arc sine of the argument.

Syntax

```
ret val = ArcSin (val)
```

Parameters

```
val Argument (-1 ... +1). FLOAT

ret_val Arc sine of the arguments in radians (-\pi/2 ... +\pi/2).
```

Notes

The execution time of the function takes $2.8\,\mu s$ with a T9, $1.4\,\mu s$ with a T10, and $0.67\,\mu s$ with a T11.

See also

```
Sin, Cos, Tan, ArcCos, ArcTan, Round
```

```
Dim val_1, val_2 As Float

Event:
  val_1 = 0.5
  val_2 = ArcSin(val_1)
  Rem Result: val_2 = 0.5236
```



ArcTan

ArcTan provides the arc tangent of the argument.

Syntax

```
ret val = ArcTan(val 1)
```

Parameters

val_1	Argument (whole range of values, see "Entering Numerical Values" on page 105).	FLOAT
ret_val	Arc tangent of the argument in radians $(-\pi/2\pi/2)$.	FLOAT

Notes

The execution time of the function takes $1.8\,\mu s$ with a T9, $0.9\,\mu s$ with a T10, and $0.42\,\mu s$ with a T11.

See also

```
Sin, Cos, Tan, ArcSin, ArcCos, Round
```

```
Dim val_1, val_2 As Float

Event:
   val_1 = 0.5
   val_2 = ArcTan(val_1)
'Result: val_2 = 0.4636
```



Asc

Asc determines the corresponding decimal value for a single ASCII character or for the first character of a character string.

Syntax

```
ret val = Asc(string)
```

Parameters

string	Character string.	STRING
ret val	ASCII number (0255) of the (first) character.	LONG

Notes

-/-

See also

String "", + String Addition, Chr, FloToStr, Flo40ToStr, LngToStr, Str-Comp, StrLeft, StrLen, StrMid, StrRight, ValF, ValI



#Begin_Debug_Mode_Disable

#Begin_Debug_Mode_Disable disables the creation of debug code if debug mode is enabled.

Syntax

```
#Begin Debug Mode Disable
```

Parameters

-/-

Notes

#Begin_Debug_Mode_Disable is a pre-processor statement and has only a function if the process is compiled with Debug mode Option (see page 67) enabled. The debug mode is used to recognize run-time errors, see chapter 5.3.1 on page 137.

The debug mode interruption is cancelled with #End_Debug_Mode_ Disable.

Debug mode interruptions cannot be nested i.e. each #End_Debug_Mode_Disable will cancel the interruption in any case. Pay attention especially to interruptions in macros and libraries.

See also

#End Debug Mode_Disable, #Include

Example

Event:

```
#Begin_Debug_Mode_Disable
Rem uncritical source code being not controlled
Rem ...
#End_Debug_Mode_Disable
Rem critical source code, which is controlled
Rem ...
```



Cast_FloatToLong

Until T11: Cast_FloatToLong changes the data type of the argument from Float into Long.

Syntax

```
ret val = Cast FloatToLong(var)
```

Parameters

ret val

var Bit pattern with data type Float.

Identical bit pattern with data type Long.

FLOAT

Notes

For processor T12, see Cast Float32ToLong.

This function does **not** execute a standard type conversion of a number (see chapter 4.4.2 "Type Conversion", page 122). Use the operator "=" for the assignment of a Float value to an integer variable.

This instruction is to be reasonably used in combination with the inverse function <code>Cast_LongToFloat</code>, if there is a bit pattern representing a Float value but given with data type <code>Long</code>. Contrary to the data type, the bit pattern will remain unchanged, so it will again be interpreted as the correct <code>Float</code> value (see also chapter 4.2.3 on page 102).

An example of practice appears with data transfer: A CAN- or RSxxx-bus only transfers 8-bit data packages of data type integer. Therefore, a 32-bit Float value has to be changed into data type Long with Cast_FloatToLong and then divided into 4 separate 8-bit packages. The receiver has to reassemble the packages again and restore the data type Float with Cast_LongToFloat.

See also

Cast_LongToFloat



Cast_LongToFloat

Until T11: Cast_LongToFloat changes the data type of the argument from Long into Float.

Syntax

```
ret val = Cast LongToFloat (val)
```

Parameters

val Bit pattern with data type Long.

ret val Identical bit pattern with data type Float.

LONG

FLOAT

Notes

For processor T12, see Cast LongToFloat32.

This function does **not** execute a standard type conversion of a number (see chapter 4.4.2 "Type Conversion", page 122). Use the operator "=" for the assignment of a Float value to an integer variable.

This instruction is to be reasonably used, if there is a bit pattern representing a Float value but given with data type Long. Contrary to the data type, the bit pattern will remain unchanged, so it will again be interpreted as the correct Float value (see also chapter 4.2.3 on page 102).

An example of practice appears with data transfer: A CAN- or RSxxx-bus only transfers 8-bit data packages of data type integer. Therefore, a 32-bit Float value has to be changed into data type Long with Cast_FloatToLong and then divided into 4 separate 8-bit packages. The receiver has to reassemble the packages again and restore the data type Float with Cast LongToFloat.

See also

Cast FloatToLong



Cast_Float32ToLong

T12 only: Cast_Float32ToLong changes the data type of the argument from Float32 into Long.

Syntax

```
ret val = Cast Float32ToLong(var)
```

Parameters

var Bit pattern with data type Float32.

FLT32 LONG

ret val Identical bit pattern with data type Long.

Notes

This function does **not** execute a standard type conversion of a number (see chapter 4.4.2 "Type Conversion", page 122). Use the operator "=" for the assignment of a Float value to an integer variable.

This instruction is to be reasonably used in combination with the inverse function <code>Cast_LongToFloat32</code>, if there is a bit pattern representing a <code>Float32</code> value but given with data type <code>Long</code>. Contrary to the data type, the bit pattern will remain unchanged, so it will again be interpreted as the correct <code>Float32</code> value (see also chapter 4.2.3 on page 102).

An example of practice appears with data transfer: A CAN- or RSxxx-bus only transfers 8-bit data packages of data type integer. Therefore, a 32-bit Float32 value has to be changed into data type Long with Cast_Float32ToLong and then divided into 4 separate 8-bit packages. The receiver has to reassemble the packages again and restore the data type Float32 with Cast LongToFloat32.

See also

Cast LongToFloat32

For processors until T11, see Cast FloatToLong.

FLT32



Cast_LongToFloat32

T12 only: Cast_LongToFloat32 changes the data type of the argument from Long into Float.

Syntax

```
ret val = Cast LongToFloat32 (val)
```

Parameters

val	Bit pattern with data type Long.	LONG

ret_val Identical bit pattern with data type Float32.

Notes

This function does **not** execute a standard type conversion of a number (see chapter 4.4.2 "Type Conversion", page 122). Use the operator "=" for the assignment of a Float value to an integer variable.

This instruction is to be reasonably used, if there is a bit pattern representing a Float32 value but given with data type Long. Contrary to the data type, the bit pattern will remain unchanged, so it will again be interpreted as the correct Float32 value (see also chapter 4.2.3 on page 102).

An example of practice appears with data transfer: A CAN- or RSxxx-bus only transfers 8-bit data packages of data type integer. Therefore, a 32-bit Float value has to be changed into data type Long with Cast_Float32ToLong and then divided into 4 separate 8-bit packages. The receiver has to reassemble the packages again and restore the data type Float32 with Cast LongToFloat32.

See also

Cast Float32ToLong

For processors until T11, see Cast LongToFloat.



Chr

Chr assigns an ASCII character with a specified decimal number to a string variable.

Syntax

```
Import String.LI*
  'T9: .LI9 / T10: .LIA / T11: .LIB / T12: .LIC
Chr(vascii,dest text)
```

Parameters

```
vascii Decimal number (0...255) of the desired ASCII LONG character.
```

dest text String variable, to which the character is assigned. STRING

Notes

If a string variable has more than one character (or element), **Chr** assigns the ASCII character only to the first element of the string.

Unicode is not supported.

See also

```
String "", + String Addition, Asc, FloToStr, Flo40ToStr, LngToStr, Str-Comp, StrLeft, StrLen, StrMid, StrRight, ValF, ValI
```



Cos

Cos provides the cosine of an angle.

Syntax

```
ret val = Cos(angle)
```

Parameters

```
angle Angle in radians (-\pi...\pi). FLOAT ret val Cosine of the angle (-1...1).
```

Notes

If you use input values, which are not in the range of $-\pi...+\pi$, the calculation error grows with the increasing value.

The execution time of the function takes $1.3\,\mu s$ with a T9, $0.7\,\mu s$ with a T10, and $0.31\,\mu s$ with a T11.

See also

```
Sin, Tan, ArcCos, ArcSin, ArcTan, Round
```



CPU_Sleep

Since processor T11: CPU_Sleep causes the processor to wait for a certain time.

Syntax

```
CPU_Sleep (val)
```

Parameters

val

Number of time units to wait in 10 ns. The value _LONG range depends on the processor:

T11: 9...715827879 $\approx 2^{30} / 1.5$) T12: 9...214748364 $\approx 2^{30} / 5$)

Notes

Please note for *ADwin-Pro II*: Alternatively, there are the instructions P1_Sleep and P2_Sleep (see also chapter 5.2.4 "Setting Waiting Times Exactly"). For processors up to T10, use Sleep.

The waiting time should always be smaller than the cycle time set with Processdelay.



In a high-priority process, CPU_Sleep cannot be interrupted. Thus,
very high values in high-priority processes can cause an interruption in
the communication to the PC.

If possible, use a constant as argument. If the argument val requires a calculation, it requires additional time; this time interval is constant and takes a few clock cycles.

The following conditions require a calculation:

- The argument is an expression with variables or array elements.
- T11 only: The variable in the argument is declared in the memory area DRAM_Extern. The time interval may vary because it depends on several conditions.
- · The argument is an array.
- The argument is a floating-point value.

See also

IO Sleep, NOP, P1 Sleep, P2 Sleep, Processdelay, Sleep



```
Event:
```

```
Rem Wait to start a subsequent measurement exactly Rem 100 \mu s after the external Event signal. 
  \textbf{CPU\_Sleep} \, (\textbf{10000}) \\ \text{Rem} \, \dots
```



Data_n

The $\operatorname{Dim} \operatorname{Data}_n[...]$ As ... instruction dimensions a global Data array.

More information about dimensioning see page 195.

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>}
```

Parameters

```
Name of the declared Data array (n: 1...200).
Data n
              Data type: Float 32, Float 64, Float (up to T11),
<arr
                  Long, String.
type>
              Array size: Number (1...2^31) of the array ele- CONST
dim1, dim2
              ments of the type Arr type.
                                                                   LONG
              Memory, where the array elements are stored:
< m \in m
              Cacheable: memory, which can provide data to the
type>
                  cache (default, T12 only).
              Uncacheable: memory, which can Not provide data to
                  the cache (T12 only).
              Until T11:
              DRAM Extern: external data memory (default).
              DM Local: internal data memory (default).
              T11 only:
              EM Local: extended program or data memory.
```

Notes

You can access the array elements 1...dim. The array element [0] must not be used since it is used for internal purpose.

The maximum array size depends on the available physical memory size of the *ADwin* system.



A global array may be declared 2-dimensional. The specifics are described in chapter 4.3.4 on page 114.

See also

Dim, FIFO, "Global Arrays" on page 107, "Variables and Arrays in the Data Memory" on page 110

```
Rem Declare the global array Data_15 with Rem 1000 long elements

Dim Data_15[1000] As Long

Rem Declare the global array Data_5 with Rem 20 x 75 Float elements

Dim Data_5[20][75] As Float

Rem Declare the global array Data_6 with Rem 20 x 1000 Float32 elements (T12)

Dim Data_6[8][1000] As Float32
```



Dec

Dec decrements the value of a Long-variable by 1.

Syntax

```
Dec (var)
```

Parameters

var

Name of a local or global Long-variable.



LONG

Notes

Dec (var) provides the same result as the program line: val=val-1 and it may have shorter execution time.

See also

```
Inc, - Subtraction
```

Dim index As Long

```
Dim Data_1[1000] As Long
Init:
index=1000

Event:
    DAC(1,Data_1[index]) 'Output the value on DAC1
    Dec(index) 'Decrement the index by 1
If (index<1) Then
    index=1000 'Start again after 1000 outputs
EndIf</pre>
```



#Define

#Define replaces a symbolic name in the source code with an expression, for instance a constant.

Syntax

#Define name expression

Parameters

Symbolic name, without quotation marks. name

CONST

Special chars are not allowed, only alphanumeric STRING characters (a...z, A...Z, 0...9) and the underscore

expressio Expression for the symbolic name, without quota- CONST tion marks.

STRING

All characters are allowed.

Notes

Place this instruction at the beginning of a source code. The replacement is done in the following source code, starting from the next line.



The instruction #Define is a preprocessor instruction that means the replacement is made when you compile the source code (even before the compiler generates the program). Use #Define in order to use names that are more descriptive in the source code instead of constants, parameters or expressions.

The first string up to a blank is interpreted as symbolic name, the following text until the carriage return is interpreted as an expression to be inserted¹. The expression is inserted exactly as you have defined it; variable names in the expression are not replaced by their value, but as a character string.

Neither name nor expression are case-sensitive.

If you want to use a mathematical term for expression, we recommend placing it in parenthesis to avoid errors (see examples).

^{1.} Text behind a comment char " '" will be ignored by the compiler.



See also

#Include

Example

```
#Define setpoint Par_1 'Comments like this are ignored
#Define measured Data_1
#Define pi 3.141592654
```

With these instructions, you can use the names setpoint, measured and pi in the source code instead of Par_1, Data_1 and 3.141592654.

```
#Define setpoint (13 + 4^3)
Par 1 = 2 * setpoint '= 2 * (13 + 4^3)
```

Without the parentheses in the #Define expression, you would get the value "90" instead of the expected "154".



Dim

Dim declares one or more

- local variables
- local one-dimensional arrays (also strings)
- global one-dimensional arrays Data n[n] (also FIFO arrays)
- global two-dimensioned arrays $Data_n[n][m]$.

Information about variables and data types can be found in chapter 4.2.3, information about FIFO arrays under the heading FIFO on page 204.



Syntax

```
Dim var1 {, var2, ...} As <var_type>
Dim array1[dim1] {, array2[dim2]} As <var_type>
     {At <mem_type>}

Dim Data_n[dim1] {, Data_n[dim2]} As <var_type>
     {As FIFO} {At <mem_type>}

Dim Data_n[dim1][dim2] As <var_type>
     {At <mem type>}
```

Parameters

```
Names of the declared variables.
var1, var2
              Names of the declared arrays. For Data n, you
array1,
              can select n from 1...200.
array2,
Data n
              Data type: Float32, Float64, Float (up to T11),
<var
                 Long. For arrays also: String.
type>
              Array size: Number (1...2^31) of the array ele- CONST
dim1, dim2
              ments of the type < var type>.
                                                                  LONG
              Memory where the variables are stored:
< m \in m
              Cacheable: memory, which can provide data to the cache
type>
                 (default, only Data arrays with T12).
              Uncacheable: memory, which can Not provide data to the cache
                 (only Data arrays with T12).
              Until T11:
              DRAM Extern: external memory (default for arrays).
              DM Local:local memory (default for variables).
              T11 only:
              EM Local: extended program or data memory.
```

Notes

The global variables Par_n and FPar_n must not be declared, because they are predefined.

If you want to access data from the computer or from several processes, you can only do this by using *global* variables and arrays.



In an array, you can access the elements 1...dim. The array element [0] must not be used, because it is used for internal purposes.

The maximum array size depends on the physical memory on the *AD-win* system.

With processor T12, only global arrays Data_x can be declared for memory types Cacheable or Uncacheable.

With data structure FIFO, please note the comments for array size dim in combination with T11 (see page 204).

String variables are arrays of type STRING (see "Strings" on page 117). They cannot be declared as FIFO.

With a T10 processor, strings may not be declared At DM_Local into the local memory.

You find notes about the use of memory areas via the cross-references below.

See also

Data_n, Event:, FIFO, Finish:, Init:, LowInit:, String "", "2-dimensional Arrays" on page 114, "Variables and Arrays in the Data Memory" on page 110, "Memory Areas (T12)" on page 113, "Memory Areas (T9...T11)" on page 111

```
Rem Dimension var1 as long variable
Dim var1 As Long

Rem Dimension the local array "array1" with 1000 long elements
Dim array1[1000] As Long

Rem Dimension the global array Data_20 with
Rem 1003 Long elements as Fifo
Dim Data_20[1003] As Long As FIFO

Rem Dimension the array TEXT with
Rem 50 elements as string variable
Dim text[50] As String
```



Do ... Until

Do...Until repeatedly executes a block of instructions until the Exit condition evaluates to "true". The block is executed at least one time.

Syntax

Parameters

```
condition Boolean abort condition with the operators <, >, =, LOGIC |
And and Or.
```

See also

```
< = > Comparison, And, Or, For ... To ... {Step ...} Next, SelectCase
```

Notes

You can nest Do...Until loops repeatedly; only the available memory size will limit the number of nested loops.

Avoid loops with long execution times in high-priority processes, because they cannot be interrupted.



End

End ends a process in the Event: section.

Syntax

End

Notes

End stops the processing of an Event: section immediately and starts processing the section Finish: (if existing). Any instructions in the Event: section following the End instruction are not processed.

In the other program sections, you should use the Exit instruction instead of End.

See also

Exit, ProcessN_Running, Restart_Process, Start_Process, Start_Process Delayed, Stop Process

```
Event:
```



#End_Debug_Mode_Disable

#End_Debug_Mode_Disable cancels the interruption of the debug mode.

Syntax

```
#End Debug Mode Disable
```

Parameters

-/-

Notes

#End_Debug_Mode_Disable is a preprocessor instruction. It has only a function if the process was compiled with enabled Debug mode Option (see page 67) and the debug mode has been interrupted with #Begin_Debug_Mode_Disable. The debug mode is used to recognize run-time errors, see chapter 5.3.1 on page 137.

The debug mode can be interrupted with #Begin_Debug_Mode_ Disable

You cannot nest debug mode interruptions. Pay attention especially to interruptions inside macros or libraries.

See also

```
#Begin Debug Mode Disable, #Include
```

Example

Event:

```
#Begin_Debug_Mode_Disable
Rem uncritical source code being not controlled
Rem ...
#End_Debug_Mode_Disable
Rem critical source code, which is controlled
Rem ...
```



Event:

The keyword **Event:** marks the start of the main program section, which is called every Event signal.

Syntax

```
Event: {At <mem type>}
```

Parameters

Notes

See also overview of program sections in chapter 4.1.1 on page 99.

The program section Event: is the central functional section, which in a process is called in (typically) regular intervals, until it is stopped. Depending on the settings the call is triggered by a cyclic timer Event signal or by an external Event signal. See more in chapter 6 "Processes in the ADwin System".

The processor type T11 can store each program section in a different memory area (see chapter 4.3.2 "Memory Areas (T9...T11)"). The huge, but slow memory area DRAM_Extern should be used for not time-critical program sections; mostly these are the sections LowInit; Init; Finish:

With processor module Pro-CPU T11, the memory area DRAM_ Extern can only be set starting with revision E04.

See also

```
Dim, LowInit:, Init:, Finish:
```

```
Dim val_1 As Float
Event:
  val 1 = -5.3
```



Exit

Exit ends a process in the sections LowInit:, Init:, or Finish:.

Syntax

Exit

Notes

Exit stops the processing of the process and the current program section immediately; the following program lines in the same section will not be executed. Even the section Finish will not be processed.

Use End in the section Event:.

See also

End, ProcessN_Running, Reset_Event, Restart_Process, Start_Process, Start Process Delayed, Stop Process

FLOAT



Exp

Exp calculates the power to the base e of the argument.

Syntax

```
ret val = Exp(val)
```

Parameters

val Argument. FLOAT

ret_val Exponential value of the argument to the base e.

Notes

The execution time of the function takes 1.3 μs with a T9, 0.7 μs with a T10, and 0.31 μs with a T11.

Euler's number has the approximate value 2.7182818.

See also

```
LN, Log, Round, Sqrt
```



FIFO

The Dim Data_n as Fifo instruction defines a global Data array as a ring buffer.

Syntax

```
Dim Data n[m] As <arr type> As FIFO
```

Parameters

```
Data_n Name of the declared DATA-field (n: 1...200).

