

ADbasic

**Real-Time Development Tool for
ADwin Systems**

ADbasic Version 4.01

December 2004

License Key:

ADwin - the fastest real-time systems under Windows

Table of contents

Table of contents.	III
Preface	1
Conventions	3
1 Introduction	5
2 Development Environment	7
2.1 Basic Steps.	7
2.1.1 Start the Development Environment	7
2.1.2 Load the <i>ADwin</i> Operating System	7
2.1.3 Basic Elements of the Development Environment	8
2.2 Working with Source Codes and Projects.	10
2.2.1 Structured Display of Source Code	10
2.2.2 Context Menu in the Source Code Window	12
2.2.3 Managing Projects	13
2.3 Menus and Dialog Boxes	13
2.3.1 File Menu	14
2.3.2 Edit Menu	15
2.3.3 View Menu	15
2.3.4 Build Menu	15
2.3.5 Options Menu	16
2.3.6 Debug Menu	25
2.3.7 Tools Menu	31
2.3.8 Window Menu	32
2.3.9 Help Menu	32
2.3.10 Project Window	33
2.3.11 The Parameter Window	34
2.3.12 The Process Window	35
2.3.13 Info window	36
2.3.14 Status Bar	37
2.4 <i>ADtools</i>	37

3 Programming Processes	39
3.1 Program Design	39
3.1.1 The Program Sections	41
3.1.2 Other Program Parts	41
3.2 Variables and Arrays	42
3.2.1 Overview	42
3.2.2 Data Structures	43
3.2.3 Data Types	44
3.2.4 Entering Numerical Values	44
3.2.5 Global Variables (Parameters)	45
3.2.6 Global Arrays	46
3.2.7 System Variables	47
3.2.8 Local Variables and Arrays	48
3.3 Variables and Arrays – Details	49
3.3.1 Variables and Arrays in the Data Memory	49
3.3.2 Memory Areas	50
3.3.3 2-dimensional Arrays	50
3.3.4 The Data Structure FIFO	52
3.3.5 Strings	53
3.4 Expressions	56
3.4.1 Evaluation of Operators	56
3.4.2 Type Conversion	58
3.5 Decision structures, Loops and Modules	59
3.5.1 Subroutine and Function Macros	60
3.5.2 Include-Files	61
3.5.3 Libraries	61
4 Optimizing Processes	63
4.1 Measuring the Processing Time	63
4.2 Useful Information	64
4.2.1 Accessing Hardware Addresses	64
4.2.2 Constants instead of Variables	64
4.2.3 Faster Measurement Function	65
4.2.4 Using Waiting Times	65
4.3 Debugging and Analysis	66
4.3.1 Finding Run-time Errors (Debug Mode)	67
4.3.2 Check the Timing Characteristics (Timing Mode)	67
4.3.3 Track the Process Flow (Trace Mode)	70

5 Processes in the <i>ADwin</i> Operating System.	73
5.1 Process Management.	74
5.1.1 Types of Processes	74
5.1.2 Processes with High-Priority	75
5.1.3 Processes with Low-Priority	75
5.1.4 Communication Process	76
5.2 Time Characteristics of Processes	76
5.2.1 Processdelay (Globaldelay)	76
5.2.2 Precise Timing of Process Cycles	78
5.2.3 Workload of the <i>ADwin</i> System	78
5.2.4 Different Operating Modes in the Operating System . . .	79
5.3 Communication.	80
5.3.1 Data Exchange between Processes	80
5.3.2 Communication between Computer and <i>ADwin</i> System	81
5.3.3 The Device Number	82
5.3.4 Communication with Development Environments	82
6 Instruction Reference	83
6.1 Instruction Syntax	83
6.2 Instructions for L16, Gold, Pro	84
6.3 <i>ADwin-Gold</i> and <i>ADwin-light-16</i>	191
6.4 <i>ADwin-light-16</i> DIO1 / <i>ADwin-Gold</i> CO1.	221
6.5 <i>ADwin-Gold-CAN</i>	277
7 How to Solve Problems?	319
Appendices.	A-1
A-1 Short-Cuts in <i>ADbasic</i>	A-1
A-2 ASCII-Character Set.	A-3
A-3 License Agreement	A-4
A-4 Command Line Calling Syntax.	A-8
A-5 Obsolete Program Parts.	A-12
A-6 Index	A-19
A-7 Instructions for <i>ADwin-Gold</i>	A-33
A-8 Instructions for <i>ADwin-light-16</i>	A-35
A-9 Instructions for <i>ADwin-Pro</i>	A-37
A-1 Instructions in this manual	A-39

Dear Reader,

ADbasic 4.0 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: introduces you to the basics of programming real-time processes for the *ADwin* system; and act as a reference manual for the *ADbasic 4.0* programming language.

The following, new features in *ADbasic* 4.0 make the development environment more convenient for the user:

New, more clearly-structured user interface, the new project management of source codes and an online-help. Debug functions are a new means for easy trouble shooting.

But most of all the compiler now supports the ADSP T10 processor.

A new section has been added to the manual. The instruction reference now contains commands for the *ADwin-Gold* and *ADwin-light-16* systems which are available when using special include files.

For the *ADwin-Pro* systems, only the instructions in chapter 6.2 apply. All other instructions are described in the "ADwin-Pro System Description, Programming in *ADbasic*" manual.

Chapters 1 and 3 are recommended for first-time users of *ADbasic* 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 2 describes the new development environment and is recommended for all users.

If you would like to provide us with suggestions on how to improve our documentation, don't 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 your *ADwin* systems. 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 in the same way as the default settings of the development environment editor.

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, we first of all would like to shortly explain the 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 3 explains the design of *ADbasic* programs.

An *ADbasic* 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

the running of processes in the operating system.

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 3 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 2).



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 TestPoint, LabVIEW, Diadem, HP-VEE, Intouch and Matlab can be used.

2 Development Environment

Processes for the *ADwin* systems are quickly and easily programmed 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.

2.1 Basic Steps


2.1.1 Start the Development Environment

The *ADbasic* development environment is started by selecting Programs ► *ADwin* ► *ADbasic 4* from the Windows start menu.


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

The *ADwin* system and processor are set 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.

2.1.2 Load the *ADwin* Operating System

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

2.1.3 Basic Elements of the Development Environment

The development environment consists of several bars and windows (see fig. 1); The dimensions of the windows may be individually adjusted.

Context-sensitive help for an element of the development environment (window, icon, menu option), is available when the button  is clicked prior to clicking the desired element.

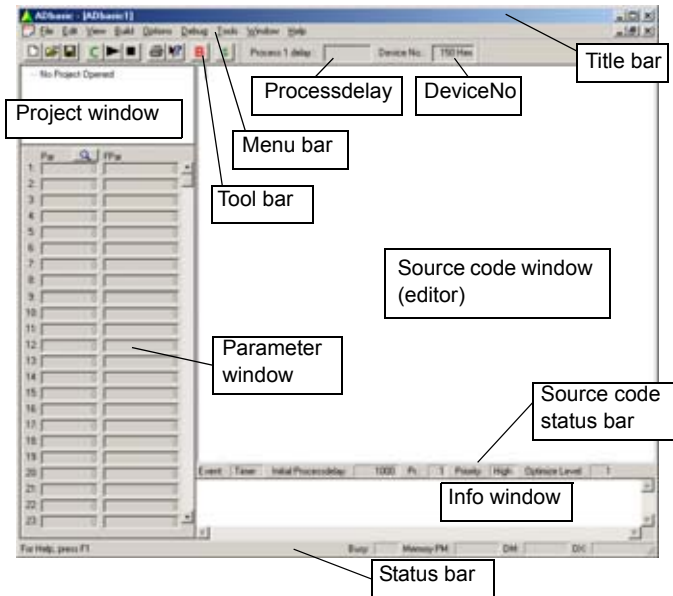


Fig. 1 – Elements of the *ADbasic* development environment

The instructions for the development environment can be found in:

- The tool bar (see fig. 2)
- The context menus of the windows (right mouse button)
- The menu bar (the definition of the instruction appears at the left side of the status bar when a menu instruction is marked).

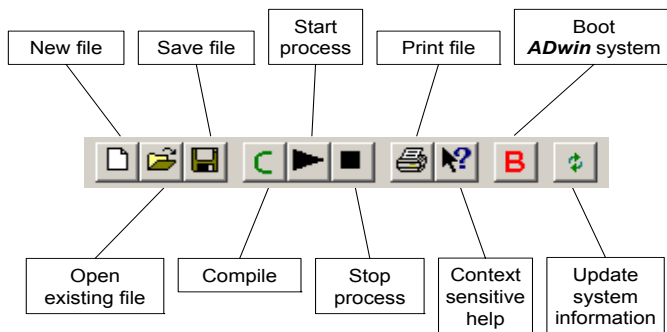



Fig. 2 – The tool bar

An instruction is selected when a menu field 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 windows. Several windows may be opened at the same 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 Window.
- The global parameters used in the source code project are highlighted in the parameter window (s. chapter 2.3.11, page 34) by clicking .
- The info window displays the compiler's error messages (highlighted in red) and warnings (see chapter 2.3.13 "Info window").

The name of an open project and the corresponding source code files are shown in the project window, otherwise it is empty if no project is open.

Some parameters of the *ADwin* system are continuously updated and displayed (only when communication has been established by the computer with the system):

- The `Processdelay` (process cycle time) for the process number of the active open source code window, displayed at the right side of the tool-bar.

- 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.
- Memory usage information, which appears in the status bar (see chapter 2.3.14).

Information about running processes are shown in separate windows:

- Process timing: Timing window (page 25)
- Run-time errors: Debug window (page 30)
- Process flow: Trace window (page 29)

2.2 Working with Source Codes and Projects

A separate source code window must be opened for each process (using **File ▶ New**).

The editor and the compiler do not bother about upper or lower case letters. However, in the examples throughout this manual - for the purpose of better differentiation - upper case letters are used for instructions and global variables and lower case letters for local variables and remarks.

For help with an *ADbasic* instruction, highlight the instruction in the source code and press [F1] to open the online help window with the appropriate information.

Numerical values may be entered in hexadecimal, binary and exponential notation, as well as in decimal (see also chapter 3.2.4).

2.2.1 Structured Display of Source Code

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. This aids in finding text positions, which is useful in longer source codes.

The editor divides the character strings you have entered, into the following syntax categories:

- **Standard**: General program text
- **Comment**: Notes and comments
- **KEYWORD**: *ADbasic* instructions
- **EXTERNAL KEYWORD**: Instructions from include and library files
- **Identifier**: Names of variables and arrays

The color design and the indentation may be changed or completely deactivated. Select the `Syntax Color Sheet` or the `Editor Sheet` from the `Options ► Settings` dialog box.

2.2.2 Context Menu in the Source Code Window

Various help functions are available from the context menu by right-clicking in the source code window (see below).

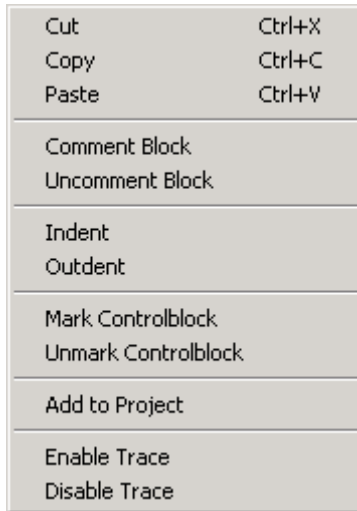


Fig. 3 – Context menu for the source code window

Once the cursor has been set to the specified program line, or the lines themselves have been highlighted, the following instructions of the context menu can be used:

- **Comment Block** inserts a comment character at the beginning of each line making those lines ineffective.
- **Indent** shifts the lines one tab stop to the right, while **Outdent** shifts the lines one tab stop to the left, allowing the source code to become more clearly-structured.
- **Mark Control block** highlights the text of a control structure. **Unmark Control block** removes the highlighting of a control structure that has already been marked. The recognized control structures are as follows:
 - **DO ... UNTIL**
 - **FOR ... TO ... {STEP} ... NEXT**
 - **IF ... THEN ... {ELSE} ... ENDIF**
 - **SELECTCASE**

- `Enable Trace` enables the lines for the Trace Mode, marking them with a question mark "?" (see also chapter 4.3.3 on page 70).
`Disable Trace` disables the lines again and removes the question marks.

2.2.3 Managing Projects

One project can manage many process source codes, for instance an application with several processes. Only one project can be open at a time.

A project allows the user to:

- Include/remove source code files in an open project
- Open all included source code files simultaneously with the saved windows settings
- View all global variables used in the project (see chapter 2.3.11 on page 34)
- Save previously used window settings

Some of these project-related capabilities can be accessed via context menu by right-clicking in the project window (see "Project Window", page 33). All other project instructions can be found in the menu `File`.

Please take into account that opening a project will cause other open source code texts to be closed. If there are unsaved files you are prompted to save these files before closing them.

2.3 Menus and Dialog Boxes

The menu bar contains these menus:

- | | | |
|------------|---|-----------|
| - File: | Manage files and projects | (page 14) |
| - Edit: | Edit source codes | (page 15) |
| - View: | Show windows and bars | (page 15) |
| - Build: | Tool for generating executable programs | (page 15) |
| - Options: | Program settings | (page 16) |
| - Debug: | Tools for error detection | (page 25) |
| - Tools: | Various help functions | (page 31) |
| - Window: | Arrange source code windows | (page 32) |
| - Help: | Help, version and license information | (page 32) |

2.3.1 File Menu

The **File** menu contains instructions for managing files and projects.

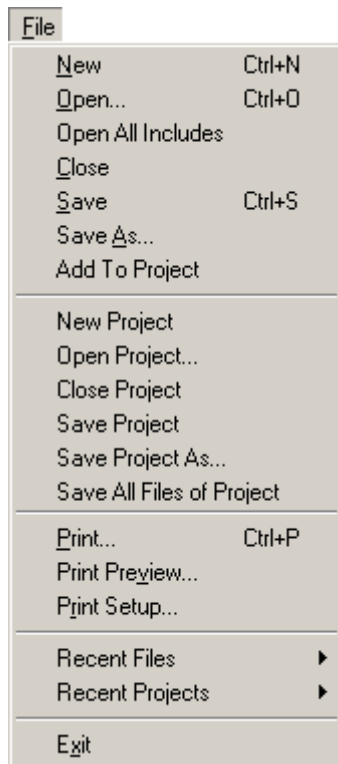
Files can be opened, saved, or new source code windows can be created. Although multiple source code windows may be open simultaneously, no more than ten processes may be loaded to the *ADwin* system at the same time.

The **Open all Includes** menu option opens all files included in the open source code using the **#INCLUDE** instruction.

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 the same time. More instructions are available in the project window (see chapter 2.3.10).

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.

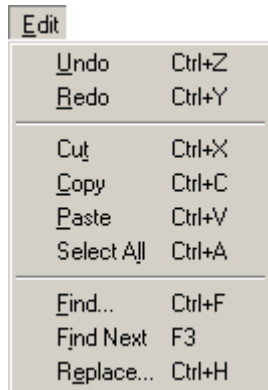


2.3.2 Edit Menu

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

Moreover the menu offers functions for searching and replacing (**Find** and **Replace**).

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.

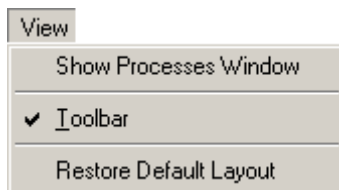


2.3.3 View Menu

In the **View** menu you may open or close

- the process window
- the toolbar.

You find further information about the process window in chapter 2.3.12 on page 35, about the toolbar see fig. 2.

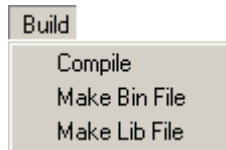


With **Restore Default Layout**, the default layout, which was active at the initial starting of the *ADbasic* program, can be restored with a single mouse-click.


2.3.4 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**



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 process window.

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



A binary file with the extension `<*.TA3>` can be transferred to an *ADwin* system equipped with a T10 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 (see chapter 5.3.4 on page 82).

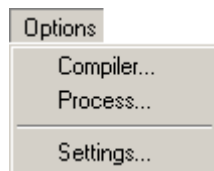
Make Lib File is also available for licensed *ADbasic* users only. It compiles the active source code into a binary file and automatically saves it as library file. The library is stored in the same directory and with the same name as the source code file, but with the file extension `.LIx`. (where *x* denotes the processor type.)

Afterwards the library can be included into other source codes that use their functions and subroutines (see chapter 3.5.1 on page 60).

2.3.5 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 window opens where the settings are entered.

•



Compiler Options Dialog Window

The parameters in this dialog window are used in every source code compilation. In particular they provide information about the *ADwin* system on which 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 window.

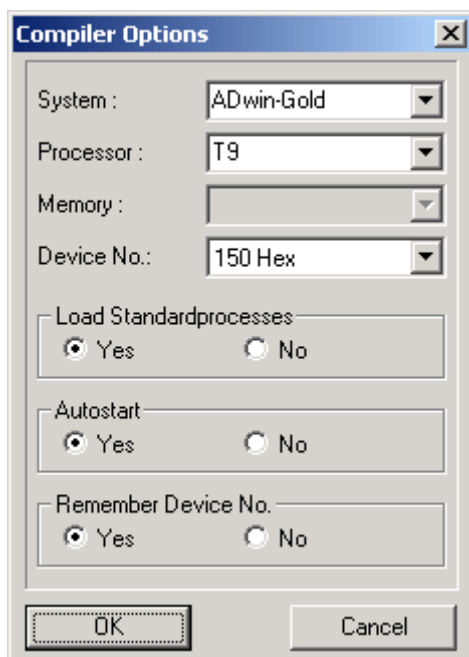


Fig. 4 – The Compiler Options Dialog Window

- **System, Processor:** Select the settings, which correspond to the *ADwin* system.

The abbreviation used under **Processor** for the processor types corresponds to the following full names:

Abbreviation	T10	T9	T8	T5	T4	T2
Full name	ADSP 21160	ADSP 21062	T805	T450	T400	T225

Fig. 5 – Processor Names

- **Memory:** This setting is not applicable for the *ADwin-Gold*, *ADwin-light-16* and *ADwin-Pro* systems with the T9 and T10 processors and therefore is not displayed.


For the old transputer systems, read the information in your hardware manual.

- **Device No.:** Select the device number with which the *ADwin* system can be accessed. The device number is set using the program <ADconfig.exe>. The default setting is 150 Hex.

With the **NONE** setting, source code can be compiled for the configured *ADwin* hardware, if it is not connected to the computer.

- **Load standard processes:** This setting is only available for the *ADwin-Gold*, *ADwin-light-16* systems.

The default setting **Yes** loads the standard processes 11, 12 and 15 (see chapter 5.1.1) into the *ADwin* system during the boot process. Selecting **No** suppresses the loading of processes 11 and 12.

- **Autostart:** Selecting **Yes** causes the binary file, generated and transferred to the *ADwin* system during compilation, to be immediately started. Selecting **No** requires the process to be started by clicking the button  in the toolbar or in the process window.
- **Remember Device No.:** The setting **Yes** saves the last used Device No. (see above) on closing *ADbasic*; the next start-up will automatically use the saved number. The setting **No** makes *ADbasic* start up with the device number **NONE**.

Process Options Dialog Window

This dialog window 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.

Each process must be configured separately by opening the **Process Options** Dialog Window for each source code window, unless using the default settings. To quickly open this window do a right-click on the source code's status bar.

Depending on the processor type set in the **Compiler Options** dialog window, for T9 or T10 ADSP processors, the dialog window shown in fig. 6 is opened. The dialog window for the T4, T5 or T8 processors differs slightly and is described in the Appendix A-5.1.

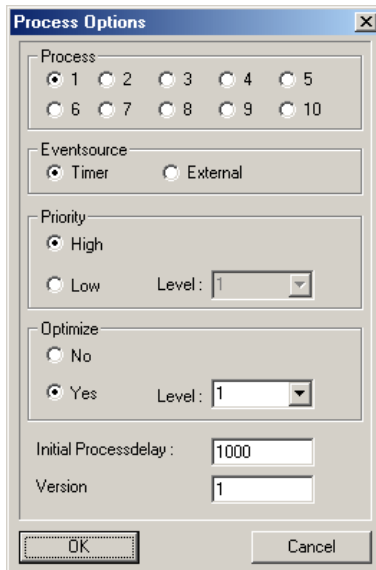


Fig. 6 – The Process Options Dialog Window

- **Process**: The number under which the transferred process should be started on one *ADwin* system, an individual number must be allocated for each process.
- **Eventsource**: The event signal that initiates the **EVENT**: section of the process.

Timer refers to the internal counter as the event signal whose rate is determined by the **GLOBALDELAY** system variable.

External indicates that the event signal is a signal at the event input of the *ADwin* system, for instance a sensor impulse. Such a process will always run with high priority. However the **Priority** option should 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 the instruction **EVENTENABLE**.

- **Priority**: The priority of the process. For more information see chapter 5.1.1 "Types of Processes".

`Level` (0...255) 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`: This 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.

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

- `Initial Processdelay`: The initial `Processdelay` (cycle time) with which the process is to be started.
- `Version`: An integer value for differentiating between several versions of a process.

Settings Dialog Window

The Settings Dialog Window has several sheets, which are activated with the tags in the upper corner.

Editor Sheet

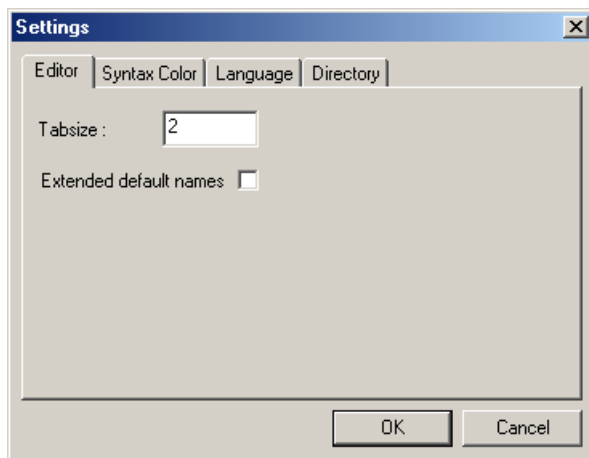


Fig. 7 – The Settings Dialog Window: Editor Sheet

Tabsize indicates the size of the tab stop. Automatic indentation is activated under the Syntax Color Sheet.

The Extended default names option, when set, automatically saves the file of any new source code in the format <ADbYYMMDD_nn.bas>, where YYMMDD is the current date and nn is a two-digit counter number, (nn is set to zero at each change of the system date).

This option helps in allocating an individual file name, when using numerous newly generated source codes.

Syntax Color Sheet

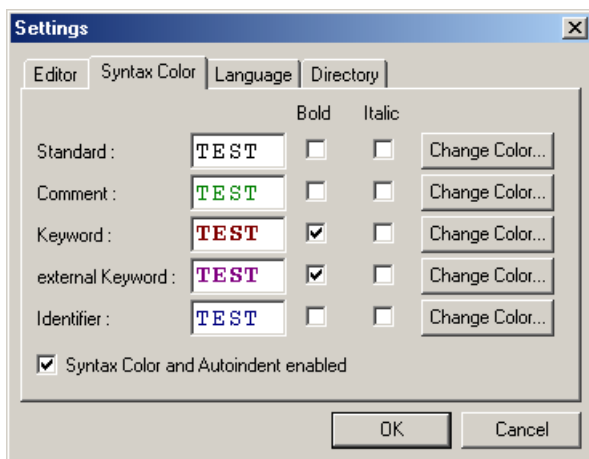


Fig. 8 – The Settings Dialog Window: Syntax Color Sheet

The Syntax Color sheet offers the settings for the text colors and emphasis of the source code window. (see chapter 2.2.1 "Structured Display of Source Code" on page 10).

For each of the categories any color (Change Color) and the font styles Bold or Italic can be set.

The structured display of the source code Syntax Color and Autoindent enabled, is the default setting. The automatic indentation uses the settings in the Editor Sheet as tab stop size.

Language Sheet

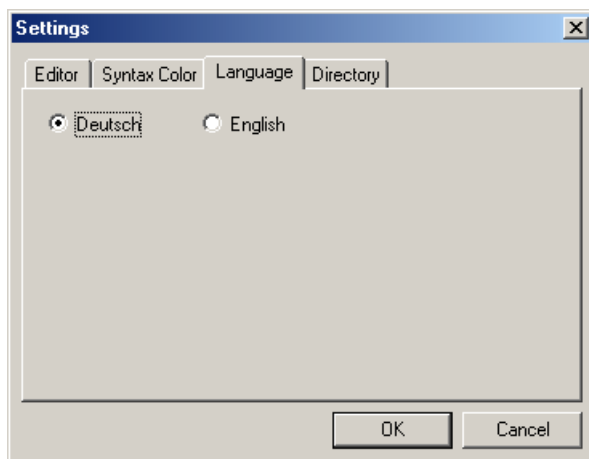


Fig. 9 – The Settings Dialog Window: Language Sheet

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

Directory Sheet

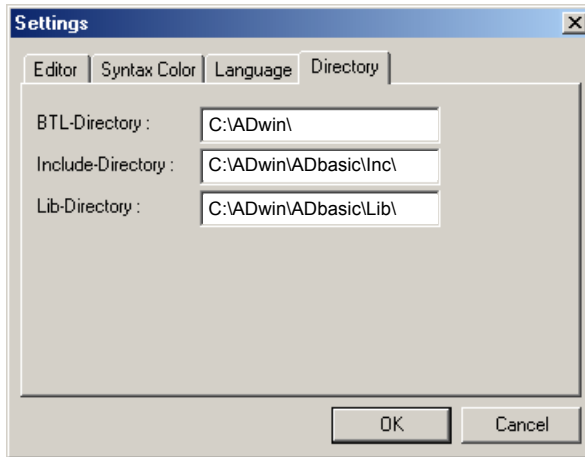


Fig. 10 – The Settings Dialog Window: Directory sheet

The Directory sheet contains the directories in which 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 2.1.2).
- **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).

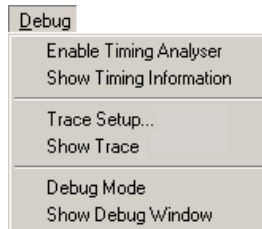


Please note: the path name must always end with a backslash (\).

It is recommended that the default directories are not to be changed. To include library and include files from other directories, indicate the correct and full path name in the include instruction.

2.3.6 Debug Menu

The Debug menu offers settings which help in finding run-time or syntactic errors. Please note that all settings will only be active after the next compilation.



Enable Timing Analyzer Option

When the `Enable Timing Analyzer` 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 `Show timing information` Menu Item). This option needs approximately 60 clock cycles (when using a T9 and T10 processor module) per event and process additionally and therefore slightly affects the timing characteristics. We recommend that the option should only be activated to compile one or only some processes and should then be deactivated again. These option settings of the processes are not saved when quitting *ADbasic*.

Show timing information Menu Item

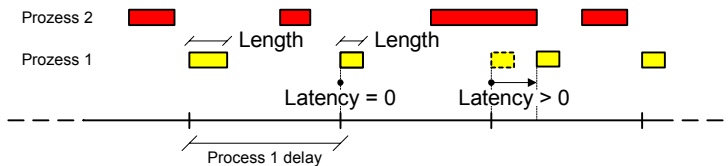
The `Show timing information` menu item opens the `Timing Information` window (with activated `Enable Timing Analyzer` Option only). For each of the processes 1...10 the window shows 7 parameters, which describe the timing characteristics of the processes since the moment it has been started. More detailed information can be found in chapter 4.3.2 "Check the Timing Characteristics (Timing Mode)".

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).

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

Fig. 11 – The "Timing information" Window

All duration values are counted in clock cycles of 25 ns. **Length** describes the time a process cycle needs (section **EVENT** :); this processing time can also be determined as described in chapter 4.1 "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, calculated as mean value from the last 1,000 length values.

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 5.2.4 "Different Operating Modes in the Operating System", page 79).

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 which is 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:
 $\text{max. latency} + \text{length} > 2 \times \text{Processdelay}$
- 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.

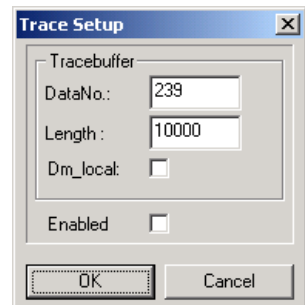
Sometimes it happens that, despite a true condition, *no* event is lost. Thus, you play it safe reducing the amount of true conditions as far as possible.

Trace Setup ... Menu Item

The Trace Setup ... menu item opens a configuration window for the trace mode. For more information about the usage of the trace mode see chapter 4.3.3 on page 70.

The trace mode will become active when using the Enabled option.

The input field DataNo indicates in which global array the process information is stored. Do not change the setting 239 (for DATA_239) if you need information about a single process only.



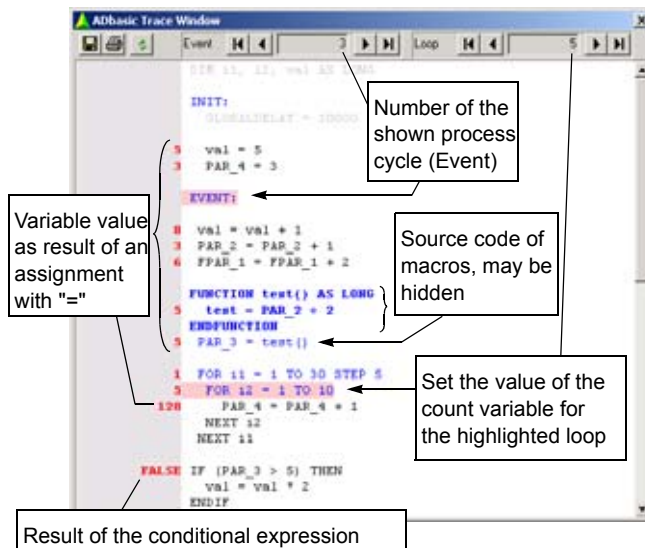
Enter the size of the global array (in LONG values) under Length so that on the one hand the size of the array is large enough for the trace information and on the other hand the ADwin system has enough remaining memory for your process variables.

With the DM_Local option active the values of the global array are stored in the local memory instead of in the external (see "Memory Areas"). The processor accesses data in the smaller (!) local memory essentially faster.


Show Trace Menu Item


The Show trace menu item opens the ADbasic Trace Window (only when the trace mode is enabled).

The trace window displays the process information left to the source code lines activated for trace mode. The most important information in the window is:



The displayed information are stored during run-time into a global array (normally [DATA_239](#), see [Enable Timing Analyzer Option](#)). The development environment then copies the array contents to the PC and displays them. Depending on the array size only few or many process cycles (events) can be stored.

When using the New Values  icon in the header line, the current process information are stored into the global array and then transferred to the PC. The previous process information are then lost.

For a later comparison the process information can either be stored  or the current screen content can be printed .

Debug mode Option

The Debug mode option, when activated, includes additional security queries into the process during the compilation of a source code (see also chapter 4.3.1 on page 67).

Activation of this option increases program execution time as well as the demand for memory. As a rule this increase has a dimension of approximately 20%, whereas greater values are also possible. Therefore, this option should only be used during program development.

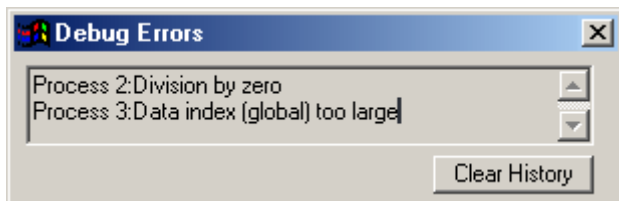


Fig. 12 – The Debug Errors Window

The window `Debug Errors` opens when a run-time error occurs in the *ADwin* system. The window can be reopened by clicking the `Show_Debug_Window` menu option after it is closed.

The operating system corrects run-time errors in a way to obtain a stable state of operation; this may cause unexpected program results. The following table shows which errors are displayed and which corrections are made.

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.
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 Access to local or global array elements which are not declared, with indices that are too large or too small.	A too small element index (<1) is replaced by 1, a too large element index by the greatest dimensioned element index.

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.

Show Debug Window Option

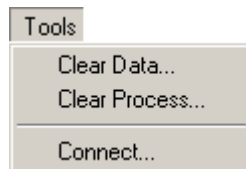
Clicking the Show Debug Window option reopens the Debug Errors window after it has been manually closed. The window opens automatically, the first time, after a run-time error occurs in the ADwin system.

In either case, the Debug mode option must be activated.

2.3.7 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 opposite of the DIM instruction which allocates memory for an array.



Clicking this option opens a dialog window requesting the data array index to be cleared, e.g. 3 for DATA_3. After entering a value and clicking "OK", the values in the data array will be lost.

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

The Connect menu option opens a dialog window for configuring the ADserver program settings for setting up a network connection to the ADwin system. A description can be found in the Appendix A-5.2.

Note: the ADserver program will no longer be updated, so it is recommended that the ADwin TCIPserver program be used instead. In this case no configurations must be made with the Connect menu option. 

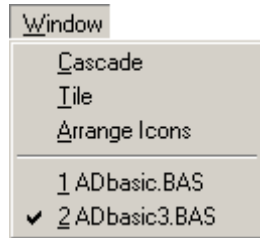
More information is available in the online help of ADwin TCIPserver (run Programs ► ADwin ► ADwinTCIPserver from the Windows start menu).

2.3.8 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.

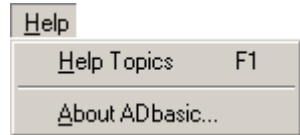
At the bottom of the menu, there is a list of open source codes; by clicking one of these menu items that source code will become the active window. The active source code is checked; in the example at right it is `ADbasic3.bas`.



2.3.9 Help Menu

The online help for *ADbasic* is accessible from the `Help Topics` menu option. The online help is also accessible using the  button or the [F1] key.

Clicking the `About ADbasic` menu option 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.



The `License key` is to be found on the cover sheet of this your *ADbasic* manual.

ADbasic will operate in demo mode, if no `License key` has been entered. In this mode the development environment only works for demonstration, test or evaluation purposes.

2.3.10 Project Window

The project window shows an opened project and the source code files included within it.

In the project window the following actions may be executed:

- Open a source code file and make it the active source code:
 - Double-click the file or
 - Highlight the file (left mouse button) then select **Open** from the context menu (right mouse button).
- Save a source code file:
Highlight the file and select **Save** from the context menu.
- Delete a source code file from the project:
Highlight the file then
 - press the [DEL] key or
 - select **Remove from Project** from the context menu.
- Hide the display of the included files:
Double-click on the project name; a [+] appears at left from the project name.

The following actions are available from the context menu only:

- Include a source code file into the project:
Select **Add to Project** from the context menu.
- Include all open source code files into the project:
Select **Add Open Files to Project** from the context menu.
- Save all open source code files of the project:
Select **Save All Files of Project** from the context menu.

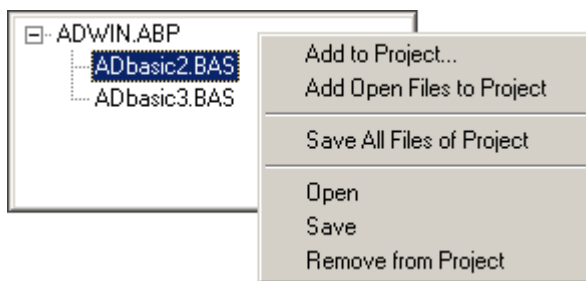


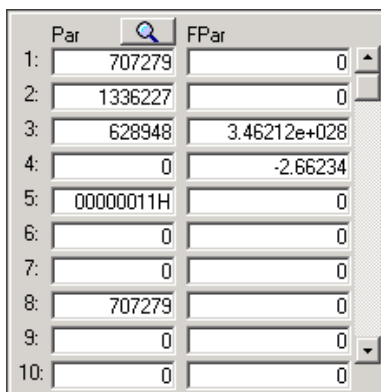


Fig. 13 – The Project Window with the Context Menu

2.3.11 The 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 right you can scroll through the parameters.


When the communication between the computer and ADwin system is active ( icon 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 grey background color when the communication is inactive (icon ).





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

Fig. 14 – The parameter window

A parameter's value (`PAR_1...PAR_80`) can be displayed in hexadecimal notation, too (see `PAR_5` in fig. 14). Do a right mouse click on the number of the variable (left of the table field) and enable / disable the option `Hexadecimal`.


Clicking the  button highlights the fields in the table that are being used in the active source code and project, using three colors. The colors have the following meaning:

- Green: The parameter is used in the active source code only.  2: 1336227
- Red: The parameter is used in the active source code, and in another source code of the project, too.  1: 707279

- Blue: The parameter is used in an inactive source code of the project, and not in the active source code.

The highlighting feature is only available when the communication between the computer and the ADwin system is set up. This feature uses saved source codes only; therefore *ADbasic* requests saving the source code when it has been changed.

2.3.12 The 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 grey.

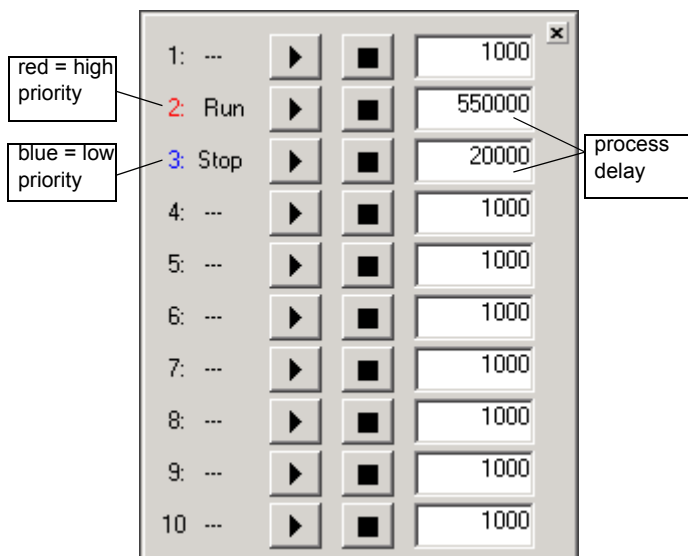




Fig. 15 – The Process Window

The status (Run or Stop) and process delay (process cycle time) are displayed for each of the processes 1 ... 10. The process delay for the active source code is also displayed in the toolbar. The priority of a process can be determined by the color of the process number, red = high priority, blue = low priority. The time units and meaning of the process delay are explained in chapter 5.2.1 "Processdelay (Globaldelay)", page 76.

It is also possible to change the process delay in the process window; after a value has been entered, it will automatically be transferred to the *ADwin* system. Please note: the system will overload if process delay values are too small.

A process can be stopped or restarted using the buttons  and  in the process window. The buttons in the toolbar have the same functions but only control the process of the active source code window.

2.3.13 Info window

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

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



The (successful) status message looks like this:

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

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.
- **Workspacesize:** Required memory size in bytes in the local data memory (DM), being used for
 - local variables and arrays
 - internal purpose (2×4 byte)

Additional memory will be required in the data memory which be calculated manually:

- Each global array requires about fourty byte in the local data memory (internal purpose).
- Each element of a global array requires 4 byte (in the external data memory; if the array be declared **AT DM_LOCAL**, the elements are stored in the local data memory).

- **Stacksize:** Internal stack size, which is used for libraries.