<arr_type>
Defined variable type: Float32, Float64,
Float (T11 only), Long.

M Array size (1...2^31): Number of elements of type arr_type in the array.
With processor T11, the range for m can be set in steps of 4 only:
m = 4 \times a + 3; a \ge 0.
```

Notes

Once a Data array is defined as FIFO ring buffer (see also chapter 4.3.4 on page 114), it cannot be used as a "normal" array.

FIFO arrays (first in, first out) are managed by data pointers. After dimensioning the array you should initialize these data pointers with FIFO_Clear, in the section LowInit: or Init:. The data in the FIFO are not changed neither by dimensioning the array nor by initializing.

If you write data into a FIFO array faster than you read it, older stored data will be overwritten and are lost. To avoid this you can use the instructions FIFO_Empty and FIFO_Full to determine the amount of space in the array.

If (with processor T11 only) the array size is set to a non-valid array size m, the FIFO array is automatically dimensioned using the next greater and valid array size. As an example, the compiler will change an array size [1000] automatically to [1003].



See also

Dim, Data_n, FIFO_Clear, FIFO_Empty, FIFO_Full

Example

Rem Dimension the global array Data_20 with Rem 1003 Long elements as fifo ringbuffer Dim Data_20[1003] As Long As FIFO



FIFO_Clear

FIFO Clear initializes the write and read pointers of a FIFO array.

Syntax

```
FIFO Clear(arraynum)
```

Parameters

arraynum Number of the DATA-FIFO array (1...200).

LONG

Notes

Initialization of the write and read pointers does not change the data in the the array.

The FIFO pointers are not initialized upon dimensioning. You should initialize the pointers in the sections LowInit: or Init: with FIFO_Clear.

Initializing the FIFO pointers during program run is useful, if you want to clear all data of the array (because of a measurement error for instance).

See also

FIFO, FIFO Empty, FIFO Full





FIFO_Empty

FIFO_Empty determines the number of empty elements in a FIFO array.

Syntax

```
ret val = FIFO Empty (arraynum)
```

Parameters

```
arraynum Number of the DATA-FIFO-array (1...200). LONG

ret val Number of the empty array elements. LONG
```

Notes

If you want to write data into a FIFO array, you can use this instruction, to determine if the FIFO still has enough empty elements.

With processor T11, please note dimensioning in steps of 4 (see page 204).

See also

```
FIFO, FIFO Clear, FIFO Full
```

Example

Dim Data 1[20003] As Long As FIFO 'Declaration



FIFO_Full

FIFO_Full determines the number of elements used in the FIFO array.

Syntax

```
ret val = FIFO Full(arraynum)
```

Parameters

```
arraynumNumber of the DATA-FIFO-array (1...200).LONGret_valNumber of the occupied array elements (0...m).LONGWith processor T11, the range for m can be set in steps<br/>of 4 only: m = 4 \times a + 3; a \ge 0.
```

Notes

Before reading out or using data from the FIFO array, you should use this instruction, to check if there is data in the FIFO. If there is no data an undefined value is returned from the FIFO array.

With processor T11, please note dimensioning in steps of 4 (see page 204).

See also

```
FIFO, FIFO_Clear, FIFO_Empty
```



Finish:

The key word Finish: marks the start of the finishing program section. The program section always has low-priority, level 1.

Syntax

```
Finish: {At mem type}
```

Parameters

Notes

See also overview of program sections in chapter 4.1.1 on page 99.

The program section Finish: is run once as soon as the process is stopped.

After having processed the last instruction in the **Finish**: section, there will be a certain delay until the process status "stopped" is valid.

The processor type T11 can store each program section in a different memory area (see chapter 4.3.2 "Memory Areas (T9...T11)"). The huge, but slow memory area DRAM_Extern should be used for not time-critical program sections; mostly these are the sections LowInit:, Init:, Finish:

With processor module Pro-CPU T11, the memory area DRAM_ Extern can only be set starting with revision E04.

See also

```
Dim, LowInit:, Init:, Event:, ProcessN Running
```

```
Dim val_1 As Float
Finish:
  val_1 = -5.3
```



FIoToStr

FloToStr converts a floating-point value into a character string.

Syntax

```
Import String.LI*
  'T9: .LI9 / T10: .LIA / T11: .LIB / T12: .LIC
FloToStr(val, string[])
```

Parameters

```
valValue to be converted.FLOATstring[]String, formatted as:<br/>{-}#.#####E{-}##.ARRAY<br/>STRING
```

Notes

The length of the returned string varies from 11 to 13 characters, depending on the sign of mantissa and exponent.

See also

```
Asc, Chr, Flo40ToStr, LngToStr, String "", StrComp, StrLeft, StrLen, Str-Mid, StrRight, ValF, ValF
```



```
Import String.LI9 'String library for the T9
 'T9: .LI9 / T10: .LIA / T11: .LIB / T12: .LIC
Dim text[13] As String
Dim pi, number As Float
Init:
 pi = 3.141592654
 FPar 1 = -pi^-20
Event:
 Rem Convert a floating-point number into a string
 FloToStr(FPar 1, text)
 REM T12 only: text[1] is not available, the following
 REM line must be deleted.
 Par 1 = text[1]
                     'String length = 13
                      'ASCII character 2Dh = "-"
 Par 2 = text[2]
                      'ASCII character 31h = "1"
 Par 3 = text[3]
                      'ASCII character 2Eh = "."
 Par 4 = text[4]
 Par 5 = text[5]
                      'ASCII character 31h = "1"
                      'ASCII character 34h = "4"
 Par 6 = text[6]
                      'ASCII character 30h = "0"
 Par 7 = text[7]
 Par 8 = text[8]
                      'ASCII character 32h = "2"
 Par 9 = text[9]
                      'ASCII character 35h = "5"
 Par_10 = text[10]
Par_11 = text[11]
                       'ASCII character 35h = "5"
                      'ASCII character 45h = "E"
                       'ASCII character 2Dh = "-"
 Par 12 = text[12]
 Par_13 = text[13]
Par_14 = text[14]
                      'ASCII character 31h = "1"
                      'ASCII character 30h = "0"
 Par_15 = text[15]
                      'String end character = 0
```



Flo40ToStr

Processor T11 only: **Flo40ToStr** converts a floating-point value into a character string.

Syntax

```
Import String.LIB '.LIB for T11
Flo40ToStr(val, string[])
```

Parameters

```
      val
      Value to be converted.
      FLOAT

      string[]
      String, formatted as: {-}#.#####E{-}##.
      ARRAY

      {-}#.#######E{-}##.
      STRING
```

Notes

The length of the returned string varies from 13 to 15 characters, depending on the sign of mantissa and exponent.

See also

```
Asc, Chr, FloToStr, LngToStr, String "", StrComp, StrLeft, StrLen, Str-Mid, StrRight, ValF, ValI
```



```
Import String.LIB 'String library for T11
Dim text[15] As String
Dim pi, number As Float
Init:
 pi = 3.141592654
 FPar 1 = -pi^-20
 Rem Convert a floating-point number into a string
 Flo40ToStr(FPar 1, text)
 REM T12 only: text[1] is not available, the following
 REM line must be deleted.
 Par 1 = text[1]
                      'String length = 13
 Par 2 = text[2]
                      'ASCII character 2Dh = "-"
 Par 3 = text[3]
                      'ASCII character 31h = "1"
 Par 4 = text[4]
                      'ASCII character 2Eh = "."
 Par 5 = text[5]
                      'ASCII character 31h = "1"
                      'ASCII character 34h = "4"
 Par 6 = text[6]
                      'ASCII character 30h = "0"
 Par 7 = text[7]
                      'ASCII character 32h = "2"
 Par 8 = text[8]
 Par 9 = text[9]
                      'ASCII character 35h = "5"
 Par 10 = text[10]
                      'ASCII character 35h = "6"
                      'ASCII character 35h = "4"
 Par 11 = text[11]
 Par 12 = text[12]
                      'ASCII character 35h = "7"
                      'ASCII character 45h = "E"
 Par 13 = text[13]
 Par 14 = text[14]
                      'ASCII character 2Dh = "-"
 Par_15 = text[15]
Par_16 = text[16]
                      'ASCII character 31h = "1"
                      'ASCII character 30h = "0"
 Par 17 = text[17]
                      'String end character = 0
```



For ... To ... {Step ...} Next

The For...Next instruction creates a program loop, which executes a specified number of times.

Syntax

```
For i = X To Y {Step Z}
... 'instruction block
Next i
```

Parameters

i	Count variable.	LONG
X	Start value of the run variable.	LONG
Y	End value of the run variable.	LONG
7.	Step length (≥1) of the run variable; default; 1.	LONG

Notes

Processors up to T11 only: The instruction block is executed at least once, even if the start value **x** is greater than the end value **y**.

Declare the count variable as Long variable.

A high priority process cannot be interrupted by another process, which is also true while executing a time intensive For ... Next loop. Since the *ADwin* system cannot respond to other events in this time, it is important to keep the number of loops small for high priority processes.



See also

```
Do ... Until, If ... Then ... {Else ...} EndIf, SelectCase
```



```
Dim index As Long
Dim sinus[360] As Float 'Array for sine values
Dim pi As Float
Init:
 pi = 3.14159
 Rem Calculate the sine values in degrees (0° to 359°)
 For index = 1 To 360
  sinus[index] = (2047*Sin((index - 1) * 2*pi/360))
 Next index
 index = 1
                      'Initialize count index
Event:
 DAC(1, sinus[index]) 'Output amplitude value
 Inc index
                      'Increase count index
 Rem From 360 degrees onward, restart at 0
 If (index > 360) Then index = 1
```



Function ... EndFunction

Function...EndFunction is used to define a function macro with passed and returned values.

Syntax

Parameters

macro_ name	Name of the function and of the return value, data type <var_type>.</var_type>	
val_1,	Names of passed parameters;	FLOAT
val_2	for arrays use the syntax with dimension brackets:	LONG
	array[] Or Data_n[].	STRING
<var_< td=""><td>Data type of the function and the return parame-</td><td></td></var_<>	Data type of the function and the return parame-	
type>	ter: Float32, Float64, Float (up to T11) or	
	Long, but not String.	

Notes

You will find general information about macros in chapter 4.5.1 on page 125.

This instruction defines a function macro, which means that the whole instruction block between Function and EndFunction is inserted any place where the macro is called.

Functions help to make your source code more clearly structured. Please note that each function call will increase the size of the compiled file.



You may insert functions at the following 3 locations:

- 1. Before the section Init:/LowInit:
- 2. After the section Finish:
- In a separate file, which you include with #Include (only in locations described in 1, and 2.).

Please note the following when defining functions:

- No process sections such as LowInit:, Init:, Event:, or Finish: can be defined.
- Local variables can be defined at the beginning, which are only available in the function and for the processing period.
 This is true even when a variable has the same name as a variable outside of the function.
- A value should be assigned to the function name, which will be the returned value for the function in the source code.

A function is called with its name and with the arguments, which you have defined; the function must be used as argument in the calling program line, e.g. in an assignment (see example). All expression types (including one- and two-dimensional arrays) are allowed as arguments, as long as they have the appropriate data type.

If you don't define arguments you nevertheless have to use the (empty) braces for the function's call: name().

If an array is used as a passed parameter, the syntax is different for call and definition:

call of function without dimension brackets:

```
ret val=name(array pass)
```

• definition of function with dimension brackets:

```
Function name(array def[])...
```

Values are assigned to elements of passed arrays as usual:

```
array def[2] = value
```



If a value is assigned to a passed parameter x within the function, the function's call must not use a constant x, but a variable or a single array element. If so, a passed parameter can be used to hold a return value.



Passed parameters in symbolic names (#Define) lead to compiler problems. Therefore, use definitions without passed parameters.

```
Rem not suitable: passed parameter 'array_def' in a Define
#Define pointer1 array_def[2]
Rem suitable alternative
#Define pointer2 2
Function name(array_def[])
   If (pointer1 > 0) Then 'line triggers compiler error
   If (array_def[pointer2] > 0) Then 'this line works fine
   ...
EndFunction
```

If a passed parameter is part of an expression inside a function, the parameter should be set in braces. This avoids problems with the order of operator evaluation.

See also

```
#Include, Sub ... EndSub, Lib_Function ... Lib_EndFunction, Lib_Sub ... Lib EndSub
```

Example

```
Function average(w1, w2, w3) As Float
Rem The function calculates the mean of the values
Rem w1, w2 and w3
Dim sum As Float
sum = w1 + w2 + w3
average = sum/3
EndFunction
```

Calling the function e.g. is done by the following program lines:

```
x = average(x1, x2, x3)

DAC(1, average(x1, x2, x3))
```

Example with array

The same function with an array as passed parameter:

```
Function average_array(array[]) As Float
  average_array=(array[1] + array[2] + array[3])/3
EndFunction
```



Calling this function is made in a similar manner (but *without* dimension brackets):

```
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))
```

For array, you can indicate a global array (as Data_1) or a local array (as my_array). Enter the array name only, without element number and brackets.



If ... Then ... {Else ...} EndIf

The If...Then control structure is used to conditionally execute a single instruction (If...Then...) or a block of instructions (If...Then... Else...EndIf).

Syntax

Parameters

```
condition Boolean condition with the operators <, <=, >, >=, LOGIC
=, <>, And and Or.

If the condition is "true", the instructions after
Then are executed.

instr
Instruction (corresponds to an instruction line).
```

Notes

You can nest If structures repeatedly; only limited by the available memory.

The instruction block after Else (if there is one) is executed faster than the one after If...Then. This can be used to speed up the total execution time of the Event: section: put the condition, which has most common state, in the Else statement, for instance when you check if limit values are exceeded.

In the single-line version, the instruction cannot call a subroutine macro (Sub) nor a function macro (Function).



See also

```
< = > Comparison, And, Or, Do ... Until, SelectCase
```

```
Dim val As Long
                      'Declaration
Event:
 val = ADC(1)
                       'Acquire measurement value
 If (val > 3000) Then 'Limit value is exceeded:
  Clear Digout(1)
                      'Reset DIGOUT 1
  Set Digout(0)
                       'Set DIGOUT 0
 Else
                      'Limit value not exceeded
  Clear Digout(0)
                      'Reset DIGOUT 0
  Set Digout(1)
                      'Set DIGOUT 1
 EndIf
                       'End of control structure
```



#If ... Then ... {#Else ... } #EndIf

This preprocessor structure is used to conditionally compile a block of instructions (#If...Then...#Else...#EndIf).

Syntax

Parameters

If the condition is "true", the instructions after Then are executed.

The system parameter <SYSPAR> and the corresponding value are shown in the table below:

<syspar></syspar>	value	Meaning
ADwin_ SYSTEM	ADWIN_CARD ADWIN_GOLD ADWIN_GOLDII ADWIN_L16 ADWIN_PRO ADWIN_PROII	System setting in the window Compiler Options.
Processor	T9 T10 T11 T12	Processor setting in the window Compiler Options.

Notes

The condition may only use the operator "="; neither Boolean conditions using And and Or nor bracing is allowed. You can nest #If structures repeatedly; only limited by the available memory.



There is no single-line version as with If...Then.

When calling the compiler via Command Line Calling (see page A-9) the system parameters refer to the command line options /Sx and /Px.

See also

```
< = > Comparison, If ... Then ... {Else ...} EndIf
```

```
Rem set low priority Processdelay to 800µs
#If Processor = T12 Then
 Rem\ T12:\ 800\mu s = 800000\ x\ 1ns
 Processdelay = 800000
#Else
#If Processor = T11 Then 'If CPU = T11
 Rem T11: 800\mu s = 240000 \times 3,3ns
 Processdelay = 240000
#If Processor = T10 Then 'If CPU = T10
 Rem T10: 800\mu s = 16 \times 50\mu s
 Processdelay = 16
#Else
                         'other CPU, here: CPU = T9
 Rem T9: 800\mu s = 8 \times 100\mu s (also other CPUs)
 Processdelay = 8
#EndIf
#EndIf
#EndIf
```



Import

Import includes functions and subroutines from the specified library file during compilation.

Syntax

```
Import {path}file
```

Parameters

file File name of the library file without quotes. The file CONST

extension is .LI9 for T9, .LIA for T10, .LIB for $_{
m STRING}$

T11.

path Path name of the library file (with drive), without CONST

quotes.

STRING

Notes

General information about include files to be found in chapter 4.5.3 on page 126.

Insert Import instructions at the beginning of your source code (before you declare the variables). If you import library files into the source code of a library A, you also have to import the other library files in the source code where library A is called.

Only those functions and subroutines, which you call in your source code are imported from the library file.

If the path name misses, only the standard directory is searched (see Options Menu, Directories, page 63). Use the back slash "\" in the path name to separate directory names.

The base directory for relative paths is—if the source code is member of a project—the directory of the project file, otherwise the directory of the source code file.

The following library files are delivered with *ADbasic*:

String.lix String instructions.

FFT.liX Instructions for fast Fourier transfor-

mation.



math.lix Special mathematics instructions.

See also

#Include, Lib_Function ... Lib_EndFunction, Lib_Sub ... Lib_EndSub

Example

Rem import the string library for the T9 processor Import String.LI9
Rem import a user library for the T10 processor Import C:\MyFiles\ADwinLibs\dig2volt.LIA



Inc

Inc increments the value of a local or global integer variable by one.

Syntax

```
Inc (var)
```

Parameters

var

Name of a local or global Long-variable.



CONST

LONG

Notes

Inc (val) is equivalent the program line: val=val+1 and it may have shorter execution time.

See also

```
Dec, + Addition
```



#Include

#Include includes all the contents of an include file into the source code.

Syntax

#Include {path}filename

Parameters

filename Name of the file to be included (with the extension CONST

.Inc), without quotes.

path Complete path with drive, or relative path.

STRING

CONST

Notes

You find general information about include files in chapter 4.5.2 on page 125.

Insert the #Include instructions at the beginning of your source code (before you declare the variables). You can import other include files in the source code of an include file.

If any include file uses library functions, you have also to include the corresponding library files with ${\tt Import}$.

If the path name misses, only the standard directory is searched (see Options Menu Directories, page 63). Use the back slash "\" in the path name to separate directory names.

The base directory for relative paths is—if the source code is member of a project—the directory of the project file, otherwise the directory of the source code file.

To include any of the include files delivered with *ADbasic*—the files contain instruction to access hardware I/Os—you enter the first characters of the instruction #Include, press [CTRL][SPACE] and select the required include file from the list. Alternatively use one of the code snippets from the "Hardware" group.

The following include files are delivered with *ADbasic* and contain the instructions to access the hardware I/Os:



ADwGCnt.inc ADwin-Gold: Instructions for CO1

add-on; insert text snippet with

IG[TAB].

ADwGCAN.inc ADwin-Gold: Instructions for CAN

add-on; insert text snippet with

IG[TAB].

ADwinGoldII.inc ADwin-Gold II: All instructions; insert

text snippet with IG2[TAB].

ADwL16.inc ADwin-light-16: All instructions; insert

text snippet with IL[TAB].

ADwinPro_All.inc ADwin-Pro, ADwin-Pro II: All instruc-

tions; insert text snippet with ${\tt IP[TAB]}.$

See also

```
#Define, Import, Function ... EndFunction, Sub ... EndSub
```

```
Rem find file in the given directory
#Include C:\Test\demofunc.Inc

Rem find file in standard directory
#Include demofunc.Inc

Rem relative path.
Rem The base directory is relative to the directory of Rem the project file (if the source file is member of Rem a project).
Rem If the source code is not a project member, the base Rem directory is the directory of the source file.
#Include .\demofunc.Inc
```



Init:

The keyword Init: marks the start of the initializing program section.

Syntax

```
Init: {At <mem type>}
```

Parameters

```
type>
T11 only: memory area, where the program sec-
type>
tion Init: is stored.