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

2.3.14 Status Bar

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



Last ADbasic action

CPU and memory usage of the ADwin system

- Left side: Information about the last *ADbasic* action.
- Right side: The current CPU and memory usage of the *ADwin* system. This information is displayed, if the communication between the computer and *ADwin* system is active.

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.
- **DM:** free internal data memory in bytes.
- **DX:** free external data memory in bytes.

2.4 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* by selecting **Programs ▸ ADwin ▸ ADtools ▸ <Toolname>** in the Windows start menu. Open the configuration menu to select the style of display and the variables to be displayed, by clicking the right mouse button.

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:

	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.
	TLed	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..
	TMeter	Global variable and array values can be viewed as an analog dial.
	TPoti	Global variable and array values can be adjusted with a potentiometer-style control.
	TProcess	Start/stop, adjust timing, and display information about the processes loaded on the <i>ADwin</i> system.
	TPar_FPar	All or selected global variables can be displayed or entered.
	TFifo	Save FIFO array data into a file..
	TBin	Up to five PAR variables can be displayed in binary (as DIL switch) and in hexadecimal notation, and adjusted.
	ADtools	Save and/or load a configuration to/from several <i>ADtools</i> .

All further information about the help programs can be found in the online help, in the program `ADtools.exe`.

3 Programming Processes


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

3.1 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 an instruction or assignment (exception see : Colon), with up to 255 (ASCII-) characters in one line.

ADbasic accepts instructions and variable names in lower and upper case letters (for more clarity all examples use upper case letters for instructions and global variables).

A program consists of up to 4 sections, which take on different tasks when executed on the *ADwin* system. fig. 16 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 be included.

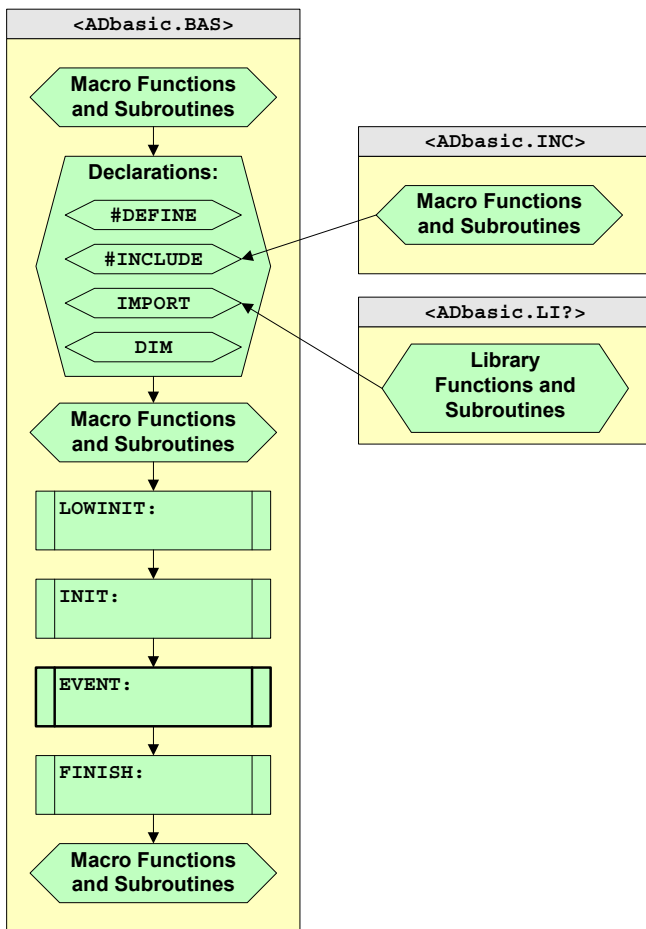



Fig. 16 – Design of an *ADbasic* program

3.1.1 The Program Sections


Each of the 4 program sections start with the following terms, 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 sections. 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.

3.1.2 Other Program Parts

Symbolic definitions

The instruction **#DEFINE** defines symbolic names. Group all of these definitions at the beginning of the file and before the start of the 4 program sections.

Arrays and Local Variables

In an *ADbasic* program the local variables and all arrays must be declared before they can be used. The global variables **PAR_n** and **FPAR_n** 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 computer, in order to exchange data between the processes or between the process and the computer.

Macros

A macro function or subroutine call inserts the macro into the program text where it is being used. However, the macro definition cannot be done within the 4 program sections. (see fig. 16 on page 40).

Libraries

Libraries must be included before the program sections that use them. Library functions and subroutines, when used more than once within a program, require less memory than similar macro functions or subroutines described above.

3.2 Variables and Arrays

3.2.1 Overview

Data structure	Name	Data type	Notes
Global variables and arrays			
Variable (Scalar)	PAR_1...PAR_80 FPAR_1...FPAR_80	LONG FLOAT	Pre-defined, not declarable,
System variable	GLOBALDELAY PROZESSn_RUNNING	LONG LONG	memory area DM
One- or two- dimensional array (vector)	DATA_1 [] [] ... DATA_200 [] []	LONG, FLOAT, STRING, FIFO	Name "DATA_" not changeable, only declaration of array number and dimension.
Local variables and arrays			
Variable (Scalar)	selectable	LONG, FLOAT	must be declared
One-dimensional array (vector)	selectable	LONG, 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 3.3.1), if not determined explicitly.

All data types have a length of 32-bit.

3.2.2 Data Structures

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

- variables (scalars)

VAR

Each variable can store one value only.

- 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 3.3.4 on page 52).

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

The compiler differentiates variables (and arrays) which are

- global:

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

- local:

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

Nearly all variables and arrays must be declared with the **DIM** instruction; this determines the data type, as well as the necessary memory place, and allocates it to the variable name. Global variables `PAR_1 ... PAR_80` and `FPAR_1 ... FPAR_80` are already pre-defined, for easier programming, 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 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.

3.2.3 Data Types

The compiler processes the following data types:

- **LONG**: 32-bit integer values with the ranges:
 $-2147483648 \dots +2147483647$ ($= -2^{31} \dots +2^{31}-1$).
- **FLOAT**: Floating-point values with the ranges:
 $-3402823 \cdot 10^{+38} \dots -1175494 \cdot 10^{-38}$ (negative values)
 $+1175494 \cdot 10^{-38} \dots +3402823 \cdot 10^{+38}$ (positive values)

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

- **STRING**: ASCII character strings, in which each character is stored as a single array element (for details see chapter 3.3.5 on page 53). A single character corresponds to an integer 8-bit value in the range 0...255.

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



When integer and floating-point values are combined, a type conversion will occur. Under certain circumstances this may cause calculation results discrepancies from expected results. More about this is found in chapter 3.4.2 on page 58.

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

3.2.4 Entering Numerical Values

You can use 4 different notations in order to enter numerical values. The following examples assign the (decimal) value 93 to a variable *x*.

1. Decimal notation:

x = 93 (integer value **LONG**) or

x = 93.0 (floating-point value **FLOAT**)

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

Please note the difference: The number 93 has the **LONG** data type, while the number 93.0 has the **FLOAT** data type. This is important when you use both data types in one expression (see chapter 3.4.2).



2. Exponential notation (**LONG** only):

```
x = 9.3E1
```

Here 9.3E1 stands for 9.3×10^1 , where "E" is followed by the exponent to the basis of 10 (max. 2 decimal places).

3. Binary notation (a b is added, **LONG** only):

```
x = 1011010b
```

4. Hexadecimal notation (an h is added, **LONG** only):

```
x = 5Ah
```

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

3.2.5 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. 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.

The system variables, also globally available, are described on page 47.

The 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 booting of the ADwin system the global variables are automatically initialized with zero.

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 **FLOAT** data type for floating-point values.

**Example**

```

PAR_5 = 700
PAR_72 = ADC(1)

```

'Parameter 5 contains the
'value 700.
'The voltage at the analog input 1
'is measured and stored into
'parameter 72.



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

3.2.6 Global Arrays

The global arrays enable the exchange of data between the processes on the *ADwin* system or the computer (see also chapter 5.3.1 "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).

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 non-sequential 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**

```

DIM DATA_5[20000] AS LONG
REM Declare the array 5 with 20000 elements of the type LONG.
DIM DATA_3[7][5] AS FLOAT
REM Declare the array 3 with 7x5 elements of the type FLOAT.

```

There is more information about 2-dimensional arrays in chapter 3.3.3 on page 50.

The maximum size of the array depends on the memory size. For instance on an *ADwin* system with 16MB memory an array of up to 4 million elements of the **LONG** type may be declared.

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



```
'The value of the 200th element from array 5 is assigned
'to the global integer variable PAR_1.
PAR_1 = DATA_5[200]

'In this program line the 345th element from the array DATA_5
'gets the value 4000.
DATA_5[345] = 4000

'This instruction assigns the value 300.1 to the 1st element 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:



```
num = 2
DATA_num[300] = 20      'WRONG !!
DATA_2[300] = 20        'CORRECT
```

The compiler determines `DATA_num` to be the name of a local array, which (probably) has not been declared and therefore is not available. Instead, use the notation `DATA_2`.

3.2.7 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.

PROZESS_n_RUNNING

Returns the status of the process `n` (with `n = 1...10`): the process is running, just being stopped or already stopped (see page 157). The variable can only be read.

GLOBALDELAY

The nominal time interval, in which time-controlled processes are called by the counter, is the processdelay (cycle time). With the system variable **GLOBALDELAY** you query and set this time, measured in clock cycles of the counter (see chapter 5.2.1 on page 76).

You read and write into the variable **GLOBALDELAY** 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.

Writing into this variable in the section **EVENT**: should just be made at the beginning of this section, because changing the variable will have an immediate effect on calling the next process cycle. Otherwise the precise processing of the process cycles in a certain time interval can become instable.



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

3.2.8 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 ("_"). 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 (255 characters).

Individual variables (scalars) can be defined as either integer values (type **LONG**) or floating-point values (type **FLOAT**), and each are 32 bits long.



Example

```
DIM value AS LONG           'Defines the variable 'value'
                               'with the data type LONG
DIM value1, value2 AS FLOAT 'Defines the variables value1
                               'and value2 with the data type 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

```
DIM value[100] AS FLOAT'Defines an array with the length
                        '100, with the name 'value',
                        'and the data type 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.

**3.3 Variables and Arrays – Details****3.3.1 Variables and Arrays in the Data Memory**

The user can explicitly determine which memory area, internal or external, to store arrays and local variables. 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**.

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

It is recommended that the internal memory be used for variables and (small) arrays for fast access. The slower, external memory is more suitable for arrays, due to its size.

The fig. 17 shows examples of declarations, in order to store variables and arrays in the different memory areas.

Variable / Array	Memory Area	Source Code Declaration
Local Variable	Internal (DM)	DIM var AS <VARTYPE> or DIM var AS ... AT DM_LOCAL
	External (DX)	DIM var AS ... AT DRAM_EXTERN
Array (global/ local)	Internal (DM)	DIM array[5] AS ... AT DM_LOCAL
	External (DX)	DIM array[5] AS ... or DIM array[5] AS ... AT DRAM_EXTERN

Fig. 17 – Allocation of the Memory Area with Declarations



The global variables `PAR_1...PAR_80` and `FPAR_1...FPAR_80` are pre-defined in the internal memory (DM), therefore they cannot be re-declared in the external memory (DX).

3.3.2 Memory Areas

The processor of the *ADwin* system uses its internal memory (SRAM) and an external memory (SDRAM) for data stored according to the following structure:

- Program memory (PM):
The 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 (standard setting).
- External data memory (DX)
The external data memory covers the external SDRAM and stores the global and local arrays (standard setting).

Data in the internal memory (DM) can be accessed faster than data in the external memory (DX) by approximately a factor of five.

The memory size is an ordering option and cannot be upgraded:

- Size of internal SRAM: 256 kB or 512 kB
- Size of external SDRAM: 8, 16, 64 or 128MB

The size of the memory areas is the only limiting factor to the size of the processes and 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, PM, DM and DX, is displayed in bytes.

3.3.3 2-dimensional Arrays

Global arrays `DATA_n` may be declared with 1 or 2 dimensions. The basic array features are described in chapter 3.2.6 "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:

```
REM use in main program
DATA_1[3][1] = 17
setpar1(DATA_1)           'sets PAR_1 = 17

REM use in library module
LIB_SUB setpar1(BYREF array[] AS LONG)
  PAR_1 = array[5]         'corresponds to DATA_1[3][1]
LIB_ENDSUB
```

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:



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

where *s* is the 2nd dimension of `DATA_n` in the declaration. In the example above there is *s*=2.

3.3.4 The Data Structure FIFO

For applications requiring a large quantity of data to be transferred continuously, it is recommended using a `DATA_n` global array with the FIFO data structure: a "First In, First Out" ring buffer.

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).

Only one-dimensional global arrays (`DATA_n`) can be declared as FIFO arrays; possible data types are `LONG` or `FLOAT`.



Example

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

This instruction declares the global array with the number 5 as FIFO ring buffer with 1000 elements of the type `LONG`.



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[1000] AS LONG AS FIFO
DATA_5 = 95
PAR_7 = DATA_5
```

'Writes the value 95 into the
'DATA_5 array which is declared as FIFO
'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[1000] 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
```

3.3.5 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**, **CHR**, **FLOATOSTR**, **LNGETOSTR**, **STRCOMP**, **STRLEFT**, **STRLEN**, **STRMID**, **STRRIGHT**, **VALI** and **VALF**. The library file <STRING.LI*> must be included, in order to use these instructions (where * indicates the processor type: 9 for T9, A for T10). It is found in the library directory (default: <C:\ADwin\ADbasic\LIB>) after the *ADwin* installation.

Moreover, strings can be concatenated using the + operator:

```
txt = txt_a + txt_b.
```

A string variable has a structure similar to an array, in which 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)
`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. The first and last elements of the string are automatically reserved by the *ADbasic* compiler.

Please note: The element number 0, here `text[0]` is not to be used!

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
<code>text [1]</code>	05h	Length of the string in characters (5)
<code>text [2]</code>	48h	ASCII value for "H"
<code>text [3]</code>	45h	ASCII value for "E"
<code>text [4]</code>	4Ch	ASCII value for "L"
<code>text [5]</code>	4Ch	ASCII value for "L"
<code>text [6]</code>	4Fh	ASCII value for "O"
<code>text [7]</code>	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:

- apostrophe ('): \x39
- quotation mark ("): \x34
- backslash (\): \x5C

Character Assignment with the Escape Sequence

The escape sequence is used to include numerical values or control characters into a string. The 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 sequences 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 00h to FFh.

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

Although it is possible to insert characters with the ASCII values `00h...1Fh` or `80h...0FFh`, for instance `[?]` or the German characters `[ß]` and `[Ö]` it is not recommended unless using the 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.



It is recommend that any control or special characters inserted into a string only be done using the escape sequence.

3.4 Expressions

3.4.1 Evaluation of Operators

An expression is what is assigned to a variable or transferred as an argument of an instruction. It consists of any possible combination of:

- simple data: constant, variable or array element
- operators being used for arguments.

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. 18).

Please take into account, that automatic Type Conversion may in some cases influence the evaluation of an expression (see page 58), too.

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

Fig. 18 – Priorities of Operator Categories
(Top = highest priority)

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.

Example

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

corresponds to

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



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
```



3.4.2 Type Conversion

In *ADbasic*, variables can (after dimensioning) generally be used without paying attention to their data types (**LONG** or **FLOAT**, see also chapter 3.2.3 "Data Types"). If necessary the data of the **LONG** type will automatically be converted into the **FLOAT** type.



Do not mix up this conversion with the instructions **CAST_FLOATTOLONG** or **CAST_LONGTOFLOAT**, which do quite a different job.

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.

- 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

```
PAR_1 = 2 / 4 * 3      'Result: PAR_1=0, because 2/4 = 0
```



Decimal places are always cut off within integer calculations, and will then be lost.

But:

```
FPAR_1 = 2 / 4 * 3      'Result: FPAR_1=1.5
```

```
PAR_1 = 2 / 4.0 * 3      'Result: PAR_1=1 (cut off!)
```

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 automatical conversion into **FLOAT**. To absolutely make calculations in **LONG**, an individual program line must be used.

Example



```
PAR_1 = 2
PAR_2 = 5
'here a conversion is made:
FPAR_3 = (PAR_2 / PAR_1) + 0.2 'FPAR_3 = 2.7
'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 conditional test within a Boolean expression in an **IF...THEN** or **DO...UNTIL** even if there are multiple tests linked with **AND** or **OR**.

Example



```
PAR_1 = 2
FPAR_2 = 5.5

'Both conditions are true, PAR_1 is not converted into
'FLOAT, therefore PAR_3 = 1.
IF ((PAR_1 / 4 * 3 = 0) AND (FPAR_2 * 1.1 > 5.5)) THEN
    PAR_3 = 1
ENDIF

'The condition with FLOAT does not influence the
'LONG calculation, therefore PAR_3 = 0.
IF (FPAR_2 * 1.1 > 5.5) THEN PAR_3 = PAR_1 / 4 * 3
```

3.5 Decision structures, Loops and Modules

When writing extensive programs, *ADbasic* provides sophisticated tools for structuring them. The following structure elements are available:

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

```
FOR ... TO ... {STEP ...} NEXT.
```

- Structures for case-by-case decisions:

```
IF ... THEN ... {ELSE} ... 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`
- 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
```
- Libraries of compiled subroutines and functions, which can be included into a user's source code, if necessary:
 - Library subroutines with `LIB_SUB ... LIB_ENDSUB`
 - Library functions with `LIB_FUNCTION ... LIB_ENDFUNCTION`

More information and examples of the instructions can be found in chapter 6 "Instruction Reference".

3.5.1 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.

You will find more information about the structure of macro modules in the instruction reference (page 127: `FUNCTION ... ENDFUNCTION`; page 180: `SUB ... ENDSUB`).

3.5.2 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 contents of an include file depends on the same rules as normal source code files. However, in most cases they 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).

Normally, the 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 40).



3.5.3 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.

Also, calling library modules several 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 3.5.1 "Subroutine and Function Macros", page 60).

Please note that a library module cannot call a library module within the same library file. It is recommended macro functions and subroutines be used instead. Alternatively, additional libraries may also be used.



When interlacing libraries (including a library within another library), the source code calling the libraries must include all levels (see fig. 19), otherwise an error message will be returned by the compiler.



Recursive calls of library functions or subroutines are not allowed.

You will find more information about the structure of the library modules in the instruction reference (page 139: **LIB_FUNCTION ... LIB_ENDFUNCTION**; page 143: **LIB_SUB ... LIB_ENDSUB**).

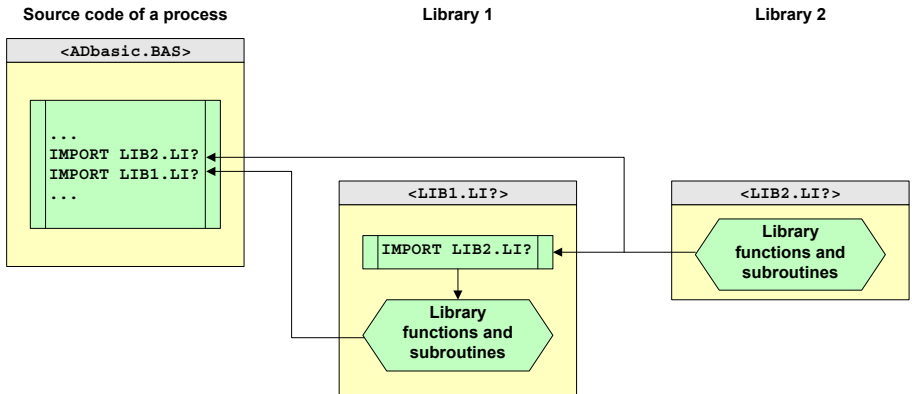


Fig. 19 – Interlaced Libraries

4 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.

4.1 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. The current counter values can be read using the **READ_TIMER** instruction; the counter corresponding to the running process's priority will automatically be read out.

When power is applied to the *ADwin* system, both counters are set to the value 0 (zero), then continually incremented in fixed clock pulses (see fig. 21).



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 **READ_TIMER** 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



```
DIM start, end AS LONG
```

```
EVENT :
```

```
...
start = READ_TIMER()
...
end = READ_TIMER()
PAR_1 = end - start -4
```

'Time-critical section
'Process time in clock pulses
'(offset = 4 clock pulses)

If `PAR_1` in the example above equals 37, the time-critical section of the high-priority process requires $37 \times 25\text{ns} = 925\text{ns}$.

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
```

4.2 Useful Information

4.2.1 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 the control and data registers can be found in the relevant hardware manual.

4.2.2 Constants instead of Variables

A calculation is executed faster when the values are specified as constants and not as variables.



Example

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

```
PAR_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 extracted and assigned to `PAR_1`.

In the second calculation the compiler already has determined the value. During run-time it will only be assigned.

4.2.3 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 chapter 6.3 "ADwin-Gold and ADwin-light-16", page 191 or "Pro-Software manual").

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)
...
START_CONV(11b)      'Wait for settling time
                     'Start conversion on both ADCs
WAIT_EOC(11b)        'Wait for end of conversion
PAR_1 = READADC(1)    'Read out ADC1
PAR_2 = READADC(2)    'Read out ADC2
```

The *ADwin-light-16* system has only one ADC.



4.2.4 Using Waiting Times

Some instructions require a certain waiting time after being called. This time can be used for other calculations.

The **SET_MUX** and **START_CONV** 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 four times the number of measurements.

The key feature in this example is that the individual steps in the conversion process are not carried out sequentially but rather in parallel. The time delay for the setting of the multiplexer is carried out in parallel with the time required for the A/D conversion of the signal from the other set of channels allowing portions of the conversion process to be overlapped.



Example

```

REM Example for Gold
INIT:
    SET_MUX(000000b)      'Set Mux for the first measurement,
                           'channels 1+3
    SLEEP(140)            'Wait 14 µs

EVENT:
    START_CONV(11b)       'Start conversion (channels 1+3)
    SET_MUX(001001b)      'Set Mux, channels 2+4
    WAIT_EOC(11b)         'Wait for end of conversion
                           ' (channels 1+3)
    PAR_1 = READADC(1)     'Read out ADC1, channel 1
    PAR_2 = READADC(2)     'Read out ADC2, channel 3

    START_CONV(11b)       'Start conversion(channels 2+4)
    SET_MUX(000000b)      'Set Mux, channels 1+3
    WAIT_EOC(11b)         'Wait for end of conversion (channels
                           '2+4)
    PAR_3 = READADC(1)     'Read out ADC1, channel 2
    PAR_4 = READADC(2)     'Read out 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.

4.3 Debugging and Analysis

Debug, timing, and trace modes are ADbasic's hands-on tools for debugging and program analysis. All modes are activated via the "Debug" menu (see page 25) and add their helping features to those programs, which are compiled with active mode.



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 that you use these tools for developing and testing of programs only.


4.3.1 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 behaviour, 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. As well, the run-time error is being corrected to maintain a stable mode of operation.

Errors being found should always be eliminated; even the automatic error correction of the debug mode is no more than a debugging tool, which does not fit for continuous operation. 

Details about activating and display of run-time errors are shown in section "Debug mode Option" on page 30.

4.3.2 Check 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 can be displayed in the Timing Information window.

Processes have good timing characteristics when the following situations *do not* (or rarely) occur:

1. An event signal does not start a process cycle immediately, but a certain (not exactly defined) time later.
2. 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 5.2.4 "Different Operating Modes in the Operating System").

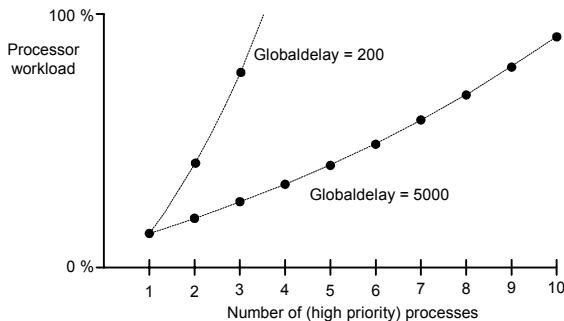
An optimal timing characteristic, especially of the high priority processes, is obtained in 2 steps by:

1. Checking Number and Priority of Processes
2. Creating 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 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 Processdelays 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 to view 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 5.2.4 "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 and T10 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 the `Show timing information` Menu Item, page 25.

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.

4.3.3 Track the Process Flow (Trace Mode)

The trace mode is a help tool for tracing the progress of program processing. This mode enables to view process information, mainly calculation results, generated during run-time on the *ADwin* system, *later* in a window of the development environment. You have to predefine the source code lines whose information you can view later.



The trace mode changes the timing characteristics of a process and needs additional memory in the data memory as well as in the program memory. This applies too, when no process information is defined and queried. With large quantities of defined data the additional requirement for time and memory can be higher than the requirement for the viewed process itself (without trace mode).




The trace mode is used as follows:

1. Activate the trace mode under : Debug ► Trace Setup and activate the `Enabled` option there.
2. Select the necessary lines in the source code and activate them for the trace mode in the context menu (right mouse button) with `Enable Trace`. Active lines have question marks ? at the beginning of each line.
Active lines can be disabled and enabled again from within a program via the instructions `TRACE_MODE_RESUME` and `TRACE_MODE_PAUSE`.
3. Compile (and start) the source code. Start first with an easy program.
4. Open the trace window with Debug ► Show Trace.

In the trace window left to the active source code lines the process information is displayed. The following information is displayed:

- A variable value as result of an assignment with the operator = (the operators `DEC` and `INC` are not supported).
- The value of a count variable in a loop, and, depending on this, the variable values in the loop (the count variable can be set at right in the header line of the trace window).
- The result of a condition: `True` or `False`.
If an `IF ... THEN` condition is followed by an assignment to a variable (single-line type of an `IF` command) and the condition is `true`, then only the variable value is displayed.
- The source code of a macro:
Do a right click on the name of the macro (`FUNCTION ...` or `SUB ...`); the macro text is then inserted directly above the line containing the macro.
The macro text is hidden in the same manner.

The displayed information is stored into a global array (normally `DATA_239`, see `Trace Setup ... Menu Item`) during run-time, that means during the time your program is running. The development environment then copies the array contents to the PC for display. Depending on the array size the information can refer to many or only a few events.

When using the `New Values`  icon in the header line, the displayed information is updated; if you would like to use the previous process information, you should first save  or print them .

By updating the information the process data in the program section **INIT**: is overwritten, too.

Note, that the trace mode only refers to the active source code, that is, imported libraries and include files are not supported. It is only possible to view one single process in the development environment; for each additional process you have to run *ADbasic* again as additional task and set the trace mode there. Please keep in mind to set a different global variable (**DATA_1** ... **DATA_200**) in each *ADbasic* task under Debug ▶ Trace Setup .



The instructions **TRACE_MODE_PAUSE** and **TRACE_MODE_RESUME** disable or enable the trace mode from within the *ADbasic* program (for active lines only). Thus, the trace mode can e.g. be activated as long as a certain condition is fulfilled.

5 Processes in the ADwin Operating 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 manipulated within the *ADwin* system and how it is transferred to and from the outside equipment 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.
2. 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.

In most cases control of the processes is done 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 boot option, it is also possible to have processes loaded and started automatically on power-up.

1. more precisely: the program section **EVENT** : .

5.1 Process Management

5.1.1 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 and
- Processes with Low-Priority

Low-priority processes are further divided into the levels 0 (low) up to 255 (high). The process priority is set via the menu `Options \ Process Options`.

Process	Function	Priority ^a
1...10	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 <i>ADwin-Pro</i> and <i>ADwin-Gold</i> systems	low, level 1
Communication	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. 20 – 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 76) 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.



5.1.2 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, even if the process is set to run with high-priority.



5.1.3 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 0 (low) up to 255 (high). Process cycles with a low level can be interrupted by those with a higher level at any time.

When low-priority processes are of the same priority level, their process cycles are processed by time slicing, which means one process cycle after the other is processed by the operating system for a short period of time. One time slice takes 2ms on average.

Low-priority processes must always be time-controlled. The cycle time (processdelay) can be set in intervals of 100 μ s (T9 processor) or 50 μ s (T10 processor).

Processes with low-priority on principle do not influence the time characteristic of high-priority processes, but vice versa they surely do.

5.1.4 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 250ms 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 5.3.2 on page 81.

5.2 Time Characteristics of Processes

5.2.1 Processdelay (Globaldelay)

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. It is usually measured in clock cycles of the system clock and called *Processdelay*, (in ear-

lier *ADbasic* versions: Globaldelay). The processdelay of each process is specified by setting the value of the system variable **GLOBALDELAY**.

The time resolution of the system clock depends on the process priority and on the processor type:

Processor	Priority	
	High	Low
T9	25ns = 0.025µs	100µs
T10		50µs

Fig. 21 – The time resolution of the system clock (units of the processdelay)

For instance, a processdelay with the value 1.000 means that for a high-priority process 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 $1.000 \times 100\mu\text{s} = 100.000\mu\text{s} = 100\text{ms}$ (with the processor T9). You can specify this event interval in the program line:

GLOBALDELAY = 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 **GLOBALDELAY**). Differences in the computing time may arise from different program sections which are run conditionally. (If, Case).

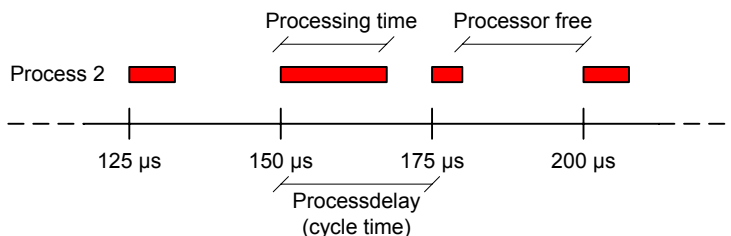


Fig. 22 – Processdelay and processing time in high-priority process cycles

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.

5.2.2 Precise Timing of Process Cycles

If you have (as shown in fig. 22) 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: $25\mu\text{s}$). 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.

If there are several high-priority processes, the actually running process cycle can influence the time schedule of the remaining process cycles. In fig. 23 for instance, the call of process 1 has to start after a delay when the processing of the active process 2 has finished.

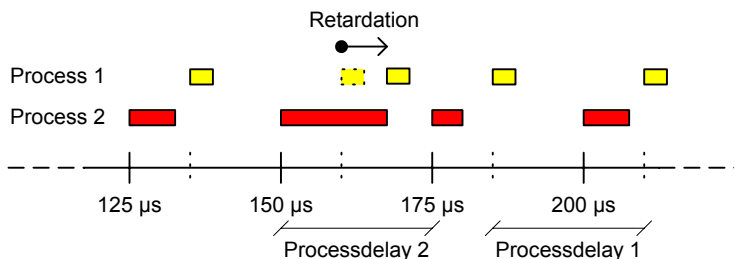


Fig. 23 – Delay of a high-priority process cycle



Keep the execution time of high-priority process cycles as short as possible. Have event loops, which require long processing time, or calculations whose result 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.

5.2.3 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. 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.

5.2.4 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.

Single Time-Controlled Process

In 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 5.2.1), 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 arrives 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 the process cycles absolutely be shorter than the cycle time (Processdelay) of the process.



Several Time-Controlled Processes

In several time-controlled processes, the operating system itself manages arriving event signals. The operating mode is working slower due to this management 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 that only few processes are used; 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 4.3.2 on page 67).

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 25). Therefore note to have short process cycles in this process (in the section **EVENT** :).

5.3 Communication

5.3.1 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 processes (see also chapter 3.5.2 "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 setpoint of the control loop.

Thus the function generator very easily controls the setpoint 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.

5.3.2 Communication between Computer and ADwin System

From PC applications and development environments, you can control the processes on the *ADwin* system, as well as requested 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 between the *ADwin* system and the PC is made via global variables (`PAR_n`, `FPAR_n`) or global arrays (`DATA_n`).

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 76).

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.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.

5.3.3 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 *ADwin32.dll* needs in order to being able to communicate with the system.

5.3.4 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 Visual Basic, C++, Delphi or C#.NET, or you may use a ready-made user interface such as TestPoint or MATLAB.

For each of these an appropriate driver software, which enables you to access the *ADwin* system is provided. 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 Computer and ADwin System" on page 81). 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*.
- 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).

6 Instruction Reference

In the following chapters the *ADbasic* instructions are listed:

- chapter 6.2 "Instructions for L16, Gold, Pro", -190

The hardware-related instructions for the *ADwin-Pro* system can be found in the documentation "*ADwin-Pro* Software".

- chapter 6.3 "ADwin-Gold and ADwin-light-16", -
- chapter 6.4 "ADwin-light-16 DIO1 / ADwin-Gold CO1", -

In these chapters the instructions are mostly listed in alphabetical order. In the annex you will find all instructions also listed alphabetically and in groups.

6.1 Instruction Syntax

Please note:

- Any expressions can be used as arguments.
- Some arguments require a specified data structure, which are labelled as follows:

CONST 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 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

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 3.4.2).

- Some instructions can only be used, when a specific library or include file is included. Under **Syntax** the relevant include-instruction is indicated (please, 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**).

6.2 Instructions for L16, Gold, Pro

+ Addition

The "+" operator is used to add two values (see also "+ String Addition").

Syntax

```
ret_val = val_1 + val_2
```

Parameters

val_1	Addend 1	<div>FLOAT</div> <div>LONG</div>
val_2	Addend 2	<div>FLOAT</div> <div>LONG</div>

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.

Example

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

+ String Addition

The "+" operator can be used to concatenate two strings (see also "+ Addition").

Syntax

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

Parameters

val_1	character string1	STRING
val_2	character string 2	STRING

Notes

If you concatenate two strings and assign them to another string, the size of the destination string must be greater or equal to the sum of the sizes of the input strings.

See also

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

Example

```
IMPORT string.li9

'Dimension 3 strings: 10, 5, 4 characters
DIM res_str[10] AS STRING
DIM str_1[5] AS STRING
DIM str_2[4] AS STRING

INIT:
  str_1 = "ADwin"      '5 characters
  str_2 = "Gold"       '4 characters

EVENT:
  res_str = str_1 + "-" + str_2 'Concatenate strings
  PAR_1 = STRLEN(res_str) 'PAR_1 = 10 (number of the characters)
```

- Subtraction

The "-" operator is used to subtract one value from another.

Syntax

```
val = val_1 - val_2
```

Parameters

val_1	Minuend	<div>FLOAT</div> <div>LONG</div>
val_2	Subtrahend	<div>FLOAT</div> <div>LONG</div>

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 3.5.1).

Example

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

* Multiplication

The "*" operator is used to multiply together two values.

Syntax

```
val = val_1 * val_2
```

Parameters

val_1 Multiplier 1

FLOAT

LONG

val_2 Multiplier 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.

Example

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

/ Division

The `"/` operator is used to divide one value by another.

Syntax

```
val = val_1 / val_2
```

Parameters

<code>val_1</code>	Dividend	<div>FLOAT</div> <div>LONG</div>
<code>val_2</code>	Divisor	<div>FLOAT</div> <div>LONG</div>

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 the divisor is a variable with a negative sign, you should use braces to ensure you get the expected result (see also chapter 3.4.1 "Evaluation of Operators" on).

Example

```
PAR_1 = 36 / 4          'PAR_1 = 9
PAR_2 = 27 / (-PAR_1)   'PAR_2 = -3
'Please note the braces in the second line
```

^ Power

The "^" operator is used to calculate the value of a number raised to a power.

Syntax

```
val = val_1 ^ val_2
```

Parameters

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

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 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 3.4.1 "Evaluation of Operators" on). This is not necessary with constants.



```
var1 = -2^2      'ret_val =
                  'ar2 = -var1^2'ret_val = -16var3 = (
                  'var1)^2'ret_val = 16
```

Example

```
PAR_1 = 9 ^ 4      'PAR_1 = 6561
```


#..., Preprocessor Statement

If an *ADbasic* instruction begins with the "#" sign, it instructs the preprocessor to treat the following source code differently. The output of the preprocessor is further processed by the compiler.

The following preprocessor 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. |

: 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.

A program line must not be longer than 255 characters (exception see **#INCLUDE** on).

It is recommend that you use this instruction only when it makes the source code more clearly-structured.

Example

```
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

```
ret_val = expr
```

Parameters

ret_val 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.

Example

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

```
INIT:
```

```
val_1 = 69                    'Assignment of a constant
```

```
EVENT:
```

```
val_2 = val_1 * 2           'Assignment of an expression
```

< = > Comparison

The operators "<", "=" and ">" are used to compare two values. In *ADbasic* these operators can only be found in conditional expressions.

Syntax

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

Parameters

val_1 Operand

FLOAT

LONG

val_2 Operand

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

Example

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

ABSF

The function **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 is 150 ns (T10: 75 ns).

See also

ABSI

Example

```
DIM val_1, val_2 AS FLOAT
EVENT:
  val_1 = -5.3
  val_2 = ABSF(val_1)   'Result: value2 = 5.3
```

ABSI

The function **ABSI** provides the absolute value of a long variable.

Syntax

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

Parameters

value	Argument	LONG
ret_val	Absolute value of the argument	LONG

Notes

The execution time of the function is 75 ns (T10: 50 ns).

See also

ABSF

Example

```
DIM val_1, val_2 AS LONG

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

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, val_2	Integer value	LONG
expr1, expr2	Boolean 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

Example

```
'Bitwise operator of LONG variables
DIM val_1, val_2, val3 AS LONG
val_1 = 0100b           '= 4
val_2 = 0110b           '= 5
val3 = val_1 AND val_2 'bitwise operator
'Result: val3 = 0100b = 4
```

Or:

```
'Boolean operation of Boolean expressions
```

```
DIM fval_1 AS FLOAT
```

```
DIM val4 AS LONG
```

```
fval_1 = 3.14
```

```
'Boolean operation: (true AND true) = true
```

```
IF ((fval_1 < 9.1) AND (fval_1 > 3.1)) THEN
```

```
    val4 = 1
```

```
ELSE
```

```
    val4 = 0
```

```
ENDIF
```

```
'Result: val4 = 1
```


ARCCOS

The function **ARCCOS** provides the arc cosine of the argument.

Syntax

```
ret_val = ARCCOS(val_1)
```

Parameters

<code>val_1</code>	Argument (-1 ... +1)	FLOAT
<code>ret_val</code>	Arc cosine of the argument in radians (0... π)	FLOAT

Notes

For `val_1` < -1 the value π (3,14159...) is returned, for `val_1` > 1 the value 0 (zero).

The execution time is up to 2.9 μ s (T10: 1.45 μ s).

See also

SIN, COS, TAN, ARCSIN, ARCTAN

Example

```
DIM val_1, val_2 AS FLOAT
```

```
EVENT:
```

```
val_1 = 0.5  
val_2 = ARCCOS(val_1)  
'Result: val_2 = 1.0472
```

ARCSIN

The function **ARCSIN** provides the arc sine of the argument.

Syntax

```
ret_val = ARCSIN (val_1)
```

Parameters

val_1	Argument (-1 ... +1)	Float
ret_val	Arc sine of the arguments in radians ($-\pi/2$... $+\pi/2$)	Float

Notes

The execution time of the instruction is up to 2.8 μ s (T10: 1.4 μ s).

See also

SIN, COS, TAN, ARCCOS, ARCTAN

Example

```
DIM val_1, val_2 AS FLOAT

EVENT:
  val_1 = 0.5
  val_2 = ARCSIN(val_1)
'Result: val_2 = 0.5236
```

ARCTAN

The function **ARCTAN** provides the arc tangent of the argument.

Syntax

```
ret_val = ARCTAN(val_1)
```

Parameters

<code>val_1</code>	Argument (whole range of values, see "Entering Numerical Values" on page 44)	FLOAT
<code>ret_val</code>	Arc tangent of the argument in radians ($-\pi/2 \dots \pi/2$)	FLOAT

Notes

The execution time of the instruction is up to 1.8 μ s (T10: 0.9 μ s).

See also

SIN, COS, TAN, ARCSIN, ARCCOS

Example

```
DIM val_1, val_2 AS FLOAT

EVENT:
    val_1 = 0.5
    val_2 = ARCTAN(val_1)
'Result: val_2 = 0.4636
```

ASC

The function **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 (0...255) of the (first) character	LONG

See also

+ String Addition, CHR, FLOTOSTR, LNGTOSTR, STRCOMP, STRLEFT, STRLEN, STRMID, STRRIGHT, VALF, VALI

Example

```
DIM text[10] AS STRING

INIT:
  text="Hello"

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

CAST_FLOATTOLONG

The instruction **CAST_FLOATTOLONG** changes the data type of the argument from FLOAT into LONG.