PM_Local: internal program memory (default).

EM_Local: extended internal program or data memory.

ory.DRAM Extern: external data memory.
```

Notes

See also overview of program sections in chapter 4.1.1 on page 99.

The program section Init: is run once as soon as the process is started and (if existing) the program section LowInit: is finished. The delay between having processed the last instruction of the Init: section and starting the Event: section is about 1...2 x Processdelay.

The program section has the priority as set for the process (menu entry "Options / Process"). With high priority, the section cannot be interrupted and should then be as short as possible.

The processor type T11 can store each program section in a different memory area (see chapter 4.3.2 "Memory Areas (T9...T11)"). The huge, but slow memory area DRAM_Extern should be used for not time-critical program sections; mostly these are the sections LowInit;, Init;, Finish:

With processor module Pro-CPU T11, the memory area DRAM_ Extern can only be set starting with revision E04.

See also

```
Dim, LowInit:, Event:, Finish:, Processdelay
```



```
Dim val_1 As Float
Init:
  val_1 = -5.3
```



IO_Sleep

IO_SLEEP causes instructions for access to inputs and outputs of a Gold II system to wait for a certain time.

Syntax

```
IO_Sleep(val)
```

Parameters

val

Number (12, 14, ...715827879 $\approx 2^{31}$ / 3) of time LONG units to wait in 10 ns.

Only even numbers are valid. An invalid number will automatically be decreased by 1.

Notes

Alternatively, there is the instruction <code>CPU_Sleep</code> (see also chapter 5.2.4 "Setting Waiting Times Exactly").

The instruction IO_SLEEP is used to wait a defined time between 2 accesses to inputs/outputs. The total waiting time is the sum of the processing time for the I/O access and the waiting time by IO SLEEP.

The waiting time should always be smaller than the cycle time set with Processdelay.



In high-priority processes, improper values can cause an interruption in the communication to the PC:

- Make sure that the argument always has a value greater than 12; else, very long waiting times can arise.
- Use very high values with care, because the communication to the PC is interrupted for a long time (danger of timeout).

If possible, use a constant as argument. If the argument val requires a calculation, it requires additional time; this time interval is constant



and takes a few clock cycles.

The following conditions require a calculation:

- The argument is an expression with variables or array elements
- The variable in the argument is declared in the memory area
 DRAM_Extern (up to T11). The time interval may vary because
 it depends on several conditions.
- The argument is an array.
- The argument is a floating-point value.

See also

```
CPU Sleep, NOP
```



Lib_Function ... Lib_EndFunction

With Lib_Function...Lib_EndFunction, a function with passed and return parameters is defined in a library file.

Syntax

```
Lib_Function lib_name(<lib_par1> {, <lib_par2>, ...})
As <fct_type>
{Dim var As <var_type>}
{#Define name expression}
... 'Instruction block
name = ...
Lib_EndFunction

Syntax of passed parameters <lib_par>:
<by_type> var_name As <var_type> {At <mem_type>}
```



Parameters

lib name Name of the library function and of the return value: data type <fct type>. <fct type> Data type: Float32, Float64, Float (up to T11), Long. Name of a passed parameter inside of library var name function; for arrays use the syntax with dimension brackets: array[] Or Data n[]. Methods for the transfer of parameters: <by type> ByRef: pass reference (pointer) to variable or array. Byval: pass value only. Data type: Float32, Float64, Float (up to <var type> T11), Long, String. Useful for processor T10 only: Type of memory, <mem type> where the passed parameters are stored; to be used only with arrays: DRAM Extern: external memory. DM Local:local memory.

Notes

You will find general information about library files in chapter 4.5.3 on page 126.

Generate library functions (and library subroutines) in a separate source code file. The compilation with "Build/Make lib file" creates the library file. With Import, those library modules are included into a process, which are being called in the process.

In a library function, you can

declare and use local variables and arrays (only one-dimensional).



Declare variables always at the beginning of the subroutine, but never outside.

- use global variables and arrays, which are passed as parameters.
- process one-dimensional arrays only.
 You can pass two-dimensional arrays as parameters, but they will be considered as one-dimensional arrays in the function (see also chapter 4.3.4 on page 114).
- assign a value to the function name, which will be the value returned for the function in the source code.

In a library function, you cannot

- define process sections such as LowInit:, Init:, Event:,
 Or Finish:.
- call a library function or subroutine from the same library file.
 If necessary, you have to put the function, which is to be called, into a new library file and Import it from there.
- use SelectCase.
- declare symbolic names using #Define.

There are 2 differing methods for passing parameters:

- ByRef: The library function can change the parameter; the changed value is available in the program (the address of the parameter is transferred).
- ByVal: The library function can only access the value of the parameter, but cannot change it. Thus, the parameter remains the same for the program that calls the function.



Passed parameters should always be declared At <mem_type>, to save valuable processor time (<mem_type> must fit with the declaration of the passed parameters in the calling program, see Dim). If not, the library function has to detect the parameter's memory type at run time.

If an array is passed as parameter, the syntax for definition and call differs:

- Definition of the library function's parameter with brackets:
 Lib Function function function array [] ...)
- Call with the parameter *without* brackets: ret val=funcname(... array ...)

If arrays are used as passed parameters, always define them as ByRef and without indicating any array size. You cannot use FIFO arrays as passed parameters.



See also



Library call

Calling the library function average is illustrated in the following example, a "moving average filter":

```
Rem Import the library 'MEAN'
Import C:\MyFiles\ADwinLibs\MEAN.LI9
#Define cnt 10 'Number of the samples
#Define samples Data 1 'Number of measm. values
#Define filtered Data 2'Number of filtered measm. values
#Define length 1000 'Length of the array
Dim samples[length] As Long'Source array
Dim filtered[length] As Long'Destination array
Dim i As Long
                      'Count variable
Init:
 i = 1
                      'Initialize counter
 Processdelay = 40000 'Measurement with 1 kHz
Event:
 samples[i] = ADC(1) 'Measure and save analog values
                      'Increment counter
 Inc i
 If (i> length) Then End'1000 measurements completed?
                       'If yes: process Finish
Finish:
 For i = 1 To (length - cnt) 'For all measm. values
  Rem Call library function "average"
  filtered[i + cnt] = average(samples,i,cnt)
  Rem Note the call with the passed array 'samples'
  Rem *without* dimension brackets
 Next i
```



Lib_Sub ... Lib_EndSub

The Lib_Sub...Lib_EndSub is used to define a subroutine with passed parameters in a library file.

Syntax

```
Lib_Sub lib_name(<lib_par1> {, <lib_par2>, ...})

{Dim var as <var_type>}

{#Define name expression}

... 'Instruction block

Lib_EndSub

Syntax of passed parameters <lib_par>:
<by_type> var_name As <var_type> {At <mem_type>}
```

Parameters

```
Name of the library subroutine.
lib name
                Name of a passed parameter inside of library
var name
                Sub:
                for arrays use the syntax with dimension bra-
                ckets: array[] or Data n[].
                Methods for the transfer of parameters:
  <by_type>
                ByRef: pass reference (pointer) to variable and array.
                Byval: pass value only.
 <var_type> Data types: Float, Long, String.
 <mem type>
                Useful for processor T10 only: Type of memory,
                where the passed parameters are stored; to be
                used only with arrays:
                DRAM Extern: external memory.
                DM Local:local memory.
```

Notes

You will find general information about library files in chapter 4.5.3 on page 126.



Generate library subroutines (and library functions) in a separate source code file. The compilation with "Build/Make lib file" creates the library file. With Import, those library modules are included into a process, which are being called in the process.

In a library subroutine, you can

- declare and use local variables and arrays (only one-dimensional).
 - Declare variables always at the beginning of the subroutine, but never outside.
- use global variables and arrays, which are passed as parameters.
- process one-dimensional arrays only.
 You can pass two-dimensional arrays as parameters, but they will be considered as one-dimensional arrays in the function (see also chapter 4.3.4 on page 114).

In a library subroutine, you cannot

- define process sections such as LowInit:, Init:, Event:, Or Finish:.
- call a library function or subroutine from the same library file.
 If necessary, you have to put the function, which is to be called, into a new library file and Import it from there.
- use SelectCase.
- declare symbolic names using #Define.

There are 2 methods for passing parameters that differ as follows:

- ByRef: The library function can change the parameter; the changed value is available in the program (the method transfers the address of the parameter).
- ByVal: The library function can only access the value of the parameter, but cannot change it. Thus, the parameter remains the same for the program that calls the function.



Refers to processor T10 only: Passed parameters should always be declared At <mem_type>, to save valuable processor time (<mem_type> must fit with the declaration of the passed parameters in the calling program, see $\tt Dim$). If not, the library subroutine has to detect the parameter's memory type at run time.



If an array is passed as parameter, the syntax for definition and call differs:

• Definition of the library subroutine's parameter *with* brackets:

```
Lib_Sub subname (... array[] ...)
```

• Call with the parameter without brackets:

```
subname(... array ...)
```

If arrays are used as passed parameters, always define them as ByRef and without indicating any array size. You cannot use FIFO arrays as passed parameters.

See also

```
Lib_Function ... Lib_EndFunction, Import, Function ... EndFunction, Sub ... EndSub
```

```
Rem save binary file as DIG2VOLT.LI9
Rem (extension according to processor)
Rem Measurement value conversion from Digits(0...65535)
Rem to 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 call

Calling the library function dig2volt is illustrated in the following example, a conversion of measurement values:

```
Rem The library 'DIG2VOLT' is imported
Import C:\MyFiles\ADwinLibs\DIG2VOLT.LI9
#Define cnt 1000
                       'Number of the samples
#Define ptr 1
                       'Start point of samples
                       'which are to be converted
#Define qain 1
                      'Gain of the PGA
#Define samples Data 1 'Memory for meas. values
#Define scaled Data_2 'Memory for converted meas. values
#Define length 1000 'array length
Dim samples[length] As Long 'Source array
Dim scaled[length] As Long 'Destination array
Dim i As Long
                       'Counter
Init:
 i = 1
                      'Initialize counter
 Processdelay = 40000 'Measurement with 1 kHz
Event:
samples[i] = ADC(1)
                      'Measure and save analog values
  Inc i
                       'Increment counter
  If (i> length) Then End'1000 measurements done?
                       'If yes: process Finish
Finish:
 Rem Convert measurement values by
 Rem calling the library subroutine 'dig2volt'
 dig2volt(samples,ptr,cnt,gain,scaled)
 Rem Note the call with the passed array 'samples'
 Rem *without* dimension brackets
```



LN

LN provides the natural logarithm (to base e) of an argument.

Syntax

```
ret val = LN(val)
```

Parameters

val Argument. FLOAT

ret val Natural logarithm of the argument.

FLOAT

Notes

The execution time of the function takes 1.45 μs with a T9, 0.7 μs with a T10, and 0.37 μs with a T11.

See also

```
Exp, Log, Round, Sqrt
```



LngToStr

LngToStr converts an integer value into a string.

Syntax

```
Import String.LI*
  'T9: .LI9 / T10: .LIA / T11: .LIB / T12: .LIC
LngToStr(value, string[])
```

Parameters

ARRAY

LONG

Notes

The length of the generated string depends on the character, which is to be converted and on the sign. String lengths of 1 to 11 characters are possible.

You will find information about the string structure in chapter 4.3.6 on page 117.

See also

String "", + String Addition, Asc, Chr, FloToStr, Flo40ToStr, StrComp, StrLeft, StrLen, StrMid, StrRight, ValF, ValF



```
Import String.LI9
 'T9: .LI9 / T10: .LIA / T11: .LIB / T12: .LIC
Dim digits[11] As String'Resulting string
Dim a As Long
Init:
 a = -1234567890
Event:
 LngToStr(a,digits) 'Convert to string
 REM T12 only: digits[1] is not available, the following
 REM line must be deleted.
 Par 1=digits[1]
                      'String length = 11
 Par 2=digits[2]
                      'ASCII character 45 = "-"
                      'ASCII character 49 = "1"
 Par 3=digits[3]
 Par 4=digits[4]
                      'ASCII character 50 = "2"
 Par 5=digits[5]
                      'ASCII character 51 = "3"
                      'ASCII character 52 = "4"
 Par 6=digits[6]
                      'ASCII character 53 = "5"
 Par 7=digits[7]
 Par 8=digits[8]
                      'ASCII character 54 = "6"
                      'ASCII character 55 = "7"
 Par_9=digits[9]
 Par 10=digits[10]
                      'ASCII character 56 = "8"
 Par 11=digits[11]
                      'ASCII character 57 = "9"
                     'ASCII character 48 = "0"
 Par 12=digits[12]
 Par 13=digits[13] 'End of string sign = 0
```



Log

Log provides the decimal logarithm (to base 10) of an argument.

Syntax

```
ret val = Log(val)
```

Parameters

val	Argument.	FLOAT
ret_val	Decimal logarithm of the argument.	FLOAT

Notes

The execution time of the function takes 1.5 μs with a T9, 0.75 μs with a T10, and 0.38 μs with a T11.

See also

```
Exp, LN, Round
```



LowInit:

The key word **LowInit**: marks the start of an initializing program section. The program section always has low-priority, level 1.

Syntax

```
LowInit: {At mem type}
```

Parameters

Notes

See also overview of program sections in chapter 4.1.1 on page 99.

The program section LowInit: is run once as soon as the process is started. The section serves to initialize, e.g. variables or data connections. LowInit: is always run before the Init: section (if existing).

The section LowInit: is suitable for huge not time-critical initialization sequences since it can be interrupted (due to low priority).



The processor type T11 can store each program section in a different memory area (see chapter 4.3.2 "Memory Areas (T9...T11)"). The huge, but slow memory area <code>DRAM_Extern</code> should be used for not time-critical program sections; mostly these are the sections <code>LowInit:, Init:, Finish:</code>.

With processor module Pro-CPU T11, the memory area DRAM_ Extern can only be set starting with revision E04.

See also

Dim, Init:, Event:, Finish:



```
Dim val_1 As Float
LowInit:
  val_1 = -5.3
```



Max_Float

Max Float returns the greater of 2 Float values.

Syntax

```
ret val = Max Float(val1, val2)
```

Parameters

val_1	Compared value 1	FLOAT
val_2	Compared value 2	FLOAT
ret val	The greater of both values.	FLOAT

Notes

-/-

See also

```
AbsF, Min_Float, Max_Long, Min_Long, ValF
```

Example

Event:

```
FPar 10 = Max Float (FPar 1, FPar 2)
```



Min_Float

Min Float returns the smaller of 2 Float values.

Syntax

```
ret val = Min Float(val1, val2)
```

Parameters

val_1	Compared value 1	FLOAT
val 2	Compared value 2	FLOAT

ret val The smaller of both values.

FLOAT

Notes

-/-

See also

```
AbsF, Max_Float, Max_Long, Min_Long, ValF
```

```
Event:
```

```
FPar_10 = Min_Float (FPar_1, FPar_2)
```



Max_Long

Max Long returns the greater of 2 integer values.

Syntax

```
ret val = Max Long(val1, val2)
```

Parameters

val_1	Compared value 1	LONG
val 2	Compared value 2	LONG

ret val The greater of both values. LONG

Notes

-/-

See also

```
Absl, Min_Float, Max_Float, Min_Long, Vall
```

```
Event:
```

```
Par 10 = Max Long(Par 1, Par 2)
```



LONG

Min_Long

Min Long returns the smaller of 2 integer values.

Syntax

```
ret val = Min Long(val1, val2)
```

Parameters

val_1	Compared value 1	LONG
val 2	Compared value 2	LONG

The smaller of both values. ret val

Notes

-/-

See also

```
Absl, Max_Long, Min_Float, Max_Float, Vall
```

Example

Event:

```
Par 10 = Min Long(Par 1, Par 2)
```



MemCpy

Since Processor T11: MemCpy copies a specified amount of array elements from a source array to a destination array.

Syntax

```
MemCpy (array1[i1], array2[i2], count)
```

Parameters

array1[]	Name of the source array.	LONG
		FLOAT
		STRING
i1	Index (≥1) of the first copied array element.	LONG
array2[]	Name of the destination array; the data type must be the	LONG
	same as of the source array.	FLOAT
		STRING
i2	Index (≥1) of the first array element to be written.	LONG
count	Number (≥1) of array elements to be copied.	LONG

Notes

MemCpy is the simple and much faster alternative to copying data in a For...Next-loop.

The instruction may be used neither with FIFO arrays nor with local variables.

Please note: The data types of source and destination array must be identical and the destination array must be declared large enough to hold all copied data.



The access to indexes out of bounds can be monitored in debug mode for the destination array (see Debug mode Option on page 67). The source array cannot be monitored.



See also

Dim

```
Dim Data_1[75], Data_2[100] As Float

Event:
   Rem Copy 70 array elements from Data_1 to Data_2
   MemCpy (Data_1[5], Data_2[30], 70)
```



NOP

 ${\tt NOP}$ (No OPeration) causes the processor to wait for one processor cycle.

Syntax

NOP

Notes

The execution time of the instruction normally is one processor cycle:

T9	25ns
T10	25ns
T11	3, 3 ns
T12	1,0ns

With this instruction, you can delay for a necessary waiting period (e.g. after Set_Mux) if there is no other use of processing time. With T11, please note chapter 5.2.4 on page 132: Setting Waiting Times Exactly.