Syntax

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

Parameters

var	Bit pattern with data type LONG	<div>FLOAT</div>
ret_val	Identical bit pattern with data type FLOAT	<div>LONG</div>

Notes

This function does **not** execute a standard type conversion of a number (see chapter 3.4.2 "Type Conversion",). 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 **CAST_LONGTOFLOAT**, 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 3.2.3).

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

The instruction **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 FLOAT	LONG
ret_val	Identical bit pattern with data type LONG	FLOAT

Notes

This function does **not** execute a standard type conversion of a number (see chapter 3.4.2 "Type Conversion",). 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 3.2.3).

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

CHR

The function **CHR** assigns an ASCII character with a specified decimal number to a string variable.

Syntax

```
IMPORT STRING.LI*  
  
CHR(vascii,dest_text)
```

Parameters

<code>vascii</code>	Decimal number (0...255) of the desired ASCII character	LONG
<code>dest_text</code>	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.

See also

+ String Addition, ASC, FLOTOSTR, LNGTHOSTR, STRCOMP, STRLEFT, STRLEN, STRMID, STRRIGHT, VALF, VALI

Example

```
IMPORT STRING.LI9  
  
DIM text_a[1], text_b[1] AS STRING  
  
EVENT:  
  CHR(13, text_a)    'Carriage Return  
  CHR(10, text_b)    'Line Feed
```

COS

The function **COS** provides the cosine of an angle.

Syntax

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

Parameters

<code>angle</code>	Angle in radians ($-\pi \dots \pi$)	FLOAT
<code>ret_val</code>	Cosine of the angle ($-1 \dots 1$)	FLOAT

Notes

If you use input values which are not in the range of $-\pi \dots +\pi$, the calculation error grows with the increasing value.

The execution time of the instruction is 1.3 μ s (T10: 0.7 μ s).

See also

ARCCOS, SIN, TAN, ARCSIN, ARCTAN

Example

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


DATA_n

DIM **DATA_n** [...] **AS** ... dimensions a global **DATA** array.

More information about dimensing with **DIM** see .

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

DATA_n	Name of the declared DATA array (n: 1...200).
< ARR_TYPE >	Data type: FLOAT , LONG , STRING
dim1, dim2	Array size: Number of the array elements of the CONST type ARR_TYPE (≥1). LONG
< MEM_TYPE >	memory, where the array elements are stored: DRAM_EXTERN : external data memory (Default) DM_LOCAL : internal 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 3.3.3.

See also

DIM, FIFO

Example

```
REM Dimension the global array DATA_20 with
REM 1000 LONG elements
DIM DATA_20[1000] AS LONG

REM Dimension the global array DATA_5 with
REM 20 x 75 FLOAT elements
DIM DATA_5[20][75] AS FLOAT
```

DEC

The instruction **DEC** decrements the value of a long-variable by 1.

Syntax

```
DEC (var)
```

Parameters

var Name of a local or global long-variable

VAR

CONST

LONG

Notes

The instruction **DEC**(var) provides the same result as the program line: `val=val-1` and this instruction has shorter execution time.

See also

INC, - Subtraction

Example

```
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 at the beginning after
ENDIF                '1000 outputs
```

#DEFINE

The function **#DEFINE** replaces a symbolic name in the source code with an expression, for instance a constant.

Syntax

```
#DEFINE name expression
```

Parameters

name	Symbolic name, <i>without</i> quotation marks	CONST STRING
expression	Expression for the symbolic name, <i>without</i> quotation marks	CONST STRING

Notes



Place this instruction at the beginning of a source code.

The function **#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 this function in order to use more descriptive names in the source code instead of constants, parameters or expressions.

The first character string up to a blank is interpreted as a symbolic name, the following text until the carriage return is interpreted as an expression to be inserted. 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.

No distinction is made between upper and lower case letters.

If you want to use a mathematical term for *expression*, we recommend it be placed in parenthesis to avoid errors (see examples).

See also

#INCLUDE

Example

```
#DEFINE setpoint PAR_1
#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

The instruction **DIM** declares one or more

- *local* variables
- *local* one-dimensional arrays (also strings)
- *global* one-dimensional arrays (`DATA_n[n]`)
- *global* two-dimensioned arrays (`DATA_n[n][m]`).

Information about variables and data types can be found in chapter 3.2.3, information about FIFO arrays under the heading FIFO on .

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

<code>var1, var2</code>	Names of the declared variables
<code>array1,</code> <code>array2,</code> <code>DATA_n</code>	Names of the declared arrays. For <code>DATA_n</code> you can select <code>n</code> from 1...200.
<VAR_TYPE>	Data type: FLOAT , LONG for arrays also: STRING
<code>dim1, dim2</code>	Array size: Number of the array elements of the CONST type VAR_TYPE (≥1). LONG
<MEM_TYPE>	Memory where the variables are stored: DRAM_EXTERN : external memory (default for arrays) DM_LOCAL : local memory (default for variables)

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 ADwin system.

Details on 2-dimensional (global) arrays are described in chapter 3.3.3.

String variables are *local* arrays of type `STRING` (see "Strings" on page 53). They cannot be declared as FIFO.

See also

DATA_n, FIFO

Example

```
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 1000 LONG elements as ring buffer
DIM DATA_20[1000] AS LONG AS FIFO

REM Dimension the array TEXT with
REM 50 elements as string variable
DIM text[50] AS STRING
```

DO ... UNTIL

The **DO...UNTIL** command is used to repeatedly execute a block of instructions until the exit condition evaluates to "true". The block is executed at least one time.

Syntax

```
DO
    ...                'Instruction block
UNTIL (condition)
```

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.

Example

```
DIM count AS LONG
DIM DATA_1[100] AS LONG AS FIFO

INIT:
    count = 1

EVENT:
    DO                                'Start loop
        DATA_1 = ADC(1,4)           'Read out measurement value
        INC count                     'Increase count variable
    UNTIL (count > 100)               'Are 100 measurements being made?
```


END

The instruction **END** ends a process in the **EVENT** : section.

Syntax

END

Notes

The instruction **END** stops the processing of an **EVENT** : section immediately and starts processing the section **FINISH** : (if it exists). 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, **PROZESSn_RUNNING**, **START_PROCESS**,
STOP_PROCESS

Example

```
EVENT:
  IF (ADC(1) > 3000) THEN 'Measure and compare
    END                'End process, but execute FINISH:
  ENDIF

FINISH:
  SET_DIGOUT(1)      'Set digital output 1
```

EXIT

The instruction **EXIT** ends a process in the sections **LOWINIT**:, **INIT**: or **FINISH**:.

Syntax

```
EXIT
```

Notes

The process is immediately stopped. Program lines following **EXIT** in the same section, will not be executed.

Use the instruction **END** in the section **EVENT**:.

See also

```
END,    PROZESSn_RUNNING,    START_PROCESS,  
STOP_PROCESS
```

Example

```
INIT:  
  IF (ADC(1) > 3000) THEN 'Measure and compare  
    SET_DIGOUT(0)          'Set digital output  
    EXIT                  'End this process  
  ENDIF
```

EXP

The function **EXP** calculates the power to the base e (exponential) of the argument.

Syntax

```
ret_val = EXP(val_1)
```

Parameters

val_1	Argument	FLOAT
ret_val	Exponential value of the argument to the base e.	FLOAT

Notes

The execution time of the instruction is 1.3 µs.

See also

LN, LOG

Example

```
DIM val_1, val_2 AS FLOAT

EVENT:
  val_1 = 5
  val_2 = EXP(val_1)    'Result: val_2 = 148.41...
```

FIFO

The **DIM DATA_n AS FIFO** instruction is used to define a global **DATA** array as a ring buffer.

Syntax

```
DIM DATA_n[dim1] AS <ARR_TYPE> AS FIFO
```

Parameters

DATA_n	Name of the declared DATA -field (n: 1...200).
<ARR_TYPE>	Defined variable type: FLOAT , LONG
dim1	Array size: Number of the elements of the type CONST ARR_TYPE in the array. LONG

Notes

Once a **DATA** array is defined as FIFO ring buffer (see also chapter 3.3.3), 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 the instruction **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 out, 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.

See also

FIFO_CLEAR, **FIFO_EMPTY**, **FIFO_FULL**

Example

```
REM Dimension the global array DATA_20 with
REM 1000 LONG elements as ring buffer
DIM DATA_20[1000] AS LONG AS FIFO
```

FIFO_CLEAR

The instruction **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

Example

```
DIM DATA_1[20000] AS LONG AS FIFO 'Declaration
DIM reinit_fifo_flag AS LONG

INIT:
    FIFO_CLEAR(1)           'Initialize the FIFO pointer

EVENT:
    REM Query the number of empty places in the FIFO array
    IF (FIFO_EMPTY(1) > 1) THEN
        REM Measure the analog input 1 and save it in the FIFO
        DATA_1 = ADC(1)
    ENDIF
    ...
    IF (reinit_fifo_flag) THEN 'e.g. error occurred
        FIFO_CLEAR(1)        'Initialize the FIFO pointer
    ENDIF
```

FIFO_EMPTY

The instruction **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.

See also

FIFO, FIFO_CLEAR, FIFO_FULL

Example

```
DIM DATA_1[20000] AS LONG AS FIFO'Declaration

INIT:
    FIFO_CLEAR(1)           'Initialize the FIFO pointer

EVENT:
    REM Query the number of empty elements in the FIFO array
    IF (FIFO_EMPTY(1) > 1) THEN
        REM Measure the analog input 1 and save it in the FIFO
        DATA_1 = ADC(1)
    ENDIF
```

FIFO_FULL

The instruction **FIFO_FULL** determines the number of elements used in the FIFO array.

Syntax

```
ret_val = FIFO_FULL(arraynum)
```

Parameters

arraynum	Number of the DATA-FIFO-array (1...200)	LONG
ret_val	Number of the occupied array elements (0...dim)	LONG

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.

See also

FIFO, FIFO_CLEAR, FIFO_EMPTY

Example

```
DIM DATA_1[20000] AS LONG AS FIFO 'Declaration

INIT:
    FIFO_CLEAR(1)           'Initialize the FIFO pointer

EVENT:
    REM Query if there are data in the FIFO
    IF (FIFO_FULL(1) > 0) THEN
        REM Output a FIFO value on the analog output 1
        DAC(1, DATA_1)
    ENDIF
```


FLOTOSTR

The function **FLOTOSTR** converts a floating point value into a character string.

Syntax

```
IMPORT STRING.LI*          '*.LI9 for T9, *.LIA for T10
FLOTOSTR(val, text[])
```

Parameters

<code>val</code>	Value to be converted	FLOAT
<code>text []</code>	String in the format: {-}#.#####E{-}#{#}	ARRAY STRING

Notes

The length of the returned string varies from 10 to 13 characters, depending on the sign and the number of digits in the exponent.

See also

+ String Addition, ASC, CHR, LNGTOSTR, STRCOMP, STRLEFT, STRLEN, STRMID, STRRIGHT, VALF, VALI

Example

```
IMPORT STRING.LI9          'String library for the T9

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)
  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"
  PAR_6 = text[6]          'ASCII character 34h = "4"
  PAR_7 = text[7]          'ASCII character 30h = "0"
  PAR_8 = text[8]          'ASCII character 32h = "2"
  PAR_9 = text[9]          'ASCII character 35h = "5"
  PAR_10 = text[10]         'ASCII character 35h = "5"
  PAR_11 = text[11]         'ASCII character 45h = "E"
  PAR_12 = text[12]         'ASCII character 2Dh = "-"
  PAR_13 = text[13]         'ASCII character 31h = "1"
  PAR_14 = text[14]         'ASCII character 30h = "0"
  PAR_15 = text[15]         'String end character = 0
```

FOR ... TO ... {STEP ...} NEXT

The **FOR...NEXT** instructions are used to create program loops which execute 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
Z	Step length (≥ 1) of the run variable; default: 1	LONG

Notes

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* processor 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

Example

```
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 the count index

EVENT:
  DAC(1, sinus[index]) 'Output the amplitude value
  INC index 'Increase the 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

```

FUNCTION name({val_1, val_2, ...}) AS <VAR_TYPE>
{
  DIM var AS <VAR_TYPE>
  ...
  'instruction block
  name = ...
ENDFUNCTION

```

Parameters

name Name of the function and of the return value,
data type **<VAR_TYPE>**

val_1, val_2 Name of the passed parameters

FLOAT
LONG
STRING

<VAR_TYPE> Data type of the function and the return parameter: **FLOAT** or **LONG**, but not **STRING**

Notes

You will find general information about macros in chapter 3.5.1.

This instruction defines a function macro, which means that the whole instruction block of the function 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. In front of the section **INIT** : / **LOWINIT** :
2. After the section **FINISH** :
3. In a separate file which you include with the instruction **#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.

All instructions between **FUNCTION** and **ENDFUNCTION** belong to the function.

A function is called with its name and with the arguments 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 definition and call:

- definition of function *with* dimension braces:
FUNCTION `name(array[])` ...
- call of function *without* dimension braces:
`ret_val=name(array)`



If a value is assigned to a passed parameter within a function, the functions call must use a variable or a single array element as argument for this parameter.

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 und 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))
```

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):

```
x = average_array(array)
DAC(1,average_array(array))
```

For `array` you can indicate a global or a local array. Enter the array name only, without element number and brackets.

GLOBALDELAY

The parameter **GLOBALDELAY** defines the process delay (cycle time) of a process.

Syntax

```
ret_val = GLOBALDELAY
```

or

```
GLOBALDELAY = expr
```

Parameters

ret_val	Current cycle time, in clock cycles of the counter	LONG
expr	Cycle time to be set, in clock cycles of the counter	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 such cycles (process delay) is counted in clock cycles and defined or queried with **GLOBALDELAY**.

Define the process delay (cycle time) for high-priority processes in units of 25 ns, and for low-priority processes in units of 100 µs (T9) or 50 µs (T10)

If you need a process with a constant process delay, program this with **GLOBALDELAY** in the section **INIT** : / **LOWINIT** : or enter the value in the dialog window "Options / Process" under "Initial Process-delay".

With high-priority processes select a sufficiently large processdelay to avoid overloading the *ADwin* system (see also chapter 5.1.4).

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 250ms, the communication between the *ADwin* system and the computer can be interrupted.

You can use the variable only once in a section.

If the parameter **GLOBALDELAY** is changed in a process cycle in the section **EVENT** :, the cycle time (processss delay) 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.

Make sure that the processor workload (display field: "Busy x%" in the status bar) is under 90 percent and must not exceed 100 percent.

See also

READ_TIMER

Example

```
INIT:  
Globaldelay = 40000      'Set EVENT processing to 1 ms  
...
```

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

```

IF (condition) THEN
    ...
    'Instruction block
{ ELSE
    ...
    'Instruction block }
ENDIF

or

IF (condition) THEN instr

```

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, by putting the condition that has most common state, into 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**

Example

```
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 is 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

```

#IF PROZESSOR = value THEN
    ...                               'instruction block
{ #ELSE                             'the else-block is optional
    ...                               'instruction block}
#ENDIF

```

Parameters

PROZESSOR = value	If the condition is "true" the instructions after THEN are executed.	<div style="border: 1px solid black; padding: 2px;">LOGIC</div>
PROZESSOR	System parameter; refers to setting of processor type in the <i>ADbasic</i> window "Compiler Options". Type in as is.	
value	Processor type (type without quotation marks): T9:processor T9 T10:processor T10	

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**.

See also

< = > Comparison, IF ... THEN ... {ELSE} ... ENDIF

Beispiel

```
REM set low priority processdelay to 800µs
#IF PROZESSOR = T10 THEN 'If CPU = T10
    '800µs = 16 x 50µs
    GLOBALDELAY = 16
#ELSE                               'other CPU, here: CPU = T9
    '800µs = 8 x 100µs
    GLOBALDELAY = 8
#ENDIF
```

IMPORT

The instruction **IMPORT** includes functions and subroutines from the specified library file during compilation.

Syntax

```
IMPORT {path}file
```

Parameters

file	File name of the library file (file extension .LI9 for T9, .LIA for T10); <i>without</i> inverted commas	CONST STRING
path	Path name of the library file (with drive), <i>without</i> inverted commas	CONST STRING

Notes

Insert **IMPORT** instructions at the beginning of your source code (before you declare the variables). If you import several library files in a program, you have to also **IMPORT** the files in any functions you call that use these instructions.

Only those functions and subroutines which you call in your source code are imported from the library file.

You should always indicate the complete path name, otherwise only the standard directory is searched. (See Options Menu Directory Sheet,). Use the back slash "\" in the path name, in order to separate directory names.

See also

#INCLUDE, LIB_FUNCTION ... LIB_ENDFUNCTION, LIB_SUB ... LIB_ENDSUB

Example

```
IMPORT STRING.LI9      'Imports the string library for
                        'the T9 processor
IMPORT C:\MyFiles\ADwinLibs\dig2volt.LIA 'Imports a user
                        'library for the T9 processor
```

INC

The instruction **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

VAR

CONST

LONG

Notes

The instruction **INC** (var) is equivalent the program line: var=val+1.
However, this instruction has a shorter execution time.

See also

DEC, + Addition

Example

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

INIT:
    index=1

EVENT:
    DATA_1[index] = ADC(1) 'Transfer the measurement value into
                           'the array
    INC(index)             'Increment index by 1
    IF (index>1000) THEN END 'End the program after
                           '1000 measurements
```

#INCLUDE

The instruction **#INCLUDE** includes all the contents of an include-file into the source code.

Syntax

```
#INCLUDE {path}filename
```

Parameters

<code>filename</code>	Name of the file to be included (with the extension <code>.inc</code>), without quotes	CONST STRING
<code>path</code>	Complete path with drive	CONST STRING

Notes

You find general information about include-files in chapter 3.5.2.

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 **IMPORT**.

You should always indicate the complete path name, otherwise only the standard directory is searched (see [Options Menu Directory Sheet](#),). Use the back slash "\" in the path name, in order to separate directory names.



Please note: A program line with an **#INCLUDE** instruction should not exceed 136 characters (maximum length for other lines see). Any further character of this line will not be processed by the compiler.

See also

```
#DEFINE, IMPORT, FUNCTION ... ENDFUNCTION,  
SUB ... ENDSUB
```

Example

```
#INCLUDE C:\Test\demofunc.inc  
#INCLUDE demofunc.inc
```


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 name(par_1 {, val_2, ...} )
  AS <FCT_TYPE>

  {DIM var AS <VAR_TYPE>}
  {#DEFINE name expression}

  ...           'Instruction block

  name = ...

LIB_ENDFUNCTION
```

Syntax of passed parameters `par_n`:

```
<BY_TYPE> par_n AS <VAR_TYPE> {AT <MEM_TYPE>}
```

Parameters

`name` Name of the library function and of the return value;
data type **<FCT_TYPE>**

<FCT_TYPE> Data type: **FLOAT, LONG**

`par_1, par_2, ...` Name of the passed parameters.

<BY_TYPE> Methods for the transfer of parameters:
 BYREF: pass reference (pointer) to variable
 BYVAL: pass value only

<VAR_TYPE> Data type: **FLOAT, LONG, STRING**

<MEM_TYPE> Type of memory, where the passed parameters are
 stored; to be used only with the method **BYREF**:
 DRAM_EXTERN: external memory
 DM_LOCAL: local memory

Notes

You will find general information about library files in chapter 3.5.3.

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.
- can use global variables and arrays which are passed as parameters.
- can 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 3.3.3).
- should assign a value to the function name, which will be the value returned for the function in the source code.
- cannot define process sections such as **LOWINIT:**, **INIT:**, **EVENT:**, or **FINISH:**.
- cannot 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.
- cannot use the instruction **SELECTCASE**.

There are 2 methods for passing parameters that differ as follows:

- **BYREF:** The library function can change the parameter, so that 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 funcname (... 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

LIB_SUB ... LIB_ENDSUB, IMPORT, SUB ... ENDSUB, FUNCTION ... ENDFUNCTION

Example

```
'----- Calculate a mean value -----  
LIB_FUNCTION average(BYREF array[] AS LONG, BYVAL ptr AS LONG,  
    BYVAL cnt AS LONG) AS LONG  
    DIM i AS LONG  
    average = 0  
    IF (cnt > 0) THEN  
        FOR i = ptr TO (ptr + cnt)  
            average = average + array[i]  
        NEXT i  
        average = average / cnt  
    ENDIF  
LIB_ENDFUNCTION
```

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 the count variable
  GLOBALDELAY = 40000    'Measurement with 1 kHz

EVENT:
  samples[i] = ADC(1)    'Measure and save analog values
  INC i                  'Increment count variable
  IF (i > length) THEN END 'Are 1000 measurements complete?
                          '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 name(par_1 {, par_2, ...})
    {DIM var AS <VAR_TYPE>}
    {#DEFINE name expression}
    ...
    'Instruction block
LIB_ENDSUB
```

Syntax of passed parameters `par_n`:

```
<BY_TYPE> par_n AS <VAR_TYPE> {AT <MEM_TYPE>}
```

Parameters

<code>name</code>	Name of the library subroutine.
<code>par_1, par_2, ...</code>	Name of the passed parameters. For arrays you have to use this syntax <i>with</i> dimension brackets: <code>array[]</code> or <code>DATA_n[][]</code>
<code><BY_TYPE></code>	Methods for the transfer of parameters BYREF : pass reference (pointer) to variable BYVAL : pass value only
<code><VAR_TYPE></code>	Data types: FLOAT , LONG , STRING
<code><MEM_TYPE></code>	Type of memory, where the passed parameters are stored; to be used only with the method BYREF : DRAM_EXTERN : external memory DM_LOCAL : local memory

Notes

You will find general information about library files in chapter 3.5.3.

Generate library subroutines (and library functions) in a separate source code file. The compilation with "Build/Make lib file" cre-

ates 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 3.3.3).
- cannot define process sections such as **LOWINIT:**, **INIT:**, **EVENT:**, or **FINISH:**.
- cannot 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.
- cannot use the instruction **SELECTCASE**.

There are 2 methods for passing parameters that differ as follows:

- **BYREF:** The library function can change the parameter, so that 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.



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 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

Example:

```
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
```

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 the samples which are
                           'to be converted
#DEFINE gain 1             'Gain of the PGA
#DEFINE samples DATA_1    'Memory for measurement values
#DEFINE scaled DATA_2     'Memory for converted measurement
                           'values
#DEFINE length 1000       'Length of the array

DIM samples[length] AS LONG 'Source array
DIM i AS LONG              'Count variable

INIT:
    i = 1                   'Initialize the count variable
    GLOBALDELAY = 40000    'Measurement with 1 kHz

EVENT:
samples[i] = ADC(1)        'Measure and save analog values
    INC i                   'Increment count variable
    IF (i > length) THEN END 'Are 1000 measurements being made?
                           'If yes: process FINISH

FINISH:
    REM Convert the 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

The function **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 instruction is 1.45 μ s (T10: 0.7 μ s). .

See also

LOG, EXP

Example

```
DIM val1, val2 AS FLOAT

EVENT:
  val1 = 5.3
  val2 = LN(val1)      'Result: val2 = 1.667...
```

LNGTOSTR

The function **LNGTOSTR** converts an integer value into a string.

Syntax

```
IMPORT STRING.LI*          '*.LI9 for T9, *.LIA for T10
LNGTOSTR(value, STRING)
```

Parameters

value	Value to be converted	LONG
string	Result string	ARRAY STRING

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 3.3.5.

See also

+ String Addition, ASC, CHR, FLOTOSTR, STRCOMP, STRLEFT, STRLEN, STRMID, STRRIGHT, VALF, VALI

Example

```
IMPORT STRING.LI9
DIM digits[11] AS STRING'Result-string
DIM a AS LONG

INIT:
  a = -1234567890

EVENT:
  LNGTOSTR(a,digits)  'Convert to string
  PAR_1=digits[1]     'String length = 11
  PAR_2=digits[2]     'ASCII character 45 = "-"
  PAR_3=digits[3]     'ASCII character 49 = "1"
  PAR_4=digits[4]     'ASCII character 50 = "2"
  PAR_5=digits[5]     'ASCII character 51 = "3"
  PAR_6=digits[6]     'ASCII character 52 = "4"
  PAR_7=digits[7]     'ASCII character 53 = "5"
  PAR_8=digits[8]     'ASCII character 54 = "6"
  PAR_9=digits[9]     'ASCII character 55 = "7"
  PAR_10=digits[10]   'ASCII character 56 = "8"
  PAR_11=digits[11]   'ASCII character 57 = "9"
  PAR_12=digits[12]   'ASCII character 48 = "0"
  PAR_13=digits[13]   'End of string sign = 0
```

LOG

The function **LOG** provides the decimal logarithm (to base 10) of an argument.

Syntax

```
ret_val = LOG(val)
```

Parameters

<code>val</code>	Argument	<div>FLOAT</div>
<code>ret_val</code>	Decimal logarithm of the argument	<div>FLOAT</div>

Notes

The execution time of the instruction is 1.5 μ s (T10: 0.75 μ s).

See also

LN, EXP

Example

```
DIM val1, val2 AS FLOAT
```

```
EVENT:
```

```
val1 = 5.3
```

```
val2 = LOG(val1)      'Result: val2 = 0.724...
```

NOP

The instruction **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; this takes 25ns with a T10 processor, and 50ns with a T9 processor.

With this instruction you can delay for a necessary waiting period (e.g. after **SET_MUX**).

See also

SLEEP

OR

The operator **OR** combines two integer values bit wise or two Boolean expressions as a Boolean operator.

Syntax

```
ret_val = val_1 OR val_2 ... val_2      'bit wise operator
IF ((expr1 OR (expr2)) THEN           'Boolean operator
```

Parameters

val_1, val_2	Integer value	LONG
expr1, expr2	Boolean 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. This is not necessary for combining of integer values.

See also

AND, NOT, XOR

Example

Bit wise operator:

```
DIM val1, val2, val3 AS LONG

val1 = 0100b
val2 = 0110b
val3 = val1 OR val2      'Result: val3 = 0110b
```

Boolean operator:

```
DIM x AS LONG
```

```
DIM val4 AS LONG
```

```
INIT:
```

```
  x = 15
```

```
EVENT:
```

```
  IF ((x < 9) OR (x > 3)) THEN
```

```
    val4 = 1
```

```
  ELSE
```

```
    val4 = 0
```

```
  ENDIF
```

```
  'Result: val4 = 1
```


PEEK

The function **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	LONG
ret_val	Contents of the memory location	LONG

Comment

You will find an overview of the register addresses in your hardware documentation.

See also

POKE, READ_TIMER

Example

The instruction indicated 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.

```
'read out memory locations of an ADwin-Gold system  
val = PEEK(30h)
```

POKE

The instruction **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 are written	LONG
value	Value to be written	LONG

Comment

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 in your hardware documentation.

See also

PEEK, READ_TIMER

Example

```
'Change memory location of an ADwin-Gold system
'Write into DAC register: 3072 (≈+5V in the range of
'±10V)
POKE(50h, 3072)
POKE(50h, 011b)      'Start output on all DACs
POKE(0C0h, 111100b)  'Set DIGOUT bits 2 to 5
```

PROZESSn_RUNNING

The system variable **PROZESSn_RUNNING** returns the current status of the specified process.

Syntax

```
ret_val = PROZESSn_RUNNING
```

Parameters

n	Number of the requested process (0...15)	CONST LONG
ret_val	Process status: 1 Process is running 0 Process is stopped -1 Process is being stopped	LONG

Notes

The result is a read only value.

See also

END, EXIT, START_PROCESS, STOP_PROCESS

Example

```
EVENT:  
  REM Get the status of process 2  
  PAR_2 = PROZESS2_RUNNING
```

READ_TIMER

The function **READ_TIMER** returns the current counter value of the *ADwin* system timer.

Syntax

```
ret_val = READ_TIMER()
```

Parameters

ret_val Current counter values

LONG

Comment

This system variable is read only.

There are 2 timers in an *ADwin* system (32-bit), which count in different units of time:

Counter time unit with process priority:	T9	T10
1. high-priority	25ns	25ns
2. low-priority	100µs	50µs

See also

GLOBALDELAY

Example

```
DIM timervalue AS LONG
```

```
EVENT:
```

```
timervalue = READ_TIMER()
```

REM, '

The compiler instructions `REM` or `"'` 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

comment	Any character strings
instr	<i>ADbasic</i> 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 `"'`.

If you want to insert a `REM` comment after an instruction, separate it from the instruction by a colon `:`. If you use `"'` a colon is not necessary.

Example

```
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
```

SELECTCASE

The **SELECTCASE** control structure is used to execute one of several instruction blocks depending on a given value.

Syntax

```

SELECTCASE var
CASE const1a{,const1b, ...}
    ...
    'Instruction block
CCASE const2a{,const2b, ...}
    ...
    'Instruction block
CASEELSE
    ...
    'Instruction block
ENDSELECT

```

Parameters

<code>var</code>	Argument to be evaluated (no expression)	LONG
<code>const1a,</code> <code>const1b,</code> <code>const2a,</code> <code>const2b</code>	Value of <code>var</code> (0...255), where the following instruction block will be executed	CONST LONG

Notes

You may nest several **SELECTCASE** structures; the only limit is the memory size.

This control structure cannot be used within a library function or subroutine.

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 following example not only the instruction **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.

See also

DO ... UNTIL, FOR ... TO ... {STEP ...} NEXT, IF ... THEN ... {ELSE}
... ENDIF

Example

```

EVENT :
  PAR_1=2
  SELECTCASE PAR_1      'Evaluate PAR_1
    CASE 0                'If PAR_1 = 0?
      PAR_10 = ADC(1)      'Read out ADC(1)
    CASE 1                'If PAR_1 = 1?
      PAR_10 = ADC(3)      'Read out ADC(3)
    CASE 2                'If PAR_1 = 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 zeroes.

Syntax

```
ret_val = SHIFT_LEFT(val, num)
```

Parameters

val	Argument	LONG
num	Number of places the argument is shifted (0...31)	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 2^n . 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

Example

```
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 zeroes.

Syntax

```
ret_val = SHIFT_RIGHT(val, num)
```

Parameter

val	Argument	LONG
num	Number of places, which are shifted (0...31)	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 is a positive number, shifting it n places to the right corresponds to a division by 2^n . 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

Example

```
DIM val1, val2 AS LONG
```

```
EVENT:
```

```
val1 = 1024
```

```
val2 = SHIFT_RIGHT(val1, 3) 'Result: val2=128
```

SIN

Syntax

```
ret_val = SIN(angle)
```

Parameters

angle	Arc angle ($-\pi \dots +\pi$)	FLOAT
ret_val	Sine of the angle ($-1 \dots 1$)	FLOAT

Notes

If you use input values which are not in the range of $-\pi \dots +\pi$, the calculation error grows with the increasing value.

The execution time of the instruction is 1.25 μs (T10: 0.63 μs).

See also

COS, TAN, ARCSIN, ARCCOS, ARCTAN

Example

```
DIM val1, val2 AS FLOAT

EVENT:
  val1 = -5.3
  val2 = SIN(val1) 'Result: val2=0.83...
```

SLEEP

The instruction **SLEEP** causes the processor to wait for a certain time.

Syntax

SLEEP (*val*)

Parameters

val Number of the time units to wait in
100ns (≥ 1)

LONG

Notes

Since this instruction 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 ADwin system will become unstable. And please consider that very high values in high-priority processes can cause an interruption in the communication to the PC.



If the arguments requires a calculation, it requires additional time; this time interval is constant and takes only additional 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

NOP

Example

```
EVENT:
SET_MUX(0)           'Set the multiplexer
SLEEP(25)            'Wait 2.5 µs (=25*100ns) = settling
                     'time of the MUX
START_CONV(1)        'Start conversion
...
```

SQRT

The function **SQRT** returns the square root of a value.

Syntax

```
ret_val = SQRT(val)
```

Parameter

val	Argument	Float
ret_val	Square root of the argument or 0 for (val<0)	Float

Notes

The execution time of the instruction is 0.9 μ s (T10: 0.45 μ s).

Example

```
DIM val1, val2 AS FLOAT

EVENT:
  val1 = 16
  val2 = SQRT(val1)      'Result: val2 = 4
```

START_PROCESS

The instruction **START_PROCESS** starts a specified process.

Syntax

```
START_PROCESS (processnum)
```

Parameters

processnum Number of the process to be started (0...12, LONG
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 from another process only. It is not possible that a process starts itself, for instance in the section **FINISH** .

See also

END, EXIT, STOP_PROCESS

Example

```
EVENT :  
  IF (ADC(1) > 3072) THEN 'threshold value exceeded?  
    START_PROCESS (2)    'Start measurement process 2  
    END  
  ENDIF
```

STOP_PROCESS

The instruction **STOP_PROCESS** stops a specified process from another running process.

Syntax

STOP_PROCESS (processnum)

Parameters

processnum Number of the process to be stopped
(0...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 or
- has the same number as the calling process.

Stopping the **EVENT**: sections happens as follows:

- First the specified process gets the status "process is being stopped" (see **PROZESSn_RUNNING**), and is set to a low priority process if processing is not complete after a certain time interval (time-out).
- If the section **EVENT**: is being in progress when the stop signal arrives, the execution continues until the **EVENT**: is completed.
- Normally the section **EVENT**: is called and processed once more.
- If there is any, the section **FINISH**: is processed (always at low-priority).
- When the **STOP_PROCESS** has completed, the specified process is not active, but can be started at any time.

If you would like the process to stop itself, use the instructions **END** or **EXIT**.

See also

END, EXIT, START_PROCESS

Example

```
EVENT:
  IF (ADC(1) > 3072) THEN 'threshold value exceeded?
    STOP_PROCESS(2)      'stop measurement process 2
  END
ENDIF
```

STRING, ""

Strings are put into quotes "".

Syntax

```
IMPORT STRING.LI*          '*.LI9 for T9, *.LIA for T10
DIM text[length] AS STRING
text = "ADwin"
```

Parameters

<code>text []</code>	Name of the text variable	<div>ARRAY</div> <div>STRING</div>
<code>length</code>	Length of the text variable	<div>CONST</div> <div>LONG</div>

Notes

Dimension the text variables with **DIM ... AS STRING** (see). 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 .

Strings can be processed with the instructions mentioned below. Also, you can add (concatenate) strings with the "+"-operator.

See also

+ String Addition, DIM, ASC, CHR, FLOTOSTR, LNGTHOSTR, STR-COMP, STRLEFT, STRLEN, STRMID, STRRIGHT, VALF, VALI

Example

```
IMPORT STRING.LI9

REM Dimension strings with 3 and 1 characters
DIM chars[3] AS STRING
DIM char[1] AS STRING

INIT:
    REM Transfer characters to the strings
    chars = "ABC"
    char = "z"

EVENT:
    PAR_1 = chars[1]      'PAR_1 = 3 number of the characters
    PAR_2 = chars[2]      'PAR_2 = 65 (= "A")
    PAR_3 = chars[3]      'PAR_3 = 66 (= "B")
    PAR_4 = chars[4]      'PAR_4 = 67 (= "C")
    PAR_5 = chars[5]      'PAR_5 = 0 end of string

    REM Conversion into upper case:
    REM Lower case: a, b, c, ..., x, y, z?
    IF (char>96 AND char<133) THEN
        REM Subtract 32 in order to convert into upper cases
        char = char - 32
        REM Output on PAR_6 (=91), for control
        PAR_6 = char
    ENDIF
```

STRCOMP

The function **STRCOMP** checks two strings to determine if they are identical.

Syntax

```
IMPORT STRING.LI*          '*.LI9 for T9, *.LIA for T10
ret_val = STRCOMP(text1[], text2[])
```

Parameters

text1[],	String	ARRAY
text2[]		STRING
		CONST
ret_val	0: Strings are identical -1: Strings are not identical	LONG

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 Addition, ASC, CHR, FLOTOSTR, LNGTOSTR, STRLEFT, STRLEN, STRMID, STRRIGHT, VALF, VALI

Example

```
IMPORT STRING.LI9

DIM text1[7], text2[7], text3[8] AS STRING

INIT:
  text1 = "ADbasic"      'ADbasic correct writing
  text2 = "ADbasci"      'ADbasic wrong writing
  text3 = "ADbasica"     'ADbasic wrong writing

EVENT:
  PAR_1 = STRCOMP(text1,text2) 'PAR_1=-1
  PAR_2 = STRCOMP(text1,text3) 'PAR_2=-1
```

STRLEFT

The function **STRLEFT** returns a specified number of characters from the left end of a string into a second string.

Syntax

```
IMPORT STRING.LI*          '*.LI9 for T9, *.LIA for T10  
STRLEFT(text1[], length, text2[])
```

Parameters

text1 []	String, from which it is copied	<div>ARRAY</div> <div>STRING</div>
length	Number of characters to be copied	<div>LONG</div>
text2 []	String, into which it is copied	<div>ARRAY</div> <div>STRING</div>

See also

+ String Addition, ASC, CHR, FLOTOSTR, LNGTOSTR, STRCOMP, STRLEN, STRMID, STRRIGHT, VALF, VALI

Example

```
IMPORT STRING.LI9

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 from the string text1
  STRLEFT(text1,14,text2)
  PAR_1 = text2[1]      'String length = 14 characters
  PAR_2 = text2[2]      'ASCII-character 4Dh = "M"
  PAR_3 = text2[3]      'ASCII-character 45h = "E"
  PAR_4 = text2[4]      'ASCII-character 47h = "G"
  PAR_5 = text2[5]      'ASCII-character 41h = "A"
  PAR_6 = text2[6]      'ASCII-character 20h = " "
  PAR_7 = text2[7]      'ASCII-character 72h = "r"
  PAR_8 = text2[8]      'ASCII-character 65h = "e"
  PAR_9 = text2[9]      'ASCII-character 61h = "a"
  PAR_10 = text2[10]     'ASCII-character 6Ch = "l"
  PAR_11 = text2[11]     'ASCII-character 2Dh = "-"
  PAR_12 = text2[12]     'ASCII-character 74h = "t"
  PAR_13 = text2[13]     'ASCII-character 69h = "i"
  PAR_14 = text2[14]     'ASCII-character 6Dh = "m"
  PAR_15 = text2[15]     'ASCII-character 65h = "e"
  PAR_16 = text2[16]     'End of string character = 0
```

STRLEN

The function **STRLEN** returns the number of characters in a string.

Syntax

```
IMPORT STRING.LI*          '*.LI9 for T9, *.LIA for T10
ret_val = STRLEN(text1[])
```

Parameters

text1[]	String whose length is determined	<div>ARRAY</div> <div>STRING</div>
ret_val	Number of characters in the string	<div>LONG</div>

See also

+ String Addition, ASC, CHR, FLOTOSTR, LNGTOSTR, STRCOMP, STRLEFT, STRMID, STRRIGHT, VALF, VALI

Example

```
IMPORT STRING.LI9
DIM text1[50] AS STRING

INIT:
    text1 = "MEGA real-time with ADwin systems"

EVENT:
    PAR_1 = STRLEN(text1) 'String length: PAR_1 = 33
```

STRMID

The function **STRMID** returns a specified number of characters from a string, starting from a certain position in the string, into a second string.

Syntax

```
IMPORT STRING.LI*          '*.LI9 for T9, *.LIA for T10
STRMID(text1[], start, length, text2[])
```

Parameters

<code>text1[]</code>	String from which it is copied	ARRAY STRING
<code>start</code>	Position of the first character which is copied	LONG
<code>length</code>	Number of characters to be copied	LONG
<code>text2[]</code>	String into which it is copied	ARRAY STRING

See also

+ String Addition, ASC, CHR, FLOTOSTR, LNGTOSTR, STRCOMP, STRLEFT, STRLEN, STRRIGHT, VALF, VALI

Example

```
IMPORT STRING.LI9

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_1 = text2[1]      'String-length = 20 characters
  PAR_2 = text2[2]      'ASCII-character 72h = "r"
  PAR_3 = text2[3]      'ASCII-character 65h = "e"
  PAR_4 = text2[4]      'ASCII-character 61h = "a"
  PAR_5 = text2[5]      'ASCII-character 6Ch = "l"
  PAR_6 = text2[6]      'ASCII-character 2Dh = "-"
  PAR_7 = text2[7]      'ASCII-character 74h = "t"
  PAR_8 = text2[8]      'ASCII-character 69h = "i"
  PAR_9 = text2[9]      'ASCII-character 6Dh = "m"
  PAR_10 = text2[10]     'ASCII-character 65h = "e"
  PAR_11 = text2[11]     'ASCII-character 20h = " "
  PAR_12 = text2[12]     'ASCII-character 77h = "w"
  PAR_13 = text2[13]     'ASCII-character 69h = "i"
  PAR_14 = text2[14]     'ASCII-character 74h = "t"
  PAR_15 = text2[15]     'ASCII-character 68h = "h"
  PAR_16 = text2[16]     'ASCII-character 20h = " "
  PAR_17 = text2[17]     'ASCII-character 41h = "A"
  PAR_18 = text2[18]     'ASCII-character 44h = "D"
  PAR_19 = text2[19]     'ASCII-character 77h = "w"
  PAR_20 = text2[20]     'ASCII-character 69h = "i"
  PAR_21 = text2[21]     'ASCII-character 6Eh = "n"
  PAR_22 = text2[22]     'End of string sign = 0
```

STRRIGHT

The function **STRRIGHT** returns a specified number of characters from the right end of a string into a second string.

Syntax

```
IMPORT STRING.LI*          '*.LI9 for T9, *.LIA for T10
STRRIGHT(text1[], length, text2[])
```

Parameters

<code>text1[]</code>	String from which it is copied	<div>ARRAY</div> <div>STRING</div>
<code>length</code>	Number of the characters to copy	<div>LONG</div>
<code>text2[]</code>	String into which it is copied	<div>ARRAY</div> <div>STRING</div>

See also

+ String Addition, ASC, CHR, FLOTOSTR, LNGTOSTR, STRCOMP, STRLEFT, STRLEN, STRMID, VALF, VALI

Example

```
IMPORT STRING.LI9

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_1 = text2[1]      'String-length = 13 characters
  PAR_2 = text2[2]      'ASCII-character 41h = "A"
  PAR_3 = text2[3]      'ASCII-character 44h = "D"
  PAR_4 = text2[4]      'ASCII-character 77h = "w"
  PAR_5 = text2[5]      'ASCII-character 69h = "i"
  PAR_6 = text2[6]      'ASCII-character 6Eh = "n"
  PAR_7 = text2[7]      'ASCII-character 2Dh = "-"
  PAR_8 = text2[8]      'ASCII-character 53h = "S"
  PAR_9 = text2[9]      'ASCII-character 79h = "y"
  PAR_10 = text2[10]     'ASCII-character 73h = "s"
  PAR_11 = text2[11]     'ASCII-character 74h = "t"
  PAR_12 = text2[12]     'ASCII-character 65h = "e"
  PAR_13 = text2[13]     'ASCII-character 6Dh = "m"
  PAR_14 = text2[14]     'ASCII-character 73h = "s"
  PAR_15 = text2[15]     'End of string sign = 0
```

SUB ... ENDSUB

The **SUB...ENDSUB** commands are used to define a subroutine macro with passed parameters.

Syntax

```
SUB name({val_1, val_2, ...})
    DIM var AS <VAR_TYPE>
    ...
    'Instruction block
ENDSUB
```

Parameters

name	Name of the subroutine	
val_1, val_2	Name of the passed parameter (for arrays use the syntax with dimension brackets: <code>array[]</code> or <code>DATA_n[] []</code>)	<div style="border: 1px solid black; padding: 2px; display: inline-block;">FLOAT</div> <div style="border: 1px solid black; padding: 2px; display: inline-block;">LONG</div>

Notes

You will find general information about macros in chapter 3.5.1.

This instruction defines a subroutine-macro, which means the whole instruction block of the subroutine 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:

1. In front of the section **INIT** : / **LOWINIT** :
2. After the section **FINISH** :
3. In a separate file which you include with the instruction **#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.

- no values can be assigned to passed parameters. Use them in the subroutine as constants.

All instructions between **SUB** and **ENDSUB** belong to the subroutine.

.A subroutine is called with its name and with all its arguments 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 you do calculations with passed parameters inside a subroutine, be sure to set the passed parameters in parentheses to avoid problems with precedence rules (e.g. BODMAS).

See also

#INCLUDE, FUNCTION ... ENDFUNCTION, LIB_SUB ...
LIB_ENDSUB, LIB_FUNCTION ... LIB_ENDFUNCTION

Example

```
SUB Fast_Dac1(val1)
    REM Outputs val1 on the 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:

```
Fast_Dac1(NewValue)
```

The same subroutine with an array as passed parameter:

```
SUB Fast_Dac1(array[]) AS FLOAT
    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
ENDFUNCTION
```

Calling this subroutine is made in a similar manner (but *without* dimension brackets):

```
Fast_Dac1(array)
```

For `array` you can indicate a global or a local array. Enter the array name only, without element number and brackets.

TAN

The function **TAN** returns the tangent of an argument.

Syntax

```
ret_val = TAN(angle)
```

Parameters

angle	Arc angle ($-\pi/2 \dots \pi/2$)	FLOAT
ret_val	Cosine of the angle (-1...1)	FLOAT

Notes

If you use input values which are not in the range of $-\pi/2 \dots +\pi/2$, the calculation error grows with the increasing value.

The execution time of the instruction is 1.33 μ s (T10: 0.67 μ s).

See also

SIN, COS, ARCSIN, ARCCOS, ARCTAN

Example

```
DIM val1, val2 AS FLOAT
```

```
EVENT:
```

```
val1 = 5.3
```

```
val2 = TAN(val1)      'Result: val2 = -1.50...
```

TRACE_MODE_PAUSE

The instruction **TRACE_MODE_PAUSE** disables the trace mode.

Syntax

TRACE_MODE_PAUSE

Notes

The instruction **TRACE_MODE_PAUSE** disables the trace mode from within an *ADbasic* program. With **TRACE_MODE_RESUME** the trace mode is enabled again. The disabling/enabling concerns trace-active program lines only, which are marked with a ?.

Both instructions allow to enable or disable the trace mode for certain program lines or program sections. Therefore the trace mode can be activated e.g. as long as a specified condition is fulfilled.

The settings for the trace mode options is described on under *Enable Timing Analyzer Option*; More information about applications can be found in chapter 4.3.3.

See also

TRACE_MODE_RESUME

Example

```
EVENT:
  PAR_1 = ADC(1,4)
  IF (PAR_1 > 32768) THEN
    TRACE_MODE_RESUME    'Trace mode enabled

    ...                  'For this program section the trace
    ...                  'mode is continuously activated

    TRACE_MODE_PAUSE    'Trace mode disabled
  ENDIF
```

TRACE_MODE_RESUME

The instruction **TRACE_MODE_RESUME** activates the trace mode beginning in the next program line.

Syntax

TRACE_MODE_RESUME

Notes

The instruction **TRACE_MODE_RESUME** enables the trace mode again after it has been disabled with **TRACE_MODE_PAUSE**. The disabling/enabling concerns trace-active program lines only, which are marked with a ?.

Both instructions allow to enable or disable the trace mode for certain program lines or program sections. Therefore the trace mode can be activated e.g. as long as a specified condition is fulfilled.

The settings for the trace mode options is described on [under Enable Timing Analyzer Option](#); More information about applications can be found in chapter 4.3.3.

See also

TRACE_MODE_PAUSE

Example

```
EVENT:
  PAR_1 = ADC(1,4)
  IF (PAR_1 > 32768) THEN
    TRACE_MODE_RESUME    'Trace mode enabled

    ...                  'For this program section the trace
    ...                  'is continuously activated

    TRACE_MODE_PAUSE    'Trace mode disabled
  ENDIF
```

VALF

The function **VALF** converts a string into a floating point number.

Syntax

```
IMPORT STRING.LI*          '*.LI9 for T9, *.LIA for T10
ret_val = VALF(text[])
```

Parameters

text [] String which is to be converted, in the following **ARRAY** **STRING** format:

Mantissa (max. 7 characters)			Exponent (0...99)		
{+}	vvvvv	.	nnnnn	e	{+} nn
-		,		E	-

ret_val Generated floating point value **FLOAT**

Notes

If you do not indicate a sign, a positive sign will be assumed.

In the mantissa only a maximum of 7 characters (pre-decimal *and* decimal places) are stored. 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:

negative values: -3,402823E38 ... -1,175494E-38

positive values: +1,175494E-38 ... +3,402823E38

Values which are not in this range are interpreted as "infinite" or zero.

If you use illegal signs (signs other than those indicated in the format) only the strings up to the first illegal sign will be evaluated.

See also

+ String Addition, ASC, CHR, FLOTOSTR, LNGTOSTR, STRCOMP, STRLEFT, STRLEN, STRMID, STRRIGHT, VALI

Example

```
IMPORT STRING.LI9

DIM text[20] AS STRING

INIT:
  text="-271.8282E-02" 'String to be converted
  PAR_1 = text[1]      'String-length
  PAR_2 = text[2]      'ASCII-character 2Dh = "-"
  PAR_3 = text[3]      'ASCII-character 32h = "2"
  PAR_4 = text[4]      'ASCII-character 37h = "7"
  PAR_5 = text[5]      'ASCII-character 2Eh = "."
  PAR_6 = text[6]      'ASCII-character 31h = "1"
  PAR_7 = text[7]      'ASCII-character 34h = "4"
  PAR_8 = text[8]      'ASCII-character 31h = "1"
  PAR_9 = text[9]      'ASCII-character 35h = "5"
  PAR_10 = text[10]     'ASCII-character 39h = "9"
  PAR_11 = text[11]     'ASCII-character 45h = "E"
  PAR_12 = text[12]     'ASCII-character 2Dh = "-"
  PAR_13 = text[13]     'ASCII-character 31h = "1"
  PAR_14 = text[14]     'ASCII-character 30h = "0"
  PAR_15 = text[15]     'End of string sign

EVENT:
  FPAR_1 = VALF(text)   'Convert string to float
```

VALI

The function **VALI** converts a string into an integer number (Long).

Syntax

```
IMPORT STRING.LI*          '*.LI9 for T9, *.LIA for T10
ret_val = VALI(text)
```

Parameters

<code>text []</code>	String to be converted in the format: Sign: + (optional) or - Pre-decimal places: max. 10 characters	ARRAY STRING
	<hr/> <div style="display: flex; justify-content: space-between;"> {+} vvvvvvvvvv </div> <hr/> <div style="display: flex; justify-content: space-between;"> - </div> <hr/>	
<code>ret_val</code>	Generated long value	LONG

Notes

If you do not indicate a sign, a positive sign will be assumed.

Please note the value range for long values:

-2.147.483.648 to +2.147.483.647

Values outside this range are interpreted as zero.

If you use illegal characters (characters other than indicated in the format) the string up to the first illegal characters will be evaluated only.

See also

+ String Addition, ASC, CHR, FLOTOSTR, LNGTHOSTR, STRCOMP, STRLEFT, STRLEN, STRMID, STRRIGHT, VALF

Example

```
IMPORT STRING.LI9

DIM text[20] AS STRING

INIT:
  text="-1234567890"      'String to be converted
  PAR_1 = text[1]         'String-length = 11
  PAR_2 = text[2]         'ASCII-character 2Dh = "-"
  PAR_3 = text[3]         'ASCII-character 31h = "1"
  PAR_4 = text[4]         'ASCII-character 32h = "2"
  PAR_5 = text[5]         'ASCII-character 33h = "3"
  PAR_6 = text[6]         'ASCII-character 34h = "4"
  PAR_7 = text[7]         'ASCII-character 35h = "5"
  PAR_8 = text[8]         'ASCII-character 36h = "6"
  PAR_9 = text[9]         'ASCII-character 37h = "7"
  PAR_10 = text[10]       'ASCII-character 38h = "8"
  PAR_11 = text[11]       'ASCII-character 39h = "9"
  PAR_12 = text[12]       'ASCII-character 30h = "0"
  PAR_13 = text[13]       'End of string sign

EVENT:
  PAR_20 = VALF(text)     'Convert string to long
```

XOR

The operator **XOR** (Exclusive-Or) combines two integer values bitwise.

Syntax

```
... val_1 XOR val_2 ...
```

Parameters

val_1, val_2 Integer value

LONG

Notes

See also

AND, NOT, OR

Example

```
DIM value AS LONG
```

```
EVENT:
```

```
value = 0100b XOR 0110b'Result: value = 4 XOR 5 = 0010b = 2
```

6.3 ADwin-Gold and ADwin-light-16

Use the following instructions only with the systems *ADwin-Gold* and *ADwin-light-16*, even if some of the instructions for the *ADwin-Pro* system are the same or similar.

Use the instructions of this section without an include file.

For *ADwin-light-16* (basic version) and the add-on *ADwin-light-16-CO1* there are additional counter instructions in chapter 6.4:

- CNT_CLEAR, page 223
- CNT_ENABLE, page 227
- CNT_LATCH, page 234
- CNT_READ, page 238
- CNT_READLATCH, page 240

ADC

The instruction **ADC** measures the voltage of an analog input via the 16-bit converter and returns the measurement result as corresponding digital value, multiplied by a gain factor if one is specified.

For the 12-/14-bit converter of the *ADwin-Gold* system use the instruction **ADC12**.

Syntax

```
ret_val = ADC(input_ch{,gain})
```

Parameters

<code>input_ch</code>	number of the analog input channel Gold: 1...16; L16: 1, 3, 5, ..., 15	LONG
<code>gain</code>	gain factor (1, 2, 4, 8)	LONG CONST
<code>ret_val</code>	measurement value in digits (0...65535)	LONG

Notes

The instruction **ADC** is a combination of consecutive functions:

- Set the multiplexer to the specified input channel (**SET_MUX**).
- Wait for the settling of the multiplexer
- Start the measurement: Convert the analog signal using the 16-bit converter - considering the gain factor - to a digital value (**START_CONV**).
- Wait for the end of conversion (**WAIT_EOC**).
- Read out the digital value from the register and return it. (**READADC**).

The execution time for the instruction depends on the system you use. You will find Information about the multiplexer settling time and the conversion time in the hardware documentation of your system.

If you set the process cycle time (**GLOBALDELAY**) to a value less than 20 μ s, the execution time of the instruction is only half as long. This is possible, because the compiler skips the waiting time for the settling of the multiplexer. It is assumed that you want to execute a measurement without setting the multiplexer.

If (at such short cycle times) you require the first measurement to be

correct, you have to set the multiplexer to the specified input channel prior to using the instruction **ADC** with **SET_MUX** for the first time. This time has to be at least as long as the multiplexer settling time.

In the following examples the instructions **SET_MUX**, **START_CONV**, **WAIT_EOC** and **READADC** should be used instead of **ADC** in the following cases:

- Very short cycle times: **GLOBALDELAY** < 240 (s.a.).
- High internal resistance (>3kΩ) of the voltage source of the measurement signal: This increases the settling time of multiplexer.
- You want to use inevitable waiting times for additional program tasks.

If you indicate a non-existing input channel the measurement result will be undefined.

The measurement range depends on the gain factor:

Gain factor	Input voltage range	Measurement range
1	-10V ... 10V	20V
2	-5V ... 5V	10V
4	-2.5V ... 2.5V	5V
8	-1.25V ... 1.25V	2.5V

With the following formula you can calculate the measured voltage from the returned digital value.

$$\text{Voltage} = (\text{Digits} - 32768_{\text{bipolar}}) \cdot \frac{\text{measurement range}}{65536}$$

The following values, shown in the table below, apply in case you have chosen a gain of 1 (measurement range of 20 Volt):

Measurement range	Return value of ADC			1 Digit is
	0	32768	65535	
20V	-10V	0V	+9.999695V	305.175µV

See also

ADC12, SET_MUX, START_CONV, WAIT_EOC, READADC

Example

```
DIM iw AS LONG           'Declaration

EVENT:
  'Measure analog input 1 with gain of 4
  iw = ADC(1,4)
  'Write measurement value into global variable, so
  'that the computer can read it
  PAR_1 = iw
```


ADC12

ADwin-Gold only: The instruction **ADC12** measures the voltage of an analog input via the 12-bit converter (revision A) or the 14-bit converter (revision B) and returns the measurement result as corresponding digital value, multiplied by a gain factor if one is specified.

For the 16-bit converter of the *ADwin-Gold* system use the instruction **ADC**.

Syntax

```
ret_val = ADC12(input_ch{,gain})
```

Parameters

<code>input_ch</code>	number of the analog input channel (1...16)	LONG
<code>gain</code>	gain factor(1, 2, 4, 8)	LONG
<code>ret_val</code>	measurement result in digits (12-bit: 0, 16, 32, ..., 65520; 14-bit: 0, 4, 8, ..., 65532)	LONG

Notes

The instruction **ADC12** is a combination of consecutive functions:

- Set the multiplexer to the specified input channel (**SET_MUX**).
- Wait for the settling of the multiplexer.
- Start the measurement: Convert the analog value within the 12 or 14 bit converter - considering the gain factor - to a digital value (**START_CONV**).
- Wait for the end of conversion (**WAIT_EOC**).
- Read out the digital value from the register and return it (**READADC12**).

The execution time for the instruction depends on the system you use. You will find Information about the multiplexer settling time and the conversion time in the hardware documentation of your system.

The steps of 16 and 4 of the returned measurement values result from the fact that the 12-bit and 14-bit conversion results are returned each as a 16-bit value: The bits 0 to 3 are always 0 (zero) with 12-bit converters and bits 0 and 1 with 14-bit converters.

In the following examples you should use the instructions **SET_MUX**, **START_CONV**, **WAIT_EOC** and **READADC12** instead of **ADC** in the following cases:

- Very short cycle times: **GLOBALDELAY** < 200: The instruction **ADC12** cannot be executed during the cycle time.
- High internal resistance (>3k Ω) of the voltage source of the measurement signal: This increases the settling time of multiplexer.
- You want to use inevitable waiting times for additional program tasks.

If you indicate a non-existing input channel the measurement result will be undefined.

The measurement range depends on the gain factor.

Gain	Input voltage range	Meas. range
1	-10 V ... 10 V	20V
2	-5 V ... 5 V	10V
4	-2.5 V ... 2.5 V	5V
8	-1.25 V ... 1.25 V	2.5V

With the following formula you can calculate the measured voltage from the returned digital value:

$$\text{Voltage} = (\text{Digits} - 32768_{\text{bipolar}}) \cdot \frac{\text{measurement range}}{65536}$$

The following values, shown in the table below, apply in case you have chosen a gain of 1 (measurement range of 20 Volt):

Measurement range	Return value of ADC12			1 Digit is
	0	32768	65535	
20V	-10V	0V	+9.99512V	4.88mV

See also:

ADC, SET_MUX, START_CONV, WAIT_EOC, READADC

Example

```
DIM iw AS LONG           'Declaration

EVENT:
  'Measure analog input 1 with a gain of 4
  iw = ADC12(1,4)
  'Write measurement value into global variable so that
  'the computer can read it.
  PAR_1 = iw
```

CLEAR_DIGOUT

The instruction **CLEAR_DIGOUT** sets one of the digital outputs to 0 (TTL low).

Syntax

CLEAR_DIGOUT (*output_no*)

Parameters

output_no Number which specifies the output to be deleted: **CONST**
LONG

<i>bit_n</i>	0	1	...	5	...	15
<i>ADwin-Gold</i>	DIO16	DIO17	...	DIO21	...	DIO31
<i>ADwin-light-16</i>	0	1	...	5	–	–

Notes

If you want to specify the output to be deleted using a variable, use the instruction **DIGOUT_WORD**.

This instruction requires that you configure the relevant channel as output. Otherwise the instruction has no effect.

With the instruction **CONF_DIO** you can configure the digital channels in groups of 8 inputs or outputs. We recommend the digital channels be configured with **CONF_DIO** (1100b): Channels 0...15 as inputs, channels 16...31 as outputs.

The instructions clears a bit in the output register of the channels DIO16...DIO31. Therefore a TTL low is set at the corresponding channel, as long as it has been defined as output.

If you want to set one of the channels 0...15 to 0, clear the corresponding bit in the output register of the channels DIO0...DIO15 (note: Configure the channel as output first). Follow these steps (see example below):

- Read out the register with **PEEK**. You will find the register number in the hardware manual.
- Clear the bit belonging to the channel (**AND** masking).
- Write the value back into the register with **POKE**.

See also

CONF_DIO, DIGOUT_WORD, SET_DIGOUT

Example

```
DIM val AS LONG           'Declaration

INIT:
    SET_DIGOUT(0)          'Set digital output DIO16 to 0

EVENT:
    val = ADC(1)           'Measurement data acquisition
    IF (val > 3000) THEN
        CLEAR_DIGOUT(0)    'Clear dig. output DIO16/0
    ENDIF
```

ADwin-Gold only: A subroutine which sets a single bit of the DIO lines 0...15 to 0 could be as follows:

```
SUB CLEAR_DIGOUT_CONN1(bitno)
    POKE(204001C0h,
    PEEK(204001C0h) AND NOT(SHIFT_LEFT(1,bitno)) )
ENDSUB
```

CONF_DIO

ADwin-Gold only: The instruction **CONF_DIO** configures the 32 digital channels in groups of 8 as inputs or outputs.

Syntax

CONF_DIO(*val*)

Parameters

val Bit pattern that configures the digital channels **CONST**
as inputs or outputs: **LONG**
Bit=0: Channels as inputs
Bit=1: Channels as outputs

Bitno. in <i>val</i>	15...4	3	2	1	0
Channels	–	DIO31	DIO23	DIO15	DIO07
	
		DIO24	DIO16	DIO08	DIO00

Notes

The digital channels of the *ADwin-Gold* system are initially configured as inputs after power-up (and cannot be used as outputs). They can only be configured in groups of 8 as inputs or outputs.

We recommend the use of the configuration **CONF_DIO**(1100b), which specifies DIO00...DIO15 as inputs and DIO16...DIO31 as outputs.

The instructions **CLEAR_DIGOUT**, **SET_DIGOUT**, **DIGIN_WORD**, **DIGOUT_WORD**, **DIGIN** are dependent on this configuration; a different configuration can interfere with or prevent the proper operation of these commands.

If you use a configuration other than the recommend configuration, you can only set and process the digital channels if you read out or write into the corresponding hardware registers with **PEEK** and **POKE** commands (see *ADwin-Gold* hardware manual).

It is recommended that you use the binary representation (suffix "b"). It shows the allocation of bits to channel groups more clearly than decimal or hexadecimal representations which can still be used if desired.

See also

CLEAR_DIGOUT, DIGIN, DIGIN_WORD, DIGOUT_WORD,
SET_DIGOUT

Example

```
'Configure DIO00...DIO15 as inputs  
'and DIO16...DIO31 as outputs  
CONF_DIO(1100b)
```

DAC

The instruction **DAC** outputs a defined voltage on a specified analog output.

Syntax

```
DAC (num, val)
```

Parameters

num	Number of the analog output (1...8)	LONG
val	Value in digits, which defines the voltage to be output (0...65535).	LONG

Notes

If you specify a value which is beyond the permissible value range, it will automatically be set to the system-specific minimum or maximum value.

Example

```
REM Digital proportional controller
DIM set_to, gain, diff, out AS LONG'Declaration

EVENT:
    set_to = PAR_1          'Setpoint
    gain = PAR_2            'Dimension
    diff = set_to - ADC(1) 'Calculate control deviation
    out = diff * gain       'Calculate actuating value
    DAC(1, out)            'Output of the actuating value
```


DIGIN

The instruction **DIGIN** returns the value of one of the digital inputs DIO00...DIO15.

Syntax

```
ret_val = DIGIN(channel_no)
```

Parameters

channel_no Number which specifies the input to be queried:

LONG

ADwin-Gold:

channel_no	0	1	...	14	15
Input No.	DIO00	DIO01	...	DIO14	DIO15

ADwin-light-16:

channel_no	0	1	...	5
Input No.	0	1	...	5

ret_val 1: TTL-level high
 0: TTL-level low

Notes

This instruction fits best for the reading of few bits. If several bits are to be read (e.g. in a loop), the usage of the instruction **DIGIN_WORD** is definitely quicker. Please remember this for time-critical applications in particular.

The following notes refer to *ADwin-Gold* only:

The instruction requires that you configure the relevant channel as input. If the channel is configured as output it will return an irrelevant value.

The instruction **CONF_DIO** can be used to configure the digital channels as inputs or outputs in groups of 8. We recommend that you configure using **CONF_DIO** (1100b) which specifies: Channels 0...15 as inputs and channels 16...31 as outputs.

If you need the value of one of the channels DIO16...DIO31, then read out the corresponding bit from the input register of these channels. These channels must be configured as inputs first. Follow these steps (see 2nd example `DIGIN_CONN2`):

- Read out the register with `PEEK`. The register number can be found in the hardware manual.
- Clear all bits except the one belonging to the channel (`AND`-masking).

See also

`CONF_DIO` (*ADwin-Gold* only), `DIGIN_WORD`, `DIGOUT_WORD`

Example

```
REM Example for Gold and L16
DIM DATA_1[10000] AS LONG AS FIFO

EVENT:
  'Is digital input 0 set?
  IF (DIGIN(0) = 1) THEN
    DATA_1 = ADC(1)      'Measurement data acquisition
  ENDIF
```

ADwin-Gold only: A function returning the value of one of the channels DIO16...DIO31 could be as follows:

```
FUNCTION DIGIN_CONN2(bitno) AS LONG
  DIGIN_CONN2=SHIFT_RIGHT(PEEK(204001B0h), bitno) AND 1
ENDFUNCTION
```

DIGIN_WORD

The instruction **DIGIN_WORD** returns the values of all digital inputs at the same time.

Syntax

```
ret_val = DIGIN_WORD()
```

Parameters

ret_val Bit pattern that corresponds to the TTL-levels LONG at the digital inputs (allocation s.b.).
 1: TTL-level high
 0: TTL-level low

ADwin-Gold:

Bit number in	31 ... 16	15	14	...	1	0
ret_val						
Input No.	–	DIO15	DIO14	...	DIO01	DIO00

ADwin-light-16:

Bit number in	31 ... 6	5	...	0
ret_val				
Input No.	–	5	...	0

Notes (ADwin-Gold only)

This instruction requires that you have configured the channels DIO00...DIO15 as inputs. If these channels are configured as output channels, no useful value is returned.

With the instruction **CONF_DIO** you can configure the digital channels as inputs or outputs in groups of 8. We recommend that you configure them using **CONF_DIO**(1100b) which specifies: Channels 0...15 as inputs, channels 16...31 as outputs.

If you need the values of the channels DIO16...DIO31, read out the input register of these channels (please note: Configure the channels as outputs first); see also 2nd example [DIGIN_WORD_CONN2](#). These channels must be configured as inputs first. The register number can be found in the hardware manual. The bits in this return value are allocated to the channels as follows:

Bit No.	31...16	15	...	1	0
Input No.	–	DIO31	...	DIO17	DIO16

See also

CONF_DIO (*ADwin-Gold* only), DIGOUT_WORD

Example

```

REM Example for Gold and L16
DIM DATA_1[10000] AS LONG AS FIFO

EVENT:
'Querying if the inputs 0 and 1 are set
IF ((DIGIN_WORD() AND 11b) = 11b) THEN
    DATA_1 = ADC(1)      'Measurement data acquisition
ENDIF

```

ADwin-Gold only: A function which returns the value of the channels DIO16...DIO31, could be as follows:

```

FUNCTION DIGIN_WORD_CONN2() AS LONG
    DIGIN_WORD_CONN2=PEEK(204001B0h)
ENDFUNCTION

```

DIGOUT_WORD

The instruction **DIGOUT_WORD** sets with a bit pattern all digital outputs to defined TTL-levels.

Syntax

DIGOUT_WORD(*val*)

Parameters

val Bit pattern that corresponds to the TTL-levels LONG
 at the digital outputs (allocation s.b.).
 1: Set to TTL-level high
 0: Set to TTL-level low

ADwin-Gold:

Bit no. in <i>val</i>	31 ... 16	15	14	...	1	0
Output No.	–	DIO31	DIO30	...	DIO17	DIO16

ADwin-light-16:

Bit no. in <i>val</i>	31 ... 6	5	...	0
Output No.	–	5	...	0

Notes (ADwin-Gold only)

This instruction requires that you have configured the channels DIO16...DIO31 as outputs. Otherwise it has no effect.

With the instruction **CONF_DIO** you can configure the digital channels as inputs or outputs in groups of 8. We recommend that you configure using **CONF_DIO** (1100b) which specifies: Channels 0...15 as inputs, channels 16...31 as outputs.

If you want to set the outputs of the channels DIO16...DIO31, write the corresponding bit pattern to the output register of these channels (please note: Configure the channels as outputs first); see also 2nd example **DIGIN_WORD_CONN1**. The register number can be found in the hardware manual.

See also

CONF_DIO (*ADwin-Gold* only), DIGIN_WORD, CLEAR_DIGOUT, SET_DIGOUT

Example

```
REM Example for Gold and L16
DIM value AS LONG

INIT:
  'Configure inputs and output (for ADwin-Gold only)
  CONF_DIO(1100b)

EVENT:
  value = ADC(1)           'Measurement data acquisition
  IF (value > 3000) THEN 'Is the limit value exceeded?
    DIGOUT_WORD(101b)     'Set outputs 0 and 2, all other
                          'outputs are cleared!

  ENDIF
```

ADwin-Gold only: A subroutine setting the TTL-levels of the channels DIO00...DIO15, could be as follows:

```
SUB DIGOUT_WORD_CONN1(value)
  POKE(204001C0h,value)
ENDSUB
```

READADC

The instruction **READADC** returns a converted value from a 16-bit A/D-converter.

Syntax

```
ret_val = READADC(num)
```

Parameters

num	Number (1, 2) of the 16-bit converter to be read.	LONG
ret_val	Measurement value in digits which corresponds to the voltage at the converter's input.	LONG

Notes

When using an *ADwin-Gold* system you read out the converted values of the 12-bit or 14-bit A/D converter using the instruction **READADC12**.

See also

ADC, READADC12, SET_MUX, START_CONV, WAIT_EOC

Example

```
EVENT:
  'Set multiplexer: ADC1 to channel 3, ADC2
  'to channel 4 (without gain)
  SET_MUX(1001b)
  ...
  START_CONV(11b)
  WAIT_EOC(11b)
  PAR_1 = READADC(1)
  PAR_2 = READADC(2)
```

'Wait for MUX settling time
'Start conversion for both ADCs
'Wait for end of conversion
'Read value of ADC1
'Read value of ADC2

READADC12

ADwin-Gold only: The instruction **READADC12** returns a converted value from one of the two 12-bit/14-bit A/D converters.

Syntax

```
ret_val = READADC12 (num)
```

Parameters

num	Number (1, 2) of the 12-bit converter to be read	LONG
ret_val	Measurement value in digits, which corresponds to the voltage at the converter's input.	LONG

Notes

Read out the converted value of the 16-bit A/D converter with the instruction **READADC**.

The A/D converters (ADC) divide the measurement range of 20 Volts into equal steps (digits), these are 4096 digits with 12-bit ADC and 16384 with 14-bit ADC.

In order to make comparing these values to the measurement values of the 16-bit ADC's easier, the instruction **READADC12** returns the result "left-aligned" descending from bit 31; the bits 3...0 (12-bit ADC) or 1...0 (14-bit ADC) have always the value 0.

Therefore using the instructions **READADC** and **READADC12** to measure the same voltage always return the same result in bits 31...4 or 31...2.

See also

ADC12, SET_MUX, START_CONV, WAIT_EOC

Example

```
DIM val1, val2 AS LONG
```

```
EVENT:
```

```
'Set multiplexer: ADC12-1 to channel 3, ADC12-2  
'to channel 4 (without gain)  
SET_MUX(1001b)  
...                               'Wait for MUX settling time  
START_CONV(11000b)               'Start conversion for both ADCs  
WAIT_EOC(11000b)                 'Wait for end of conversion  
val1 = READADC12(1)              'Read value of ADC12-1  
val2 = READADC12(2)              'Read value of ADC12-2
```

SET_DIGOUT

The instruction **SET_DIGOUT** sets one of the digital outputs to 1 (TTL-level high).

Syntax

SET_DIGOUT(channelno)

Parameters

channelno Number which specifies the output to be set: CONST
LONG

channelno	0	1	...	5	...	15
<i>ADwin-Gold</i>	DIO16	DIO17	...	DIO21	...	DIO31
<i>ADwin-light-16</i>	0	1	...	5	–	–

Notes

This instruction fits best for the setting of few bits. If several bits are to be set (e.g. in a loop), the usage of the instruction **DIGOUT_WORD** is definitely quicker. Please remember this for time-critical applications in particular.

The following notes refer to *ADwin-Gold* only:

If you want to set the output using a variable, use the instruction **DIGOUT_WORD**.

This instruction requires that you have previously configured the corresponding channel as an output. Otherwise it performs no action. With the instruction **CONF_DIO** you can configure the digital channels as inputs or outputs in groups of 8. We recommend that you configure them using **CONF_DIO**(1100b) which specifies: Channels 0...15 as inputs, channels 16...31 as outputs.

This instruction sets one bit in the output register of the channels DIO16...DIO31. If you have set the corresponding channel as output it will generate a TTL-level high.

If you want to set one of the channels 0...15 to 1, set the corresponding bit in the output register of the channels DIO0...DIO15 using the **POKE**

command (note: Configure the channel as output first). Follow these steps (see 2nd example `SET_DIGOUT_CONN1`):

- Read out the register with **PEEK**. The register number can be found in the hardware manual.
- Set the bit belonging to the channel (**OR**-masking).
- Write the value with **POKE** into the register.

See also

`CONF_DIO` (*ADwin-Gold* only), `CLEAR_DIGOUT`, `DIGOUT_WORD`

Example

```
REM Example for Gold and L16
DIM val AS LONG

INIT:
  'Configure digital inputs/output (ADwin-Gold only)
  CONF_DIO(1100b)

EVENT:
  val = ADC(1)           'Measurement data acquisition
  IF (val > 3000) THEN
    SET_DIGOUT(0)        'Set digital output DIO16 / 0
  ENDIF
```

ADwin-Gold only: A subroutine which sets a single bit of the DIO-lines 0...15 to 1 could be as follows:

```
SUB SET_DIGOUT_CONN1(bitno)
  POKE(204001C0h, PEEK(204001C0h) OR SHIFT_LEFT(1,bitno) )
ENDSUB
```

SET_MUX

The instruction **SET_MUX** sets one or more A/D input multiplexers and (*ADwin-Gold* only) the corresponding gain for the specified measurement channel.

Syntax

SET_MUX(*pattern*)

Parameters

pattern Bit pattern for the allocation of measurement channels and gain LONG

Bitno.	9	8	7	6	5	4	3	2	1	0
	PGA 2		PGA 1		MUX 2			MUX 1		

PGA 1 / 2 *ADwin-Gold* only: With 2 bits (6...7 / 8...9) each you determine the gain factor of the multiplexer:

2 Bits PGA 1 / PGA 2

00: Factor 1

01: Factor 2

10: Factor 4

11: Factor 8

MUX 1 / 2 With 3 bits each (0...2 / 3...5) you determine the channel to which the multiplexer is set:

3 bits	MUX 2	MUX 1
000:	channel 2	channel 1
001:	channel 4	channel 3
010:	channel 6	channel 5
011:	channel 8	channel 7
100:	channel 10	channel 9
101:	channel 12	channel 11
110:	channel 14	channel 13
111:	channel 16	channel 15

Notes

Please consider that when setting the multiplexer to another channel a specified settling time is required. You should only start the conversion after this settling time has elapsed. Please use the necessary

settling time (as well as the conversion time) from the hardware documentation of your system.

It is preferable to use a binary code (suffix "b") for the bit pattern. This will make it easier to display the bit pattern than if you use a decimal or hexadecimal representation although it is still possible to use these.

See also

ADC, ADC12, READADC, READADC12, START_CONV, START_CONV

Example

To set the multiplexer of ADC1 to channel 5 and to gain 8 and at the same time the multiplexer of ADC2 to channel 10 and gain 2, you need the bit pattern: 0111011010b (decimal: 474).

```
DIM val AS LONG
```

```
EVENT:
```

```
    SET_MUX(0111100010b) 'Set multiplexer (s.a.)  
    'Wait here for the settling time of the multiplexer  
    'by inserting some instructions.  
    START_CONV(1)         'Start AD-conversion ADC1  
    WAIT_EOC(1)           'Wait for end of conversion of  
    'ADC1  
    val = READADC(1)      'Read value of ADC1
```

START_CONV

The instruction **START_CONV** is used to start the conversion of one or more A/D converters as well as all the D/A converters.

Syntax

START_CONV(pattern)

Parameters

pattern

Bit pattern that specifies which converters should be started (only bits 0...4 can be used).

CONST
LONG

Bit no.	4	3	2	1	0	Systems
ADC1, 16-bit	–	–	–	–	x	Gold, L16
ADC2, 16-bit	–	–	–	x	–	Gold
all DACs	–	–	x	–	–	Gold, L16
ADC1, 12-bit	–	x	–	–	–	Gold
ADC1, 14-bit						
ADC2, 12-bit	x	–	–	–	–	Gold
ADC2, 14-bit						

Notes

Please note that ADC1 and ADC2 can either be 12-bit, 14-bit or 16-bit analog-to-digital converters. For more information see your hardware manual.

Also note that you can only use constants as parameters, variables are not allowed as an argument.

It is preferable to use a binary code (suffix "b") for the bit pattern. This will make it easier to display the bit pattern than if you use a decimal or hexadecimal representation although it is still possible to use these.