See also

CPU_Sleep, P1_Sleep, P2_Sleep, Sleep



Not

Not inverts the bits of an argument.

Syntax

```
ret val = Not(val)
```

Parameters

val Value to be inverted (no logic expression).

LONG

ret val Inverted argument.

LONG

Notes

If possible, use this function only with integer values (of the type Long). floating-point values (of the type Float) are converted into integer values before they are inserted: The decimal places are truncated and the value rounded if necessary before the Not operation.

Not runs with bits only, not with Boolean expressions. Therefore, you cannot negate logic expressions (true / false) with it. Not allowed: $Not(Par_2 > 2)$.

See also

```
And, If ... Then ... {Else ...} EndIf, Or, XOr
```



Or

The operator or combines two integer values bit wise or two Boolean expressions as a Boolean operator.

Syntax

Parameters

```
val_1, val_2Integer value.LONGexpr1, expr2Boolean expression with the value "true" or "false".LOGIC
```

Notes

With Or, you can only combine expressions of the same type (integer or Boolean) with each other, mixing them is not possible.

You can use Boolean operators only in statements such as If ... Then ... Else or Do ... Until (variables cannot have Boolean values).

If you use several Boolean operators in one line, you have to put each operation into parentheses. For bit wise combining of integer values, parentheses are not required.

See also

```
And, If ... Then ... {Else ...} EndIf, Not, XOr
```

Example

Bit wise operator:



Boolean operator:



P1_Sleep

Since Processor T11: **P1_Sleep** causes the Pro I bus to wait for a certain time.

Syntax

```
P1 Sleep (val)
```

Parameters

val Number of the time units to wait in 10ns:

LONG

with constants: 7...715827879. with variables: 9...715827879.

Notes

Alternatively, there are the instructions CPU_Sleep and P2_Sleep (see also chapter 5.2.4 "Setting Waiting Times Exactly"). For processors up to T10, use Sleep.

With processor T12, P1_Sleep has only effect on the TTL signal inputs Dig I/O 0 and Dig I/O 1.

P1_Sleep is used to wait a defined time between 2 accesses to modules on the Pro I bus.

The waiting time should always be smaller than the cycle time set with Processdelay.

In a high-priority process, P1_Sleep cannot be interrupted. Thus, very high values in high-priority processes can cause an interruption in the communication to the PC.



Do not use values lower than the given range of values.

If possible, use a constant as argument. If the argument val requires
a calculation, it requires additional time; this time interval is constant



and takes a few clock cycles.

The following conditions require a calculation:

- The argument is an expression with variables or array elements.
- The variable in the argument is declared in the memory area DRAM_Extern. The time interval may vary because it depends on several conditions.
- The argument is an array.
- The argument is a floating-point value.

See also

```
CPU_Sleep, NOP, P2_Sleep, Sleep
```



P2 Sleep

Since Processor T11: P2 Sleep causes the Pro II bus to wait for a certain time.

Syntax

```
P2 Sleep (val)
```

Parameters

val

Even number (14...715827878) of the time units _LONG to wait in 10 ns. An odd number is not allowed

Notes

Alternatively, there are the instructions CPU Sleep and P1 Sleep (see also chapter 5.2.4 "Setting Waiting Times Exactly"). For processors up to T10, use Sleep.

P2 Sleep is used to wait a defined time between 2 accesses to modules on the Pro II bus.

The waiting time should always be smaller than the cycle time set with Processdelay.

In a high-priority process, P2_Sleep cannot be interrupted. Thus, very high values in high-priority processes can cause an interruption in the communication to the PC.



If possible, use a constant as argument. If the argument val requires a calculation, it requires additional time; this time interval is constant and takes a few clock cycles.

The following conditions require a calculation:

- The argument is an expression with variables or array elements.
- T11 only: The variable in the argument is declared in the memory area DRAM Extern. The time interval may vary because it depends on several conditions.
- The argument is an array.
- The argument is a floating-point value.

See also

```
CPU Sleep, NOP, P1 Sleep, Sleep
```



LONG



Peek

Peek reads the contents of a specified memory location of the *ADwin* system.

Syntax

```
ret val = Peek(addr)
```

Parameters

addr Address of the memory location to be read out.

Contents of the memory location.

LONG

Notes

You will find an overview of the register addresses (*Gold* and *Light-16*) in your hardware documentation.

See also

```
Poke, Read_Timer
```

ret val

Example

The instruction below reads the value of the memory address 30h, which is the data register of the ADC1 on the *ADwin-Gold* system and contains the converted analog value.

```
Rem read out memory locations of an ADwin-Gold system
val = Peek(30h)
```



Poke

Poke writes a value into a specified memory location of the ADwin system.

Syntax

```
Poke (addr, value)
```

Parameters

addr Address of the memory location, into which values __LONG_

are written.

Value to be written. value

LONG

Notes

With Poke, you are overwriting the specified memory address. Information stored there will be lost.



Do not write to memory addresses whose functions you do not know. If you do, it is possible that important data, processes or even the operating system will be destroyed.

If this should happen, existing measurement data is lost. To recover, you must reboot the ADwin system and reload the processes.

You will find an overview of the register addresses (*Gold* and *Light-16*) in your hardware documentation.

See also

```
Peek, Read Timer
```

```
'Change memory locations of an ADwin-Gold system
'Write into DAC register 1: 3072 (=+5V in the range ±10V)
Poke (20400050h, 3072)
Poke(20400010h, 011b) 'Start output on all DACs
Poke (204000C0h, 111100b) 'Set outputs DI018...DI021 to High
```



Processdelay

The system variable Processdelay defines the process delay (cycle time) of a process.

Processdelay replaces the system variable Globaldelay, which is still valid for reasons of compatibility.

Syntax

```
ret_val = Processdelay
or
Processdelay = expr
```

Parameters

ret_val	Current cycle time in clock cycles.	LONG
expr	Cycle time to be set: Number (≥1) of clock cycles.	LONG

Notes

In a time-controlled process, the section **Event:** is called repeatedly and in fixed time intervals by the internal counter. The time interval between two cyclic calls is called process delay and is counted in clock cycles.

The time interval of the Processdelay depends on the process priority and the processor type:

Processor	Priority		
	High	Low	
Т9	25ns	100 µs	
T10	25ns	50 µs	
T11	3, 3 ns	$3,\overline{3}$ ns = $0,00\overline{3}$ µs	
T12	1,0ns	1,0ns	

With high-priority processes, select a sufficiently large process delay to avoid overloading the *ADwin* system (see also chapter 6.1.4 on page 146). As a rule of thumb the processor workload (display field: "Busy x%" in the status bar) should be under 90 percent and must not exceed 100 percent.



If the time needed for processing the section Event: is larger than the process delay, the next counter call and following will be delayed. If this delay cannot be caught up within 250 ms, the communication between the *ADwin* system and the computer can be interrupted.

You may set a constant process delay by assigning a value to the variable Processdelay in the section Init:/LowInit:. You will then overwrite the default value you have set in the dialog window "Options / Process" under "Initial Processdelay".

You can set the variable only once in a section.

Writing into this variable in the section <code>Event:</code> should just be made at the beginning of this section. If the parameter <code>Processdelay</code> is changed in a process cycle in the section <code>Event:</code>, the cycle time (processdelay) will be changed immediately. This may be critical especially when the cycle time has been shortened: Make sure that the execution time of the program remains less than the newly set cycle time.

See also

Read Timer

```
Init:
```

```
Rem Set cycle time

Processdelay = 40000

Rem For T9 and T10, high priority: 1 ms

Rem For T11, high+low priority: 0.133 ms

Rem ...
```



EndIf

If you need a longer cycle time than may be set with Processdelay, you can use an auxiliary variable:

```
Init:
    Rem Set max. cycle time
    Processdelay = 2147483647
    Rem For T9 and T10, high priority: 53.7s
    Rem For T11, high+low priority: 7.2s
    Rem initialize auxiliary variable
    Par_1 = 0

Event:
    Inc Par_1
    Rem use 100fold cycle time
    Rem For T9 and T10, high priority: 89.5 min
    Rem For T11, high+low priority: 12min
    If (Par_1 = 100) Then
    Par_1 = 0
    Rem run program
```



Process Error

Process Error returns the previously occurred error of the current process.

Syntax

```
ret val = Process Error
```

Parameters

ret val

Number of the previously occurred error in the LONG process. Some error numbers:

- 0: no error
- 1: Division by zero
- 2: Square root from negative value
- 10: Accessing a too high element number of a global
- 11: Accessing a too small element number (≤0) of a global array.
- 12: Accessing a too high element number of a local
- 13: Accessing a too small element number (≤0) of a local array.
- 20: Transputer only: Timing error, i.e. Processdelay is too small.
- 30: FIFO index is not a FIFO.

Notes

The return value is defined only if debug mode is enabled (see Debug mode Option, page 67). The variable is read-only.

See also

```
ProcessN Running, Start Process, Stop Process
```

```
Event:
```

```
Par 10 = Sqrt(Par 12)
Rem read previous error number in the process
Par 2 = Process Error
```



ProcessN_Running

The system variable Processn_Running returns the current status of the specified process.

Syntax

```
ret val = Processn Running
```

Parameters

n Number of the requested process (0...12, 15).

CONST LONG

LONG

ret val F

Process status:

- 1 Process is running.0 Process is stopped.
- -1 Process is being stopped.

Notes

The system variable is read only.

See also

```
End, Exit, Restart_Process, Start_Process, Start_Process_Delayed, Stop Process
```

Example

Event:

```
Rem Get the status of process 2
Par 2 = Process2 Running
```



Read_Timer

Read_Timer returns the current counter value of the *ADwin* system timer.

Syntax

Parameters

ret val Current counter value.

LONG

Notes

The counter value cannot be written.

There are 2 timers in an *ADwin* system (32-bit), which count in different units of time:

process priority	Т9	T10	T11	T12
high	25ns	25ns	3, 3 ns	1,0ns
low	100 µs	50µs	3, 3 ns	1,0ns

You may determine a time interval from the difference of 2 timer values. Please note that any read timer value will be reached again after a certain time interval, which depends on the units of time given above:

process priority	Т9	T10	T11	T12
high	107.4s	107.4s	14.3s	4,3s
low	119.3h	59.7h	14.3s	4,3s

Please note that with processor T12 the time measurement inside of a program section follows special rules. Find more under "Measuring the Processing Time".

See also

Processdelay, Read_Timer_Sync



Example

```
Dim timervalue As Long

Event:
   timervalue = Read_Timer()
```

See related example seconds_timer.bas in folder C:\ADwin\ADbasic\samples_ADwin.



Read_Timer_Sync

T12 only: Read_Timer_Sync returns the current counter value of the *ADwin* system timer.

Syntax

```
#Include ADwinPro_All.inc
ret val = Read Timer Sync()
```

Parameters

ret val Current counter value in units of 1 ns.

LONG

Notes

The counter value can only be read.

You may determine a time interval from the difference of 2 timer values. Please note that any read timer value will be reached again after 4.3 seconds.

Please note that the time measurement inside of a program section follows special rules. Find more under "Measuring the Processing Time" and especially "Annotations for T12".

See also

```
Processdelay, Read Timer
```

Example

```
Dim timervalue As Long
Event:
   timervalue = Read_Timer_Sync()
```

See also related example seconds_timer.bas in folder

C:\ADwin\ADbasic\samples ADwin



Rem, '

The compiler instructions Rem or " $^{\prime\prime}$ " make it possible to insert comments into the source code for a program. Any text in a program line following the instruction is ignored by the compiler.

Syntax

```
Rem comment
instr : Rem comment
instr 'comment
```

Parameters

Any character strings.

ADbasic instruction.

Notes

The instruction only applies to the line, in which it is used. If a comment requires more than one text line, then you must begin each line with the instructions Rem or " r".

If you want to insert a Rem comment after an instruction, separate it from the instruction by a colon ":". If you use " $^{\prime\prime}$ ", a colon is not necessary.

You can set the horizontal position of a comment after an instruction; find more under Positioning comments (page 23).

```
Rem This is a comment that needs more than
Rem one text line
'This is a comment line, too
Dim min As Long: Rem comment after an instruction
Dim max As Long
'Also a comment after an instruction
```



Reset_Event

Reset_Event deletes all external Event signals, which are to be processed.

Syntax

```
Reset Event
```

Notes

The instruction is only valid for externally controlled processes and in the Init: section.

We recommend running the instruction at the end of the Init: section; this is even mandatory with processor T11. This prevents a too early Event signal (coming up during initialization) from starting the main program (Event: section) too early.

More about the operating mode of the operating system for externally controlled processes see section "Externally Controlled Process" on page 153.

See also

End, Exit, ProcessN_Running, Start_Process, Stop_Process



Restart_Process

Since Processor T11: Restart_Process starts the same process again.

Syntax

```
Restart Process
```

Notes

The instruction is valid in the program section Finish: only.

All lines of the program section after **Restart_Process** will be executed, before the process starts anew. For better readability, we recommend putting the instruction at the end of the program section.

The instruction may cause an endless loop. Prevent an endless loop by using Restart Process inside of a conditional block.



See also

```
End, Exit, If ... Then ... {Else ...} EndIf, Start_Process, Start_Process_ Delayed, Stop Process
```

```
Event:
    Rem ...
Finish:
    Rem ...
    If (cond = 2) Then
        Rem If condition is true, the process is started anew
        Restart_Process
EndIf
```



Round

Round returns the nearest integer of a value of data type Float.

Syntax

```
ret val = Round(value)
```

Parameters

ret val

value Variable or expression.

Rounded integer value.

FLOAT

Notes

Use the instruction Round before assigning a float value to a variable of data type Long. Otherwise, the decimal places of the float value are cut off. This is important especially for float values a bit less than the next integer value e.g. the value 0.99999.

Rounding is executed according to IEEE 754 standard that is to the nearest integer. If the float value is half-way between two integers, the result will be the nearest even integer.

See also

```
chapter 4.4.2 "Type Conversion", AbsF
```



SelectCase

The SelectCase control structure is used to execute one of several instruction blocks depending on a given value.

Syntax

Parameters

var	Argument to be evaluated (no expression).	LONG
constla, constlb, const2a, const2b	Value of var (0255), where the following instruction block will be executed.	CONST

Notes

This control structure cannot be used within a library function or sub-routine.

You may nest several SelectCase structures; the only limit is the memory size.

Depending on the argument you can replace multiple nested IF structures with SelectCase so that they will be more clearly structured; another benefit is this structure is executed faster than several consecutive If structures

If the argument to be evaluated does not correspond to one of the Case constants, only the CaseElse instruction block is executed (if



there is any). This is also true when the argument to be evaluated is beyond the value range of the constant.

CCase means "Continue Case": If a Case or CCase instruction block has been executed, then a directly following CCase instruction block is executed, too.

In the example below, not only ADC(5), but also ADC(7) are executed. However, if Par 1=3, then only ADC(7) will be executed.

If you change variables in the instruction blocks in such a manner that the value of the argument is changed, this will only be considered at the next SelectCase query.

The SelectCase structure creates an internal branch table located in the data memory (DM), whose memory requirements correspond to the greatest used Case-/CCase-constant. In order to limit the memory requirements to a minimum, the value range of constants is restricted to 0...255. There is:

Memory requirement in bytes = $[(greatest constant value)+1] \times 4$

As an example the memory requirement with a max. Case constant 200 is $(200 + 1) \times 4 = 804$ Bytes; the maximum possible memory requirement is 1 KiB.

See also

```
Do ... Until, For ... To ... {Step ...} Next, If ... Then ... {Else ...} EndIf
```



```
Event:
 Par 1=2
 SelectCase Par_1
                   'Evaluate Par 1
                     'If Par 1 = 0?
  Case 0
    Case 1
    Par 10 = ADC(3) 'Read out ADC(3)
                     'If Par_1 = 2?
  Case 2
    Par_10 = ADC(5) 'read out ADC(5) and ADC(7), too
                     '(by CCase)
  CCase 3
                     'If Par 1 = 3?
    Par_11 = ADC(7)
                     'Read out ADC(7)
  Case 4,5,6,7,16
                  'If Par 1 = 4,5,6,7,or 16?
    Par_2 = Digin_Word() 'read digital inputs
  CaseElse
                     'Par 1: other values
    Digout Word(Par 10) 'Output value of Par 10 to
                     'the digital outputs
 EndSelect
                     'End of selection
```



Shift_Left

The **Shift_Left** instruction shifts all bits of a value by a specified number of places to the left. The empty bits at the right are filled with zeros.

Syntax

```
ret val = Shift Left(val, num)
```

Parameters

val	Argument.	LONG
num	Number of places the argument is shifted (031).	LONG
ret_val	Argument with shifted bits or. 0 for (num<0) and for (num>31).	LONG

Notes

Use only integer values for the argument if possible. Floating-point values (of the type Float) are converted into integer values before shifting them. The decimal places are truncated and the value is rounded if necessary.

Shifting the bits n places to the left corresponds to the multiplication with 2ⁿ. A possible overflow is not taken into account, which means, a set bit is lost if it is left-shifted beyond the length of an argument.

The execution time is similar to that one of a comparable multiplication operator.

See also

```
Shift Right
```

```
Dim val1, val2 As Long

Event:
  val1 = 1024
  val2 = Shift Left(val1, 2) 'Result: val2=4096
```



Shift_Right

The **Shift_Right** instruction shifts all bits of a value by a specified number of places to the right. The empty bits at the left are filled with zeros.

Syntax

```
ret val = Shift Right(val,num)
```

Parameters

val	Argument.	LONG
num	Number of places, which are shifted (031).	LONG
ret_val	Argument with shifted bits or. 0 for (num<0) and for (num>31).	LONG

Notes

Use only integer values for the argument if possible. Floating-point values (of the type Float) are converted into integer values before shifting them. The decimal places are truncated and the value is rounded.

If the argument val is a positive number, shifting it num places to the right corresponds to a division by 2ⁿ. A possible division remainder is not taken into account, which means, a set bit is lost if it is right-shifted beyond the length of an argument.

The execution time is shorter than the execution time of a comparable division. For instance, $val_2 = Shift_Right(val_1, 3)$ is faster than $val_2 = val_1 / 8$.

See also:

```
Shift Left
```

```
Dim val1, val2 As Long

Event:
  val1 = 1024
  val2 = Shift Right(val1, 3) 'Result: val2=128
```



Sin

Sin provides the sine of an angle.

Syntax

```
ret val = Sin(angle)
```

Parameters

```
angle Arc angle (-\pi...+\pi). FLOAT ret_val Sine of the angle (-1...1).
```

Notes

If you use input values, which are not in the range of $-\pi...+\pi$, the calculation error grows with the increasing value.

The execution time of the function takes $1.25\,\mu s$ with a T9, $0.63\,\mu s$ with a T10, and $0.28\,\mu s$ with a T11.

See also

Cos, Tan, ArcSin, ArcCos, ArcTan, Round



Sleep

Processors until T10 only: Sleep causes the processor to wait for a certain time.