See also

ADC, ADC12, READADC, READADC12, SET_MUX, WAIT_EOC

Example

```
DIM val1 AS LONG
```

```
EVENT:
```

```
SET_MUX(0)           'Set multiplexer to channel 1
'Bypass the settling time with command lines
START_CONV(1)         'Start ADC1 A/D-conversion
WAIT_EOC(1)           'Wait for end of conversion
val1 = READADC(1)     'Read out value
```

WAIT_EOC

The instruction **WAIT_EOC** waits for the end of the conversion cycle of a specified A/D-converter.

Syntax

WAIT_EOC (*pattern*)

Parameters

pattern

Bit pattern that specifies which converters are to be waited for (only bits 0...4 can be used).

CONST

LONG

Bit no.	4	3	2	1	0	Systems
ADC1, 16-bit	–	–	–	–	x	Gold, L16
ADC2, 16-bit	–	–	–	x	–	Gold
ADC1, 12/14-bit	–	x	–	–	–	Gold
ADC2, 12/14-bit	x	–	–	–	–	Gold

Notes

If you set more than one of the bits, you have to wait for the conversion to finished for all of the relevant ADCs.

Always select the bits of existing ADCs. Otherwise the communication in a high-priority process between *ADwin* system and computer will be interrupted.

See also

ADC, ADC12, READADC, READADC12, SET_MUX, START_CONV

Example

```
DIM val AS LONG
```

```
EVENT:
```

```
SET_MUX(001000b)      'Set MUX of ADC2 to channel 4  
'Bypass the settling time of the multiplexer with  
'command lines  
START_CONV(2)          'Start A/D-conversion ADC2  
WAIT_EOC(2)            'Wait for end of conversion at 'ADC2  
val = READADC(2)       'Read out value
```


6.4 ADwin-light-16 DIO1 / ADwin-Gold CO1

The instructions of this section are divided into groups:

- **counter instructions** (**CNT_...**; page 223)
for *ADwin-light-16* (basic, CO1, DIO1) and *ADwin-Gold* (CO1).

The counters are numbered ascending from 1. Some instructions use a bit pattern where counters are allocated to bits as is illustrated below:

Bit no.	31...4	3	2	1	0
Counter no.	–	4 ^a	3 ^a	2 ^b	1

a. for *ADwin-Gold* CO1 only

b. not with *ADwin-light-16* CO1 add-on.

Use for the bit pattern preferably the binary code (suffix "b"). The indication "10b" or "1100b" illustrates more clearly which counter is accessed and which is not, than in decimal or hexadecimal code which you may of course equally use.

- **digital channel instructions** (**DIG_...**; page 250)
applicable for *ADwin-light-16* DIO1 only.
- **CAN bus instructions** (page 259)
applicable for *ADwin-light-16* DIO1 only.

Inside these groups instructions be sorted alphabetically.

Please keep in mind to include the relevant include file for each system.

Instructions in this section

The instructions in this section are valid for the following *ADwin* systems:

Instruction	Gold	L16		
	CO1	Basis	CO1	DIO1
CNT_CLEAR (page 221)	x	x	x	x
CNT_CLEARENABLE (page 225)				x
CNT_ENABLE (page 227)	x	x	x	x
CNT_GETSTATUS (page 229)	x			x
CNT_INPUTMODE (page 232)	x			x
CNT_LATCH (page 234)	x	x	x	x

Instruction	Gold CO1	L16		
		Basis	CO1	DIO1
CNT_MODE (page 236)	x			x
CNT_READ (page 238)	x	x	x	x
CNT_READLATCH (page 240)	x	x	x	x
CNT_READFLATCH (page 242)	x			x
CNT_RESETSTATUS (page 244)	x			
CNT_SE_DIFF (page 246)	x			
CNT_SET (page 248)	x			x
CONF_DIO_E (page 250 ff)				x
DIGIN_WORD1_E, DIGIN_WORD2_E DIGOUT_SET1_E, DIGOUT_SET2_E, DIGOUT_RESET1_E, DIGOUT_RESET2_E DIGOUT_WORD1_E, DIGOUT_WORD2_E				
INIT_CAN (page 259 ff) EN_INTERRUPT, EN_RECEIVE, EN_TRANSMIT CAN_MSG, READ_MSG, TRANSMIT SET_CAN_BAUDRATE, GET_CAN_REG, SET_CAN_REG				x

CNT_CLEAR

The instruction **CNT_CLEAR** sets one or more counters to zero, according to the bit pattern in *pattern*.

Syntax

```
#INCLUDE ADWGCNT.INC      'ADwin-Gold only
#include ADWL16.INC        'ADwin-light-16 only

CNT_CLEAR(pattern)
```

Parameters

pattern Bit pattern LONG
 Bit = 0: no influence
 Bit = 1: set counter to zero

Bit no.	31...4	3	2	1	0
Counter no.	—	4 ^a	3 ^a	2 ^b	1

a. for *ADwin-Gold* CO1 only

b. not with *ADwin-light-16* CO1 add-on.

Notes

After the instruction has been executed the bit pattern is automatically reset to 0 (zero), so the counters start counting from 0.

See also

CNT_CLEARENABLE, CNT_ENABLE, CNT_GETSTATUS, CNT_INPUTMODE, CNT_LATCH, CNT_MODE, CNT_READ, CNT_READLATCH, CNT_READFLATCH, CNT_RESETSTATUS, CNT_SE_DIFF, CNT_SET

Example

```

#INCLUDE ADWGCNT.INC      'ADwin-Gold only
#INCLUDE ADWL16.INC      'ADwin-light-16 only

DIM old_1, new_1 AS LONG 'Dimension
DIM old_2, new_2 AS LONG 'the variables

INIT:
  old_1 = 0                'Initialize
  old_2 = 0                'the variables
  CNT_MODE(0)              'All counters on external clock input
  CNT_SET(11b)             'counters 1+2 with clock (CLK) and
                           'direction (DIR) input
  CNT_INPUTMODE(0)        'Determine functionality CLR/LATCH:
                           'All as CLR input
  CNT_CLEARENABLE(11b)    'Enables the CLR function of
                           'counters 1+2
  CNT_CLEAR(11b)          'Reset counters 1+2 to 0
  CNT_ENABLE(11b)        'Start counters 1+2

EVENT:
  CNT_LATCH(11b)          'Latch counters 1+2 simultaneously
  new_1 = CNT_READLATCH(1) 'read out Latch A counter 1 and...
  new_2 = CNT_READLATCH(2) 'Latch A counter 2.
  PAR_1 = new_1 - old_1    'Calculate the difference
                           '(f = impulses / time)
  PAR_2 = new_2 - old_2    '  "-
  old_1 = new_1            'Save new counter values as old
  old_2 = new_2            '  "-

```

CNT_CLEARENABLE

L16-DIO1 only: **CNT_CLEARENABLE** disables or enables the CLR input of one or more counters according to the bit pattern in *pattern*.

Syntax

```
#INCLUDE ADWL16.INC           'ADwin-light-16 only
CNT_CLEARENABLE(pattern)
```

Parameters

pattern

Bit pattern

LONG

Bit = 0: disable CLR input at the counter

Bit = 1: enable CLR input at the counter

Bit no.	31...2	1	0
Counter no.	–	2	1

Notes

This instruction affects all counters at the same time. It only works if the CLR mode is set by **CNT_INPUTMODE**.

Use this instruction only if the counter is disabled.

See also

CNT_CLEAR, CNT_ENABLE, CNT_GETSTATUS,
CNT_INPUTMODE, CNT_LATCH, CNT_MODE, CNT_READ,
CNT_READLATCH, CNT_READFLATCH, CNT_SET

Example

```

#INCLUDE ADWL16.INC

DIM old_1, new_1 AS LONG 'Dimension
DIM old_2, new_2 AS LONG ' the variables

INIT:
  old_1 = 0           'Initialize
  old_2 = 0           ' the vaiables
  CNT_MODE(0)         'All counters on external clock input
  CNT_SET(11b)        'counters 1+2 with clock (CLK) and
                      'direction (DIR) input
  CNT_INPUTMODE(0)    'Determine functionality CLR/LATCH:
                      'All with CLR input
  CNT_CLEARENABLE(11b) 'Enables the CLR-function of
                      'counters 1+2
  CNT_CLEAR(11b)      'Reset counters 1+2 to 0
  CNT_ENABLE(11b)     'Start counters 1+2

EVENT:
  CNT_LATCH(11b)      'Latch counters 1+2 at the same time
  new_1 = CNT_READLATCH(1) 'read out Latch A counter 1 and...
  new_2 = CNT_READLATCH(2) 'Latch A counter 2.
  PAR_1 = new_1 - old_1 'Calculate the difference
                      '(f = impulses / time)
  PAR_2 = new_2 - old_2 ' -"-
  old_1 = new_1         'Save new counter values as old
  old_2 = new_2         ' -"-

```


CNT_ENABLE

The instruction **CNT_ENABLE** disables or enables the counters set by *pattern*, to count incoming impulses.

Syntax

```
#INCLUDE ADWGCNT.INC      'ADwin-Gold only
#include ADWL16.INC       'ADwin-light-16 only

CNT_ENABLE(pattern)
```

Parameters

pattern Bit pattern LONG
 Bit = 0: stop counter
 Bit = 1: enable counter

Bit no.	31...4	3	2	1	0
Counter no.	—	4 ^a	3 ^a	2 ^b	1

a. for *ADwin-Gold* CO1 only

b. not with *ADwin-light-16* CO1 add-on.

See also

CNT_CLEAR, CNT_CLEARENABLE, CNT_GETSTATUS,
 CNT_INPUTMODE, CNT_LATCH, CNT_MODE, CNT_READ,
 CNT_READLATCH, CNT_READFLATCH, CNT_RESETSTATUS,
 CNT_SE_DIFF, CNT_SET

Example

```

#INCLUDE ADWGCNT.INC      'ADwin-Gold only
#INCLUDE ADWL16.INC      'ADwin-light-16 only

DIM old_1, new_1 AS LONG 'Dimension
DIM old_2, new_2 AS LONG ' the variables

INIT:
  old_1 = 0                'Initialize
  old_2 = 0                ' the variables
  CNT_MODE(0)              'All counters on external clock input
  CNT_SET(11b)             'Counters 1+2 with clock (CLK) and
                           'direction (DIR) inputs
  CNT_INPUTMODE(0)         'Determine functionality: At all
                           'counters as CLR-input
  CNT_CLEARENABLE(11b)    'Enables the CLR-function of
                           'counters 1+2
  CNT_CLEAR(11b)          'Reset counters 1+2 to 0
  CNT_ENABLE(11b)         'Start counters 1+2

EVENT:
  CNT_LATCH(11b)          'Latch counters 1+2 simultaneously
  new_1 = CNT_READLATCH(1) 'read out Latch A counter 1 and...
  new_2 = CNT_READLATCH(2) 'Latch A counter 2.
  PAR_1 = new_1 - old_1    'Calculate the difference
                           '(f = impulses / time)
  PAR_2 = new_2 - old_2    '-"-
  old_1 = new_1            'Save new counter values as old
  old_2 = new_2            '-"-

```

CNT_GETSTATUS

CNT_GETSTATUS reads out and returns the status register of the counter.

Syntax

```
#INCLUDE ADWGCNT.INC      'ADwin-Gold only
#include ADWL16.INC       'ADwin-light-16 only

ret_val = CNT_GETSTATUS(CounterNo)
```

Parameters

<code>CounterNo</code>	Counter number (L16-DIO1: 1...2, Gold-CO1: 1...4)	LONG
<code>ret_val</code>	Contents of the status register: In case of error, refer to the table for the meaning of the individual bits.	LONG

Table ADwin-Gold

Bit Nr.	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
Signal	-	-	-	-	-	-	-	-	N4	N3	N2	N1	-	-	-	-

Bit Nr.	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Signal	L4	C4	L3	C3	L2	C2	L1	C1	B4	A4	B3	A3	B2	A2	B1	A1

- : don't care (signal status is not defined (mask out with 0F 0F 00 33h))
 Ax: Signal A (signal is not changing states)
 Bx: Signal B (signal is not changing states)
 Cx: Correlation error (signals A and B are identical, they are not phase-shifted by approx. 90°)
 Lx: Line error (cable not connected or the line is broken)
 Nx: CLR-/LATCH-input (signal is not changing state)
 x: Counter number (1, 2, 3 or 4)

Table ADwin-light-16

Bit no.	31...28	27	26	25	24	23...20	19	18	17	16	15...06	05	04	03...02	01	00
Signal	-	L2	C2	L1	C1	-	B2	A2	B1	A1	-	N2	N1	-	R2	R1

- : don't care (signal status is not defined (mask out with 0F 0F 00 33h)

Ax: Signal A (signal is not changing states)

Bx: Signal B (signal is not changing states)

Cx: Correlation error* (signals A and B are identical, they are not phase-shifted by approx. 90°)

Lx: Line error* (cable not connected or the line is broken)

Nx: CLR-/LATCH-input (signal is not changing states)

Rx: Reset-Enable (value which was set by **CNT_CLEARENABLE**)

x: Counter number (1 or 2)

* Auto-Reset (is reset during reading out)

Notes

A line error (Lx) can only be detected at differential inputs! For TTL-inputs these bits are always 0.

ADwin-Gold only: The status register is not reset by reading it; use the instruction **CNT_RESETSTATUS** instead.

See also

CNT_CLEAR, CNT_CLEARENABLE, CNT_ENABLE,
CNT_INPUTMODE, CNT_LATCH, CNT_MODE, CNT_READ,
CNT_READLATCH, CNT_READFLATCH, CNT_RESETSTATUS,
CNT_SE_DIFF, CNT_SET

Example (ADwin-light-16 DIO1 only)

```

#INCLUDE ADWL16.INC
DIM error AS LONG

INIT:
    CNT_MODE(0)           'All counters at external clock input
    CNT_SET(0)            'All counters with A/B-input (for
                          'instance for incremental encoder)
    CNT_INPUTMODE(0)      'Determine functionality CLR/LATCH: At
                          'all counters as CLR-input
    CNT_CLEARENABLE(11b)  'Enables the CLR-function of
                          'counters 1+2
    CNT_CLEAR(11b)        'Reset counters 1+2 to 0
    CNT_ENABLE(1)         'Start counter 1
    error = 0             'Reset error indicator

EVENT:
    PAR_1 = CNT_READ(1)    'Read out counter 1
    PAR_2 = CNT_GETSTATUS(1) AND 0F0F0033h 'Read out status
                          'register counter 1
    IF (PAR_2 AND 2000000h = 2000000h) THEN 'Line or cable error
                          'counter 1?
        INC PAR_3          'Number of line or cable errors until
                          'now...
        error = 1          'Set error indicator
    ENDIF
    IF (PAR_2 AND 1000000h = 1000000h) THEN 'Correlation error
                          'counter 1?
        INC PAR_4          'Number of correlation errors until
                          'now...
        error = 1          'Set error indicator
    ENDIF
    PAR_5 = SHIFT_RIGHT(PAR_2 AND 10h,4)
                          'current status of CLR-input
    PAR_6 = SHIFT_RIGHT(PAR_2 AND 10000h,16)
                          'current status of input A.
    PAR_7 = SHIFT_RIGHT(PAR_2 AND 20000h,17)
                          'current status of input B.

```

CNT_INPUTMODE

The instruction **CNT_INPUTMODE** sets the function of the CLR/LATCH input of one or more counters.

Syntax

```
#INCLUDE ADWGCNT.INC      'ADwin-Gold only
#include ADWL16.INC       'ADwin-light-16 only

CNT_INPUTMODE(pattern)
```

Parameters

<code>pattern</code>	Bit pattern	LONG
	Bit = 0: Set CLR-mode	
	Bit = 1: Set LATCH-mode	

Notes

Use this instruction only when the counter is not enabled.

See also

CNT_CLEAR, CNT_CLEARENABLE, CNT_ENABLE,
CNT_GETSTATUS, CNT_LATCH, CNT_MODE, CNT_READ,
CNT_READLATCH, CNT_READFLATCH, CNT_RESETSTATUS,
CNT_SE_DIFF, CNT_SET

Example

```

#INCLUDE ADWGCNT.INC      'ADwin-Gold only
#INCLUDE ADWL16.INC      'ADwin-light-16 only

DIM old_1, new_1 AS LONG 'Dimension...
DIM old_2, new_2 AS LONG 'variables

INIT:
  old_1 = 0                'Initialize...
  old_2 = 0                'variables
  CNT_MODE(0)              'All counters on external clock input
  CNT_SET(11b)             'Counters 1+2 with clock (CLK) and
                           'direction (DIR) input
  CNT_INPUTMODE(0)         'Determine functionality CLR/LATCH: As
                           'CLR-input at all counters
  CNT_CLEARENABLE(11b)    'Enables the CLR-function of
                           'counters 1+2
  CNT_CLEAR(11b)           'Reset counters 1+2 to 0
  CNT_ENABLE(11b)         'Start counters 1+2

EVENT:
  CNT_LATCH(11b)           'Latch counters 1+2 simultaneously
  new_1 = CNT_READLATCH(1) 'Read out latch A counter 1 and...
  new_2 = CNT_READLATCH(2) 'latch A counter 2.
  PAR_1 = new_1 - old_1    'Calculate the difference
                           '(f = impulses / time)
  PAR_2 = new_2 - old_2    ' -"-
  old_1 = new_1            'Save new counter values as old
  old_2 = new_2            ' -"-

```

CNT_LATCH

The instruction **CNT_LATCH** transfers the current counter values of one or more counters into the relevant Latch A, depending on the bit pattern in **pattern**.

Syntax

```
#INCLUDE ADWGCNT.INC      'ADwin-Gold only
#include ADWL16.INC       'ADwin-light-16 only

CNT_LATCH(pattern)
```

Parameters

pattern Bit pattern LONG
 Bit = 0: no function
 Bit = 1: transfer counter values into Latch A

Bit no.	31...4	3	2	1	0
Counter no.	—	4 ^a	3 ^a	2 ^b	1

a. for *ADwin-Gold* CO1 only

b. not with *ADwin-light-16* CO1 add-on.

Notes

After the instruction has been executed the bit pattern is automatically reset to 0 (zero).

Latch A is read out into a variable with **CNT_READLATCH** command.

See also

CNT_CLEAR, CNT_CLEARENABLE, CNT_ENABLE,
 CNT_GETSTATUS, CNT_INPUTMODE, CNT_MODE, CNT_READ,
 CNT_READLATCH, CNT_READFLATCH, CNT_RESETSTATUS,
 CNT_SE_DIFF, CNT_SET

Example

```

#INCLUDE ADWGCNT.INC      'ADwin-Gold only
#INCLUDE ADWL16.INC      'ADwin-light-16 only

DIM old_1, new_1 AS LONG 'Dimension...
DIM old_2, new_2 AS LONG 'the variables

INIT:
  old_1 = 0                'Initialize
  old_2 = 0                'the variables
  CNT_MODE(0)              'All counters on external clock input
  CNT_SET(11b)             'Counters 1+2 with clock (CLK) and
                           'direction (DIR) input
  CNT_INPUTMODE(0)         'Determine functionality CLR/LATCH: As
                           'CLR-input at all counters
  CNT_CLEARENABLE(11b)    'Enables the CLR-function of
                           'counters 1+2
  CNT_CLEAR(11b)          'Reset counters 1+2 to 0
  CNT_ENABLE(11b)         'Start counters 1+2

EVENT:
  CNT_LATCH(11b)           'Latch counters 1+2 simultaneously
                           'and then...
  new_1 = CNT_READLATCH(1) 'read out Latch A counter 1 and...
  new_2 = CNT_READLATCH(2) 'Latch A counter 2.
  PAR_1 = new_1 - old_1    'Calculate the difference
                           ' (f = impulses / time)
  PAR_2 = new_2 - old_2    '-''-
  old_1 = new_1            'Save new counter values as old
  old_2 = new_2            '-''-

```

CNT_MODE

The instruction **CNT_MODE** defines the operating mode of all counters by selecting which clock input they use according to the bit pattern in **pattern**.

Syntax

```
#INCLUDE ADWGCNT.INC      'ADwin-Gold only
#include ADWL16.INC       'ADwin-light-16 only

CNT_MODE(pattern)
```

Parameters

pattern Bit pattern LONG
 Bit = 0: external clock input (CLK/DIR or A/B)
 Bit = 1: internal clock input (5 MHz or 20 MHz)

Bit no.	31...4	3	2	1	0
Counter no.	–	4 ^a	3 ^a	2	1

a. for ADwin-Gold CO1 only

Notes

Determine the mode of the selected clock input with **CNT_SET**.

Please use this instruction only when the counter is disabled.

See also

CNT_CLEAR, CNT_CLEARENABLE, CNT_ENABLE,
 CNT_GETSTATUS, CNT_INPUTMODE, CNT_LATCH, CNT_READ,
 CNT_READLATCH, CNT_READFLATCH, CNT_RESETSTATUS,
 CNT_SE_DIFF, CNT_SET

Example

```

#INCLUDE ADWGCNT.INC      'ADwin-Gold only
#INCLUDE ADWL16.INC      'ADwin-light-16 only

DIM old_1, new_1 AS LONG 'Dimension
DIM old_2, new_2 AS LONG 'the variables

INIT:
    old_1 = 0              'Initialize
    old_2 = 0              'the variables
    CNT_MODE(0)            'All counters on external clock input
    CNT_SET(1)             'Counter 1 with 20 MHz
    CNT_INPUTMODE(0)       'Determine the functionality CLR/LATCH
                           ' As CLR-input at all counters
    CNT_CLEARENABLE(11b)   'Enables the CLR-function of
                           ' counters 1+2
    CNT_CLEAR(11b)         'Reset counters 1+2 to 0
    CNT_ENABLE(11b)        'Start counters 1+2

EVENT:
    CNT_LATCH(11b)         'Latch counters 1+2 simultaneously
                           'and then...
    new_1 = CNT_READLATCH(1) 'Read out Latch A counter 1 and...
    new_2 = CNT_READLATCH(2) 'Latch A counter 2.
    PAR_1 = new_1 - old_1   'Calculate the difference
                           ' (f = impulses / time)
    PAR_2 = new_2 - old_2   '-''-
    old_1 = new_1           'Save new counter values as old
    old_2 = new_2           '-''-

```

CNT_READ

CNT_READ transfers current counter values into Latch A and returns them as return value.

Syntax

```
#INCLUDE ADWGCNT.INC      'ADwin-Gold only
#include ADWL16.INC       'ADwin-light-16 only

ret_val = CNT_READ(CounterNo)
```

Parameters

CounterNo	Counter number (L16, L16-DIO1: 1...2, L16-CO1: 1; Gold-CO1: 1...4)	LONG
ret_val	Counter values	LONG

Notes

Use the return value in calculations only with variables of the type **LONG** (e.g. differences or count direction).

See also

CNT_CLEAR, CNT_CLEARENABLE, CNT_ENABLE,
CNT_GETSTATUS, CNT_INPUTMODE, CNT_LATCH, CNT_MODE,
CNT_READLATCH, CNT_READFLATCH, CNT_RESETSTATUS,
CNT_SE_DIFF, CNT_SET

Example

```

#INCLUDE ADWGCNT.INC      'ADwin-Gold only
#INCLUDE ADWL16.INC      'ADwin-light-16 only

DIM old_1, new_1 AS LONG 'Dimension...
DIM old_2, new_2 AS LONG 'the variables

INIT:
  old_1 = 0                'Initialize...
  old_2 = 0                'the variables
  CNT_MODE(0)              'All counters on external clock input
  CNT_SET(11b)             'Counters 1+2 with clock (CLK) and
                           'direction (DIR) inputs
  CNT_INPUTMODE(0)        'Determine functionality CLR/LATCH: At
                           'all as CLR-input
  CNT_CLEARENABLE(11b)    'Enables the CLR-function of
                           'counters 1+2
  CNT_CLEAR(11b)         'Reset counters 1+2 to 0
  CNT_ENABLE(11b)        'Start counters 1+2

EVENT:
  new_1 = CNT_READ(1)      'Latch counter 1 and read out Latch A
                           'afterward
  new_2 = CNT_READ(2)      'Latch counter 2 and read out Latch A
                           'afterward
  PAR_1 = new_1 - old_1     'Calculate the difference
                           '(f = impulses / time)
  PAR_2 = new_2 - old_2    ' -"-
  old_1 = new_1            'Save new counter values as old
  old_2 = new_2            ' -"-

```

CNT_READLATCH

CNT_READLATCH returns the value of a counter previously stored in Latch A.

Syntax

```
#INCLUDE ADWGCNT.INC      'ADwin-Gold only
#include ADWL16.INC        'ADwin-light-16 only

ret_val = CNT_READLATCH(CounterNo)
```

Parameters

<code>CounterNo</code>	Counter number (L16, L16-DIO1: 1...2, L16-CO1: 1, Gold-CO1: 1...4)	LONG
<code>ret_val</code>	Contents of Latch A	LONG

Notes

Use the return value in calculations only with variables of the type **LONG** (e.g. differences or count direction).

See also

CNT_CLEAR, CNT_CLEARENABLE, CNT_ENABLE,
CNT_GETSTATUS, CNT_INPUTMODE, CNT_LATCH, CNT_MODE,
CNT_READ, CNT_READFLATCH, CNT_RESETSTATUS,
CNT_SE_DIFF, CNT_SET

Notes

The point of time when the current counter value is latched depends on the **CNT_MODE** settings:

- External clock input (**CNT_MODE** bit = 0): Only the instruction **CNT_LATCH** latches the counter.
- Internal clock input (**CNT_MODE** bit = 1): Any edge of the external measurement signal latches the counter.

At a positive edge of the input signal the counter values are latched into Latch A, whereas at a negative edge of the input signal the counter values are latched into Latch B.

Example

```

#INCLUDE ADWGCNT.INC      'ADwin-Gold only
#INCLUDE ADWL16.INC      'ADwin-light-16 only
DIM rise, rise_old, fall, fall_old AS LONG
#DEFINE high PAR_1
#DEFINE low PAR_2
#DEFINE T PAR_9
#DEFINE f PAR_10

INIT:
    rise_old = 0           'Initialize the variables
    fall_old = 0
    CNT_MODE(11b)         'Counters 1+2 on internal clock input
    CNT_SET(0)            'All counters with 20 MHz internal
                           'reference clock
    CNT_INPUTMODE(11b)   'Determine functionality CLR/LATCH: At
                           'counters 1+2 as LATCH input
    CNT_CLEARENABLE(0)   'Disables the CLR-function of all
                           'counters
    CNT_CLEAR(11b)      'Reset counters 1+2 to 0
    CNT_ENABLE(1)       'Start couner 1

EVENT:
    rise = CNT_READLATCH(1) 'Read out Latch A counter 1
    fall = CNT_READFLATCH(1) 'Read out Latch B counter 1
    IF (rise <> rise_old) THEN 'Is a rising edge detected?
        T = rise - rise_old 'Period duration in nanoseconds
        f = 1E9 / T         'Frequency in Hertz
        IF (fall <> fall_old) THEN 'Is a falling edge detected?
            high = (fall - rise) * 25 'Impulse duration in nanoseconds
            low = (rise - fall_old) * 25 'Pause duration in
                                   'nanoseconds
        ELSE
            'No falling edge is detected
            high = (fall - rise_old) * 25 'Impulse duration in
                                   'nanoseconds
            low = (rise - fall) * 25 'Pause duration in nanoseconds
        ENDIF
    ENDIF
    rise_old = rise         'Save contents of the latch
    fall_old = fall         'Save contents of the latch

```

CNT_READFLATCH

CNT_READFLATCH returns the value of a counter previously stored in Latch B.

Syntax

```
#INCLUDE ADWGCNT.INC      'ADwin-Gold only
#include ADWL16.INC       'ADwin-light-16 only

ret_val = CNT_READFLATCH(CounterNo)
```

Parameters

CounterNo	Counter number (L16-DIO1: 1...2, Gold-CO1: 1...4)	LONG
ret_val	Contents of Latch B	LONG

Comment

Use the return value in calculations only with variables of the type **LONG** (e.g. differences or count direction).

The point of time when the current counter value is latched depends on the **CNT_MODE** settings:

- External clock input (**CNT_MODE** bit = 0): Only the instruction **CNT_LATCH** latches the counter.
- Internal clock input (**CNT_MODE** bit = 1): Any edge of the external measurement signal latches the counter.

At a positive edge of the input signal the counter values are latched into Latch A, whereas at a negative edge of the input signal the counter values are latched into Latch B.

See also

CNT_CLEAR, CNT_CLEARENABLE, CNT_ENABLE,
CNT_GETSTATUS, CNT_INPUTMODE, CNT_LATCH, CNT_MODE,
CNT_READ, CNT_READLATCH, CNT_RESETSTATUS,
CNT_SE_DIFF, CNT_SET

Example

```

#INCLUDE ADWGCNT.INC      'ADwin-Gold only
#INCLUDE ADWL16.INC      'ADwin-light-16 only
DIM rise, rise_old, fall, fall_old AS LONG
#DEFINE high PAR_1
#DEFINE low PAR_2
#DEFINE T PAR_9
#DEFINE f PAR_10

INIT:
    rise_old = 0           'Initialize...
    fall_old = 0           ' the variables
    CNT_MODE(11b)          'Counters 1+2 on internal clock input
    CNT_SET(0)             'All counters with 20 MHz internal
                           'clock reference
    CNT_INPUTMODE(11b)     'Determine functionality CLR/LATCH: At
                           'counters 1+2 as LATCH inputs
    CNT_CLEARENABLE(0)     'Disables the CLR-function of all
                           'counters
    CNT_CLEAR(11b)        'Reset counters 1+2 to 0
    CNT_ENABLE(1)         'Start counter 1

EVENT:
    rise = CNT_READLATCH(1) 'Read out Latch A counter 1
    fall = CNT_READFLATCH(1) 'Read out Latch B counter 1
    IF (rise <> rise_old) THEN 'Is a rising edge detected?
        T = rise - rise_old    'Period duration in nanoseconds
        f = 1E9 / T           'Frequency in Hertz
        IF (fall <> fall_old) THEN 'Is a falling edge detected?
            high = (fall - rise) * 25 'Impulse duration in nanoseconds
            low = (rise - fall_old) * 25 'Pause duration in
                                     'nanoseconds
        ELSE
            'No falling edge detected
            high = (fall - rise_old) * 25 'Impulse duration in
                                     'nanoseconds
            low = (rise - fall) * 25 'Pause duration in nanoseconds
        ENDIF
    ENDIF
    rise_old = rise           'Save contents of the latch
    fall_old = fall           'Save contents of the latch

```

CNT_RESETSTATUS

ADwin-Gold only: The instruction **CNT_RESETSTATUS** clears the status register of all four 32 bit-counters.

Syntax

```
#INCLUDE ADWGCNT.INC      'ADwin-Gold only
CNT_RESETSTATUS()
```

Comment

The status register is read out with the instruction **CNT_GETSTATUS**.

See also

CNT_CLEAR, CNT_ENABLE, CNT_GETSTATUS,
CNT_INPUTMODE, CNT_LATCH, CNT_MODE, CNT_READ,
CNT_READLATCH, CNT_READFLATCH, CNT_SE_DIFF, CNT_SET

Example

```

#INCLUDE ADWGCNT.INC      'ADwin-Gold only

DIM error AS LONG

DIM old_1, new_1 AS LONG 'Dimensioning...
DIM old_2, new_2 AS LONG ' variables

INIT:
  CNT_ENABLE(0)           'Stop all counters
  CNT_CLEAR(1111b)        'Clear all counters
  CNT_SE_DIFF(11b)        'Set all counters to diff. inputs
  CNT_MODE(0)             'Set external event input
  CNT_SET(0)              'Set mode 4 edge evaluation
  CNT_INPUTMODE(0)        'Enable CLR counter input
  CNT_ENABLE(1111b)       'Start all counters
  old_1 = 0               'Initialize...
  old_2 = 0               ' variables
  error = 0               'Initialize error flag

EVENT:
  PAR_1 = CNT_READ(1)      'Read out counter 1
  PAR_2 = CNT_GETSTATUS(1) AND 0FFFF00F0h 'Read out and mask
                                     'status register counter 1
  IF (PAR_2 AND 2000000h = 2000000h) THEN 'Line or cable error
                                     'counter 1?
    INC PAR_3              'Number of line or cable errors until
                                     'now...
    error = 1              'Set error flag
  ENDIF
  IF (PAR_2 AND 1000000h = 1000000h) THEN 'Correlation error
                                     'counter 1?
    INC PAR_4              'Number of correlation errors until
                                     'now...
    error = 1              'Set error flag
  ENDIF
  CNT_RESETSTATUS()       'Clear bits of line and correlation
                                     'errors
  PAR_5 = SHIFT_RIGHT(PAR_2 AND 10h,4) 'status of CLR-input
  PAR_6 = SHIFT_RIGHT(PAR_2 AND 10000h,16) 'status of input A
  PAR_7 = SHIFT_RIGHT(PAR_2 AND 20000h,17) 'status of input B

```

CNT_SE_DIFF

ADwin-Gold only: The instruction **CNT_SE_DIFF** sets counter inputs to the input mode single-ended or differential as pairs.

Syntax

```
#INCLUDE ADWGCNT.INC      'ADwin-Gold only
CNT_SE_DIFF (CounterNo)
```

Parameter

CounterNo Bit pattern to choose the counter pairs (see LONG table) and set the input mode:
 Bit = 0: Run inputs single-ended
 Bit = 1: Run inputs differential

Bit no. in CounterNo	31 ... 2	1	0
Inputs of counters no.	–	3 + 4	1 + 2

Comment

After start-up of an *ADwin-Gold*, the operating mode of the counter inputs is undefined; all of the counter inputs have to be set to the desired operating mode.

See also

CNT_CLEAR, CNT_ENABLE, CNT_GETSTATUS,
 CNT_INPUTMODE, CNT_LATCH, CNT_MODE, CNT_READ,
 CNT_READLATCH, CNT_READFLATCH, CNT_RESETSTATUS,
 CNT_SET

Example

```

#INCLUDE ADWGCNT.INC      'ADwin-Gold only

DIM error AS LONG          'Dimensioning...
DIM old_1, new_1 AS LONG ' variables
DIM old_2, new_2 AS LONG

INIT:
  CNT_ENABLE(0)             'Stop all counters
  CNT_CLEAR(1111b)          'Clear all counters
  CNT_SE_DIFF(11b)          'Set all counters to diff. inputs
  CNT_MODE(0)               'Set external event input
  CNT_SET(0)                'Set mode 4 edge evaluation
  CNT_INPUTMODE(0)          'Enable CLR counter input
  CNT_ENABLE(1111b)         'Start all counters
  old_1 = 0                 'Initialize...
  old_2 = 0                 ' variables
  error = 0                 'Initialize error flag

EVENT:
  PAR_1 = CNT_READ(1)       'Read out counter 1
  PAR_2 = CNT_GETSTATUS(1) AND 0FFFF00F0h 'Read out and mask
                                         'status register counter 1
  IF (PAR_2 AND 2000000h = 2000000h) THEN 'Line or cable error
                                         'counter 1?
    INC PAR_3                'Number of line or cable errors until
                                         'now...
    error = 1                'Set error flag
  ENDIF
  IF (PAR_2 AND 1000000h = 1000000h) THEN 'Correlation error
                                         'counter 1?
    INC PAR_4                'Number of correlation errors until
                                         'now...
    error = 1                'Set error flag
  ENDIF
  CNT_RESETSTATUS()         'Clear bits of line and correlation
                                         'errors
  PAR_5 = SHIFT_RIGHT(PAR_2 AND 10h,4)   'current status of CLR-input
  PAR_6 = SHIFT_RIGHT(PAR_2 AND 10000h,16) 'current status of input A.
  PAR_7 = SHIFT_RIGHT(PAR_2 AND 20000h,17) 'current status of input B.

```

CNT_SET

The instruction **CNT_SET** defines the operating mode for all counters (depending on **CNT_MODE**) according to the bit pattern in **pattern**.

Syntax

```
#INCLUDE ADWGCNT.INC      'ADwin-Gold only
#include ADWL16.INC       'ADwin-light-16 only

CNT_SET(pattern)
```

Parameters

pattern Bit pattern, for the meaning of the bits LONG
see table below.

Bit value in pattern	External clock input Bit = 0 in CNT_MODE	Internal clock input Bit = 1 in CNT_MODE
Bit = 0	4-edge evaluation	Reference clock 20 MHz
Bit = 1	Clock and direction input	Reference clock 5 MHz

Bit no.	31...4	3	2	1	0
Counter no.	—	4 ^a	3 ^a	2	1

a. for *ADwin-Gold* CO1 only

Comment

Please use this instruction only when the counter is disabled.

See also

CNT_CLEAR, CNT_CLEARENABLE, CNT_ENABLE,
CNT_GETSTATUS, CNT_INPUTMODE, CNT_LATCH, CNT_MODE,
CNT_READ, CNT_READLATCH, CNT_READFLATCH,
CNT_RESETSTATUS, CNT_SE_DIFF, CNT_SET

Example

```
#INCLUDE ADWGCNT.INC      'ADwin-Gold only
#INCLUDE ADWL16.INC       'ADwin-light-16 only

INIT:
  CNT_MODE(0)             'All counters on external clock input
  CNT_SET(11100b)         'Counters 3+4 (Gold only) with clock/
                           'direction evaluation, Counters 1+2
                           'with 4 edge evaluation
  CNT_CLEAR(11100b)       'Set counters 3+4 (Gold only) to 0
  CNT_ENABLE(11100b)      'Enable counters 3+4 (Gold only),
                           'disable counters 1+2
```

CONF_DIO_E

The instruction **CONF_DIO_E** configures the digital channels as inputs or outputs in groups of 8.

Syntax

```
#INCLUDE ADWL16.INC
```

```
CONF_DIO_E(setup)
```

Parameters

setup

Bit pattern, that configures
the digital channels as inputs or outputs:

LONG

Bit=0: Channels as inputs

Bit=1: Channels as outputs

Bit no. in val	15...4	3	2	1	0
Channels	–	DIO31	DIO23	DIO15	DIO07
	
		DIO24	DIO16	DIO08	DIO00

Comment

After power-up all digital I/O-lines are configured as inputs.

See also

DIGIN_WORD1_E, DIGIN_WORD2_E, DIGOUT_RESET1_E,
DIGOUT_RESET2_E, DIGOUT_SET1_E, DIGOUT_SET2_E,
DIGOUT_WORD1_E, DIGOUT_WORD2_E

Example

```
#INCLUDE ADWL16.INC
```

```
INIT:
```

```
CONF_DIO_E(1100b)    'Configures DIOs 15:00 as inputs and  
                     'DIOs 31:16 as outputs.
```


DIGIN_WORD1_E

The instruction **DIGIN_WORD1_E** returns the values of the digital inputs 0...15 at the same time.

Syntax

```
#INCLUDE ADWL16.INC

ret_val = DIGIN_WORD1_E()
```

Parameters

ret_val Bit pattern, that corresponds to the TTL-level at the digital inputs.
 1: TTL-level high
 0: TTL-level low

LONG

Bit number in	31 ... 16	15	14	...	1	0
ret_val						
Input No.	–	DIO15	DIO14	...	DIO01	DIO00

Comment

If you have configured the channels as outputs, the contents of the output register of these bits is returned.