Syntax

Sleep (val)

Parameters

val

Number (\geq 1) of time units to wait in 100ns.

LONG

Notes

For processor T11, Sleep must be replaced by one of the instructions CPU_Sleep, P1_Sleep or P2_Sleep (see also chapter 5.2.4 "Setting Waiting Times Exactly"); mostly P1 Sleep is best.

Since **Sleep** is executed as a count loop, it cannot be interrupted in high-priority process.

Please make sure (especially when using variables) that the argument does not have a value less than 1, otherwise the *TiCo* processor *ADwin* system will become unstable. In addition, please consider that very high values in high-priority processes can cause an interruption in the communication to the PC.



If possible, use a constant as argument. If the argument val requires a calculation, it requires additional time; this time interval is constant and takes a few clock cycles.

The following conditions require a calculation:

- The argument is an expression with variables or array elements.
- The variable in the argument is declared in the memory area DRAM Extern.
- The argument is an array.
- The argument is a floating-point value.

See also

CPU_Sleep, NOP, P1_Sleep, P2_Sleep





Sqrt

Sgrt returns the square root of a value.

Syntax

```
ret val = Sqrt(val)
```

Parameters

val	Argument (≥0).	FLOAT
ret_val	Square root of the argument.	FLOAT

Notes

The execution time of the function takes $0.9\mu s$ with a T9, $0.45\mu s$ with a T10, and $0.26\mu s$ with a T11.

With an invalid argument, the return value is undefined.

See also

```
^ Power, Exp, LN, Log, Round
```



LONG

Start_Process

Start Process starts a specified process.

Syntax

```
Start Process(processnum)
```

Parameters

```
processnu Number of the process to be started (1...12, 15).
```

Notes



Please assure that the process is transferred to the *ADwin* system before you start it.

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

- · is already running or
- has the same number as the calling process.

You can start a process with Start_Process from another process
only (except for Restart_Process). It is not possible that a process
starts itself, for instance in the section Finish:.

Start_Process launches a process but does not wait until it is actually running. If required you can query with **ProcessN_Running** if the launched process really has started working.

See also

End, Exit, Restart Process, Start Process Delayed, Stop Process

```
Event:
```



Start_Process_Delayed

Since processor T11: Start_Process_Delayed starts a specified process (section Event:) with the defined delay.

Syntax

```
Start Process Delayed (processno, delay)
```

Parameters

processno Number of the process to be started (1...10).

LONG

delay

Delay time (>30) in clock cycles of the timer.

With T11, one clock cycle takes $3,\overline{3}$ ns, with T12 1,0 ns.

LONG

Notes

Please assure that the process is transferred to the *ADwin* system before you start it.



The instruction may only start a time-controlled process with high priority; it has no effect, if you indicate the number of a process, where one of the following is true:

- The process is externally controlled.
- · The process has low priority.
- The process is running already.
- The process has the same number as the calling process.

You may start a process with Start_Process_Delayed from a different process only (except for Restart_Process).

A delayed started process always begins with the Event: section, the sections Init: and LowInit: will not be executed.

These items apply to the wanted starting time:

- The delay until starting time starts being counted with processing Start_Process_Delayed; the processing time of the instruction is 30 clock cycles.
- From a high-priority program section the starting time can only
 be maintained, if the delay time delay is greater than the
 remaining processing time for the rest of the section.
 Any subsequent lines of the section must be processed, before
 the selected process can start. The starting time therefore is
 additionally delayed by a too long remaining processing time.



See also

Restart_Process, Start_Process, Stop_Process

Example

Event:

```
Rem ...

If (cond = 2) Then

Rem If condition is true, process 2 is started

Rem with a delay of 100 clock cycles.

Start_Process_Delayed(2,100)
```

Rem There are NO MORE program lines here to surely maintain $\mbox{\it Rem}$ the wanted starting time.



Stop_Process

Stop_Process stops a specified process from another running process.

Syntax

Stop Process (processnum)

Parameters

processnu Number of the process to be stopped (1...12,15).

LONG

Notes

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

- · has already been stopped,
- has not yet been loaded to the ADwin system.

Stopping the Event: section happens as follows:

- First, the specified process gets the status "process is being stopped" (see Processn_Running); with low priority processes this will take some time (time-out).
- If the Event: section is being processed when the stop signal arrives, the execution of the Event: section is yet completed.
- Normally the Event: section is called and processed once again.
- If existing, the Finish: section is processed (always at low priority).
- When Stop_Process has completed, the specified process is inactive, but can be started at any time.

If you like the process to stop itself, use the instructions ${\tt End}$ or ${\tt Exit}$.



See also

End, Exit, ProcessN_Running, Restart_Process, Start_Process, Start_Process_Delayed





String ""

Strings are put into quotes " ".

Syntax

```
Import String.LI*
  'T9: .LI9 / T10: .LIA / T11: .LIB / T12: .LIC
Dim text[length] as String
text = "ADwin"
```

Parameters

text [] Name of the text variable.

ARRAY

length Length of the text variable.

CONST

LONG

Notes

Dimension text variables with Dim ... As String (see page 195). A string you want to assign to a variable is put in quotes.

More information about text variables and the structure of strings can be found under "Strings" on page 117.

Strings can be processed with the instructions mentioned below. Also, you can add (concatenate) strings with the "+"-operator.

Please note with processor T12 that you may not access the array element 1 of a string (string length). Use **StrLen** to determine the string length.

See also

+ String Addition, Dim, Asc, Chr, FloToStr, Flo40ToStr, LngToStr, Str-Comp, StrLeft, StrLen, StrMid, StrRight, ValF, ValI



```
Import String.LI9
 'T9: .LI9 / T10: .LIA / T11: .LIB / T12: .LIC
Rem Dimension strings with 3 and 1 characters
Dim chars[3] As String
Dim char[1] As String
 Rem Transfer characters to the strings
 chars = "ABC"
 char = "z"
Event:
 REM T12 only: chars[1] is not available, so the following
 REM line must be deleted.
 Par_2 = chars[2]
Par_3 = chars[3]
                     'Par \ 3 = 66 \ (= "B")
                     'Par\ 4 = 67\ (=\ "C")
 Par 4 = chars[4]
 Par 5 = chars[5]
                     'Par 5 = 0 end of string
 Rem Conversion into upper Case:
 Rem Lower Case: a, b, c, ..., x, y, z?
 Par 6 = Asc(char)
 If (Par 6>96 And Par 6<133) Then
  Rem Subtract 32 in order to convert into upper cases
  Chr (Par 6-32, char)
 EndIf
```



StrComp

StrComp checks two strings to determine if they are identical.

Syntax

```
Import String.LI*
  'T9: .LI9 / T10: .LIA / T11: .LIB / T12: .LIC
ret_val = StrComp(string1[], string2[])
```

Parameters

LONG

ARRAY

STRING CONST

Notes

If the strings do not have the same lengths, a negative value is returned, even if the shorter string is included in the longer one.

See also

```
String "", + String Addition, Asc, Chr, FloToStr, Flo40ToStr, LngToStr, StrLeft, StrLen, StrMid, StrRight, ValF, ValI
```



StrLeft

StrLeft returns a specified number of characters from the left end of a string into a second string.

Syntax

```
Import String.LI*
  'T9: .LI9 / T10: .LIA / T11: .LIB / T12: .LIC
StrLeft(string1[], length, string2[])
```

Parameters

 ${\tt string1[]} \quad {\tt String, from which is copied.}$

ARRAY

length Number of characters to be copied.

LONG

string2[] String, into which is copied.

STRING

See also

```
String "", + String Addition, Asc, Chr, FloToStr, Flo40ToStr, LngToStr, StrComp, StrLen, StrMid, StrRight, ValF, ValI
```



```
Import String.LI9
 'T9: .LI9 / T10: .LIA / T11: .LIB / T12: .LIC
Rem Dimension the source and destination strings
Dim text1[32], text2[14] As String
Init:
 Rem Define source string
 text1 = "MEGA real-time with ADwin systems"
Event:
 Rem Get 14 characters from the left of string text1
 StrLeft(text1,14,text2)
 REM T12 only: text2[1] is not available, the following
 REM line must be deleted.
 Par 1 = text2[1]
                  'String length = 14 chars
 Par 2 = text2[2]
                   'ASCII-char 4Dh = "M"
                   'ASCII-char 45h = "E"
 Par 3 = text2[3]
                    'ASCII-char 47h = "G"
 Par 4 = text2[4]
 Par 5 = text2[5]
                   'ASCII-char 41h = "A"
 Par_6 = text2[6]
                    'ASCII-char 20h = " "
 Par 7 = text2[7]
                   'ASCII-char 72h = "r"
 Par 8 = text2[8]
                   'ASCII-char 65h = "e"
 Par 9 = text2[9]
                   'ASCII-char 61h = "a"
```



StrLen

StrLen returns the number of characters in a string.

Syntax

```
Import String.LI*
  'T9: .LI9 / T10: .LIA / T11: .LIB / T12: .LIC
ret_val = StrLen(string[])
```

Parameters

```
string[] String whose length is determined.
```

ARRAY
STRING

```
ret_val Number of characters in the string.
```

Notes

Please note with processor T12 that you may not access the array element 1 of a string to determine the string length (possible until T11). Use Strien instead.

See also

```
String "", + String Addition, Asc, Chr, FloToStr, Flo40ToStr, LngToStr, StrComp, StrLeft, StrMid, StrRight, ValF, Vall
```

```
Import String.L19
  'T9: .L19 / T10: .L1A / T11: .L1B / T12: .L1C
Dim text1[50] As String

Init:
  text1 = "MEGA real-time with ADwin systems"

Event:
  Par_1 = StrLen(text1) 'String length: Par_1 = 33
```



StrMid

StrMid returns a specified number of characters from a string into a second string, starting from a certain position in the string.

Syntax

```
Import String.LI*
  'T9: .LI9 / T10: .LIA / T11: .LIB / T12: .LIC
StrMid(string1[], start, length, string2[])
```

Parameters

string1[]	String, from which is copied.	ARRAY
		STRING
start	Position of the first character, which is copied.	LONG
length	Number of characters to be copied.	LONG
string2[]	String, into which is copied.	ARRAY
		STRING

See also

```
String "", + String Addition, Asc, Chr, FloToStr, Flo40ToStr, LngToStr, StrComp, StrLeft, StrLen, StrRight, ValF, ValF
```



```
Import String.LI9
 'T9: .LI9 / T10: .LIA / T11: .LIB / T12: .LIC
Rem Dimension source and destination strings:
Dim text1[32], text2[20] As String
Init:
 Rem Define source string
 text1 = "MEGA real-time with ADwin systems"
Event:
 Rem Copy 20 characters beginning at the 6. character from
 Rem the string text1
 StrMid(text1,6,18,text2)
 Par 2 = text2[2]
                  'ASCII-char 72h = "r"
 Par 3 = text2[3]
                  'ASCII-char 65h = "e"
                  'ASCII-char 61h = "a"
 Par 4 = text2[4]
                  'ASCII-char 6Ch = "1"
 Par 5 = text2[5]
 Par 6 = text2[6]
                  'ASCII-char 2Dh = "-"
                  'ASCII-char 74h = "t"
 Par 7 = text2[7]
                  'ASCII-char 69h = "i"
 Par 8 = text2[8]
                  'ASCII-char 6Dh = "m"
 Par 9 = text2[9]
                  'ASCII-char 65h = "e"
 Par 10 = text2[10]
 Par 11 = text2[11]
                  'ASCII-char 20h = " "
                  'Ascii-char 77h = "w"
 Par 12 = text2[12]
                  'ASCII-char 69h = "i"
 Par 13 = text2[13]
                  'ASCII-char 74h = "t"
 Par 14 = text2[14]
 Par 15 = text2[15]
                  'ASCII-char 68h = "h"
```



StrRight

StrRight returns a specified number of characters from the right end of a string into a second string.

Syntax

```
Import String.LI*
  'T9: .LI9 / T10: .LIA / T11: .LIB / T12: .LIC
StrRight(string1[], length, string2[])
```

Parameters

string1[]	String, from which it is copied.	ARRAY
Der mig 1 []		STRING
length	Number of the characters to copy.	LONG
string2[]	String, into which it is copied.	ARRAY
		STRING

See also

```
String "", + String Addition, Asc, Chr, FloToStr, Flo40ToStr, LngToStr, StrComp, StrLeft, StrLen, StrMid, VaIF, VaII
```



```
Import String.LI9
 'T9: .LI9 / T10: .LIA / T11: .LIB / T12: .LIC
Rem Dimension the source and destination string:
Dim text1[32], text2[13] As String
Init:
 Rem Define the source string
 text1 = "MEGA real-time and ADwin systems"
Event:
 Rem Get 13 characters from the string text1,
 Rem starting at right
 StrRight(text1, 13, text2)
 Par 2 = text2[2]
                  'ASCII-char 41h = "A"
 Par 3 = text2[3]
                   'ASCII-char 44h = "D"
                   'ASCII-char 77h = "w"
 Par 4 = text2[4]
                   'ASCII-char 69h = "i"
 Par 5 = text2[5]
 Par 6 = text2[6]
                  'ASCII-char 6Eh = "n"
 Par_7 = text2[7]
                   'ASCII-char 2Dh = "-"
 Par 8 = text2[8]
                   'ASCII-char 53h = "S"
 Par_9 = text2[9]
                   'ASCII-char 79h = "y"
 Par 10 = text2[10]
                  'ASCII-char 73h = "s"
 Par 11 = text2[11]
                  'ASCII-char 74h = "t"
```



Sub ... EndSub

The Sub...EndSub commands are used to define a subroutine macro with passed parameters.

Syntax

Parameters

Notes

You will find general information about macros in chapter 4.5.1 on page 125.

This instruction defines a subroutine-macro, which means the whole instruction block between <u>Sub</u> and <u>EndSub</u> is inserted in the place where the macro is called.

Subroutines help to make your source code more clearly structured. Please note that each subroutine call will enlarge the compiled file.

You may insert subroutines at the following 3 places:

- Before section Init:/LowInit:
- 2. After section Finish:
- In a separate file, which you include with #Include (only at the locations 1. and 2.).



Be aware that in subroutines:

- no process sections such as LowInit:, Init:, Event:, Or Finish: can be defined,
- local variables can be defined at the beginning, which are only available in the function and for the processing period.
 This is true even when a variable has the same name as a variable outside the function.

If a passed parameter is part of an expression inside a subroutine, the parameter should be set in braces. This avoids problems with precedence rules (e.g. BODMAS).

A subroutine is called with its name and with all its arguments, which you have defined. Valid arguments include every expression (also arrays), as long as it has the appropriate data type.

If you do not define arguments, you have to use the empty parentheses when calling the subroutine: name().

If an array (not an array element) is used as a passed parameter, the syntax is different for call and definition:

- Subroutine call without dimension brackets:
 - subname(array_pass)
- Subroutine definition with dimension brackets:

```
Sub subname(array_def[])...
```

Values are assigned to elements of passed arrays as usual:

```
array pass[2] = value
```



If a value is assigned to a passed parameter $\mathbf x$ within the subroutine, the subroutine's call must not use a constant $\mathbf x$, but a variable or a single array element. If so, a passed parameter can be used to hold a return value.

See also

```
#Include, Function ... EndFunction, Lib_Sub ... Lib_EndSub, Lib_Function ... Lib_EndFunction
```

```
Sub Fast_Dac1(val1)

Rem Outputs val1 on analog output 1 of an ADwin-Gold

Poke(20400050h, (val1))'Write value into the

'output register

Poke(20400010h, 11011b) 'Start conversion

EndSub
```



Calling the subroutine Fast_Dac1 is made with the program line:

```
Event:
   Fast Dac1(NewValue)
```

Example 2

The same subroutine with an array as passed parameter:

```
Sub Fast_Dac1(array[])
  Rem Outputs element 3 of the array on the
  Rem analog output 1 of an ADwin-Gold
  Poke(20400050h, (array[3]))'Write value to output
  Poke(20400010h, 11011b) 'Start conversion
EndSub
```

Calling this subroutine is made in a similar manner (but *without* dimension brackets):

```
Event:
  Fast Dacl(array)
```

For array, you can indicate a global or a local array. Enter the array name only, without element number and brackets.



Tan

Tan returns the tangent of an argument.

Syntax

```
ret val = Tan(angle)
```

Parameters

```
angle Arc angle (-\pi/2...\pi/2). FLOAT ret_val Cosine of the angle (-1...1).
```

Notes

If you use input values, which are not in the range of $-\pi/2...+\pi/2$, the calculation error grows with the increasing value.

The execution time of the function takes $1.33\,\mu s$ with a T9, $0.67\,\mu s$ with a T10, and $0.31\,\mu s$ with a T11.

See also

```
Sin, Cos, ArcSin, ArcCos, ArcTan, Round
```



User_Version

T12 only: User Version return the version number of a program.

Syntax

```
ret val = User Version
```

Parameters

-/-

Notes

User_Version is a system constant and may only be read but not be changed via software.

You set the version number in the Process Options dialog box (page 56).



ValF

Valf converts a string into a floating-point number.

Syntax

```
Import String.LI*
  'T9: .LI9 / T10: .LIA / T11: .LIB / T12: .LIC
ret val = Valf(String[])
```

Parameters

String [] String, which is to be converted, in the following format: ARRAY

STRING

Mantissa (max. 10 characters)			Exponent (099)			
{+}	vvvvv		nnnnn	e	{+}	nn
-		,		Е	-	

ret val Generated floating-point value.

FLOAT

Notes

If you do not indicate a sign, a positive sign will be assumed.

The character "E" divides mantissa from exponent. With T9 and T10, in the mantissa only a maximum of 7 characters (pre-decimal *and* decimal places) are evaluated, with T11 and T12 a maximum of 10 characters. If you have more characters, the last of them will be lost. As decimal separator either the dot or the comma are allowed.

Please note the value range for float values in chapter 4.2.3 on page 102. Values outside the value range are interpreted as "infinite" or zero.

If you use illegal characters (characters other than indicated in the format above), only the strings up to the first illegal sign will be evaluated.

See also

String "", + String Addition, Asc, Chr, FloToStr, Flo40ToStr, LngToStr, StrComp, StrLeft, StrLen, StrMid, StrRight, Vall



```
Import String.LI9
 'T9: .LI9 / T10: .LIA / T11: .LIB / T12: .LIC
Dim text[20] As String
Init:
 text="-27.14159E-10" 'String to be converted
 Par 1 = text[1]
                      'String-length
                      'ASCII-char 2Dh = "-"
 Par_2 = text[2]
 Par 3 = text[3]
                      'ASCII-char 32h = "2"
                      'ASCII-char 37h = "7"
 Par 4 = text[4]
 Par 5 = text[5]
                      'ASCII-char 2Eh = "."
 Par 6 = text[6]
                      'ASCII-char 31h = "1"
 Par 7 = text[7]
                      'ASCII-char 34h = "4"
                      'ASCII-char 31h = "1"
 Par 8 = text[8]
 Par 9 = text[9]
                      'ASCII-char 35h = "5"
 Par 10 = text[10]
                      'ASCII-char 39h = "9"
                     'ASCII-char 45h = "E"
 Par 11 = text[11]
                     'ASCII-char 2Dh = "-"
 Par 12 = text[12]
 Par 13 = text[13]
                     'ASCII-char 31h = "1"
 Par 14 = text[14]
                     'ASCII-char 30h = "0"
 Par 15 = text[15] 'End of string sign
Event:
 FPar 1 = ValF(text) 'Convert string to Float
```



Vall

ValI converts a string into an integer number (Long).