See also

CONF_DIO_E, DIGIN_WORD2_E, DIGOUT_RESET1_E, DIGOUT_RESET2_E, DIGOUT_SET1_E, DIGOUT_SET2_E, DIGOUT_WORD1_E, DIGOUT_WORD2_E

Example

```
#INCLUDE ADWL16.INC

INIT:
    CONF_DIO_E(1100b)      'Configures DIOs 15:00 as inputs and
                           'DIOs 31:16 as outputs

EVENT:
    PAR_1 = DIGIN_WORD1_E() 'Read low-word (bits 15:00)
```

DIGIN_WORD2_E

The instruction **DIGIN_WORD2_E** returns the values of the digital inputs 16...31 at the same time.

Syntax

```
#INCLUDE ADWL16.INC

ret_val = DIGIN_WORD2_E()
```

Parameters

ret_val Bit pattern, that corresponds to the TTL-level at the digital inputs.
 1: TTL-level high
 0: TTL-level low

LONG

Bit number in	31 ... 16	15	14	...	1	0
ret_val						
Input No.	—	DIO31	DIO30	...	DIO17	DIO16

Comment

If you have configured the channels as outputs, the contents of the output register of these bits is returned.

See also

CONF_DIO_E, DIGIN_WORD1_E, DIGOUT_RESET1_E, DIGOUT_RESET2_E, DIGOUT_SET1_E, DIGOUT_SET2_E, DIGOUT_WORD1_E, DIGOUT_WORD2_E

Example

```
#INCLUDE ADWL16.INC

INIT:
  CONF_DIO_E(0)           'Configures DIOs 31:00 as inputs(also
                           'in the power-up status!)

EVENT:
  PAR_1 = DIGIN_WORD1_E() 'Read low-word (bits 15:00)
  PAR_2 = DIGIN_WORD2_E() 'Read high-word (bits 31:16)
```

DIGOUT_RESET1_E

The instruction **DIGOUT_RESET1_E** sets the selected digital outputs 0...15 to TTL-level low.

Syntax

```
#INCLUDE ADWL16.INC
DIGOUT_RESET1_E(setup)
```

Parameters

setup Bit pattern for setting specified outputs: LONG
 Bit = 1: Set to TTL-level low
 Bit = 0: no influence

Bit number	31 ... 16	15	14	...	1	0
in setup						
Input No.	–	DIO15	DIO14	...	DIO01	DIO00

See also

CONF_DIO_E, DIGIN_WORD1_E, DIGIN_WORD2_E,
 DIGOUT_RESET2_E, DIGOUT_SET1_E, DIGOUT_SET2_E,
 DIGOUT_WORD1_E, DIGOUT_WORD2_E

Example

```
#INCLUDE ADWL16.INC

INIT:
  CONF_DIO_E(11b)      'Configures DIOs 15:00 as outputs and
                        'DIOs 31:16 as inputs

INIT:
  PAR_1 = 5555h         'Delete all odd-numbered bits of the
                        'low-word upon output.
  DIGOUT_WORD1_E(0FFFFh) 'Output DIO-bits 15:00

EVENT:
  DIGOUT_RESET1_E(PAR_1) 'Delete DIO-bits equivalent to PAR_1
  PAR_1 = PAR_1 XOR 0FFFFh 'Invert output-word
  DIGOUT_WORD1_E(PAR_1) 'Output DIO-bits 15:00
```

DIGOUT_RESET2_E

The instruction **DIGOUT_RESET2_E** sets the selected digital outputs 16...31 to TTL-level low.

Syntax

```
#INCLUDE ADWL16.INC
DIGOUT_RESET2_E(setup)
```

Parameters

setup Bit pattern for setting specified outputs: LONG
 Bit = 1: Set to TTL-level low
 Bit = 0: no influence

Bit number in setup	31 ... 16	15	14	...	1	0
Input No.	–	DIO31	DIO30	...	DIO17	DIO16

See also

CONF_DIO_E, DIGIN_WORD1_E, DIGIN_WORD2_E,
 DIGOUT_RESET1_E, DIGOUT_SET1_E, DIGOUT_SET2_E,
 DIGOUT_WORD1_E, DIGOUT_WORD2_E

Example

```
#INCLUDE ADWL16.INC

INIT:
  CONF_DIO_E(1100b)      'Configures DIOs 15:00 as inputs and
                          'DIOs 31:16 as outputs

INIT:
  PAR_2 = 5555h           'Clear all odd-numbered bits of the
                          'high-word during output.
  DIGOUT_WORD1_E(0FFFFh) 'Output the DIO bits 15:00

EVENT:
  DIGOUT_RESET2_E(PAR_2) 'Clear the DIO bits according to
                          'PAR_2.
  PAR_2 = PAR_2 XOR 0FFFFh 'Invert output-word
  DIGOUT_WORD2_E(PAR_2) 'Output the DIO bits 31:16
```

DIGOUT_SET1_E

The instruction **DIGOUT_SET1_E** sets the selected digital outputs 0...15 to TTL-level high.

Syntax

```
#INCLUDE ADWL16.INC
DIGOUT_SET1_E(setup)
```

Parameters

setup Bit pattern to set specified outputs: LONG
 Bit = 1: Set to TTL-level high
 Bit = 0: No change

Bit number	31 ... 16	15	14	...	1	0
in setup						
Input No.	–	DIO15	DIO14	...	DIO01	DIO00

See also

CONF_DIO_E, DIGIN_WORD1_E, DIGIN_WORD2_E,
 DIGOUT_RESET1_E, DIGOUT_RESET2_E, DIGOUT_SET2_E,
 DIGOUT_WORD1_E, DIGOUT_WORD2_E

Example

```
#INCLUDE ADWL16.INC

INIT:
  CONF_DIO_E(1100b)    'Configures DIOs 15:00 as outputs and
                        'DIOs 31:16 as input
  PAR_1 = 0AAAAh       'Set all even-numbered bits of the
                        'low-word during the output
  DIGOUT_WORD1_E(0)    'Output the DIO bits 15:00

EVENT:
  DIGOUT_SET1_E(PAR_1) 'Set the DIO bits according to PAR_1
  PAR_1 = PAR_1 XOR 0FFFFh 'Invert output-word
  DIGOUT_WORD1_E(PAR_1) 'Output the DIO bits 15:00
```

DIGOUT_SET2_E

The instruction **DIGOUT_SET2_E** sets the selected digital outputs 16...31 to TTL-level high.

Syntax

```
#INCLUDE ADWL16.INC
DIGOUT_SET2_E(setup)
```

Parameters

setup Bit pattern to set specified outputs: LONG
 Bit = 1: Set to TTL-level high
 Bit = 0: No change

Bit number in setup	31 ... 16	15	14	...	1	0
Input No.	–	DIO31	DIO30	...	DIO17	DIO16

See also

CONF_DIO_E, DIGIN_WORD1_E, DIGIN_WORD2_E,
 DIGOUT_RESET1_E, DIGOUT_SET1_E, DIGOUT_SET2_E,
 DIGOUT_WORD1_E, DIGOUT_WORD2_E

Example

```
#INCLUDE ADWL16.INC

INIT:
  CONF_DIO_E(1100b)    'Configures DIOs 15:00 as outputs and
                        'the DIOs 31:16 as inputs
  PAR_1 = 0AAAAh       'Set all even-numbered bits of the
                        'low-word during the output
  DIGOUT_WORD2_E(0)    'Output the DIO bits 15:00

EVENT:
  DIGOUT_SET2_E(PAR_2) 'Set the DIO bits according to PAR_1
  PAR_2 = PAR_2 XOR 0FFFFh 'Invert output-word
  DIGOUT_WORD2_E(PAR_2) 'Output the DIO bits 15:00
```

DIGOUT_WORD1_E

The instruction **DIGOUT_WORD1_E** sets all digital outputs 0...15 to specified TTL-levels using a bit pattern.

Syntax

```
#INCLUDE ADWL16.INC
DIGOUT_WORD1_E(setup)
```

Parameters

setup Bit pattern, corresponding to the
TTL level desired at the digital outputs.
Bit = 1: Set to TTL-level high
Bit = 0: Set to TTL-level low

LONG

Bit number	31 ... 16	15	14	...	1	0
in setup						
Input No.	–	DIO15	DIO14	...	DIO01	DIO00

See also

CONF_DIO_E, DIGIN_WORD1_E, DIGIN_WORD2_E,
DIGOUT_RESET1_E, DIGOUT_RESET2_E, DIGOUT_SET1_E,
DIGOUT_SET2_E, DIGOUT_WORD2_E

Example

```
#INCLUDE ADWL16.INC

INIT:
    CONF_DIO_E(0011b)    'Configures DIOs 15:00 as outputs and
                          'DIOs 31:16 as inputs
    PAR_1 = 5555h        'Set all odd-numbered bits of the
                          'low-word

EVENT:
    DIGOUT_WORD1_E(PAR_1) 'Output the DIO bits 15:00
```

DIGOUT_WORD2_E

The instruction **DIGOUT_WORD2_E** sets all the digital outputs 16...31 to specified TTL-levels using a bit pattern.

Syntax

```
#INCLUDE ADWL16.INC
DIGOUT_WORD2_E(setup)
```

Parameters

setup Bit pattern, corresponding to the LONG
 TTL level desired at the digital outputs.
 Bit = 1: Set to TTL-level high
 Bit = 0: Set to TTL-level low

Bit number in setup	31 ... 16	15	14	...	1	0
Input No.	—	DIO31	DIO30	...	DIO17	DIO16

See also

CONF_DIO_E, DIGIN_WORD1_E, DIGIN_WORD2_E,
 DIGOUT_RESET1_E, DIGOUT_RESET2_E, DIGOUT_SET1_E,
 DIGOUT_SET2_E, DIGOUT_WORD1_E

Example

```
#INCLUDE ADWL16.INC

INIT:
  CONF_DIO_E(12)      'Configures DIOs 15:00 as inputs and
                      'DIOs 31:16 as outputs
  PAR_2 = 0AAAAh      'Set all even-numbered bits of the
                      'low-word.

EVENT:
  DIGOUT_WORD2_E(PAR_2) 'Output the DIO bits 31:16
```


CAN_MSG

`CAN_MSG[]` is a one-dimensional array, consisting of 9 elements, where the message objects are stored.

Syntax

```
#INCLUDE ADWL16.INC
```

```
CAN_MSG[n] = value
```

or

```
value = CAN_MSG[n]
```

Parameters

<code>n</code>	Element number in the field <code>CAN_MSG</code> (1...9)	LONG
<code>value</code>	Value (8 bit), which is to be written into or read from the message object.	LONG

Comment

The elements of the array `CAN_MSG[]` have the following functions:

Element no. in <code>CAN_MSG</code>	1...8	9
Contents	Message object(s) = databyte(s)	Number (0...8) of allocated databytes

Enter the values to be transferred into the field `CAN_MSG[]`, *before* transferring them with **TRANSMIT**.

See also

EN_INTERRUPT, EN_RECEIVE, EN_TRANSMIT, GET_CAN_REG, INIT_CAN, READ_MSG, SET_CAN_BAUDRATE, SET_CAN_REG, TRANSMIT

Example

```
#INCLUDE ADWL16.INC
```

INIT:

```
INIT_CAN()           'Initialize CAN controller
EN_TRANSMIT(6,40,0)   'Enable message object 6 with the
                      'identifier 40 (11 bit)for sending

CAN_MSG[1] = 40h      'Byte 4 (MSB) &
CAN_MSG[2] = 49h      'Byte 3 & ...
CAN_MSG[3] = 0Fh      'Byte 2 & ...
CAN_MSG[4] = 0DBh     'Byte 1 equal the number Pi as a FLOAT
                      'number (32-bit)
CAN_MSG[9] = 4        'Length of the message in bytes
```

EVENT:

```
TRANSMIT(6)           'Sends the message object 6
```

EN_INTERRUPT

CAN bus: The instruction **EN_INTERRUPT** configures a specified message object in such a manner that an external event is generated when the message arrives.

Syntax

```
#INCLUDE ADWL16.INC  
  
EN_INTERRUPT(objectno)
```

Parameters

<code>objectno</code>	Number (1...15) of the message object	<div>LONG</div>
-----------------------	---------------------------------------	-----------------

See also

CAN_MSG, EN_RECEIVE, GET_CAN_REG

Example

```
#INCLUDE ADWL16.INC  
  
INIT:  
    INIT_CAN()                'Initialization of the CAN controller  
    EN_RECEIVE(1200,0)        'Initialize the message object 1 to  
                                'receive CAN messages with the  
                                'identifier 200  
    EN_INTERRUPT(1)           'Enables the triggering of interrupts  
                                '(ext. EVENT) when receiving the  
                                'message object 1
```

EN_RECEIVE

CAN bus: The instruction **EN_RECEIVE** enables a specified message object to receive messages.

Syntax

```
#INCLUDE ADWL16.INC

EN_RECEIVE(objectno, id, extend)
```

Parameters

<code>objectno</code>	Number (1...15) of the message object	LONG
<code>id</code>	Identifier ($0 \dots 2^{11}$ or $0 \dots 2^{29}$) of the messages, which can be received in this message object.	LONG
<code>extend</code>	Length of the identifier: 0: 11 bits 1: 29 bits	LONG

See also

CAN_MSG, EN_TRANSMIT, GET_CAN_REG

Comment

A message object can only receive messages from the CAN bus when you have previously enabled it to receive with **EN_RECEIVE**.

The message object only receives messages with the identifier you have specified.

Example

```
#INCLUDE ADWL16.INC

INIT:
    INIT_CAN()                'Initialization of the CAN controller
    EN_RECEIVE(1200,0)        'Initialize the message object 1 to
                                'receive CAN messages with the
                                'identifier 200
```

EN_TRANSMIT

CAN bus: The instruction **EN_TRANSMIT** enables a specified message object to send messages.

Syntax

```
#INCLUDE ADWL16.INC  
EN_TRANSMIT(objectno, id, extend)
```

Parameters

objectno	Number (1...14) of the message object	LONG
id	Identifier which is sent with the messages of this message object.	LONG
extend	Length of the identifier: 0: 11 bits 1: 29 bits	LONG

See also

CAN_MSG, EN_RECEIVE, GET_CAN_REG

Comment

A message object can only send messages to the CAN bus when you have it previously enabled to send with **EN_TRANSMIT**.

Example

```
#INCLUDE ADWL16.INC  
  
INIT:  
  INIT_CAN()           'Initialization of the CAN controller  
  EN_TRANSMIT(6,40,0)  'Initialize the message object 6 to  
                       'send CAN messages with identifier 40
```

GET_CAN_REG

(DIO1 only);

CAN bus: The instruction **GET_CAN_REG** reads the value of a specified register in the CAN controller.

Syntax

```
#INCLUDE ADWL16.INC  
  
ret_val = GET_CAN_REG(regno)
```

Parameters

regno	Register number in the CAN controller (0...255)	LONG
ret_val	Contents of the register (transfer to the lower 8 bits)	LONG

See also

SET_CAN_BAUDRATE, SET_CAN_REG

Comment

You will find the register list of the CAN controller in the Intel® AN82527 datasheet.

Example

```
#INCLUDE ADWL16.INC  
INIT:  
    INIT_CAN()           'Initialization of the CAN controller  
    PAR_1 = GET_CAN_REG(0) 'Read out the control register
```

INIT_CAN

CAN bus: The instruction **INIT_CAN** initializes the CAN controller.

Syntax

```
#INCLUDE ADWL16.INC  
  
INIT_CAN()
```

Comment

The instruction carries out the following steps:

- Reset (hardware reset of the CAN controller)
- All filters are set to "must match".
- Clockout register is set to 0 (= the external frequency is not divided).
- The register "Bus Configuration" is set to 0.
- The transfer rate for the CAN bus is set to 1 MBit/s.
- All message objects are disabled.

You have to execute this instruction before you access the CAN controller with other instructions. We recommend you place this instruction in the process section **LOWINIT**: or **INIT**:

See also

CAN_MSG, EN_RECEIVE, EN_TRANSMIT, GET_CAN_REG

Example

```
#INCLUDE ADWL16.INC  
  
INIT:  
    INIT_CAN()           'Initialize the CAN controller 1 on CAN  
                          'module 1
```

READ_MSG

CAN bus: The instruction **READ_MSG** checks if new messages have been received in a specified message object. If so, the message is saved in **CAN_MSG** and the identifier of the message is returned.

Syntax

```
#INCLUDE ADWL16.INC

ret_val = READ_MSG(msgno)
```

Parameters

msgno	Number (1...15) of the message object	LONG
ret_val	-1: No new message >0: New message; value = identifier of the message	LONG

Comment

You can read out a message you have received only once.

You have to enable the message object you want to read out with **EN_RECEIVE** before, so that it will be able to receive.

See also

CAN_MSG, EN_RECEIVE, EN_TRANSMIT, GET_CAN_REG

Example

REM Receives a message object and arranges
REM the individual bytes, in case a new object
REM with the right identifier is received.

```
#INCLUDE ADWL16.INC
```

```
DIM n AS LONG
```

```
INIT:
```

```
PAR_1 = 0
```

```
  INIT_CAN()           'Initialization of the CAN controller
```

```
  EN_RECEIVE(1,40,0)   'Initialize the message object 1 to  
                        'send CAN messages with identifier 40
```

```
EVENT:
```

```
  REM If the message is changed, read out the received data  
  REM from object 1 and transfer the identifier to parameter 9.  
  REM The data bytes are in the array CAN_MSG[].
```

```
  PAR_9 = READ_MSG(1)
```

```
  IF (PAR_9 = 40) THEN
```

```
    REM New message for message object with the identifier 40  
    REM has arrived
```

```
    PAR_1 = CAN_MSG[1] 'Read out high-bytes
```

```
    FOR n = 2 TO 4      'Arrange the remaining 3 bytes to
```

```
      PAR_1 = SHIFT_LEFT(PAR_1,8) + CAN_MSG[n] 'a 32-bit number
```

```
    NEXT n
```

```
  ENDIF
```

SET_CAN_BAUDRATE

CAN bus: The instruction **SET_CAN_BAUDRATE** sets the Baud rate of the CAN controller.

Syntax

```
#INCLUDE ADWL16.INC

ret_val = SET_CAN_BAUDRATE(rate)
```

Parameters

rate	Baud rate in bits/second	LONG
ret_val	0: Baud rate is set 1: Baud rate invalid	LONG

Comment

The available baud rates (bus frequencies) are given in the table "Available Baud rates". Please use the table's notation exactly, i.e. non-integer baud rates with 4 decimal places; values with different notation will be rejected as not allowed.

The instruction executes the following actions:

- Checks if the transferred Baud rate is allowed. If not then set the return value to 1 and stop processing.
- Set the registers of the CAN controller for the Baud rate.
- Set sampling mode to 0: One sample per bit.
- Select the settings in such a way that the sample point is always between 60% and 72% of the total bit length.
- Set the jump width for synchronization to 1.

In special cases it may be of interest to set a baud rate in a different way than the instruction works. The hardware manual gives an explanation how to do this.



The instruction should be called in the program sections **LOWINIT**: or **INIT**:, after the instruction **INIT_CAN**, because otherwise the set Baud rate will be overwritten by the default setting (1 MBit/s).

See also

GET_CAN_REG, SET_CAN_REG

Example

```
#INCLUDE ADWL16.INC

INIT:
    INIT_CAN()           'Initialization of the CAN controller
    SET_CAN_BAUDRATE(125000) 'Set the Baud rate of 125 kBit/s
```

Available Baud rates				
1000000.0000	888888.8889	800000.0000	727272.7273	666666.6667
615384.6154	571428.5714	533333.3333	500000.0000	470588.2353
444444.4444	421052.6316	400000.0000	380952.3810	363636.3636
347826.0870	333333.3333	320000.0000	307692.3077	296296.2963
285714.2857	266666.6667	250000.0000	242424.2424	235294.1176
222222.2222	210526.3158	205128.2051	200000.0000	190476.1905
181818.1818	177777.7778	173913.0435	166666.6667	160000.0000
156862.7451	153846.1538	148148.1481	145454.5455	142857.1429
140350.8772	133333.3333	126984.1270	125000.0000	123076.9231
121212.1212	117647.0588	115942.0290	114285.7143	111111.1111
106666.6667	105263.1579	103896.1039	102564.1026	100000.0000
98765.4321	95238.0952	94117.6471	90909.0909	88888.8889
87912.0879	86956.5217	84210.5263	83333.3333	81632.6531
80808.0808	80000.0000	78431.3725	76923.0769	76190.4762
74074.0741	72727.2727	71428.5714	70175.4386	69565.2174
68376.0684	67226.8908	66666.6667	66115.7025	64000.0000
63492.0635	62500.0000	61538.4615	60606.0606	60150.3759
59259.2593	58823.5294	57971.0145	57142.8571	55944.0559
55555.5556	54421.7687	53333.3333	52631.5789	52287.5817
51948.0519	51282.0513	50000.0000	49689.4410	49382.7160
48484.8485	47619.0476	47337.2781	47058.8235	46783.6257
45714.2857	45454.5455	44444.4444	43956.0440	43478.2609
42780.7487	42328.0423	42105.2632	41666.6667	41025.6410
40816.3265	40404.0404	40000.0000	39215.6863	38647.3430
38461.5385	38277.5120	38095.2381	37037.0370	36363.6364
36199.0950	35714.2857	35555.5556	35087.7193	34782.6087
34632.0346	34482.7586	34188.0342	33613.4454	33333.3333
33057.8512	32921.8107	32388.6640	32258.0645	32000.0000
31746.0317	31620.5534	31372.5490	31250.0000	30769.2308
30651.3410	30303.0303	30075.1880	29629.6296	29411.7647
29304.0293	29090.9091	28985.5072	28673.8351	28571.4286

Available Baud rates				
28070.1754	27972.0280	27777.7778	27681.6609	27586.2069
27210.8844	27027.0270	26936.0269	26755.8528	26666.6667
26315.7895	26143.7908	25974.0260	25806.4516	25641.0256
25396.8254	25078.3699	25000.0000	24844.7205	24767.8019
24691.3580	24615.3846	24390.2439	24242.4242	24024.0240
23809.5238	23668.6391	23529.4118	23460.4106	23391.8129
23255.8140	23188.4058	22988.5057	22857.1429	22792.0228
22727.2727	22408.9636	22222.2222	22160.6648	22038.5675
21978.0220	21739.1304	21680.2168	21621.6216	21505.3763
21390.3743	21333.3333	21276.5957	21220.1592	21164.0212
21052.6316	20833.3333	20779.2208	20671.8346	20512.8205
20460.3581	20408.1633	20202.0202	20050.1253	20000.0000
19851.1166	19753.0864	19704.4335	19656.0197	19607.8431
19512.1951	19323.6715	19230.7692	19138.7560	19047.6190
18912.5296	18867.9245	18823.5294	18648.0186	18604.6512
18518.5185	18433.1797	18390.8046	18306.6362	18181.8182
18140.5896	18099.5475	18018.0180	17857.1429	17777.7778
17738.3592	17582.4176	17543.8596	17429.1939	17391.3043
17316.0173	17241.3793	17204.3011	17094.0171	17021.2766
16949.1525	16913.3192	16842.1053	16806.7227	16771.4885
16666.6667	16632.0166	16563.1470	16528.9256	16460.9053
16393.4426	16326.5306	16260.1626	16227.1805	16194.3320
16161.6162	16129.0323	16000.0000	15873.0159	15810.2767
15779.0927	15686.2745	15625.0000	15594.5419	15503.8760
15473.8878	15444.0154	15384.6154	15325.6705	15238.0952
15180.2657	15151.5152	15122.8733	15094.3396	15065.9134
15037.5940	15009.3809	14842.3006	14814.8148	14705.8824
14652.0147	14571.9490	14545.4545	14519.0563	14492.7536
14414.4144	14336.9176	14311.2701	14285.7143	14260.2496
14184.3972	14109.3474	14035.0877	13986.0140	13937.2822
13913.0435	13888.8889	13840.8304	13793.1034	13722.1269

Available Baud rates				
13675.2137	13605.4422	13582.3430	13559.3220	13513.5135
13468.0135	13445.3782	13377.9264	13333.3333	13289.0365
13223.1405	13157.8947	13136.2890	13114.7541	13093.2897
13071.8954	13008.1301	12987.0130	12903.2258	12882.4477
12820.5128	12800.0000	12759.1707	12718.6010	12698.4127
12578.6164	12558.8697	12539.1850	12500.0000	12422.3602
12403.1008	12383.9009	12345.6790	12326.6564	12307.6923
12288.7865	12195.1220	12158.0547	12121.2121	12066.3650
12030.0752	12012.0120	11994.0030	11922.5037	11904.7619
11851.8519	11834.3195	11764.7059	11730.2053	11695.9064
11661.8076	11627.9070	11611.0305	11594.2029	11544.0115
11494.2529	11477.7618	11428.5714	11396.0114	11379.8009
11363.6364	11347.5177	11299.4350	11220.1964	11204.4818
11188.8112	11111.1111	11080.3324	11034.4828	11019.2837
10989.0110	10943.9124	10928.9617	10884.3537	10869.5652
10840.1084	10810.8108	10796.2213	10781.6712	10752.6882
10695.1872	10666.6667	10638.2979	10610.0796	10582.0106
10540.1845	10526.3158	10457.5163	10430.2477	10416.6667
10389.6104	10335.9173	10322.5806	10296.0103	10269.5764
10256.4103	10230.1790	10204.0816	10101.0101	10088.2724
10062.8931	10025.0627	10012.5156	10000.0000	9937.8882
9925.5583	9876.5432	9852.2167	9828.0098	9803.9216
9791.9217	9768.0098	9756.0976	9696.9697	9685.2300
9661.8357	9615.3846	9603.8415	9569.3780	9523.8095
9456.2648	9433.9623	9411.7647	9400.7051	9367.6815
9356.7251	9324.0093	9302.3256	9291.5215	9259.2593
9227.2203	9216.5899	9195.4023	9153.3181	9142.8571
9090.9091	9070.2948	9049.7738	9039.5480	9009.0090
8958.5666	8928.5714	8918.6176	8888.8889	8879.0233
8869.1796	8859.3577	8771.9298	8743.1694	8714.5969
8695.6522	8658.0087	8648.6486	8620.6897	8602.1505

Available Baud rates				
8592.9108	8556.1497	8547.0085	8510.6383	8483.5631
8474.5763	8465.6085	8456.6596	8421.0526	8403.3613
8385.7442	8333.3333	8281.5735	8264.4628	8255.9340
8230.4527	8205.1282	8196.7213	8163.2653	8130.0813
8113.5903	8105.3698	8097.1660	8088.9788	8080.8081
8064.5161	8000.0000	7976.0718	7944.3893	7936.5079
7905.1383	7843.1373	7812.5000	7804.8780	7797.2710
7774.5384	7751.9380	7736.9439	7729.4686	7714.5612
7692.3077	7662.8352	7655.5024	7619.0476	7590.1328
7575.7576	7561.4367	7547.1698	7532.9567	7518.7970
7469.6545	7441.8605	7421.1503	7407.4074	7400.5550
7386.8883	7352.9412	7326.0073	7285.9745	7272.7273
7259.5281	7246.3768	7187.7808	7168.4588	7142.8571
7136.4853	7130.1248	7111.1111	7098.4916	7092.1986
7054.6737	7017.5439	6993.0070	6956.5217	6944.4444
6926.4069	6902.5022	6896.5517	6861.0635	6820.1194
6808.5106	6802.7211	6791.1715	6779.6610	6734.0067
6688.9632	6683.3751	6666.6667	6611.5702	6578.9474
6568.1445	6562.7564	6557.3770	6535.9477	6530.6122
6493.5065	6456.8200	6451.6129	6441.2238	6410.2564
6400.0000	6379.5853	6349.2063	6324.1107	6289.3082
6274.5098	6269.5925	6250.0000	6245.1210	6211.1801
6172.8395	6163.3282	6153.8462	6144.3932	6102.2121
6060.6061	6046.8632	6037.7358	5997.0015	5961.2519
5952.3810	5925.9259	5895.3574	5865.1026	5847.9532
5818.1818	5797.1014	5772.0058	5747.1264	5714.2857
5702.0670	5681.8182	5649.7175	5614.0351	5610.0982
5555.5556	5521.0490	5517.2414	5464.4809	5434.7826
5423.7288	5376.3441	5333.3333	5291.0053	5245.9016
5208.3333	5161.2903	5079.3651	5000.0000	

SET_CAN_REG

CAN bus: The instruction `SET_CAN_REG` writes a value into a specified register of the CAN controller.

Syntax

```
#INCLUDE ADWL16.INC  
  
SET_CAN_REG(regno, value)
```

Parameters

<code>regno</code>	Register number in the CAN controller (0...255)	LONG
<code>value</code>	Value (8 bits), which is written into the register.	LONG

Comment

The register list of the CAN controller can be found in the Intel® AN82527 datasheet.

See also

SET_CAN_BAUDRATE, GET_CAN_REG

Example

```
#INCLUDE ADWL16.INC  
  
INIT:  
    INIT_CAN()           'Initialization of the CAN controller  
    SET_CAN_REG(0,1)     'Set control register to the value 1
```


TRANSMIT

CAN bus: The instruction **TRANSMIT** sends the message in `CAN_MSG` via the specified message object.

Syntax

```
#INCLUDE ADWL16.INC
```

```
TRANSMIT(msgno)
```

Parameters

`msgno`

Number (1...14) of the message object

LONG

Comment

Enter the message - data bytes and number of data bytes - into the array `CAN_MSG`, before you start sending.

You have to enable the message object with **EN_TRANSMIT**, so that it will be able to send messages.

With this instruction the message is sent as soon as the message object has received access rights to the CAN bus.

See also

INIT_CAN, READ_MSG, EN_TRANSMIT

Example

```
REM Sends a message object (the number Pi as 32-bit
REM FLOAT number)
#INCLUDE ADWL16.INC

INIT:
    INIT_CAN()                'Initialization of the CAN controller
    EN_TRANSMIT(6,40,0)        'Initialize the message object 6 to
                                'send CAN messages with identifier 40
    CAN_MSG[1] = 40h           'Byte 4 (MSB) &
    CAN_MSG[2] = 49h           'Byte 3 & ...
    CAN_MSG[3] = 0Fh           'Byte 2 & ...
    CAN_MSG[4] = 0DBh          'Byte 1 are equal to Pi as 32-bit float
                                'number
    CAN_MSG[9] = 4              'Length of the message in bytes

EVENT:
    TRANSMIT(6)                'Send the message object 6
```

6.5 ADwin-Gold-CAN

This section describes the instructions of the CAN add-on of the *ADwin-Gold* system.

Use the instructions of this section with the include file `ADWGCAN.inc`.

The CAN add-on is equipped with 4 SSI decoder interfaces, 2 CAN interfaces and 2 RSxxx interfaces. The following instructions are available:

CAN interfaces

CAN_MSG	page 278	INIT_CAN	page 284
EN_CAN_INTERRUPT	page 280	READ_MSG	page 285
EN_RECEIVE	page 281	SET_CAN_BAUDRATE	page 287
EN_TRANSMIT	page 282	SET_CAN_REG	page 293
GET_CAN_REG	page 283	TRANSMIT	page 294

RSxxx interfaces

CHECK_SHIFT_REG	page 296	RS_RESET	page 302
GET_RS	page 298	RS485_SEND	page 303
READ_FIFO	page 298	SET_RS	page 304
RS_INIT	page 299	WRITE_FIFO	page 305

SSI decoders

SSI_MODE	page 307	SSI_SET_CLOCK	page 312
SSI_READ	page 309	SSI_START	page 314
SSI_SET_BITS	page 311	SSI_STATUS	page 316

CAN_MSG

`CAN_MSG []` is a one-dimensional array, consisting of 9 elements, where the message objects are stored.

Syntax

```
#INCLUDE ADWGCAN.INC
```

```
CAN_MSG[n] = value
```

or

```
value = CAN_MSG[n]
```

Parameters

<code>n</code>	Element number in the field <code>CAN_MSG</code> (1...9)	LONG
<code>value</code>	Value (8 bit), which is to be written into or read from the message object.	LONG

Notes

The elements of the array `CAN_MSG []` have the following functions:

Element no.	1...8	9
Contents	Message objects = data bytes	Number (0...8) of allocated data bytes

Enter the values to be transferred into the field `CAN_MSG []`, *before* transferring them with **TRANSMIT**.

See also

EN_CAN_INTERRUPT, EN_RECEIVE, EN_TRANSMIT, GET_CAN_REG, INIT_CAN, READ_MSG, SET_CAN_BAUDRATE, SET_CAN_REG, TRANSMIT

Example

REM Sends a 32 Bit FLOAT-value (here: Pi) as sequence of
REM 4 bytes in a message object
REM (Receiving of a float value see Example at READ_MSG)

```
#INCLUDE ADWGCAN.INC
#DEFINE pi 3.14159265
DIM i AS LONG

INIT:
    INIT_CAN(2)                'Initialize CAN controller 2

    REM Enable message object 6 of controller 2 with the
    REM for sending with the identifier 40 (11 bit)
    EN_TRANSMIT(2, 6,40,0)

    REM Create bit pattern of Pi with data type Long
    PAR_1 = CAST_FLOATTOLONG(pi)

    REM divide bit pattern (32 Bit) into 4 bytes
    CAN_MSG[4] = PAR_1 AND 0FFh 'assign LSB
    FOR i = 1 TO 3
        CAN_MSG[4-i] = SHIFT_RIGHT(PAR_1,8) AND 0FFh
    NEXT i
    CAN_MSG[9] = 4              'message length in bytes

EVENT:
    TRANSMIT(2,6)              'Sends the message object 6
```

EN_CAN_INTERRUPT

CAN bus: The instruction **EN_CAN_INTERRUPT** configures a specified message object of a CAN interface in such a manner that an external event is generated when the message arrives.

Syntax

```
#INCLUDE ADWGCAN.INC

EN_CAN_INTERRUPT (Can_No, objectno)
```

Parameters

Can_No	Number (1, 2) of the CAN interface	LONG
objectno	Number (1...15) of the message object	LONG

See also

CAN_MSG, EN_RECEIVE, EN_TRANSMIT

Example

```
#INCLUDE ADWGCAN.INC

INIT:
  INIT_CAN(1)           'Initialization of CAN controller 1
  EN_RECEIVE(1,1,1200,0) 'Initialize the message object 1 of
                        'controller 1 to receive CAN messages
                        'with the identifier 200
  EN_CAN_INTERRUPT(1,1) 'Enables the triggering of interrupts
                        '(ext. EVENT) when receiving the
                        'message object 1
```

EN_RECEIVE

CAN bus: The instruction **EN_RECEIVE** enables a specified message object of a CAN interface to receive messages.

Syntax

```
#INCLUDE ADWGCAN.INC

EN_RECEIVE(Can_No, objectno, id, extend)
```

Parameters

Can_No	Number (1, 2) of the CAN interface	LONG
objectno	Number (1...15) of the message object	LONG
id	Identifier (0...2 ¹¹ or 0...2 ²⁹) of the messages, which can be received in this message object.	LONG
extend	Length of the identifier: 0: 11 bits 1: 29 bits	LONG

See also

CAN_MSG, EN_CAN_INTERRUPT, EN_TRANSMIT, GET_CAN_REG

Notes

A message object can only receive messages from the CAN bus when you have previously enabled it to receive with **EN_RECEIVE**.

The message object only receives messages with the identifier you have specified.

Example

```
#INCLUDE ADWGCAN.INC

INIT:
  INIT_CAN(1)           'Initialization of CAN controller 1
  EN_RECEIVE(1,1,1200,0) 'Initialize the message object 1 of
                        'controller 1 to receive CAN messages
                        'with the identifier 200
```

EN_TRANSMIT

CAN bus: The instruction **EN_TRANSMIT** enables a specified message object of a CAN interface to send messages.

Syntax

```
#INCLUDE ADWGCAN.INC

EN_TRANSMIT(Can_No, objectno, id, extend)
```

Parameters

Can_No	Number (1, 2) of the CAN interface	LONG
objectno	Number (1...14) of the message object	LONG
id	Identifier which is sent with the messages of this message object.	LONG
extend	Length of the identifier: 0: 11 bits 1: 29 bits	LONG

See also

CAN_MSG, EN_RECEIVE, GET_CAN_REG, TRANSMIT

Notes

A message object can only send messages to the CAN bus when you have it previously enabled to send with **EN_TRANSMIT**.

Example

```
#INCLUDE ADWGCAN.INC

INIT:
  INIT_CAN(1)           'Initialization of CAN controller 1
  REM Initialize message objects 6 of controller 1:
  REM Object 1 to receive with identifier 200
  REM Object 1 to send with identifier 40
  EN_RECEIVE(1,1,200,0)
  EN_TRANSMIT(1,6,40,0)
```


GET_CAN_REG

(CAN);

CAN bus: The instruction **GET_CAN_REG** reads the value of a specified register in one of the CAN controllers.

Syntax

```
#INCLUDE ADWGCAN.INC  
  
ret_val = GET_CAN_REG(Can_No, regno)
```

Parameters

Can_No	Number (1, 2) of the CAN interface	LONG
regno	Register number in the CAN controller (0...255)	LONG
ret_val	Contents of the register (transferred in the lower 8 bits)	LONG

See also

INIT_CAN, SET_CAN_BAUDRATE, SET_CAN_REG

Notes

You will find the register list of the CAN controller in the Intel® AN82527 data sheet.

Example

```
#INCLUDE ADWGCAN.INC  
INIT:  
  INIT_CAN(1)           'Initialization of CAN controller 1  
  PAR_1 = GET_CAN_REG(1,0) 'Read out the control register
```

INIT_CAN

CAN bus: The instruction **INIT_CAN** initializes one of the CAN controllers.

Syntax

```
#INCLUDE ADWGCAN.INC  
  
INIT_CAN(Can_No)
```

Parameters

Can_No Number (1, 2) of the CAN interface

LONG

Notes

The instruction carries out the following steps:

- Reset (hardware reset of the CAN controller)
- All filters are set to "must match".
- Clockout register is set to 0 (= the external frequency is not divided).
- The register "Bus Configuration" is set to 0.
- The transfer rate for the CAN bus is set to 1 MBit/s.
- All message objects are disabled.

You have to execute this instruction before you access the CAN controller with other instructions. We recommend you place this instruction in the process section **LOWINIT**: or **INIT**:

See also

CAN_MSG, EN_RECEIVE, EN_TRANSMIT, GET_CAN_REG

Example

```
#INCLUDE ADWGCAN.INC  
  
INIT:  
    INIT_CAN(1)                      'Initialize CAN controller 1
```

READ_MSG

CAN bus: The instruction **READ_MSG** checks if new messages have been received in a specified message object of the CAN interface.

If so, the message is saved in **CAN_MSG** and the identifier of the message is returned.

Syntax

```
#INCLUDE ADWGCAN.INC  
  
ret_val = READ_MSG(Can_No, msgno)
```

Parameters

Can_No	Number (1, 2) of the CAN interface	LONG
msgno	Number (1...15) of the message object	LONG
ret_val	-1: No new message >0: New message received; value = identifier of the message	LONG

Notes

To receive a message you have to follow the correct order:

- Enable the message object with **EN_RECEIVE** for receiving (only once).
- Check for a received message and save to **CAN_MSG** with **READ_MSG**.

You can read a received message only once.