Syntax

```
Import String.LI*
  'T9: .LI9 / T10: .LIA / T11: .LIB / T12: .LIC
ret_val = ValI(String[])
```

Parameters

```
String [] String to be converted in the format:
Sign: + (optional) or -.
Pre-decimal places: max. 10 characters.

{+} vvvvvvvvv
-.

ret val Generated long value.
```

LONG

ARRAY

STRING

Notes

If you do not indicate a sign, a positive sign will be assumed.

Please note the value range for long values:

-2147483648 to +2147483647

Values outside this range are interpreted as zero.

If you use illegal characters (characters other than indicated in the format above), the string up to the first illegal characters will be evaluated only.

See also

```
String "", + String Addition, Asc, Chr, FloToStr, Flo40ToStr, LngToStr, StrComp, StrLeft, StrLen, StrMid, StrRight, ValF
```



```
Import String.LI9
 'T9: .LI9 / T10: .LIA / T11: .LIB / T12: .LIC
Dim text[20] As String
Init:
 text="-1234567890"
                      'String to be converted
 Par 1 = text[1]
                       'String-length = 11
                       'ASCII-char 2Dh = "-"
 Par_2 = text[2]
 Par 3 = text[3]
                      'ASCII-char 31h = "1"
 Par 4 = text[4]
                       'ASCII-char 32h = "2"
 Par 5 = text[5]
                      'ASCII-char 33h = "3"
 Par 6 = text[6]
                      'ASCII-char 34h = "4"
 Par 7 = text[7]
                      'ASCII-char 35h = "5"
 Par 8 = text[8]
                      'ASCII-char 36h = "6"
 Par 9 = text[9]
                      'ASCII-char 37h = "7"
 Par 10 = text[10]
                      'ASCII-char 38h = "8"
                      'ASCII-char 39h = "9"
 Par 11 = text[11]
                      'ASCII-char 30h = "0"
 Par 12 = text[12]
 Par 13 = text[13]
                     'End of string sign
Event:
 Par 20 = ValI(text) 'Convert string to long
```



XOr

The operator XOr (Exclusive-Or) combines two integer values bitwise.

Syntax

```
... val 1 XOr val 2 ...
```

Parameters

LONG

See also

```
And, Not, Or
```

```
Dim value As Long
Event:
  value = 0100b XOr 0110b
  Rem Result: value = (4 XOr 6) = 0010b = 2
```



8 How to Solve Problems?

If problems already occur during installation, please refer to the documentation for your *ADwin* system. Make sure all settings have been carried out properly and completely. Also check if the base address, the processor type, etc. are set correctly in the menu <code>Options\Compiler</code>. If your problems still persist, please give your local technical support office a call.

If you need help of a more substantial nature, you can contact us directly; you find the address inside the manual's cover page.





Appendix

A.1 Short-Cuts in ADbasic

To display short-cuts of code snippets, open <ADbasicCS.xml> in the folder $C:\ADwin\ADbasic\Common\$ with a browser.

Short cut key	Function	Matching menu item
F1	Show help topic for	
	marked instruction.	
CTRL-F1	Show online help con-	Help▶ Content
	tent.	
F2	Show declaration of	
	marked instruction.	
CTRL-F2	Jump to declaration of	
	marked instruction.	
F3	Find next forward.	Edit Find Next
SHIFT-F3	Find next backwards.	
CTRL-F3	Find Text at cursor posi-	
	tion forward.	
CTRL-SHIFT-F3	Find Text at cursor posi-	
	tion backwards.	
CTRL-F5	Boot ADwin system.	
F6	Create library.	Build▶Make Lib
		File
F7	Create binary file.	Build▶Make Bin
		File
CTRL-F7	Create binary files of the	Build▶Make All Bin
	project.	Files
F8	Compile source code.	Build▶ Compile
CTRL-F8	Start process.	
F9	Stop process.	
CTRL-SPACE	Insert or complete a dec-	
	laration.	
CTRL-SHIFT-	Show parameters of a	
SPACE	sub / function.	



Short cut key	Function	Matching menu item
CTRL-A	Select all.	Edit Select All
CTRL-A	Comment marked lines	Source context menu:
CIRL-B	Comment marked lines	
		Comment Block
CTRL-SHIFT-B	Uncomment marked	
	lines	Uncomment Block
CTRL-C	Сору.	Edit ▶ Copy
CTRL-F	Find text.	Edit ▶ Find
CTRL-G	Jump to a line.	
CTRL-H	Replace text.	Edit ▶ Replace
CTRL-I	Indent marked lines	Source context menu:
		Indent
CTRL-SHIFT-I	Outdent marked lines	Source context menu:
		Outdent
CTRL-N	New source code file.	File▶ New
CTRL-0	Open source code file.	File▶ Open
CTRL-P	Print source code file.	File Print
CTRL-R	Colour mark used	Parameter window:
	parameters	Icon 🔑
CTRL-S	Save source code file.	File Save
CTRL-U	Lower case.	
CTRL-SHIFT-U	Upper case.	
CTRL-V	Paste.	Edit Paste
CTRL-X	Cut.	Edit ▶ Cut
CTRL-Z	Undo input.	Edit ▶ Undo
CTRL-SHIFT-Z	Redo input.	Edit ▶ Redo
CTRL-K + K	Insert / delete bookmark.	
CTRL-K + N	Jump to next bookmark.	
CTRL-K + P	Jump to previous book-	
	mark.	
CTRL-K + X	Insert a code snippet.	_
I egend:		

Legend:



A-B: Press keys A and B at the same time.

A+B: Press key A first, release and then press key B.



A.2 ASCII-Character Set

NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL
00h 0	01h 1	02h 2	03h 3	04h 4	05h 5	06h 6	07h 7
BS ¹	TAB ²	LF ³	VT	FF	CR4	so	SI
08h 8	09h 9	0Ah 10	0Bh 11	0Ch 12	0Dh 13	0Eh 14	0Fh 15
DLE	DC1				NAK	-	
10h 16	11h 17	12h 18	13h 19	14h 20	15h 21	16h 22	17h 23
CAN	EM	SUB	ESC	FS	GS	RS	บร
18h 24	19h 25	1Ah 26	1Bh 27	1Ch 28	1Dh 29	1Eh 30	1Fh 31
SPC ⁵	!	"	#	\$	용	&	,
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	1	2	3	4	5	6	7
30h 48	31h 49	32h 50	33h 51	34h 52	35h 53	36h 54	37h 55
8	9	:	;	<	=	>	?
38h 56	39h 57	3Ah 58	3Bh 59	3Ch 60	3Dh 61	3Eh 62	3Fh 63
@	A	В	С	D	E	F	G
		B 42h 66	_	_	_	_	_
		_	_	_	_	_	_
40h 64 H	41h 65 I	42h 66	43h 67 K	44h 68	45h 69 M	46h 70 N	47h 71
40h 64 H	41h 65 I	42h 66	43h 67 K	44h 68	45h 69 M	46h 70 N	47h 71
40h 64 H 48h 72 P	41h 65 I 49h 73 Q	42h 66 J 4Ah 74	43h 67 K 4Bh 75 S	44h 68 L 4Ch 76 T	45h 69 M 4Dh 77 U	46h 70 N 4Eh 78	47h 71 O 4Fh 79
40h 64 H 48h 72 P 50h 80	41h 65 I 49h 73 Q	42h 66 J 4Ah 74 R	43h 67 K 4Bh 75 S	44h 68 L 4Ch 76 T	45h 69 M 4Dh 77 U	46h 70 N 4Eh 78	47h 71 O 4Fh 79
40h 64 H 48h 72 P 50h 80	41h 65 I 49h 73 Q 51h 81 Y	42h 66 J 4Ah 74 R 52h 82	43h 67 K 4Bh 75 S 53h 83	44h 68 L 4Ch 76 T 54h 84	45h 69 M 4Dh 77 U 55h 85	N 4Eh 78 V 56h 86	47h 71 O 4Fh 79 W 57h 87
40h 64 H 48h 72 P 50h 80	41h 65 I 49h 73 Q 51h 81 Y	42h 66 J 4Ah 74 R 52h 82	43h 67 K 4Bh 75 S 53h 83	44h 68 L 4Ch 76 T 54h 84	45h 69 M 4Dh 77 U 55h 85	N 4Eh 78 V 56h 86	47h 71 O 4Fh 79 W 57h 87 5Fh 95
40h 64 H 48h 72 P 50h 80 X 58h 88	41h 65 I 49h 73 Q 51h 81 Y 59h 89	### 42h 66 ### J ### 4Ah 74 ### R ### 52h 82 ### Z ### 5Ah 90	43h 67 K 48h 75 S 53h 83 [58h 91	44h 68 L 4Ch 76 T 54h 84 \ 5Ch 92	45h 69 M 4Dh 77 U 55h 85] 5Dh 93	N 4Eh 78 V 56h 86 ^ 5Eh 94	47h 71 O 4Fh 79 W 57h 87 5Fh 95
40h 64 H 48h 72 P 50h 80 X 58h 88	41h 65 I 49h 73 Q 51h 81 Y 59h 89	42h 66 J 4Ah 74 R 52h 82 Z 5Ah 90 b	43h 67 K 48h 75 S 53h 83 [58h 91	44h 68 L 4Ch 76 T 54h 84 \ 5Ch 92	45h 69 M 4Dh 77 U 55h 85] 5Dh 93	N 4Eh 78 V 56h 86 ^ 5Eh 94	47h 71 O 4Fh 79 W 57h 87 5Fh 95
40h 64 H 48h 72 P 50h 80 X 58h 88 . 60h 96 h	41h 65 I 49h 73 Q 51h 81 Y 59h 89 a 61h 97	42h 66 J 4Ah 74 R 52h 82 Z 5Ah 90 b 62h 98	43h 67 K 48h 75 S 53h 83 [58h 91 C 63h 99	44h 68 L 4Ch 76 T 54h 84 \ 5Ch 92 d 64h 100 1	45h 69 M 4Dh 77 U 55h 85 J 5Dh 93 e 65h 101 m	46h 70 N 4Eh 78 V 56h 86 ^ 5Eh 94 £ 66h 102 n	47h 71 O 4Fh 79 W 57h 87 5Fh 95 G 67h 103
40h 64 H 48h 72 P 50h 80 X 58h 88 . 60h 96 h	41h 65 I 49h 73 Q 51h 81 Y 59h 89 a 61h 97	42h 66 J 4Ah 74 R 52h 82 Z 5Ah 90 b 62h 98	43h 67 K 48h 75 S 53h 83 [58h 91 C 63h 99	44h 68 L 4Ch 76 T 54h 84 \ 5Ch 92 d 64h 100 1	45h 69 M 4Dh 77 U 55h 85 J 5Dh 93 e 65h 101 m	46h 70 N 4Eh 78 V 56h 86 ^ 5Eh 94 £ 66h 102 n	47h 71 O 4Fh 79 W 57h 87 5Fh 95 G 67h 103
40h 64 H 48h 72 P 50h 80 X 58h 88 60h 96 h 68h 104	41h 65 I 49h 73 Q 51h 81 Y 59h 89 a 61h 97 i 69h 105	42h 66 J 4Ah 74 R 52h 82 Z 5Ah 90 b 62h 98 j 6Ah 106	43h 67 K 48h 75 S 53h 83 [58h 91 C 63h 99 k 68h 107	44h 68 L 4Ch 76 T 54h 84 \ 5Ch 92 d 64h 100 1 6Ch 108	45h 69 M 4Dh 77 U 55h 85] 5Dh 93 e 65h 101 m 6Dh 109 u	46h 70 N 4Eh 78 V 56h 86 ^ 5Eh 94 f 66h 102 n 6Eh 110	47h 71 O 4Fh 79 W 57h 87 5Fh 95 G 67h 103 O 6Fh 111 W
40h 64 H 48h 72 P 50h 80 X 58h 88 60h 96 h 68h 104 P 70h 112	41h 65 I 49h 73 Q 51h 81 Y 59h 89 a 61h 97 i 69h 105 q 71h 113	42h 66 J 4Ah 74 R 52h 82 Z 5Ah 90 b 62h 98 j 6Ah 106 r	43h 67 K 48h 75 S 53h 83 [58h 91 C 63h 99 k 68h 107 S 73h 115	44h 68 L 4Ch 76 T 54h 84 \ 5Ch 92 d 64h 100 L 6Ch 108 t 74h 116	45h 69 M 4Dh 77 U 55h 85] 5Dh 93 e 65h 101 m 6Dh 109 u 75h 117 }	46h 70 N 4Eh 78 V 56h 86 ^ 5Eh 94 f 66h 102 n 6Eh 110 V 76h 118	47h 71 O 4Fh 79 W 57h 87 5Fh 95 G 67h 103 O 6Fh 111 W 77h 119



A.3 License Agreement

Between the buyer of *ADbasic*—termed the Licensee— and Jäger Computergesteuerte Messtechnik GmbH, Rheinstraße 2-4, 64653 Lorsch—termed hereinafter Jäger Messtechnik GmbH—the following license agreement is concluded:

1. OBJECT OF THE LICENSE AGREEMENT

- 1.1 Object of the license agreement is the software of the compiler and the development system ADbasic (hereinafter termed ADbasic software) as well as the printed user manual "ADbasic: The Real-Time Development Tool for ADwin Systems" (hereinafter termed "printed materials").
- 1.2 The company Jaeger Messtechnik GmbH draws your attention to the fact that it is not possible according to the state of the art to develop computer software in such a way that no errors occur in all applications and combinations. Only a computer software, which is basically practicable according to the user documentation is object of the license agreement.

2. EXTENT OF USAGE

- 2.1 Jaeger Messtechnik GmbH grants the Licensee a single, non-exclusive and individual right of use. This means that you may use the enclosed copy of the *ADbasic* software only on a single computer and only in one single location. The Licensee may transfer the *ADbasic* software in physical form (that is stored on a storage device) from one computer to another computer, provided that it is only used individually on one single computer at any time. A usage other than these restrictions is not permitted.
- 2.2 Programs generated by the Licensee with the *ADbasic* software, may be distributed and used without restriction.

3. SPECIAL RESTRICTIONS

The Licensee is not permitted to



- a) pass or otherwise give to any third party access to the ADbasic software without prior written consent of Jaeger Messtechnik GmbH,
 - 4. electronically transfer the *ADbasic* software from one computer to another over a network or a data transfer channel,
 - change or modify, translate, reverse engineer, decompile or disassemble the ADbasic software without prior written consent of Jaeger Messtechnik GmbH.

OWNERSHIP

- 6.1 Upon purchasing the product, only title to the physical storage device, where the ADbasic software has been stored, is passed to the Licensee. No title to the rights of the ADbasic software itself is passed to the Licensee.
- 6.2 Jaeger Messtechnik GmbH reserves all rights for publication, copying, processing and commercialization of the *ADbasic* software.

COPYRIGHTS

7.1 The ADbasic software and the printed materials are protected by copyright.

For backup purposes, the Licensee may generate a single copy of the *ADbasic* software. He must reproduce the copyright notice of Jaeger Messtechnik GmbH on the copy. The copyright notice on the *ADbasic* software must not be removed.

7.2 It is expressly not permitted to fully or partially copy or reproduce the ADbasic software as well as the printed materials in its original or modified form or merged or included in other software.

GRANT OF LICENSE

8.1 The right to use the ADbasic software can only be granted to a third party with prior written consent of Jaeger Messtechnik GmbH. The Licensee must then completely delete the software, which he has installed and pass it to the third party. (The transfer has to include the original data carrier with the documentation, backup version included). The license may furthermore only be transferred to a third party, if the latter agrees for the benefit of Jaeger Messtechnik GmbH to the terms and conditions of this License Agreement and to the General Conditions of the company Jaeger Messtechnik GmbH.



8.2 You must not rent, lease or lend the ADbasic software.

9. PERIOD OF AGREEMENT

- 9.1 The period of the License Agreement is unlimited.
- 9.2 The right of the Licensee for using the ADbasic software voids automatically without notice of termination, if he violates a condition of this License Agreement. Upon termination of the license, the Licensee must destroy the original data medium and all copies of the ADbasic software, possible modified copies included, as well as the printed materials.

10.CLAIM FOR DAMAGES AND PENALTY UPON VIOLATION OF THE CONTRACT

- 10.1If the Licensee violates conditions of this License Agreement he must pay damages.
- 10.2Notwithstanding, Jaeger Messtechnik GmbH will charge a penalty of 20,000.00 EURO for violation of the copyright, unauthorized usage of the software, and unauthorized distribution of the software to third parties.
- 10.3The title to omission on completion of the contract is not influenced by the claim for damages and the penalties.

11. MODIFICATIONS AND UPDATES

Jaeger Messtechnik GmbH is entitled to update the *ADbasic* software upon its own discretion. Jaeger Messtechnik GmbH is not obliged to have updates of the *ADbasic* software available for the Licensee.

For extensive updates, Jaeger Messtechnik GmbH reserves the right to charge an additional fee.

12.WARRANTY AND LIABILITY OF JAEGER MESSTECHNIK GMBH

 a) Jaeger Messtechnik GmbH assumes warranty to the Licensee that at the moment of delivery the data medium, on which the ADbasic software is stored, is error-free in accordance with the accompanying



materials, when applied under normal operating conditions and under normal maintenance conditions.

- 13. If the data medium is faulty, the Licensee is granted a replacement within the warranty period of 6 months from the date of delivery. He must return the data medium as well as a copy of the invoice to Jaeger Messtechnik GmbH or to the distributor from whom he has purchased the product.
- 14. If a fault as described in Section 10 b) is not eliminated within an adequate period of time by replacement of the product, the Licensee may choose between either allowance (price reduction) or conversion (rescission of the License Agreement). The Licensee is not entitled to any further claims.
- 15. For the reasons mentioned in Section 1.2, Jaeger Messtechnik GmbH does not assume liability for the absence of defects with regards to the *ADbasic* software. In particular, Jaeger Messtechnik GmbH does not assume warranty for the fact that the *ADbasic* software meets the requirements and purposes of the Licensee or is compatible to other programs he is working with. The Licensee is responsible for the correct choice and the consequences of using the *ADbasic* software, as well as for the results he intends to obtain or has obtained. The same applies for the printed materials, which are delivered with the *ADbasic* software.
- 16. Jaeger Messtechnik does not assume liability for damages, unless Jäger Messtechnik GmbH has caused damages by intention or by gross negligence. Liability because of properties assured by Jaeger Messtechnik GmbH remains unaffected. Liability is excluded for consequential damages, which are not part of the assurance given above.
- 17. Jaeger Messtechnik GmbH does not assume liability for damages caused by viruses, which are passed on by the data medium. The Licensee is hold responsible for checking the data medium for viruses, before installing the *ADbasic* software on his computer.

18.FINAL CONDITIONS



The invalidity of some individual conditions does not affect the validity of the License Agreement.

In addition to the conditions of this License Agreement, the General Terms and Conditions of Jaeger Messtechnik GmbH apply.

1.

A.4 Command Line Calling

The *ADbasic* compiler cannot only be activated through the user interface, but it can also be directly called in Windows or DOS (with a so-called "command line call"). The compiler works the same in both cases; it can compile a source code file and generate a binary or library file.

The compiler will only be called after you have entered your license key in *ADbasic*.



The command line call has changed since *ADbasic 4*. Thus, you have to check the syntax of previously written calls.



Please note the general hints about Command line calls in Windows on page 15.

A.4.1 Syntax

There are command line calls to create binary files (main option /M) and to create a library file (main option /L).

You add command line options, beginning with a slash /, some of which have optional parameters. If an option is missing, the compiler will use a default setting; nevertheless, we recommend typing all options to avoid ambiguities¹.

As an alternative, options of a single call may be written into a makefile and the compiler called with main option /MAKE.

At last there are the main options $/ {\tt H}$ to display a short help text, and $/ {\tt VER}$ to display the compiler version number.

The command line call is entered in a single line; option letters are case sensitive.

1. As an example, a call with all options given remains correct, even when a default setting is being changed.



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"

ADbasic /H

ADbasic /VER
```

Please note: With processor type T12 (option /P12), you have to write ADbasic_C instead of ADbasicCompiler.

Optional settings are given in brackets []. The character | separates options, which are mutually exclusive.

File names can be written without, with relative or with absolute path names. The base directory for a file name without or with relative path name is the working directory, from which the command line is called.



Main Options

/M Generate a binary file with the extension . Txn.

x Processor type; see option /Px.

n Process number; see option / PNx.

/L Generate a library file with the extension .LIx.

x Processor type; see option /Px.

/MAKE Read main option, file name and other options of a

single call from the makefile.

The text in the makefile may be written using several lines. Options outside the makefile are not

permitted

/H Display a short help text.

/VER Display compiler version number.

Options

src.bas File name of the source code to be compiled; type

with suffix .bas.

Compiler warnings are written into the file src.wrn, error messages into the file src.err.

/A"dest" [Path and] name of the binary or library file <dest>,

which is to be generated, without suffix. The default

is the file name src.

The file suffix . Txn (binary file) or . LIx (library file)

is attached automatically.

/IP"path" Directory, where include files are searched.

This setting overwrites the ADbasic standard direc-

tory and should thus be used with caution.

/LP"path" Directory, where library files are searched.

This setting overwrites the ADbasic standard direc-

tory and should thus be used with caution.



/Lx	Languag	ge for warnings and error messages.
	/LE	English. Default.
	/LG	German
/Sx	Hardware, for which the file is compiled:	
	/SC	Cards
	/SL	Light-16
	/SG /SGII	Gold; Default Gold II
	/SD	Pro
	/SPII	Pro II
/Px	Process	or type, for which the file is compiled:
	/P2	Processor T2
	/P4	Processor T4
	/P5	
	/P8 /P9	Processor T9; Default
	/P10	Processor T10
	/P11	Processor T11
/ET		imer-triggered process, see also chapter 6 143. Default.
	Exclude	S/EE.
/EE		externally triggered process, see also chappage 143.
	Exclude	S /ET.
/PNx	Number	\times (110) of the process. Default: 1.
/PH	Create process with high priority. Default. See also chapter 6.1.2 on page 145.	
/PL		process with low priority and priority level 1 ggered process only). See also chapter
	•	page 145.
/PLx		process with low priority and priority level \mathbf{x}
	(-1010	
/PDx	•	e time (Processdelay) of the process to x. 1000, T11: 3000. See also chapter 6.2.1 on 8.



/Ox	Set optimize level x (0, 1, 2) of the compile also Process Options dialog box (page 56).	er, see
	/00 Optimize level 0 (=don't optimize)	
	/01 Optimize level 1 (Default)	
	/O2 Optimize level 2	
/Vx	Set process version x , see Process Options box (page 56). Default: 1.	dialog

A.4.2 Notes

The order of options is arbitrary. Command line calls are case sensitive.

If option /A is not used, the generated binary or library file is saved in the same directory, as the source code.

If warnings or errors occur during compilation, they are saved in the files <src.WRN> and <src.ERR>. The error messages are the same as those that *ADbasic* displays in the info window (see chapter 3.11.1 on page 86).

We recommend deleting the files containing the warnings and error messages before compilation, so that you can very easily check if the compilation has proceeded without any errors.

A.4.3 Examples