See also

CAN_MSG, **EN_RECEIVE**, **EN_TRANSMIT**, **GET_CAN_REG**

Example

REM If a new message with the correct identifier is received
 REM the data is read out. The first 4 bytes of the message are
 REM combined to a float value of length 32 bit. (Sending a
 REM float value see example of TRANSMIT).

```
#INCLUDE ADWGCAN.INC
```

```
DIM n AS LONG
```

```
INIT:
```

```
PAR_1 = 0
```

```
  INIT_CAN(1)           'Initialization of CAN controller 1
```

```
  EN_RECEIVE(1,1,40,0) 'Initialize the message object 1 of  

                           'controller 1 to send CAN messages with  

                           'identifier 40
```

```
EVENT:
```

```
REM If the message is changed, read out the received data  

REM from object 1 and transfer the identifier to parameter 9.  

REM The data bytes are in the array CAN_MSG[].
```

```
PAR_9 = READ_MSG(1,1)
```

```
IF (PAR_9 = 40) THEN
```

```
  REM New message for message object with the identifier 40  

  REM has arrived
```

```
  PAR_1 = CAN_MSG[1]  'Read out high-byte
```

```
  FOR n = 2 TO 4      'Combine with remaining 3 bytes to
```

```
    PAR_1 = SHIFT_LEFT(PAR_1,8) + CAN_MSG[n] 'a 32-bit value
```

```
  NEXT n
```

```
  REM Convert the bit pattern in PAR_1 to data type FLOAT and  

  REM assign to the variable FPAR_1.
```

```
  FPAR_1 = CAST_LONGTOFLOAT(PAR_1)
```

```
ENDIF
```

SET_CAN_BAUDRATE

CAN bus: The instruction **SET_CAN_BAUDRATE** sets the Baud rate of one of the CAN controllers.

Syntax

```
#INCLUDE ADWGCAN.INC  
  
ret_val = SET_CAN_BAUDRATE (Can_No, rate)
```

Parameters

Can_No	Number (1, 2) of the CAN interface	LONG
rate	Baud rate in bits/second	LONG
ret_val	0: Baud rate is set 1: Baud rate invalid	LONG

Notes

The available baud rates (bus frequencies) are given in the table "Available Baud rates". Please use the table's notation exactly, i.e. non-integer baud rates with 4 decimal places; values with different notation will be rejected as not allowed.

The instruction executes the following actions:

- Checks if the transferred Baud rate is allowed. If not then set the return value to 1 and stop processing.
- Set the registers of the CAN controller for the Baud rate.
- Set sampling mode to 0: One sample per bit.
- Select the settings in such a way that the sample point is always between 60% and 72% of the total bit length.
- Set the jump width for synchronization to 1.

In special cases it may be of interest to set a baud rate in a different way than the instruction works. The hardware manual gives an explanation how to do this.

The instruction should be called in the program sections **LOWINIT** : or **INIT** : , after the instruction **INIT_CAN**, because otherwise the set Baud rate will be overwritten by the default setting (1MBit/s).



See also

GET_CAN_REG, SET_CAN_REG

Example

```
#INCLUDE ADWGCAN.INC
```

```
INIT:
```

```
    INIT_CAN(1)           'Initialization of CAN controller 1
```

```
    SET_CAN_BAUDRATE(1,125000)'Set the Baud rate of 125 kBit/s
```

Available Baud rates				
1000000.0000	888888.8889	800000.0000	727272.7273	666666.6667
615384.6154	571428.5714	533333.3333	500000.0000	470588.2353
444444.4444	421052.6316	400000.0000	380952.3810	363636.3636
347826.0870	333333.3333	320000.0000	307692.3077	296296.2963
285714.2857	266666.6667	250000.0000	242424.2424	235294.1176
222222.2222	210526.3158	205128.2051	200000.0000	190476.1905
181818.1818	177777.7778	173913.0435	166666.6667	160000.0000
156862.7451	153846.1538	148148.1481	145454.5455	142857.1429
140350.8772	133333.3333	126984.1270	125000.0000	123076.9231
121212.1212	117647.0588	115942.0290	114285.7143	111111.1111
106666.6667	105263.1579	103896.1039	102564.1026	100000.0000
98765.4321	95238.0952	94117.6471	90909.0909	88888.8889
87912.0879	86956.5217	84210.5263	83333.3333	81632.6531
80808.0808	80000.0000	78431.3725	76923.0769	76190.4762
74074.0741	72727.2727	71428.5714	70175.4386	69565.2174
68376.0684	67226.8908	66666.6667	66115.7025	64000.0000
63492.0635	62500.0000	61538.4615	60606.0606	60150.3759
59259.2593	58823.5294	57971.0145	57142.8571	55944.0559
55555.5556	54421.7687	53333.3333	52631.5789	52287.5817
51948.0519	51282.0513	50000.0000	49689.4410	49382.7160
48484.8485	47619.0476	47337.2781	47058.8235	46783.6257
45714.2857	45454.5455	44444.4444	43956.0440	43478.2609
42780.7487	42328.0423	42105.2632	41666.6667	41025.6410
40816.3265	40404.0404	40000.0000	39215.6863	38647.3430
38461.5385	38277.5120	38095.2381	37037.0370	36363.6364
36199.0950	35714.2857	35555.5556	35087.7193	34782.6087
34632.0346	34482.7586	34188.0342	33613.4454	33333.3333
33057.8512	32921.8107	32388.6640	32258.0645	32000.0000
31746.0317	31620.5534	31372.5490	31250.0000	30769.2308
30651.3410	30303.0303	30075.1880	29629.6296	29411.7647
29304.0293	29090.9091	28985.5072	28673.8351	28571.4286

Available Baud rates				
28070.1754	27972.0280	27777.7778	27681.6609	27586.2069
27210.8844	27027.0270	26936.0269	26755.8528	26666.6667
26315.7895	26143.7908	25974.0260	25806.4516	25641.0256
25396.8254	25078.3699	25000.0000	24844.7205	24767.8019
24691.3580	24615.3846	24390.2439	24242.4242	24024.0240
23809.5238	23668.6391	23529.4118	23460.4106	23391.8129
23255.8140	23188.4058	22988.5057	22857.1429	22792.0228
22727.2727	22408.9636	22222.2222	22160.6648	22038.5675
21978.0220	21739.1304	21680.2168	21621.6216	21505.3763
21390.3743	21333.3333	21276.5957	21220.1592	21164.0212
21052.6316	20833.3333	20779.2208	20671.8346	20512.8205
20460.3581	20408.1633	20202.0202	20050.1253	20000.0000
19851.1166	19753.0864	19704.4335	19656.0197	19607.8431
19512.1951	19323.6715	19230.7692	19138.7560	19047.6190
18912.5296	18867.9245	18823.5294	18648.0186	18604.6512
18518.5185	18433.1797	18390.8046	18306.6362	18181.8182
18140.5896	18099.5475	18018.0180	17857.1429	17777.7778
17738.3592	17582.4176	17543.8596	17429.1939	17391.3043
17316.0173	17241.3793	17204.3011	17094.0171	17021.2766
16949.1525	16913.3192	16842.1053	16806.7227	16771.4885
16666.6667	16632.0166	16563.1470	16528.9256	16460.9053
16393.4426	16326.5306	16260.1626	16227.1805	16194.3320
16161.6162	16129.0323	16000.0000	15873.0159	15810.2767
15779.0927	15686.2745	15625.0000	15594.5419	15503.8760
15473.8878	15444.0154	15384.6154	15325.6705	15238.0952
15180.2657	15151.5152	15122.8733	15094.3396	15065.9134
15037.5940	15009.3809	14842.3006	14814.8148	14705.8824
14652.0147	14571.9490	14545.4545	14519.0563	14492.7536
14414.4144	14336.9176	14311.2701	14285.7143	14260.2496
14184.3972	14109.3474	14035.0877	13986.0140	13937.2822
13913.0435	13888.8889	13840.8304	13793.1034	13722.1269

Available Baud rates				
13675.2137	13605.4422	13582.3430	13559.3220	13513.5135
13468.0135	13445.3782	13377.9264	13333.3333	13289.0365
13223.1405	13157.8947	13136.2890	13114.7541	13093.2897
13071.8954	13008.1301	12987.0130	12903.2258	12882.4477
12820.5128	12800.0000	12759.1707	12718.6010	12698.4127
12578.6164	12558.8697	12539.1850	12500.0000	12422.3602
12403.1008	12383.9009	12345.6790	12326.6564	12307.6923
12288.7865	12195.1220	12158.0547	12121.2121	12066.3650
12030.0752	12012.0120	11994.0030	11922.5037	11904.7619
11851.8519	11834.3195	11764.7059	11730.2053	11695.9064
11661.8076	11627.9070	11611.0305	11594.2029	11544.0115
11494.2529	11477.7618	11428.5714	11396.0114	11379.8009
11363.6364	11347.5177	11299.4350	11220.1964	11204.4818
11188.8112	11111.1111	11080.3324	11034.4828	11019.2837
10989.0110	10943.9124	10928.9617	10884.3537	10869.5652
10840.1084	10810.8108	10796.2213	10781.6712	10752.6882
10695.1872	10666.6667	10638.2979	10610.0796	10582.0106
10540.1845	10526.3158	10457.5163	10430.2477	10416.6667
10389.6104	10335.9173	10322.5806	10296.0103	10269.5764
10256.4103	10230.1790	10204.0816	10101.0101	10088.2724
10062.8931	10025.0627	10012.5156	10000.0000	9937.8882
9925.5583	9876.5432	9852.2167	9828.0098	9803.9216
9791.9217	9768.0098	9756.0976	9696.9697	9685.2300
9661.8357	9615.3846	9603.8415	9569.3780	9523.8095
9456.2648	9433.9623	9411.7647	9400.7051	9367.6815
9356.7251	9324.0093	9302.3256	9291.5215	9259.2593
9227.2203	9216.5899	9195.4023	9153.3181	9142.8571
9090.9091	9070.2948	9049.7738	9039.5480	9009.0090
8958.5666	8928.5714	8918.6176	8888.8889	8879.0233
8869.1796	8859.3577	8771.9298	8743.1694	8714.5969
8695.6522	8658.0087	8648.6486	8620.6897	8602.1505

Available Baud rates				
8592.9108	8556.1497	8547.0085	8510.6383	8483.5631
8474.5763	8465.6085	8456.6596	8421.0526	8403.3613
8385.7442	8333.3333	8281.5735	8264.4628	8255.9340
8230.4527	8205.1282	8196.7213	8163.2653	8130.0813
8113.5903	8105.3698	8097.1660	8088.9788	8080.8081
8064.5161	8000.0000	7976.0718	7944.3893	7936.5079
7905.1383	7843.1373	7812.5000	7804.8780	7797.2710
7774.5384	7751.9380	7736.9439	7729.4686	7714.5612
7692.3077	7662.8352	7655.5024	7619.0476	7590.1328
7575.7576	7561.4367	7547.1698	7532.9567	7518.7970
7469.6545	7441.8605	7421.1503	7407.4074	7400.5550
7386.8883	7352.9412	7326.0073	7285.9745	7272.7273
7259.5281	7246.3768	7187.7808	7168.4588	7142.8571
7136.4853	7130.1248	7111.1111	7098.4916	7092.1986
7054.6737	7017.5439	6993.0070	6956.5217	6944.4444
6926.4069	6902.5022	6896.5517	6861.0635	6820.1194
6808.5106	6802.7211	6791.1715	6779.6610	6734.0067
6688.9632	6683.3751	6666.6667	6611.5702	6578.9474
6568.1445	6562.7564	6557.3770	6535.9477	6530.6122
6493.5065	6456.8200	6451.6129	6441.2238	6410.2564
6400.0000	6379.5853	6349.2063	6324.1107	6289.3082
6274.5098	6269.5925	6250.0000	6245.1210	6211.1801
6172.8395	6163.3282	6153.8462	6144.3932	6102.2121
6060.6061	6046.8632	6037.7358	5997.0015	5961.2519
5952.3810	5925.9259	5895.3574	5865.1026	5847.9532
5818.1818	5797.1014	5772.0058	5747.1264	5714.2857
5702.0670	5681.8182	5649.7175	5614.0351	5610.0982
5555.5556	5521.0490	5517.2414	5464.4809	5434.7826
5423.7288	5376.3441	5333.3333	5291.0053	5245.9016
5208.3333	5161.2903	5079.3651	5000.0000	

SET_CAN_REG

CAN bus: The instruction **SET_CAN_REG** writes a value into a specified register of one of the CAN controllers.

Syntax

```
#INCLUDE ADWGCAN.INC  
  
SET_CAN_REG(Can_No, regno, value)
```

Parameters

Can_No	Number (1, 2) of the CAN interface	LONG
regno	Register number in the CAN controller (0...255)	LONG
value	Value (8 bits), which is written into the register.	LONG

Notes

The register list of the CAN controller can be found in the Intel® AN82527 datasheet.

See also

SET_CAN_BAUDRATE, GET_CAN_REG

Example

```
#INCLUDE ADWGCAN.INC  
  
INIT:  
    INIT_CAN(1)           'Initialization of CAN controller 1  
    SET_CAN_REG(1,0,1)    'Set control register to the value 1
```

TRANSMIT

CAN bus: The instruction **TRANSMIT** sends the message in `CAN_MSG` via the specified message object of a CAN controller.

Syntax

```
#INCLUDE ADWGCAN.INC  
  
TRANSMIT (Can_No, msgno)
```

Parameters

<code>Can_No</code>	Number (1, 2) of the CAN interface	LONG
<code>msgno</code>	Number (1...14) of the message object	LONG

Notes

To send a message you have to follow the correct order:

- Enable the message object with **EN_TRANSMIT** for sending (only once).
- Enter the message into the array `CAN_MSG`: Data bytes and number of data bytes.
- Send the message with **TRANSMIT**.

The CAN interface will send the message as soon as the message object has received access rights to the CAN bus.

See also

INIT_CAN, READ_MSG, EN_TRANSMIT

Example

REM Sends a 32 bit FLOAT value (here: Pi) as sequence of
REM 4 bytes in a message object
REM (Receiving of a float value see Example of READ_MSG)

```
#INCLUDE ADWGCAN.INC
#DEFINE pi 3.14159265
DIM i AS LONG

INIT:
    INIT_CAN(2)                'Initialize CAN-Controller 2

    REM Initialize message object 6 of controller 2
    REM for sending of CAN messages with the identifier 40
    EN_TRANSMIT(2, 6,40,0)

    REM Create bit pattern of Pi with data type Long
    PAR_1 = CAST_FLOATTOLONG(pi)

    REM divide bit pattern (32 Bit) into 4 bytes
    CAN_MSG[4] = PAR_1 AND 0FFh 'assign LSB
    FOR i = 1 TO 3
        CAN_MSG[4-i] = SHIFT_RIGHT(PAR_1,8) AND 0FFh
    NEXT i
    CAN_MSG[9] = 4              'message length in bytes

EVENT:
    TRANSMIT(2,6)              'Sends the message object 6
```

CHECK_SHIFT_REG

RSxxx: The instruction `CHECK_SHIFT_REG` returns, if all data has been sent, which was written into the send-FIFO of the RSxxx interface.

Syntax

```
#INCLUDE ADWGCAN.INC

ret_val = CHECK_SHIFT_REG(interface)
```

Parameters

<code>interface</code>	number (1, 2) of the RSxxx interface that is to be read out	LONG
<code>ret_val</code>	Sending status: LONG 0: Data has been sent (= no more data in the send-FIFO). 1: Not yet all data sent (= the send-FIFO still contains data).	

Notes

With the return value 0 both the send FIFO and the output shift register are empty. With the return value 1 there is at least one bit to be sent.

We recommend to use this instruction only after you have more experience about how the controller operates (data-sheet of the manufacturer Texas Instruments). For more common applications more comfortable instructions are available in the include file.

See also

GET_RS, RS_INIT, RS_RESET, WRITE_FIFO

Example

```
#INCLUDE ADWGCAN.INC

EVENT:
...
PAR_1 = CHECK_SHIFT_REG(1) 'Check if RSxxx interface 1 still
                             'has data to send
...
```

GET_RS

RSxxx: The instruction **GET_RS** reads out a specified controller register.

Syntax

```
#INCLUDE ADWGCAN.INC  
ret_val = GET_RS(reg_addr)
```

Parameters

reg_addr	Address of the controller register to read	LONG
ret_val	Contents of the controller register	LONG

Notes

We recommend to use this instruction only after you have more experience about how the controller operates (data-sheet of the manufacturer Texas Instruments). For more common applications more comfortable instructions are available in the include file.

See also

CHECK_SHIFT_REG, RS_INIT, RS_RESET, SET_RS

Example

-/-

READ_FIFO

RSxxx: The instruction **READ_FIFO** reads a value from the input FIFO of a specified RSxxx interface.

Syntax

```
#INCLUDE ADWGCAN.INC

result = READ_FIFO(interface)
```

Parameters

<code>interface</code>	number (1, 2) of the RSxxx interface that is to be read out	LONG
<code>ret_val</code>	Contents of the input FIFO: -1: FIFO is empty ≥0: Transferred value	LONG

Notes

-/-

See also

RS_INIT, RS_RESET, RS485_SEND, WRITE_FIFO

Example

```
#INCLUDE ADWGCAN.INC

INIT:
  RS_RESET()
  RS_INIT(1,9600,0,8,0,1) 'Initialization of RSxxx interface 1
                          'with 9600 Baud, without parity,
                          '8 data bits, 1 stop bit and
                          'hardware handshake.

EVENT:
  PAR_1 = READ_FIFO(1) 'Get a value from the FIFO. If
                      'the FIFO is empty, -1 is returned.
```


RS_INIT

RSxxx: The instruction **RS_INIT** initializes one RSxxx interface.

The following parameters are set:

- Transfer rate in Baud
- Use of test bits
- Data length
- Amount of stop bits
- Transfer protocol (handshake)

Syntax

```
#INCLUDE ADWGCAN.INC

RS_INIT(interface,baud,parity,bits,stop,
        handshake)
```

Parameters

interface	Number of RSxxx interface (1, 2), which is to be initialized	LONG
baud	Transfer rate in Baud	LONG
parity	Use of test bits: 0: without parity bit 1: even parity 2: odd parity	LONG
bits	Amount of data bits (5, 6, 7 or 8)	LONG
stop	Amount of stop bits 0: 1 stop bit 1: 1½ stop bits at 5 data bits; 2 stop bits at 6, 7 or 8 data bits	LONG
handshake	Transfer protocol: 0: RS232, No handshake 1: RS232, Hardware handshake (RTS/CTS) 2: RS232, Software handshake (Xon/Xoff) 3: RS485 (default)	LONG

Notes

This instruction is necessary before working first with the selected RSxxx interface, in order to set the interface parameters. They must be identical to the remote station, in order to verify a correct transfer.

The initialization is necessary after you have executed a hardware reset with the instruction **RS_RESET**.

If transfer protocol RS485 is set, the transfer direction must be set, too (with **RS485_SEND**).

See also

CHECK_SHIFT_REG, GET_RS, RS485_SEND, RS_RESET, SET_RS

Example

```
#INCLUDE ADWGCAN.INC
```

```
INIT:
```

```
  RS_RESET()           'Reset RSxxx controller
  RS_INIT(1,9600,0,8,0,1) 'Initialization of RSxxx interface 1
                          'with 9600 Baud, without parity,
                          '8 data bits, 1 stop bit and
                          'hardware handshake.
```

RS_RESET

RSxxx: The instruction **RS_RESET** executes a hardware reset and deletes the settings for all RSxxx interfaces.

Syntax

```
#INCLUDE ADWGCAN.INC  
  
RS_RESET()
```

Notes

The instruction sends a reset impulse to the input of the controller TL16C754. In the data-sheet of the controller 16C754 from Texas Instruments it is described, to which values the registers have been set after the hardware reset.

After a hardware reset an initialization with **RS_INIT** must follow, in order to initialize the controller and to set the interface parameters.

See also

CHECK_SHIFT_REG, GET_RS, RS_INIT, SET_RS

Example

```
#INCLUDE ADWGCAN.INC  
  
INIT:  
    RS_RESET()                'Reset RSxxx controller  
    RS_INIT(1,9600,0,8,0,1) 'Initialization of RSxxx interface 1  
                             'with 9600 Baud, without parity,  
                             '8 data bits, 1 stop bit and  
                             'hardware handshake.
```

RS485_SEND

RSxxx: The instruction **RS485_SEND** determines the transfer direction for a specified RSxxx interface.

Syntax

```
#INCLUDE ADWGCAN.INC  
RS485_SEND(interface,dir)
```

Parameters

interface	RSxxx interface to be set (1, 2)	LONG
dir	Transfer direction of the RSxxx interface: 0: Set RSxxx interface to receive 1: Set RSxxx interface to send 2: Set RSxxx interface to send and to receive its sent data 3: Mute RSxxx interface, i.e. the interface works as receiver but doesn't put data into the input FIFO.	LONG

Notes

Setting the transfer direction means:

- Receiver: The RSxxx interface can only read data, even if data are in the output FIFO of the controller for this RSxxx interface.
- Sender: The RSxxx interface transfers data to the bus which are read by other devices.
- Sender/receiver: The RSxxx interface can transfer data to the bus and back at the same time. Thus, the sent data can be checked.

See also

CHECK_SHIFT_REG, GET_RS, RS_INIT, RS_RESET, SET_RS

Example

-/-

SET_RS

RSxxx: The instruction **SET_RS** writes a value into a specified register of the controller.

Syntax

```
#INCLUDE ADWGCAN.INC  
  
SET_RS(reg_addr,value)
```

Parameters

reg_addr	Number of the register, into which data are written	LONG
value	Value to be written into the register	LONG

Notes

We recommend to use this instruction only after you have more experience about how the controller operates (data-sheet of the manufacturer: TL16C754 from Texas Instruments). For more common applications more comfortable instructions are available in the include file.

See also

GET_RS, RS_INIT, RS_RESET

Example

-/-

WRITE_FIFO

RSxxx: The instruction **WRITE_FIFO** writes a value into the send-FIFO of a specified RSxxx interface.

Syntax

```
#INCLUDE ADWGCAN.INC  
  
ret_val = WRITE_FIFO(interface,value)
```

Parameters

interface	RSxxx interface number (1, 2) to whose send-FIFO data are transferred	LONG
value	Value to be written into the send-FIFO	LONG
ret_val	Status message: 0: Data are transferred successfully. 1: Data were not transferred, send-FIFO is full.	LONG

Notes

The instruction checks first if there is at least one free memory cell in the send-FIFO. If so, the transferred value is written into the FIFO (return value 0); otherwise a 1 is returned, indicating that the FIFO is full and writing is not possible.

See also

CHECK_SHIFT_REG, READ_FIFO, RS_INIT, RS_RESET, RS485_SEND

Example

```
#INCLUDE ADWGCAN.INC
DIM val AS LONG

INIT:
  RS_RESET()
  RS_INIT(1,9600,0,8,0,1)'Initialization of RSxxx interface 1
                          'with 9600 Baud, no parity,
                          '8 data bits, 1 stop bit and
                          'hardware handshake.

EVENT:
  PAR_1 = WRITE_FIFO(1,val)'If the FIFO is not full, [val]
                          'is written into the FIFO. Otherwise
                          'a 1 in PAR_1 indicates that writing
                          'into the FIFO ist not possible
                          '(FIFO full).
```


SSI_MODE

SSI: The instruction **SSI_MODE** sets the modes of all SSI decoders, either "single shot" (read out once) or "continuous" (read out continuously).

Syntax

```
#INCLUDE ADWGCAN.INC
```

```
SSI_MODE(pattern)
```

Parameters

pattern

Operation mode of the SSI decoders, indicated LONG as bit pattern. A bit is assigned to each of the decoders (see table).

Bit = 0: "Single shot" mode, the encoder is read out once.

Bit = 1: "Continuous" mode, the encoder is read out continuously.

Bit no.	31:2	3	2	1	0
SSI decoder	–	4	3	2	1

Notes

If you select the mode "continuous", reading the encoder is started immediately. The instruction **SSI_START** is not necessary for this.

Using the "continuous" mode, some encoder types occasionally return the wrong counter value 0 (zero) instead of the correct counter value. This error does not occur with the "single shot" mode.

See also

SSI_READ, SSI_SET_BITS, SSI_SET_CLOCK, SSI_START, SSI_STATUS

Example

```
#INCLUDE ADWGCAN.INC
```

INIT:

```
SSI_SET_CLOCK(1,200) 'clock rate for decoder 1 = 50 kHz
SSI_SET_CLOCK(2,200) 'clock rate for decoder 2= 50 kHz
SSI_MODE(11b)        'Set continuous-mode
                     '(for encoders 1+2)

SSI_SET_BITS(1,23)    '23 encoder bits for encoder 1
SSI_SET_BITS(2,23)    '23 encoder bits for encoder 2
```

EVENT:

```
PAR_1 = SSI_READ(1)  'Read out position value (encoder 1)
PAR_2 = SSI_READ(2)  'Read out position value (encoder 2)
```

SSI_READ

SSI: The instruction **SSI_READ** returns the last saved counter value of a specified SSI counter.

Syntax

```
#INCLUDE ADWGCAN.INC  
  
ret_val = SSI_READ(dcdr_no)
```

Parameters

dcdr_no	Number (1...4) of the SSI decoder whose counter value is to be read.	LONG
ret_val	Last counter value of the SSI counter (= absolute value position of the encoder).	LONG

Notes

An encoder value is saved when the bits indicated by **SSI_SET_BITS** are read.

See also

SSI_MODE, SSI_SET_BITS, SSI_SET_CLOCK, SSI_START, SSI_STATUS

Example

```

#INCLUDE ADWGCAN.INC
DIM m, n, y AS LONG

INIT:
    SSI_SET_CLOCK(1,50)    'clock rate for decoder 1 = 200 kHz
    SSI_MODE(1)            'Set continuous-mode (encoder 1)
    SSI_SET_BITS(1,23)     '23 encoder bits for encoder 1

EVENT:
    PAR_1 = SSI_READ(1)    'Read out position value (encoder 1)

    REM Change value from Gray-code into a binary value:
    m = 0                    'delete value of the last conversion
    y = 0                    ' _"-
    FOR n = 1 TO 32          'Check all 32 possible bits
        m = (SHIFT_RIGHT(PAR_1, (32 - n)) AND 1) XOR m
        y = (SHIFT_LEFT(m, (32 - n))) OR y
    NEXT n
    PAR_9 = y                'The result of the Gray/binary
                              'conversion in PAR_9

```

SSI_SET_BITS

SSI: The instruction **SSI_SET_BITS** sets for an SSI counter the amount of bits which generate a complete encoder value.

The number of bits should be equal to the resolution of the encoder.

Syntax

```
#INCLUDE ADWGCAN.INC

SSI_SET_BITS(dcdr_no,bit_no)
```

Parameters

<code>dcdr_no</code>	Number (1...4) of the SSI decoder whose resolution is to be set.	LONG
<code>bit_no</code>	Amount of bits (1...32) of the bits which are to be read for the encoder (corresponds to the encoder resolution).	LONG

Notes

The resolution (amount of bits) of the SSI encoder should be similar to the amount of bits which are transferred.

See also

SSI_MODE, SSI_READ, SSI_SET_CLOCK, SSI_START, SSI_STATUS



Example

```
#INCLUDE ADWGCAN.INC

INIT:
  SSI_SET_CLOCK(1,50)  'clock rate for decoder 1 = 200 kHz
  SSI_SET_CLOCK(2,50)  'clock rate for decoder 2= 200 kHz
  SSI_MODE(11b)        'Set continuous-mode (encoders 1+2)
  SSI_SET_BITS(1,10)   '10 encoder bits for encoder 1
  SSI_SET_BITS(2,25)   '25 encoder bits for encoder 2

EVENT:
  PAR_1 = SSI_READ(1)  'Read out position value (encoder 1)
  PAR_2 = SSI_READ(2)  'Read out position value (encoder 2)
```

SSI_SET_CLOCK

SSI: The instruction **SSI_SET_CLOCK** sets the clock rate (approx. 40kHz to 1 MHz) , with which the encoder is clocked.

Syntax

```
#INCLUDE ADWGCAN.INC

SSI_SET_CLOCK(dcdr_no,prescale)
```

Parameters

<code>dcdr_no</code>	Number (1...4) of the SSI decoder whose clock rate is to be set.	LONG
<code>prescale</code>	scale factor (10...255) for setting the clock rate according to the equation: Clock rate = 10MHz / <code>prescale</code>	LONG

Notes



The setting of the clock rate is always identical for both encoders, which are connected to the and cannot be set separately. If necessary, the clock has to consider the clock rate of the slowest encoder.

Scale factors < 10 are automatically corrected to the value 10; from values > 255 only the least significant 8 bits are used as scale factor.

The possible clock frequency depends on the length of the cable, cable type, and the send and receive components of the encoder or decoder. Basically the following rule applies: The higher the clock frequency the shorter the cable length.

See also

SSI_MODE, SSI_READ, SSI_SET_BITS, SSI_START, SSI_STATUS

Example

```
#INCLUDE ADWGCAN.INC

INIT:
    SSI_SET_CLOCK(1,10)    'clock rate for decoder 1 = 1 MHz
    SSI_SET_CLOCK(2,20)    'clock rate for decoder 2 = 0,5 MHz
    SSI_MODE(11b)          'Set continuous-mode for encoder 1+2
    SSI_SET_BITS(1,10)     '10 encoder bits for encoder 1
    SSI_SET_BITS(2,25)     '25 encoder bits for encoder 2

EVENT:
    PAR_1 = SSI_READ(1)    'Read out position value (encoder 1)
    PAR_2 = SSI_READ(2)    'Read out position value (encoder 2)
```

SSI_START

SSI: The instruction **SSI_START** starts the reading of one or both SSI encoders (only in mode "single shot").

Syntax

```
#INCLUDE ADWGCAN.INC
```

```
SSI_START(dcdr_no)
```

Parameters

dcdr_no Bit pattern for selecting the SSI decoders which are to be started:
 Bit = 0: No function
 Bit = 1: Start reading of the SSI decoder

LONG

Bit no.	31:2	3	2	1	0
SSI decoder	–	4	3	2	1

Notes

In the continuous mode this instruction has no function, because the encoder values are nevertheless read out continuously.

An encoder value will be saved only when the amount of bits is read which is set by **SSI_SET_BITS**.



A complete encoder value is always transferred, even if the operation mode is changing meanwhile.

See also

SSI_MODE, SSI_READ, SSI_SET_BITS, SSI_SET_CLOCK, SSI_STATUS

Example

```
#INCLUDE ADWGCAN.INC

INIT:
    SSI_SET_CLOCK(1,250)  'clock rate for decoder 1 = 40 kHz
    SSI_SET_CLOCK(2,250)  'clock rate for decoder 2= 40 kHz
    SSI_MODE(0)           'Set single shot-mode (all counters)
    SSI_SET_BITS(1,23)     '23 encoder bits for encoder 1
    SSI_SET_BITS(2,23)     '23 encoder bits for encoder 2

EVENT:
    SSI_START(11b)         'Read position value of encoders 1 & 2
    DO                     'for encoder 1:
    UNTIL (SSI_STATUS(1) = 0) 'If position value is read
                             'completely, then ...
    PAR_1 = SSI_READ(1)     'read out and display position value
    DO                     'For encoder 2:
    UNTIL (SSI_STATUS(2) = 0) 'If position value is read
                             'completely, then ...
    PAR_1 = SSI_READ(2)     'read out and display position value
```

SSI_STATUS

SSI: The instruction **SSI_STATUS** returns the current read-status on the specified module for a specified decoder.

Syntax

```
#INCLUDE ADWGCAN.INC  
  
ret_val = SSI_STATUS (dcd_r_no)
```

Parameters

dcd_r_no	Number (1...4) of the SSI decoder whose status is to be queried.	LONG
ret_val	Read-status of the decoder: 0: Decoder is ready, that is a complete value was has been read. 1: Decoder is reading an encoder value.	LONG

Notes

Use the status query only in the SSI mode "single shot". In the mode "continuous" querying the status is not useful.

See also

SSI_MODE, SSI_READ, SSI_SET_BITS, SSI_SET_CLOCK, SSI_START

Example

```
#INCLUDE ADWGCAN.INC

INIT:
  SSI_SET_CLOCK(1,250) 'clock rate for decoder 1 = 40 kHz
  SSI_SET_CLOCK(2,250) 'clock rate for decoder 2= 40 kHz
  SSI_MODE(0) 'Set single shot-mode (all counters)
  SSI_SET_BITS(1,23) '23 encoder bits for encoder 1
  SSI_SET_BITS(2,23) '23 encoder bits for encoder 2

EVENT:
  SSI_START(11b) 'Read position value of encoders 1 & 2
  DO 'For encoder 1:
  UNTIL (SSI_STATUS(1) = 0) 'If position value is completely
    'read, then ...
  PAR_1 = SSI_READ(1) 'Read out and display position value
  DO 'For encoder 2:
  UNTIL (SSI_STATUS(2) = 0) 'If position value is completely
    'read, then ...
  PAR_1 = SSI_READ(2) 'Read out and display position value
```



7 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 `Options\Compiler`. 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.

Appendicies

A-1 Short-Cuts in ADbasic

Short cut key	Function	Matching menu item
CTRL + F5	<i>Boot ADwin system</i>	Build► Boot
F8	Compile source code	Build► Compile
CTRL + F8	Start process	Build► Start
F9	Stop process	Build► Stop
CTRL + R	Colour mark used parameters	Parameter window: Icon 
CTRL + B	Comment marked lines	Source context menu: Comment Block
CTRL + SHIFT + B	Uncomment marked lines	Source context menu: Uncomment Block
TAB	Indent marked lines	Source context menu: Indent
SHIFT + TAB	Outdent marked lines	Source context menu: Outdent
CTRL + N	New source code file	File► New
CTRL + O	Open source code file	File► Open
CTRL + S	Save source code file	File► Save
CTRL + P	Print source code file	File► Print
CTRL + Z	Undo input	Edit► Undo
CTRL + Y	Redo input	Edit► Redo
CTRL + X	Cut	Edit► Cut
CTRL + C	Copy	Edit► Copy
CTRL + V	Paste	Edit► Paste
CTRL + A	Select all	Edit► Select All
CTRL + F	Find text	Edit► Find
F3	Continue search text	Edit► Find Next
CTRL + H	Replace text	Edit► Replace

Short cut key	Function	Matching menu item
F1	Call help topic for marked instruction	Help ► Help Topics

A-2 ASCII-Character Set

NUL 00h 0	SOH 01h 1	STX 02h 2	ETX 03h 3	EOT 04h 4	ENQ 05h 5	ACK 06h 6	BEL 07h 7
BS ¹ 08h 8	TAB ² 09h 9	LF ³ 0Ah 10	VT 0Bh 11	FF 0Ch 12	CR ⁴ 0Dh 13	SO 0Eh 14	SI 0Fh 15
DLE 10h 16	DC1 11h 17	DC2 12h 18	DC3 13h 19	DC4 14h 20	NAK 15h 21	SYN 16h 22	ETB 17h 23
CAN 18h 24	EM 19h 25	SUB 1Ah 26	ESC 1Bh 27	FS 1Ch 28	GS 1Dh 29	RS 1Eh 30	US 1Fh 31
SPC ⁵ 20h 32	! 21h 33	" 22h 34	# 23h 35	\$ 24h 36	% 25h 37	& 26h 38	' 27h 39
(28h 40) 29h 41	* 2Ah 42	+ 2Bh 43	, 2Ch 44	- 2Dh 45	. 2Eh 46	/ 2Fh 47
0 30h 48	1 31h 49	2 32h 50	3 33h 51	4 34h 52	5 35h 53	6 36h 54	7 37h 55
8 38h 56	9 39h 57	: 3Ah 58	; 3Bh 59	< 3Ch 60	= 3Dh 61	> 3Eh 62	? 3Fh 63
@ 40h 64	A 41h 65	B 42h 66	C 43h 67	D 44h 68	E 45h 69	F 46h 70	G 47h 71
H 48h 72	I 49h 73	J 4Ah 74	K 4Bh 75	L 4Ch 76	M 4Dh 77	N 4Eh 78	O 4Fh 79
P 50h 80	Q 51h 81	R 52h 82	S 53h 83	T 54h 84	U 55h 85	V 56h 86	W 57h 87
X 58h 88	Y 59h 89	Z 5Ah 90	[5Bh 91	\ 5Ch 92] 5Dh 93	^ 5Eh 94	_ 5Fh 95
` 60h 96	a 61h 97	b 62h 98	c 63h 99	d 64h 100	e 65h 101	f 66h 102	g 67h 103
h 68h 104	i 69h 105	j 6Ah 106	k 6Bh 107	l 6Ch 108	m 6Dh 109	n 6Eh 110	o 6Fh 111
p 70h 112	q 71h 113	r 72h 114	s 73h 115	t 74h 116	u 75h 117	v 76h 118	w 77h 119
x 78h 120	y 79h 121	z 7Ah 122	{ 7Bh 123	 7Ch 124	} 7Dh 125	~ 7Eh 126	□ 7Fh 127

¹ Backspace, ² Tabulator, ³ Linefeed,

⁴ Carriage Return, ⁵ Space

A-3 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,
- b) electronically transfer the *ADbasic* software from one computer to another over a network or a data transfer channel,

- c) change or modify, translate, reverse engineer, decompile or disassemble the *ADbasic* software without prior written consent of Jaeger Messtechnik GmbH.

4. OWNERSHIP

- 4.1 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.

- 4.2 Jaeger Messtechnik GmbH reserves all rights for publication, copying, processing and commercialization of the *ADbasic* software.

5. COPYRIGHTS

- 5.1 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.

- 5.2 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.

6. GRANT OF LICENSE

- 6.1 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.

- 6.2 You must not rent, lease or lend the *ADbasic* software.

7. PERIOD OF AGREEMENT

- 7.1 The period of the License Agreement is unlimited.
- 7.2 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.

8. CLAIM FOR DAMAGES AND PENALTY UPON VIOLATION OF THE CONTRACT

8.1 If the Licensee violates conditions of this License Agreement he must pay damages.

8.2 Notwithstanding, 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.

8.3 The title to omission on completion of the contract is not influenced by the claim for damages and the penalties.

9. 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.

10. 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.
- b) 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.
- c) 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.

- d) 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.
- e) 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.
- f) 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.

11. 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.

A-4 Command Line Calling Syntax

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 term and functionality "command line call" come 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. Not all compiler settings can be made via command line call.

A-4.1 Syntax

A command line call consists of at least the name of the program you are calling and of the file which is to be compiled (each with path and file name). You can add command line options, beginning with a slash `/`, some of which have optional parameters.

The command line call is entered in a single line.

Syntax

```
{ [LW:\] [path\]}ADbasic{.exe} /L /M  
{ [LW:\] [path\]}infile{.bas} {/Sx} {/Px}  
{/A{ [LW:\] [path\]}outfile}
```

Options

[LW:\]	Optional: Drive or hard disk. For the program <ADbasic.exe> it usually is C:\.
[path\]	Optional: Subdirectory where the program <ADbasic.exe> or the source code files are located. With standard installation this is C:\ADwin\ADbasic\.
infile	File name of the source code you want to compile.
/L	Compile the source code and generate a library file with the extension <code>LIx</code> (excludes the option <code>/M</code>).
x	Stands for the processor on which the compiled file is to run (see option <code>/P</code>).
/M	Compile the source code and generate a binary file with the extension <code>Txn</code> (excludes the option <code>/L</code>).
x	Stands for the processor on which the compiled file is to run (see option <code>/P</code>).
n	Stands for the process number of the compiled file (is read from the source code file).
/Sx	ADwin system for which the file is compiled: /SC Cards (ISA bus;16-bit resolution = No) /SL Light-16 (16-bit resolution = Yes) /SG Gold (16-bit resolution = Yes); default /SP Pro (16-bit resolution = Yes)
/Px	Processor type for which the file is compiled: /P2 Processor T2 /P4 Processor T4 /P5 Processor T5 /P8 Processor T8 /P9 Processor T9 (ADSP); default /P10 Processor T10 (ADSP)

`/Aoutfile` Path and name of the binary or library file `<outfile>` which is to be generated.