```
\label{eq:c:ADwinADbasicADbasiccompiler.exe} $$C:\ADwin\ADbasic\ADbasiccompiler.exe /L $$Z:\Myfiles\test.bas$
```



This command line compiles the source code <code><test.bas></code> and generates the library file <code><test.LI9></code> in the directory <code><Z:\Myfiles\></code>.

Since nothing else is indicated, the default setting is used:

- save generated file in the directory of the source code file.
- use English warnings and error messages.
- Hardware: ADwin-Gold.
- Processor: T9.
- Optimize level: 1.



If you do the call from the directory $< C : \Delta Dwin \Delta Dbasic>$, you can shorten this line to:

ADbasicCompiler.exe /L Z:\Myfiles\test.bas

The shortest version is when the source code is stored in the directory <C:\ADwin\ADbasic> (here without file name extension):

ADbasicCompiler /L test.bas

Anyway, we recommend the complete version—at least for automation of the call:

ADbasiccompiler /L test.bas /A"test" /LE /SG /P9 /O1

-\u00fc- ADbasiccompiler /L Z:\Myfiles\String.bas /SP /O1

This command line compiles the source code <string.bas> into a library file for a *Pro* system with processor T9. It is a timer-triggered process with number 1 and high priority.

The same call, for processor T10 only, is as follows:

ADbasiccompiler /L Z:\Myfiles\String.bas /P10 /SL /O1

ADbasicCompiler /M C:\ADwin\ADbasic\samples_ADwin\bas_dmo6f.bas /LE /SG /P9 /ET /PN3 /PH /O1

Compiles the demo file <bas_dmo6f.bas> into a binary file for a Gold system with T9 processor. It is a timer-triggered process with number 3 and high priority.

ADbasiccompiler /M C:\ADwin\ADbasic\samples_ADwin\bas_dmo6 /LE /P8 /SL /O1

Compiles the demo file <bas_dmo6.bas> into a binary file for a Light-16 card with processor T8, without optimization. It is a timer-triggered process with number 2 and low priority

- C:\ADwin\ADbasic\ADbasic /M C:\user\my_file.bas /LE /P4 /SC /A"your_file" /O1

This instruction compiles the file <my_file.bas> for an *ADwin*-Card with processor T4. It is an externally triggered process with number 5 and low priority. The generated binary file has the



name <code><your_file.T45></code> and can be found in the same directory where the source code is saved: <code><C:\user></code>.

ADbasicCompiler /M C:\user\my_file.bas /LE /SG /P9 /A"Y:\somewhere\your file" /ET /PN3 /PH /O1



The binary file now is saved as <Y:\somewhere\your_file.T93>; It is a timer-triggered process with number 3 and high priority.

A.4.4 Command line calls in Windows

The term and functionality "command line call" originates from DOS, where commands to the operating system (DOS) had to be entered in command lines. Entering such command lines is still possible under Windows.

There are several ways to enter commands under Windows:

 Open a Command Prompt window (from Windows start menu, directory Programs / Accessories).

The compiler call needs the Windows environment anyway. Thus, the call works only from the Command Prompt window, not from original DOS-mode.



- Select Run in the start menu and enter a command line in the input window.
- For frequently needed command lines, create an icon on the desktop. When you generate an icon, enter the command line directly.

One or more command lines can be combined in one batch file <*.bat>, for example in order to compile several source code files of a project with only one call.

When you call a command line, you have to transfer the relevant options and parameters.



A.5 Obsolete Program Parts

For compatibility reasons, the development environment also offers settings for *ADwin* systems with transputer processors (T4, T5, T8).

Dialog Window Process Options

In this dialog window, you set compiler options for the currently open source code window that is you set the properties of the process, which is compiled from the current source code and transferred to the *ADwin* system.

You must make the necessary settings separately for each of the source code windows by opening the dialog window again (unless you want to use the default settings).

If you have set the processor types T4, T5 or T8 in the dialog window Compiler Options, the dialog window shown in fig. 1 is opened.



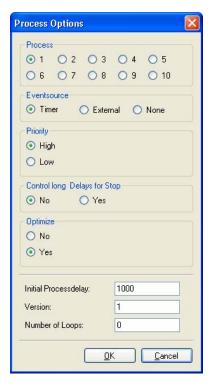


Fig. 1 – The Dialog Window Process Options for processors T4 ...
T8

Event: Here you set, which event signal is to start the section
 EVENT: of your process.

With the setting Timer, you define the number of counts of the internal counter as the event signal. In this case, you use the system variable PROCESSDELAY to define the time interval triggering an event signal.

With Extern, you determine that a signal at the event input of your *ADwin* hardware starts the process. This could be for instance an impulse of a sensor. Such a process must run at high priority. In this case, set the option Priority to High.



How to use an external event input with an *ADwin-Pro* system, is described in the software documentation under **EVENTENA-BLE**.

With the setting None, the process starts immediately after it has been transferred to the system. The section **EVENT:** is—independent of any event signals—it is restarted immediately after the execution (infinite loop).

In a high-priority process, you have to assure that the process also provides computing time for other tasks (e.g. communication with the computer).

- Process: Set the number (1...10), with which the transferred process is accessed on the system.
 - If several processes are running simultaneously on the *ADwin* system, you must assign a separate number to each of the processes.
- Number of Loops: If you like, you can set here the number of times the program cycles through the event loop before it stops.
 When this number is reached, the process stops automatically.
 A setting you have changed will be active upon the next start of the process (not in the currently running process); you need not recompile your program.

If you enter the value "0", the program is repeated until you stop the process with:

- the instruction END.
- the instruction STOP PROCESS or
- the stop icon in the development environment.
- Version: Here you enter an integer value, in order to differentiate between different versions of your program.
- Priority: Set here the priority of the process. You will find more information about this subject in chapter 6.1 "Process Management". The setting Level does not exist for the transputer processor type.
- Control long Delays for Stop: This setting is only available when you use the processors T2 ... T8.



The stopping of a process is delayed, if it is not called frequently (cycle time interval > 5 milliseconds). We recommend using the option in this case, because this option will speed up the stop procedure.

- Optimize: The optional optimization shortens the process execution time of up to 20 percent. A higher setting under Level leads to shorter execution times.
 - If unexpected compiler or run-time errors occur, you can sometimes avoid them by setting a lower Level for the optimization.
- Delay: Set here the processdelay (cycle time), before the process is to begin.



A.6 List of Debug Error messages

The following error messages can be displayed, if the option Debug mode is enabled in *ADbasic*; see Debug mode Option, page 67.

Run-time error

Division by zero

SQRT from negative number

Datan: Index is too large / Datan: Index is less than 1 Array index is too large / Array index is less than 1

Access to local or global array elements, which are not declared, i.e. with indices that are too large or too small.

A trailing (inc) in the error message says that an instruction of a system include file was accessed wrongly (as described above).

Fifo index is no fifo

The array with the given index is not declared as FIFO or not declared at all.

Address of Pro II module is >15 or <1

P2_Burst_xxx¹: "startadr" is not divisable by 4

P2_Burst_xxx¹: Number of values is not divisable by 4

P2_Burst_Init¹: Number of values is not divisable by 4 / by 8

P2_Burst_Read_Unpacked1: Number of values is not divisable by 8

P2_Burst_Read_Unpacked2: Number of values is not divisable by 4

P2_Burst_Read_Unpacked8: Number of values is not divisable by 2

P2_Burst_Read: Number of values smaller than 1 / than 4



Run-time error

P2_Burst_(C)Read_xxx: The number of channels being set by P2_Burst_Init does not fit to this read command.

Use a command, which reads the data of the set number of channels.

P2 GetData/SetData Long: TiCo DATA does not exist

P2_GetData/SetData_Long: TiCo DATA has wrong datatype

P2_GetData/SetData_Long: TiCo DATA index too large

P2_GetData/SetData_Long: TiCo DATA index < 1

P2 Digout Fifo Write: timestamp difference < 2

P2_Get/Set_TiCo_Ringbuffer: TiCo Ringbuffer is declared as READ and WRITE

P2_TDRV_Init: TiCo is not present, no TiCo prg. loaded or invalid information in TiCo memory

Media Read / Media Write:

start_block + count_blocks128 > num_blocks
start block < 0</pre>

Access to an invalid range of the storage media, with a block number that is too large or too small

P2 Seq Read:

An odd number of read measurement values is not allowed

^{1.} Valid for P2_Burst_Init, P2_Burst_Read, P2_Burst_Write



A.7 Index	ActiveX · 155
	communication to ADwin
Symbols	system · 154
-· 162	use from a development
# · 167	environment · 155
	ADbasic
#Begin_Debug_Mode_Disable · 18	license key · 10
	ADbasic 5: changes · 7
#Define · 193	ADconfig · 155
#Else · 223	add file/folder shortcut · 44
#End_Debug_Mode_Disable · 200	Add Open Files to Project · 74
#Endlf · 223	Add to Project
#If · 223	context menu · 18
#Include · 228	project window · 74
* · 163	addition · 159
+ · 159	additional memory (EM) · 112
+ (String) · 160	ADsim
.NET · 155	Alternative to ADbasic · 5
/· 164	communication
: 168	with Simulink model · 155
<=>· 170	ADtools
= · 169	overview · 95
^ · 165	set bar · 64
'(Rem) · 273	ADWIN CARD · 223
	ADWIN GOLD · 223
Numerics	ADWIN_GOLDII · 223
150h, see device no.	ADWIN_L16 · 223
2-dimensional arrays · 114	ADWIN PRO · 223
40 bit accuracy · 103	ADWIN PROII · 223
64 bit accuracy (float) · 103	ADWIN SYSTEM · 223
	ADwin32.dll · 154
Α	ADwin64.dll · 154
Abs · 172	Alternative to ADbasic · 5
AbsF · 173	analyze
Absl : 174	general · 137
absolute value	run-time error · 137
all data types · 172	timing · 137
floating point number · 173	And · 175
integer number · 174	arc cosine: ArcCos · 177
integer number 1/4	arc sine: ArcSin · 178
	arc tangent: ArcTan · 179
	• • • • • • • • • • • • • • • • • • •



arithmetic functions	bar
- · 162	ADtools · 95
* · 163	editor · 19
+ · 159	menu · 49
/· 164	status bar · 80
^ · 165	base e · 203
Dec · 192	Begin_Debug_Mode_Disable see
Exp · 203	#Begin_Debug_Mode_Disable
Inc · 227	
LN · 243	binary file
Log · 246	see also library
Sqrt · 285	create · 52
Array-Index (local) too large / <1,	from command line · 9
see run-time error	from TiCoBasic · 53
arrays	use from development
2-dimensional · 114	environment · 155
allocate memory area · 110	binary notation · 105
copy · 253	bit pattern of floating-point
Data_n · 190	numbers · 105
FIFO · 204	bit shifting
global · 107	left · 280
first element · 108	right · 281
initialize · 100	bookmark · 38
local · 109	booting · 11
first element · 110	break, see stop process
overview · 101	BTL file
(Dim) AS · 195	directory settings · 63
Asc · 180	busy display · 80
ASCII-character set · 4	bypass waiting time · 255
assign a value · 105	
assignment (=) · 169	C
(Dim) AT · 195	C#.NET, C++ · 155
autocomplete, instruction or	Cacheable, memory · 113
variable · 40	call graph · 82
autoindent · 60	call tree
automatic type conversion · 123	create info file · 70
autoSave · 52	show · 70
autostart · 52	carriage return (escape
	sequence) · 120
В	case sensitivity · 16
backslash (escape sequence) · 120	Case, CCase, CaseElse (Select-
the formal sections of the section o	-, , (- 5)



Case) · 277	compiler
Cast_Float32ToLong 184	autoSave · 52
Cast_FloatToLong · 182	call · 52
Cast_LongToFloat · 183	command line call · 9
Cast LongToFloat32 · 185	compiler message, error /
change license key · 10	status · 86
check	prebuild, postbuild · 66
number and priority of	set options · 54
processes · 138	store project options · 64
Chr · 186	compiler instructions
clear parameter scan · 44 CM	#Begin_Debug_Mode_Disable 181
Dim · 195	#Define · 193
CM, see Cacheable	#End_Debug_Mode_Disable · 20
code size · 86	0
code snippets · 41	#If Then · 223
color settings · 61	#Include · 228
command line	overview · 167
call · 9	conditional jump
line length	If Then · 221
standard · 97	SelectCase · 277
upper case / lower case · 97	constant · 100
comment	context menu
comment Block · 23	project window · 74
position · 23	source code window · 18
see remarks	control block
communication	context menu · 18
between processes · 153	mark · 37
process in the ADwin	control characters In strings · 119
system · 146	control structures
time-Out · 146	overview · 124
with a development	toggle folding · 27
environment · 155	cosine: Cos · 187
with Simulink model · 155	counter
with the PC · 154	internal, clock cycle · 148
compare macros to libraries · 127	read · 270
comparison	CPU_Sleep · 188
< = > · 170	cursor position · 80
strings · 293	cut off decimal places · 122
	cycle time · 148



D	debug
data exchange	general · 137
between processes · 153	debug mode · 137
with Simulink model · 155	enable timing mode · 67
with the development	menu · 67
environment · 155	timing mode · 137
with the PC · 154	timing window · 88
data loss	debug errors · 67
FIFO · 116	debug mode · 67
from booting · 11	Dec · 192
•	decimal logarithm · 246
data memory	decimal notation · 105
see also memory	decimal places, cut off · 122
2-dim. arrays in ~ · 115	decimal places, float values · 63
additional demand by	decimal separator · 105
debug mode · 137	declaration
timing mode · 140	display all · 93
allocate · 110	jump to · 38
overview, internal, external · 111	see dimensioning
data structures	show all · 43
FIFO · 115	show single info · 43
global arrays · 107	decrement · 192
memory fragmentation · 146	Define, see #Define
global arrays, 2-dimensional · 114	defining foldable text range · 26
global variables · 106	definition of macros, position in the
local variables and arrays · 109	program · 100
overview · 101	Delphi · 155
data types	demo mode · 10
overview · 102	design of an TiCoBasic
string · 117	program · 97
type conversion · 122	Deutsch · 63
data word, numbering of bits · 2	development environment
Data_n · 107	bars and windows · 12
dimensioning · 195	communication with C, Delphi,
global arrays, 2-dimensional · 114	Matlab etc. · 155
overview · 190	directory settings · 63
Data-Index (global) too large / <1,	short-cuts · 1
see run-time error	source directory · 12
	start · 9



device no.	E
definition · 155	editor
set · 55	bar · 19
DIAdem · 155	general · 60
Dim · 195	print settings · 63
dimensioning	syntax colors · 61
instruction Dim · 195	e-function Exp · 203
memory area · 110	Else (If Then) · 221
position in the program · 100	EM, see memory
directory	EM Local
settings · 63	 Dim · 195
with standard installation · 12	Event · 201
display	Finish · 210
all declarations · 43	Init · 230
current information · 15	LowInit · 247
declarations · 93	End · 199
memory usage T12: CPU, CM,	End_Debug_Mode_Disable see
UM · 80	#End_Debug_Mode_Disable
memory usage: CPU, PM, EM,	EndFunction · 217
DM, DX · 80	EndIf (If Then) · 221
passed parameters · 42	EndSelect (SelectCase) · 277
single declaration info · 43	EndSub · 301
syntax highlighting · 21	enter license key · 10
division	equal to = · 170
by 2 · 281	
simple · 164	error
Division by zero, see run-time error	see also run-time error
DM, see memory	data loss with FIFO · 116
DM_LOCAL	forced by Cut&Paste · 51
Dim · 195	process overwritten · 145
Do Until · 198	run-time · 67
documenting self-defined	time-Out · 146
instructions/variables · 24	try lower optimization level · 58
DRAM_Extern	error message, compiler · 86
Dim : 195	escape sequence · 119
Event · 201	Ethernet · 155
Finish · 210	evaluate
Init · 230	operators · 121
LowInit · 247	
DX. see memory	



event	FIFO
external signal · 143	check number of elements · 117
external signal: reset · 274	data loss · 116
lost event signals: check · 91	design of data structure · 115
lost signal	dimensioning · 195
externally controlled	initialize · 206
process · 153	overview · 204
several time-controlled	query empty elements · 208
processes · 152	query full elements · 209
single time-controlled	FIFO_Clear · 206
process · 152	FIFO_Empty · 208
measure time difference · 130	FIFO_Full · 209
set signal source · 58	file name
Event, program section · 201	binary file · 53
exclusive Or operation · 310	library · 53
Exit · 202	file, add shortcut · 44
exponential function: Exp · 203	find
exponential notation · 105	declaration of
expressions	instruction/variable · 38
evaluate · 121	examples · 33
separate evaluation · 123	regular expressions · 35
symbolic names · 100	text · 30
extensive initialization · 99	text quickly · 29
external data memory (DX) · 112	Finish: · 210
external event signal · 143	Flo40ToStr · 213
external memory	floating-point numbers
T12 · 113	bit pattern · 105
external memory (SDRAM) · 111	decimal notation · 105
	decimal places in parameter
F	window · 63
F1: call help · 16	exponential notation · 105
fast	value range · 103
optimization level · 58	FloToStr · 211
•	fold text ranges · 27
	foldable text range, define · 26
	folder, add shortcut · 44
	font settings · 61
	For Next · 215
	formatting, smart · 23
	FPar_n · 106
	fragmentation, memory · 146



Function · 217	Inc · 227
function	Include · 228
documenting · 24	include
general features · 125	directory settings · 63
library	include a file: #Include · 228
definition · 234	include a library: Import · 225
general · 126	include-file, general · 125
macro · 217	increment · 227
position in the program · 100	indent
valid characters · 100	comments · 23
	lines · 23
G	TiCoBasic sections · 60
german · 63	#INF (infinity, floating-point
global arrays, see arrays, global	numbers) · 105
global variables, see variables, glo-	info range · 86
bal	info window · 86
global variables, window · 92	Init: • 230
Globaldelay · 265	initialization, boot · 11
goto line · 38	initialize · 99
greater than >, >= · 170	input license key · 10
	insert code snippets · 41
Н	installation, standard directory · 12
halt, <i>see</i> stop process	instruction
hardware access	autocomplete · 40
read · 263	declaration info · 43
write · 264	display passed parameters · 42
header, print · 63	documenting self-defined ~ · 24
help	jump to declaration · 38
call selected · 16	measure processing time · 129 separator (:) · 168
F1 · 16	instruction reference · 157
stay on top · 64	Integer (data type) · 104
hexadecimal notation · 105	integer numbers
	binary notation · 105
	hexadecimal notation · 105
IEEE floating-point format · 103	type conversion · 122
If · 221	value range · 102
see also #If · 223	internal counter
Import · 225	clock cycle · 148
import 223	2.22 0,0.0



internal memory	library
additional (EM) · 112	create
data (DM) · 112	from command line · 9
SRAM · 111	from TiCoBasic · 53
T12 · 113	directory settings · 63
interrupt, see stop process	function · 234
IO_Sleep · 232	general · 126
	Import · 225
J	position in the program · 100
Java · 155	subroutine · 239
Jump forward/backward · 39	toggle folding · 27
jump to declaration · 38	valid characters · 100
jump to program line · 38	versus macros · 127
jump, conditional	library file
If Then · 221	create · 52
SelectCase · 277	license agreement · 5
277	license key · 10
Κ	line comment, indent · 23
	line feed (escape sequence) · 120
Kallisté · 155	line length, max.
keyboard, settings display · 80	standard · 97
	lines
L	change to comment · 23
language · 63	indenting · 23
latency (timing window) · 89	jump to · 38
layout, print · 63	numbering · 60
length (timing window) · 89	smart format · 23
less than <, <= · 170	LN · 243
Lib_EndFunction · 234	LngToStr · 244
Lib_EndSub · 239	Log · 246
Lib_Function · 234	logarithm
Lib_function	decimal · 246
documenting · 24	natural · 243
Lib_Sub · 239	logic functions
Lib_sub	And · 175
documenting · 24	Not · 256
	Or · 257
	Shift_Left · 280
	Shift_Right · 281
	XOr · 310
	long, see integer numbers



memory
see also data memory
additional demand by
debug mode · 137
timing mode · 140
allocate · 110
areas (PM, DM, EM, DX) · 111
areas CM, UM · 113
calculate need of · 86
fragmentation · 146
string · 118
T12 · 113
workload · 80
menu
bar · 49
build · 52
debug · 67
edit · 51
file · 50
help · 72
options · 54
select · 13
tools · 71
view · 51
window · 72
Min Float · 250
Min_Long · 252
minimum
float values · 250
integer values · 252
multiplication
by 2 · 280
simple · 163
5 p.:0
N
names
arrays and local variables · 109 macros, libraries · 100
•
natural logarithm · 243
negative sign · 122
news in ADbasic 6 · 7



Next (For) · 215	options setting
NOP · 255	ADtools · 64
Not · 256	compiler · 54
not equal to <> · 170	directory · 63
notation of numbers · 105	editor · 60
notes, see remarks	general · 60, 63
number of processes, check · 138	help · 64
number, see device no.	language · 63
numerical values, notation · 105	print · 63
	process · 56
0	project · 64
operating system	description · 65
directory settings · 63	general · 65
load, see booting	prebuild, postbuild · 66
operators	syntax colors · 61
And : 175	Or · 257
evaluate · 121	outdent lines · 23
negative sign · 122	overload of processor · 151
Or · 257	
priority · 121	Р
XOr · 310	P1 Sleep · 259
optimal timing	P2_Sleep · 261
one process · 139	Par n · 106
several processes · 138	parameter scan · 44
optimize	parameter window · 76
calculate polynoms quickly · 165	decimal places, FPar · 63
constants instead of	parameters, see variables, global
variables · 132	parse and indent · 60
general · 129	passed parameters, display · 42
measure faster · 132	Peek · 263
measure processing time · 129	PM, see memory
register access · 131	PM_Local
run-time error · 137	Event · 201
setting waiting time · 132	Finish · 210
T11 memory access · 136	Init · 230
timing · 137	LowInit · 247
use waiting times · 135	Poke · 264
	polynoms, calculate quickly · 165
	position, comments · 23
	postbuild · 66



power · 165	process
base e · 203	autostart · 52
replace in polynom · 165	check number and priority · 138
prebuild · 66	communication · 153
pre-processor instructions	communication process · 146
#Begin_Debug_Mode_Disable ·	load anew · 146
181	memory use · 146
#Define · 193	number · 144
#End_Debug_Mode_Disable · 20	operating modes for timing · 152
0	optimal timing, one process · 139
#If Then · 223	optimal timing, several
#Include · 228	processes · 138
pre-processor, see compiler	options, show · 13
instructions · 167	priority
print settings · 63	communication · 146
priority	high · 145
low-priority processes with T11 /	low · 145
T12 · 150	low with T11 / T12 · 150
of processes, check · 138	overview · 144
operators · 121	processing time · 149
process, see process, priority	query status · 269
problems	read out error · 268
slow editor · 60	setting options · 56
	several · 149
	standard processes 11, 12 · 144
	start
	delayed · 287
	other process · 286
	stop, see stop process
	time characteristic · 148
	process control
	End · 199
	Exit · 202
	Process_Error · 268
	ProcessN_Running · 269
	Reset_Event · 274
	Restart_Process · 275
	Start_Process · 286
	Start_Process_Delayed · 287
	Stop_Process · 289



process cycle	program structure
call	overview · 124
by event · 143	display · 82
time interval · 148	include-file 125
precise timing · 149	library · 126
process optimization, see optimize	module (macro) · 125
Process_Error · 268	toggle folding · 27
Processdelay · 148	project
syntax · 265	compile all source code files · 53
time resolutions · 148	general · 46
Processn_Running · 269	highlight used parameters · 44
Processor · 223	options setting
program architecture	description · 65
jump	general · 65
If Then · 221	prebuild, postbuild · 66
SelectCase · 277	window · 74
library	Prozessn_Running · 269
function · 234	
Lib_Sub · 239	Q
loop	QNaN (floating-point
Do Until · 198	numbers) · 105
For Next · 215	Hamberey 103
modules	R
function · 217	
subroutine Sub · 301	Read_Timer · 270
remarks Rem · 273	Read_Timer_Sync · 272
program design · 97	redo · 19
program improvement, see optimize	register access · 131
program line, jump to · 38	regular expressions · 35
program memory · 112	Rem · 273
additional demand by	remarks · 273
debug mode · 137	replace
timing mode · 140	examples · 34
program section	regular expressions · 35
Event: · 99	text · 30
Finish: · 99	Reset_Event · 274
Init: · 99	Restart_Process · 275
LowInit: · 99	ring buffer · 115
overview · 99	root · 285
	Round · 276



run-time error	Simulink			
see also debug mode	Alternative to ADbasic · 5			
display · 67	communication			
find · 137	with model · 155			
	sine: Sin · 282			
S	size			
Save All Files of Project · 74	optimization level · 58			
SDRAM, see memory	Sleep · 283			
search · 30	see also P1_Sleep			
declaration of	smart format · 23			
instruction/variable · 38	SNaN (floating-point			
	numbers) · 105			
examples · 33	snippets · 41			
regular expressions · 35 SelectCase · 277	source code			
self-defined instructions/variables	change tab order · 16			
documenting · 24	creating · 16			
separator: 168	editor window · 80			
settings	formatting · 20, 21			
ADtools · 64	information · 13			
compiler · 54	structured display · 21			
directory · 63	to do's · 88			
editor · 60	use in a project · 74			
general · 60	source code status bar · 13			
help: 64	special char, find · 35			
print · 63	springen			
process · 56	anderes Sprungziel · 39			
syntax colors · 61	Sprungziel, wechseln zwischen · 39			
Shift Left · 280	Sqrt (square root) · 285			
Shift_Right · 281	Sqrt from negative value, see run-			
(bit) shifting	time error			
left · 280	SRAM, see memory			
right · 281	stack size			
Short · 104	until T11 · 86			
short-cuts · 1	Stack Test			
show	Analysis · 84			
declarations of a file · 43	display · 70			
declarations window · 93	Stack test			
line numbers · 60	set option · 69			
process options window · 13	Start_Process · 286			
F. 11330 Option	Start_Process_Delayed · 287			
	starting TiCoBasic · 9			



status bar · 80	T12 · 118
status bar of source code	StrLeft · 294
window · 13	StrLen · 296
status message, compiler · 86	StrMid · 297
stay on top, help window · 64	StrRight · 299
Step (For) · 215	structure
stop process	Coloured display of source
itself	code · 21
in Event: · 199	indent lines · 23
in LowInit:, Init:, Finish: · 202	program sections · 124
others · 289	toggle folding · 27
Stop_Process · 289	Sub · 301
StrComp · 293	subroutine
string	documenting · 24
assign values normally · 118	general features · 125
assignment not being	library
recommended · 120	definition (Lib_Sub) · 239
control character · 119	general · 126
definition of data type · 104	macro · 301
escape sequence · 119	position in the program · 100
Unicode · 104	valid characters · 100
variable structure · 118	subtraction · 162
String instruction	switch to ADbasic 6 · 7
addition · 160	symbolic names · 100
ASCII value into char · 186	syntax
char into ASCII value · 180	colors · 61
comparison · 293	highlighting · 21
dimensioning · 291	system variable
Float to string · 211	Globaldelay see
Float to string (40 bit) · 213	Processdelay · 265
length of a string · 296	overview · 108
Long to string · 244	Process_Error · 268
partial string	Processdelay · 265
left · 294	ProcessN_Running · 269
midst · 297	
right · 299	T
String to Float · 306	T11
String to long · 308	low-priority processes · 150
syntax · 291	setting waiting time · 133
string, element 1 forbidden with	



T12	time saving
clock cycle · 148	constants instead of
float values · 103	variables · 132
low-priority processes · 150	measure faster · 132
measure processing time · 130	register access · 131
memory areas · 113	setting waiting time · 132
query version number · 305	use waiting times · 135
setting waiting time · 133	timer event · 143
Stack size · 59	timer, see counter
string, element 1 · 118	timing
tab	see optimize
escape sequence · 120	changed by
size · 60	debug mode · 137
tab order, change · 16	timing mode · 140
tangent: Tan · 304	operating modes
TCP/IP	externally controlled
see Ethernet	process · 153
terminate, see stop process	general 152
Testpoint · 155	several time-controlled
text	processes · 152
define foldable range · 26	single time-controlled
find And replace · 30	process · 152
find quickly · 29	optimal, several processes · 138
fold ranges · 27	optimal, with one process · 139
indenting · 23	optimize · 137
smart format · 23	query information 139
Then (If Then) · 221	timing mode
TiCoBasic	additional processor time · 140
demo mode · 10	enable · 67
license agreement · 5	use · 137
license key · 10	window · 88
start · 9	To (For) · 215
TiCoBasicCompiler, command	to do list · 88
line · 9	toggle folding · 27
time	tool bar · 13
cycle time · 148	toolbox · 73
precise cycle timing · 149	
time-Out · 146	



tools	Until (Do) · 198
TBin · 95	upper / lower case letters · 16
TButton · 95	USB · 155
TDigit · 95	user defined instructions and
TFifo · 95	variables · 99
TGraph · 95	user surface · 12
TLed · 95	User_Version · 305
TMeter · 95	utility programs, see ADtools
TPar_FPar · 95	
TPoti · 95	V
TProcess · 95	ValF · 306
transputer settings · 16	Vall : 308
trigonometric functions	valid characters
ArcCos · 177	macros, libraries · 100
ArcSin · 178	variables · 109
ArcTan · 179	value range · 102
Cos · 187	value range 102
Sin · 282	variables
Tan · 304	autocomplete · 40
type conversion	declaration info · 43
ASCII value into char · 186	display · 76
automatical · 122	global · 106
Float to Long (data type	copy a great number of · 253
only) · 182	highlight used · 44
Float to Long (only data	name · 101
type) · 183, 185	initialization by booting · 11
Float to string · 211	initialize · 100
Float to string (40 bit) · 213	jump to declaration · 38
Float32 to Long (data type	local · 109
only) · 184	allocate memory area · 110
Long to string · 244	name length · 109
String to Float · 306	overview · 101
String to long · 308	switch hex/decimal display · 77
	symbolic names · 100
U	valid characters · 100, 109
UM, see Uncacheable	see also system variable
Uncacheable, memory · 113	variables, documenting · 24
uncomment Block · 23	Verlauf der Sprungziele · 39
undo · 19	version management
Unicode · 104	query version number · 305
unmark Control block · 37	



view	window
to do list · 88	call graph · 82
Visual Basic · 155	compiler options · 54
	debug errors · 67
W	declarations · 93
wait	global variables · 92
IO_Sleep · 232	info range · 86
NOP · 255	info window · 86
P1 Sleep: Pro I-Bus · 259	overview · 12
P2_Sleep: Pro II-Bus · 261	parameter · 76
Processor T11: CPU_Sleep · 188	process Options · 56
setting waiting time exactly · 132	project · 74
Sleep: processors until T10 · 283	project options · 64
205	description · 65
	general · 65
	prebuild, postbuild · 66
	source code · 80
	source code information · 13
	source code status bar · 13
	Stack Test Analysis · 84
	status bar · 80
	timing analyzer · 88
	to do list · 88
	toolbox · 73
	work load
	100%, memory
	fragmentation · 146
	workload
	definition · 151
	display · 80
	influence of number of
	processes · 138
	workspace size · 86
	X
	XOr · 310

Symbols		Do Until	189	Р	
< = > (comparison)	162	E-F		P1_Sleep	249
+ (addition)	151	End	190	P2 Sleep	251
+ (String addition)	152	Event:	192	Peek	253
- (subtraction)	154	Exit	193	Poke	254
* (multiplication)	155	Exp	194	Processdelay	255
/ (division)	156	FIFO	195	Processn_Running	259
^ (power)	157	FIFO_Clear	197	Process Error	258
= (assignment)	161	FIFO_Empty	199	R	
: colon	160	FIFO Full	200	Read_Timer	260
" " (String)	281	Finish:	201	Read_Timer_Sync	262
#Begin_Debug_Mode)	Flo40ToStr	204	Rem	263
Disable	_ 172	FloToStr	202	Reset_Event	264
#Define	184	For To {Step		Restart Process	265
#End_Debug_Mode_	Dis-	Next	·	Round	266
able	191	Function EndFunc			200
#If Then {#Else		208	lion	S	
#EndIf	213			SelectCase	267
#Include	218	G-J		Shift_Left	270
#, compiler stateme		If Then {Else		Shift_Right	271
159		dlf	211	Sin	272
A-B		Import	215	Sleep	273
	400	Inc	217	Sqrt	275
Abs	163	Init:	220	Start_Process	276
AbsF	164	IO_Sleep	222	Stop_Process	279
Absl	165	K-L		" " (String)	281
And	166	Lib_Function Lib_	End-	StrComp	283
ArcCos	168	Function	224	StrLeft	284
ArcSin	169	Lib_Sub Lib_EndS	Sub	StrLen	286
ArcTan	170	229		StrMid	287
Asc	171	LN	233	StrRight	289
С		LngToStr	234	Sub EndSub	291
Cast_Float32ToLong	175	Log	236	T-Z	
Cast_FloatToLong	173	LowInit:	237	Tan	294
Cast_LongToFloat	174	M-O		User_Version	295
Cast_LongToFloat32	176	Max_Float	239	ValF	296
Chr	177	Max_Long	239	Vall	298
Cos	178	Min_Float	240	XOr	300
CPU_Sleep	179	Min Long	242		
D		NOP	245		
Data n	181	Not	246		
Dec	183	Or	247		
Dim	186	Ol	247		
		-		•	