A-4.2 Notes

Optional information in the Syntax is set into braces. The order of options can be arranged any way you like. Command lines are not case sensitive.

The `Debug Mode` option is never active when a compilation is effected via command line.

If the option `/A` is not used, the generated binary or library file is saved in the same directory, as the source code.

The following options are mutually exclusive:

Option	excluded then:
<code>/L</code>	<code>/M</code>
<code>/M</code>	<code>/L</code>
<code>/SG, /SL</code>	<code>/P2, /P4, /P5, /P8</code>
<code>/SP</code>	<code>/P2</code>

A-4.3 Examples



```
C:\ADwin\ADbasic\ADbasic.exe /L Z:\Myfiles\test.bas
```

This command line compiles the source code `<test.bas>` and generates the library file `<test.lib>` in the directory `<Z:\Myfiles\>`. Since nothing else is indicated, the default setting is used:

- Processor T9
- Gold system (16-bit system = Yes)
- save generated file in the directory of the source code file

If you are already in the directory `<C:\ADwin\ADbasic>`, you can shorten this line to:

```
ADbasic.exe /L Z:\Myfiles\test.bas
```

The shortest version is when the source code is in the same directory `<C:\ADwin\ADbasic>` (here without file name extension):

```
ADbasic /L test
```



```
C:\ADwin\ADbasic\ADbasic /L Z:\Myfiles\test.bas /SL
```

This command line compiles the source code `<string.bas>` into a library file for a *Light-16* system with the processor T9.

The same call, for the processor T10 only, is as follows:

```
C:\ADwin\ADbasic\ADbasic /L Z:\Myfiles\test.bas /P10 /SL
```



```
C:\ADwin\ADbasic\ADbasic /M Z:\Myfiles\test.bas  
C:\ADwin\ADbasic\samples_ADwin\bas_dmo6f /P9 /SG
```



Compiles the demo file <bas_dmo6f.bas> into a binary file for a *Gold* system with T9 processor.

```
C:\ADwin\ADbasic\ADbasic /M  
C:\ADwin\ADbasic\samples_ADwin\bas_dmo6 /P8 /SL
```



Compiles the demo file <bas_dmo6.bas> into a binary file for a *Light-16* card with the processor T8.

```
C:\ADwin\ADbasic\ADbasic /M C:\user\my_file.bas /P4 /SC  
/Ayour_file
```



This instruction compiles the file <my_file.bas> for an *ADwin-Card* with the processor T4. The generated binary file has the name <your_file.T41> and can be found in the same directory where the source code is saved: <C:\user>.

```
C:\ADwin\ADbasic\ADbasic.exe /M C:\user\my_file.bas  
/AY:\somewhere\your_file
```



The binary file now has the name <your_file.T91> and can be found in the directory <Y:\somewhere>.

A-4.4 Special Settings and Messages

Special Settings

Take into account, where the compiler gets its information. Some compiler settings cannot be entered via command line, but rather only in the development environment *ADbasic*.



- The development environment (transparently) saves the settings Event, Process, Priority, Optimize in the source code file.
- The paths for Include-Directory and Lib-Directory are saved in the registry.

It is possible that the registry may have changed, especially if you or another user has worked with a different project on your computer. Therefore we recommend you check these settings before you execute a command line.

Warnings and Error Messages

If warnings or errors occur during compilation, they are saved in the files `<filename.WRN>` and `<filename.ERR>`. The error messages are the same as those that *ADbasic* displays in the info window (see chapter 2.3.13 on page 36).

We recommend you delete 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-5 Obsolete Program Parts

For compatibility reasons the development environment *ADbasic 4* also offers settings for *ADwin* systems with transputer processors (T4, T5, T8), as well as help programs with new updates from the previous version of *ADbasic*.

A-5.1 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.

Processors T2 ... T8 only

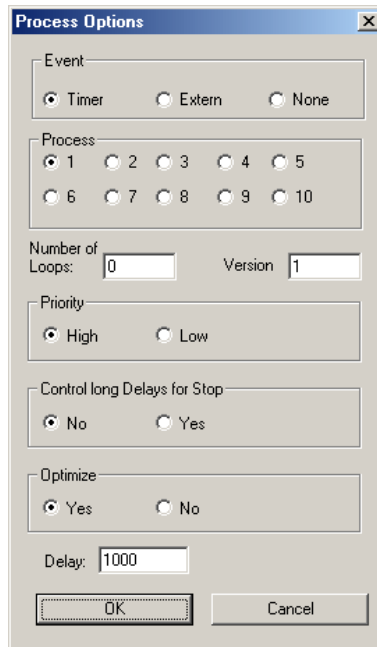


Fig. 1 – The Dialog Window Process Options

- **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 **GLOBALDELAY** to define time intervals which triggers 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 the instruction **EVENTENABLE**.

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 needn't 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 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 5.1 "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 you use the option in this case, because this option will speed up the stop procedure.

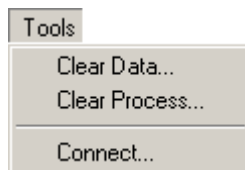
- **Optimize**: The optional optimization shortens the process execution time of up to 20 percent. A higher setting under `Level` leads to shorter execution times.

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-5.2 The Menu Item `Connect`

In the menu `Tools` you can find among other things the item `Connect`. It opens the dialog window of the same name, where you make the settings for the program *ADserver* (which is no longer being updated). With *ADserver* you setup a network connection to the *ADwin* system.



We recommend you use the program *ADwin TCP/IPserver* instead of *ADserver*. In this case, do not make any settings in the dialog window, but close it!

You will find more information in the online help of *ADwin TCP/IPserver*.

If you still want to use the program *ADserver*:

From *ADbasic* you can access an *ADwin* system, connected to any PC, via a network (LAN, ISDN, Internet, ...). The program *ADserver* must be started on the PC before trying to access the *ADwin* system. Next you enter the network settings in the dialog window.

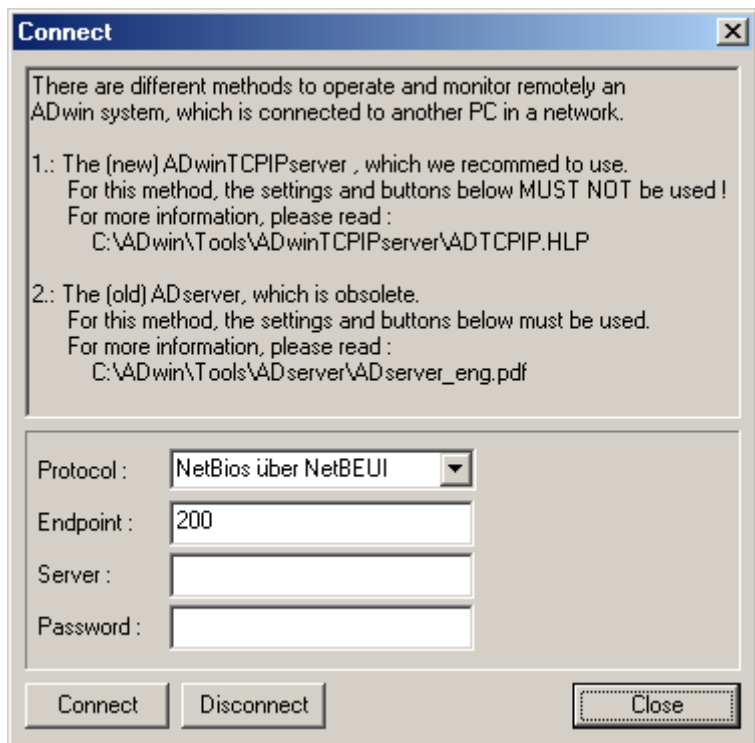


Fig. 2 – The dialog window Connect

- **Protocol:** The protocol, your network uses. It must be installed on your computer.
- **Endpoint:** End point for the network communication.
- **Server:** Name or address of the network computer you want to setup a connection to.
- **Password:** Password for the program *ADserver*. Pay attention this is case sensitive.

The settings **Protocol**, **Endpoint** and **Password** must be identical to the settings in the program *ADserver* on the network computer.

As soon as you click the **Connect** button, the connection to the network computer and the connected *ADwin* system, is set up. All further actions of the

development environment are transferred to this system. By clicking `Disconnect` the connection is disconnected again.

A-6 Index

Symbols

- · 87
- # · 91
- #DEFINE · 110
- #ELSE · 134
- #ENDIF · 134
- #IF · 134
- #INCLUDE · 138
- * · 88
- + · 85
- + (String) · 86
- .NET · 82
- / · 89
- : · 92
- < = > · 94
- = · 93
- ^ · 90
- ' (REM) · 159

Numerics

- 150h, see device no.
- 2-dimensional arrays · 50

A

- ABSF · 95
- ABSI · 96
- absolute value
 - floating point number · 95
 - integer number · 96
- ActiveX · 82
 - communication to the ADwin system · 81
 - use from a development environment · 82
- ADbasic
 - start · 7
- ADC · 192

- ADC12 · 195
- ADC14, see ADC12
- Add Open Files to Project · 33
- Add to Project · 12, 33
- addition · 85
- ADtools · 37
- ADwin32.dll · 81
- analog in-/outputs
 - ADC:measure a channel · 192
 - DAC: output one value · 202
 - read converted value · 209
 - read converted value (12 Bit) · 210
 - set multiplexer · 214
 - start a conversion · 216
 - wait for end of conversion · 218
- analyze
 - general · 66
 - process flow · 70
 - run-time error · 67
 - timing · 67
- AND · 97
- arc cosine: ARCCOS · 99
- arc sine: ARCSIN · 100
- arc tangent: ARCTAN · 101
- arithmetic functions
 - · 87
 - * · 88
 - + · 85
 - / · 89
 - ^ · 90
 - DEC · 109
 - EXP · 117
 - INC · 137
 - LN · 147
 - LOG · 150
 - SQRT · 166
- Array-Index (local) too large / <1,
see run-time error

arrays

- 2-dimensional · 50
- allocate memory area · 49
- DATA_n · 107
- FIFO · 118
- global · 46
 - first element · 47
- initialize · 41
- local · 48
 - first element · 49
- overview · 42
- (DIM) AS · 112
- ASC · 102
- ASCII-character set · 3
- assign a value · 44
- assignment (=) · 93
- (DIM ...) AT · 112
- autoindent option · 22
- automatical type conversion · 58

B

- backslash (control character) · 55
- base e · 117
- binary file
 - see also library
 - see menu, build
 - create
 - from ADbasic · 16
- binary file see library
- binary notation · 45
- bit shifting
 - left · 162
 - right · 163
- booting · 7
- break, see stop process
- BTL file
 - directory settings · 24
- bypass waiting time · 151

C

- C#.NET, C++ · 82
- CAN bus (Gold CAN)
 - CAN_MSG · 278
 - EN_CAN_INTERRUPT · 280
 - EN_RECEIVE · 281
 - EN_TRANSMIT · 282
 - GET_CAN_REG · 283
 - INIT_CAN · 284
 - READ_MSG · 285
 - SET_CAN_BAUDRATE · 287
 - SET_CAN_REG · 293
 - TRANSMIT · 294
- CAN bus (L16 DIO1)
 - CAN_MSG · 259
 - EN_INTERRUPT · 261
 - EN_RECEIVE · 262
 - EN_TRANSMIT · 263
 - GET_CAN_REG · 264
 - INIT_CAN · 265
 - READ_MSG · 266
 - SET_CAN_BAUDRATE · 268
 - SET_CAN_REG · 274
 - TRANSMIT · 275
- CAN_MSG (Gold CAN) · 278
- CAN_MSG (L16 DIO1) · 259
- carriage return (control character) · 55
- case sensitivity · 10
- CASE, CCASE, CASEELSE (SELECTCASE ...) · 160
- CAST_FLOATTOLONG · 103
- CAST_LONGTOFLOAT · 104
- check
 - number and priority of processes · 68
- CHECK_SHIFT_REG · 296
- CHR · 105
- CLEAR_DIGOUT · 198
- CNT_CLEAR · 223
- CNT_CLEARENABLE · 225

- CNT_ENABLE · 227
- CNT_GETSTATUS · 229
- CNT_INPUTMODE · 232
- CNT_LATCH · 234
- CNT_MODE · 236
- CNT_READ · 238
- CNT_READFLATCH · 242
- CNT_READLATCH · 240
- CNT_RESETSTATUS · 244
- CNT_SE_DIFF · 246
- CNT_SET · 248
- color settings · 22
- command line
 - line length
 - standard · 39
 - with #INCLUDE · 138
 - upper case / lower case · 39
- command line call · 8
- Comment Block · 12
- comment, see remarks
- communication
 - between processes · 80
 - process in the ADwin system · 76
 - time-out · 76
 - with a development environment · 82
 - with the computer · 81
- comparison
 - < = > · 94
 - Strings · 172
- compiler
 - command line call · 8
 - instruction #DEFINE · 110
 - instruction #INCLUDE · 138
 - preprocessor statement · 91
 - set options · 16
- Compiler-Anweisungen
 - #IF ... THEN · 134
- conditional jump
 - IF ... THEN · 132
 - SELECTCASE · 160

- CONF_DIO · 200
- CONF_DIO_E · 250
- context menu
 - project window · 33
 - source code window · 12
- control characters in strings · 55
- control structures · 59
- Controlblock · 12
- conversion, start of · 216
- cosine: COS · 106
- counter
 - internal, clock cycle · 77
 - read · 158
- cut off decimal places · 58
- cycle time · 76

D

- DAC · 202
- data exchange
 - between processes · 80
 - with the computer · 81
 - with the development environment · 82
- data loss
 - FIFO · 52
 - from booting · 7
- data memory
 - 2-dim. arrays in ~ · 51
 - additional demand by
 - debug mode · 67
 - timing mode · 70
 - trace mode · 70
 - allocate · 49
 - overview, internal, external · 50
- data structures
 - FIFO · 52
 - global arrays · 46
 - global arrays, 2-dimensional · 50
 - global variables · 45
 - local variables and arrays · 48
 - overview · 43

- data types
 - overview · 44
 - string · 53
 - type conversion · 58
- data word
 - numbering of bits · 3
- DATA_n · 46
 - global arrays, 2-dimensional · 50
 - overview · 107
- Data-Index (global) too large / <1,
 - see run-time error
- debug
 - general · 66
 - debug mode · 67
 - enable timing mode · 25
 - enable trace mode · 28
 - menu · 25
 - timing mode · 67
 - timing window · 25
 - trace mode · 70
 - trace window · 29
 - TRACE_MODE_PAUSE · 184
 - TRACE_MODE_RESUME · 185
- DEC · 109
- decimal logarithm · 150
- decimal notation · 44
- decimal places, cut off · 58
- decimal separator · 44
- declaration, see dimensioning
- decrement · 109
- DEFINE, see #DEFINE
- definition of macros
 - position in the program · 42
- Delphi · 82
- design of an ADbasic program · 39
- development environment
 - bars and windows · 8
 - communication with C, Delphi, Matlab etc. · 82
 - directory settings · 24
 - short-cuts · 1
 - source directory · 7
 - start · 7
- device no.
 - definition · 82
 - set · 18
- DIAdem · 82
- DIGIN · 203
- DIGIN_WORD · 205
- DIGIN_WORD1_E (L16-DIO) · 251
- DIGIN_WORD2_E (L16 DIO1) · 252
- digital in-/outputs
 - clear one output · 198
 - configure · 200
 - read all inputs · 205
 - read one input · 203
 - set all outputs · 207
 - set one output · 212
- DIGOUT_RESET1_E (L16 DIO1) · 253
- DIGOUT_RESET2_E (L16 DIO1) · 254
- DIGOUT_SET1_E (L16 DIO1) · 255
- DIGOUT_SET2_E (L16 DIO1) · 256
- DIGOUT_WORD · 207
- DIGOUT_WORD1_E (L16 DIO1) · 257
- DIGOUT_WORD2_E (L16 DIO1) · 258
- DIM · 112
- dimensioning
 - instruction DIM · 112
 - memory area · 49
 - position in the program · 41
- directory
 - with standard installation · 7

- directory settings · 24
- Disable Trace · 12
- display
 - current information · 9
- division
 - by 2 · 163
 - simple · 89
- Division by zero, see run-time error
- DM, see memory
- DM_LOCAL (DIM ...) · 112
- DO ... UNTIL · 114
- DRAM_EXTERN (DIM ...) · 112
- DX, see memory

E

- ELSE (IF ... THEN) · 132
- EN_CAN_INTERRUPT (Gold CAN) · 280
- EN_INTERRUPT (L16 DIO1) · 261
- EN_RECEIVE (Gold CAN) · 281
- EN_RECEIVE (L16 DIO1) · 262
- EN_TRANSMIT (Gold CAN) · 282
- EN_TRANSMIT (L16 DIO1) · 263
- Enable Trace · 12
- enable trace mode · 28
- END · 115
- ENDFUNCTION · 127
- ENDIF (IF ... THEN) · 132
- ENDSELECT (SELECTCASE ...) · 160
- ENDSUB · 180
- equal to = · 94
- error
 - see also run-time error
 - data loss with FIFO · 52
 - forced by Cut&Paste · 15
 - process overwritten · 75
 - run-time · 30
 - time-out · 76
 - try lower optimization level · 20
- escape sequence · 55

- Ethernet · 81
- event
 - lost event signals
 - check · 27
 - lost signal
 - externally controlled
 - process · 80
 - several time-controlled
 - processes · 79
 - single time-controlled
 - process · 79
 - measure time difference · 64
- EVENT: · 41
- exclusive OR operation · 190
- EXIT · 116
- exponential function: EXP · 117
- exponential notation · 45
- expressions
 - evaluate · 56
 - separate evaluation · 59
- extensive initialization · 41
- external data memory (DX) · 50
- external memory (SDRAM) · 50

F

- F1-Help · 10
- FIFO
 - check number of elements · 53
 - data loss · 52
 - design of data structure · 52
 - dimensioning · 112
 - initialize · 119
 - overview · 118
 - query empty elements · 121
 - query full elements · 122
- FIFO_CLEAR · 119
- FIFO_EMPTY · 121
- FIFO_FULL · 122
- file name
 - binary file · 16
 - library · 16

FINISH: · 41
floating-point numbers
 decimal notation · 44
 exponential notation · 45
 value range · 44
FLOTOSTR · 123
font settings · 22
FOR ... NEXT · 125
FPAR_n · 45
FUNCTION · 127
function
 general features · 60
 library
 definition · 139
 general · 61
 macro · 127
 position in the program · 42

G

GET_CAN_REG (Gold CAN) · 283
GET_CAN_REG (L16 DIO1) · 264
GET_RS · 297
global arrays, see arrays, global
global variables, see variables, global
GLOBALDELAY
 system variable · 130
 time resolutions · 76
greater than >, >= · 94

H

halt, see stop process
hardware access
 read · 155
 write · 156
help
 context-sensitive · 8
 instruction (F1) · 10
hexadecimal notation · 45

I

IEEE floating-point format · 44
IF · 132
 see also #IF · 134
IMPORT · 136
INC · 137
INCLUDE · 138
include
 include a file: #INCLUDE · 138
 include a library: IMPORT · 136
 include-file, general · 61
include file
 directory settings · 24
increment · 137
INIT: · 41
INIT_CAN (Gold CAN) · 284
INIT_CAN (L16 DIO1) · 265
initialization
 boot · 7
installation, standard directory · 7
instruction
 measure processing time · 63
instruction separator (:) · 92
integer numbers
 binary notation · 45
 hexadecimal notation · 45
 type conversion · 58
 value range · 44
internal counter
 clock cycle · 77
internal data memory (DM) · 50
internal memory (SRAM) · 50
interrupt, see stop process

J

jump, conditional
 IF ... THEN · 132
 SELECTCASE · 160

L

- latency (timing window) · 26
- length (timing window) · 26
- less than <, <= · 94
- LIB_ENDFUNCTION · 139
- LIB_ENDSUB · 143
- LIB_FUNCTION · 139
- LIB_SUB · 143
- library
 - create
 - from ADbasic · 16
 - directory settings · 24
 - function · 139
 - general · 61
 - IMPORT · 136
 - position in the program · 42
 - subroutine · 143
- license agreement · 4
- line feed (control character) · 55
- line length, max.
 - standard · 39
 - with #INCLUDE · 138
- LN · 147
- LNGTOSTR · 148
- LOG · 150
- logarithm
 - decimal · 150
 - natural · 147
- logic functions
 - AND · 97
 - NOT · 152
 - OR · 153
 - SHIFT_LEFT · 162
 - SHIFT_RIGHT · 163
 - XOR · 190
- Long, see integer numbers
- LOWINIT: · 41

M

- macro
 - function · 127
 - general features · 60
 - position in the program · 42
- Mark Controlblock · 12
- Matlab · 82
- matrix, 2-dimensional · 50
- maximum line length
 - standard · 39
 - with #INCLUDE · 138
- measure processing time · 63
- measurement graph · 37
- memory
 - additional demand by
 - debug mode · 67
 - timing mode · 70
 - trace mode · 70
 - allocate · 49
 - areas (PM, DM, DX) · 50
 - see also data memory
 - string · 53
- menu
 - build · 15
 - choose · 9
 - debug · 25
 - edit · 15
 - file · 14
 - help · 32
 - options · 16
 - tools · 31
 - view · 15
 - window · 32
- multiplexer
 - set · 214
- multiplication
 - by 2 · 162
 - simple · 88

N

- names
 - local variables · 48
- natural logarithm · 147
- negative sign · 57
- NEXT (FOR ...) · 125
- NOP · 151
- NOT · 152
- not equal to <> · 94
- notation of numbers · 44
- notes, see remarks
- number of processes, check · 68
- number, see device no.
- numerical values
 - notation · 44

O

- operating system
 - directory settings · 24
 - load, see booting
- operators
 - evaluate · 56
 - negative sign · 57
 - priority · 57
 - XOR · 190
- optimal timing
 - one process · 69
 - several processes · 68
- optimize
 - constants instead of
 - variables · 64
 - general · 63
 - measure faster · 65
 - measure processing time · 63
 - register access · 64
 - run-time error · 67
 - timing · 67
 - use waiting times · 65

- optimize, see also debug
- option setting
 - editor · 21
 - general · 21
 - structured display · 22
- options setting
 - compiler · 16
 - directory · 24
 - language · 23
 - process · 18
- OR · 153
- OR operation · 153

P

- PAR_n · 45
- parameter window · 34
- parameters, see variables, global
- PEEK · 155
- PM, see memory
- PM_LOCAL (DIM ...) · 112
- POKE · 156
- power · 90
 - base e · 117
- Präprozessor-Anweisungen
 - #IF ... THEN · 134
- pre-processor instructions
 - #DEFINE · 110
 - #INCLUDE · 138
- preprocessor statement · 91
- priority
 - of processes, check · 68
 - operators · 57
 - process, see process, priority

- process
 - check number and priority · 68
 - communication · 80
 - communication process · 76
 - number · 74
 - operating modes for timing · 79
 - optimal timing, one process · 69
 - optimal timing, several processes · 68
 - priority
 - communication · 76
 - high · 75
 - low · 75
 - overview · 74
 - processing time · 77
 - query status · 157
 - setting options · 18
 - several · 78
 - standard processes 11, 12 · 75
 - start · 167
 - stop
 - see stop process · 115
 - time characteristic · 76
- process control
 - END · 115
 - EXIT · 116
 - PROZESSn_RUNNING · 157
 - START_PROCESS · 167
 - STOP_PROCESS · 168
- process cycle
 - call
 - by event · 73
 - time interval · 77
 - precise timing · 78
- process flow
 - track · 70
- process optimization, see optimize
- processdelay · 76
- program architecture
 - jump
 - IF ... THEN · 132
 - SELECTCASE · 160
 - library
 - function · 139
 - subroutine · 143
 - loop
 - DO ... UNTIL · 114
 - FOR ... NEXT · 125
 - modules
 - FUNCTION · 127
 - subroutine SUB · 180
 - remarks REM · 159
- program design · 39
- program improvement, see optimize
- program memory · 50
 - additional demand by
 - debug mode · 67
 - timing mode · 70
 - trace mode · 70
- program section
 - EVENT: · 41
 - FINISH: · 41
 - INIT: · 41
 - LOWINIT: · 41
 - overview · 41
- program structure
 - overview · 59
 - include-file · 61
 - library · 61
 - module (macro) · 60
- project
 - colour mark used variables · 34
 - general · 13
 - window · 33
- PROZESSn_RUNNING · 157

R

- READ_FIFO · 298
- READ_MSG (Gold CAN) · 285

READ_MSG (L16 DIO1) · 266
READ_TIMER · 158
READADC · 209
READADC12 · 210
register access · 64
REM · 159
remarks · 159
ring buffer · 52
root · 166
RS_INIT · 299
RS_RESET · 302
RS485_SEND · 303
run-time error
 see also debug mode
 display · 30
 find · 67

S

Save All Files of Project · 33
SDRAM, see memory
SELECTCASE · 160
separator : · 92
SET_CAN_BAUDRATE (Gold CAN) · 287
SET_CAN_BAUDRATE (L16 DIO1) · 268
SET_CAN_REG (Gold CAN) · 293
SET_CAN_REG (L16 DIO1) · 274
SET_DIGOUT · 212
SET_MUX · 214
SET_RS · 304
settling time see multiplexer · 214
SHIFT_LEFT · 162
SHIFT_RIGHT · 163
(bit) shifting
 left · 162
 right · 163
short-cuts · 1
sine: SIN · 164
SLEEP · 165
source code
 information · 9
 structured display · 10
 use in a project · 33
 working with · 10
SQRT · 166
SQRT from negative value, see run-time error
square root · 166
SRAM, see memory
SRAM_EXTERN (DIM ...) · 112
SSI_MODE · 307
SSI_READ · 309
SSI_SET_BITS · 311
SSI_SET_CLOCK · 312
SSI_START · 314
SSI_STATUS · 316
start of conversion · 216
START_CONV · 216
START_PROCESS · 167
starting
 ADbasic · 7
STEP (FOR ...) · 125
stop process
 itself
 in Event: · 115
 in LowInit, Init, Finish: · 116
 others · 168
STOP_PROCESS · 168
STRCOMP · 172
string
 assign values normally · 54
 assignment not being recommended · 56
 control character · 55
 definition of data type · 44
 escape sequence · 55
 variable structure · 53

- string instruction
 - addition · 86
 - ASCII value into char · 105
 - char into ASCII value · 102
 - comparison · 172
 - dimensioning · 170
 - float to string · 123
 - length of a string · 175
 - long to string · 148
 - partial string
 - left · 173
 - midst · 176
 - right · 178
 - string to float · 186
 - string to long · 188
 - syntax · 170
- STRLEFT · 173
- STRLEN · 175
- STRMID · 176
- STRIGHT · 178
- structure
 - Coloured display of source
 - code · 10
 - indent lines · 12
 - program sections · 59
- SUB · 180
- subroutine
 - general features · 60
 - library
 - definition · 143
 - general · 61
 - macro · 180
 - position in the program · 42
- subtraction · 87
- system variable
 - GLOBALDELAY · 130
 - PROZESSn_RUNNING · 157
- system variables
 - overview · 47

T

- tab (control character) · 55
- tabsize · 21
- tangent: TAN · 183
- TBin · 37
- TButton · 37
- TCP/IP
 - See Ethernet
- TDigit · 37
- terminate, see stop process
- Testpoint · 82
- TFifo · 37
- TGraph · 37
- THEN (IF ... THEN) · 132
- time
 - cycle time · 76
 - precise cycle timing · 78
 - time-out · 76
- time saving
 - constants instead of
 - variables · 64
 - measure faster · 65
 - register access · 64
 - use waiting times · 65
- timer, see counter
- timing
 - changed by
 - debug mode · 67
 - timing mode · 70
 - trace mode · 70
 - operating modes
 - externally controlled
 - process · 80
 - general · 79
 - several time-controlled
 - processes · 79
 - single time-controlled

- process · 79
- optimal, several processes · 68
- optimal, with one process · 69
- optimize · 67
- query information · 69
- timing mode
 - additional processor time · 70
 - enable · 25
 - use · 67
 - window · 25
- timing, see optimize
- TLed · 37
- TMeter · 37
- TO (FOR ...) · 125
- tool bar · 9
- TPar_FPar · 37
- TPoti · 37
- TProcess · 37
- trace mode
 - additional processor time and memory demand · 70
 - apply from within program · 72
 - enable · 28
 - TRACE_MODE_PAUSE · 184
 - TRACE_MODE_RESUME · 185
 - update information · 71
 - use · 70
 - window · 29
- TRACE_MODE_PAUSE · 184
- TRACE_MODE_RESUME · 185
- TRANSMIT (Gold CAN) · 294
- TRANSMIT (L16 DIO1) · 275
- transputer
 - settings · 12
- trigonometric functions
 - ARCCOS · 99
 - ARCSIN · 100
 - ARCTAN · 101
 - COS · 106
 - SIN · 164
 - TAN · 183

- type conversion
 - ASCII value into char · 105
 - automatical · 58
 - Float to Long (only data type) · 104
 - float to string · 123
 - Float toLong (only data type) · 103
 - long to string · 148
 - string to float · 186
 - string to long · 188

U

- Uncomment Block · 12
- Unmark Controlblock · 12
- UNTIL (DO ...) · 114
- upper / lower case letters · 10
- USB · 81
- use trace mode · 70
- user surface · 8
- utility programs, see *ADtools*

V

- VALF · 186
- VALI · 188
- value range · 44
- variables
 - colour mark used · 34
 - display · 34
 - global · 45
 - name · 42
 - initialization by booting · 7
 - initialize · 41
 - local · 48
 - allocate memory area · 49
 - name length · 48
 - overview · 42
 - PROZESSn_RUNNING · 157
 - switch hex/decimal display · 34
- Visual Basic · 82

W

wait

NOP · 151

SLEEP · 165

WAIT_EOC · 218

Window

source code information · 9

window

compiler options · 16

overview · 8

parameter · 34

process Options · 18

project · 33

workload

definition · 78

influence of number of
processes · 68

WRITE_FIFO · 305

X

XOR · 190

A-7 Instructions for ADwin-Gold

Symbols

< = > (comparison)

+ (Addition)

+ (String-Addition)

- (Subtraktion)

* (multiplication)

/ (Division)

^ (power)

= (assignment)

: Colon

#DEFINE

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

#INCLUDE

A

ABSF

ABSI

ADC

ADC12 (ADC14)

AND

ARCCOS

ARCSIN

ARCTAN

ASC

C

CAN_MSG (CAN)

CAST_FLOATTOLONG

CAST_LONGTOFLOAT

CHECK_SHIFT_REG (CAN)

CHR

CLEAR_DIGOUT

CNT_CLEAR (CO1)

CNT_ENABLE (CO1)

CNT_GETSTATUS (CO1)

CNT_INPUTMODE (CO1)

CNT_LATCH (CO1)

CNT_MODE (CO1)

CNT_READ (CO1)

CNT_READFLATCH (CO1)

CNT_READLATCH (CO1)

CNT_RESETSTATUS (CO1)

CNT_SET (CO1)

CNT_SE_DIFF (CO1)

CONF_DIO

COS

D

DAC

DATA_n

DEC

DIGIN

DIGIN_WORD

DIGOUT_WORD

DIM

DO ... UNTIL

E

END

EN_CAN_INTERRUPT (CAN)

EN_RECEIVE (CAN)

EN_TRANSMIT (CAN)

EXIT

EXP

F

FIFO

FIFO_CLEAR

FIFO_EMPTY

FIFO_FULL

FLOTOSTR

FOR ... TO ... {STEP ...} NEXT

FUNCTION ... ENDFUNCTION

G

GET_CAN_REG (CAN)
GET_RS (CAN)
GLOBALDELAY

I

IF ... THEN ... {ELSE ...} ENDIF
IMPORT
INC
INIT_CAN (CAN)

L

LIB_FUNCTION ...
LIB_ENDFUNCTION
LIB_SUB ... LIB_ENDSUB
LN
LNGTOSTR
LOG

N-P

NOP
NOT
OR
PEEK
POKE
PROZESSn_RUNNING

R

READADC
READADC12
READ_FIFO (CAN)
READ_MSG (CAN)
READ_TIMER
REM
RS485_SEND (CAN)
RS_INIT (CAN)
RS_RESET (CAN)

S

SELECTCASE
SET_CAN_BAUDRATE (CAN)
SET_CAN_REG (CAN)
SET_DIGOUT
SET_MUX
SET_RS (CAN)
SHIFT_LEFT
SHIFT_RIGHT
SIN
SLEEP
SQRT
SSI_MODE (CAN)
SSI_READ (CAN)
SSI_SET_BITS (CAN)
SSI_SET_CLOCK (CAN)
SSI_START (CAN)
SSI_STATUS (CAN)
START_CONV
START_PROCESS
STOP_PROCESS
STRCOMP
String " "
STRLEFT
STRLEN
STRMID
STRRIGHT
SUB ... ENDSUB

T

TAN
TRACE_MODE_PAUSE
TRACE_MODE_RESUME
TRANSMIT (CAN)

V-Z

VALF
VALI
WAIT_EOC
WRITE_FIFO (CAN)

XOR

A-8 Instructions for ADwin-light-16

Symbols

< = > (comparison)
+ (Addition)
+ (string addition)
- (subtraction)
* (multiplication)
/ (Division)
^ (power)
= (assignment)
: Colon
#DEFINE
#IF ... THEN ... {#ELSE ...} #ENDIF
#INCLUDE

A

ABSF
ABSI
ADC
AND
ARCCOS
ARCSIN
ARCTAN
ASC

C

CAN_MSG (DIO1 only)
CAST_FLOATTOLONG
CAST_LONGTOFLOAT
CHR
CLEAR_DIGOUT
CNT_CLEAR
CNT_CLEARENABLE (DIO1 only)
CNT_ENABLE
CNT_GETSTATUS (DIO1 only)
CNT_INPUTMODE (DIO1 only)
CNT_LATCH

CNT_MODE (DIO1 only)
CNT_READ
CNT_READFLATCH (DIO1 only)
CNT_READLATCH
CNT_SET (DIO1 only)
CONF_DIO_E (DIO1 only)
COS

D

DAC
DATA_n
DEC
DIGIN
DIGIN_WORD
DIGIN_WORD1_E (DIO1 only)
DIGIN_WORD2_E (DIO1 only)
DIGOUT_RESET1_E (DIO1 only)
DIGOUT_RESET2_E (DIO1 only)
DIGOUT_SET1_E (DIO1 only)
DIGOUT_SET2_E (DIO1 only)
DIGOUT_WORD
DIGOUT_WORD1_E (DIO1 only)
DIGOUT_WORD2_E (DIO1 only)
DIM
DO ... UNTIL

E

END
EN_INTERRUPT (DIO1 only)
EN_RECEIVE (DIO1 only)
EN_TRANSMIT (DIO1 only)
EXIT
EXP

F

FIFO

FIFO_CLEAR
FIFO_EMPTY
FIFO_FULL
FLOTOSTR
FOR ... TO ... {STEP ...} NEXT
FUNCTION ... ENDFUNCTION

G

GET_CAN_REG (DIO1 only)
GLOBALDELAY

I

IF ... THEN ... {ELSE ...} ENDIF
IMPORT
INC
INIT_CAN (DIO1 only)

L

LIB_FUNCTION ...
LIB_ENDFUNCTION
LIB_SUB ... LIB_ENDSUB
LN
LNGTOSTR
LOG

N-P

NOP
NOT
OR
PEEK
POKE
PROZESSn_RUNNING

R

READADC
READ_MSG (DIO1 only)

READ_TIMER
REM

S

SELECTCASE
SET_CAN_BAUDRATE (DIO1 only)
SET_CAN_REG (DIO1 only)
SET_DIGOUT
SET_MUX
SHIFT_LEFT
SHIFT_RIGHT
SIN
SLEEP
SQRT
START_CONV
START_PROCESS
STOP_PROCESS
STRCOMP
String " "
STRLEFT
STRLEN
STRMID
STRRIGHT
SUB ... ENDSUB

T

TAN
TRACE_MODE_PAUSE
TRACE_MODE_RESUME
TRANSMIT (DIO1) (DIO1 only)

V-Z

VALF
VALI
WAIT_EOC
XOR

A-9 Instructions for ADwin-Pro

The following overview contains those instructions only , which are processed in the Pro-CPU modules directly.

You find any other instructions for Pro modules in a separate manual "Pro-Software" (for lack of space).

Symbols

< = > (comparison)
+ (Addition)
+ (String-Addition)
- (Subtraktion)
* (multiplication)
/ (Division)
^ (power)
= (assignment)
: Colon
#DEFINE
#IF ... THEN ... {#ELSE ...} #ENDIF
#INCLUDE

A

ABSF
ABSI
AND
ARCCOS
ARCSIN
ARCTAN
ASC

C

CAST_FLOATTOLONG
CAST_LONGTOFLOAT
CHR
COS

D

DATA_n
DEC
DIM
DO ... UNTIL

E

END
EXIT
EXP

F

FIFO
FIFO_CLEAR
FIFO_EMPTY
FIFO_FULL
FLOTOSTR
FOR ... TO ... {STEP ...} NEXT
FUNCTION ... ENDFUNCTION

G

GLOBALDELAY

I

IF ... THEN ... {ELSE ...} ENDIF
IMPORT
INC

L

LIB_FUNCTION ...
LIB_ENDFUNCTION
LIB_SUB ... LIB_ENDSUB
LN
LNGTOSTR
LOG

N-P

NOP
NOT
OR
PEEK
POKE
PROZESSn_RUNNING

R

READ_TIMER
REM

S

SELECTCASE
SHIFT_LEFT

SHIFT_RIGHT
SIN
SLEEP
SQRT
START_PROCESS
STOP_PROCESS
STRCOMP
String " "
STRLEFT
STRLEN
STRMID
STRRIGHT
SUB ... ENDSUB

T

TAN
TRACE_MODE_PAUSE
TRACE_MODE_RESUME

V-Z

VALF
VALI
XOR

Symbols

< = > (comparison)	94
+ (addition)	85
+ (string addition)	86
- (subtraction)	87
* (multiplication)	88
/ (division)	89
^ (power)	90
= (assignment)	93
: colon	92
" " (String)	170
#DEFINE	110
#IF ... THEN ... {#ELSE ...} #ENDIF	134
#INCLUDE	138
#..., preprocessor statement	91

A-B

ABSF	95
ABSI	96
ADC	192
ADC12 (ADC14)	195
AND	97
ARCCOS	99
ARCSIN	100
ARCTAN	101
ASC	102

C

CAN_MSG	
Gold CAN	278
L16 DIO1	259
CAST_FLOATTOLONG	103
CAST_LONGTOFLOAT	104
CHECK_SHIFT_REG	296
CHR	105
CLEAR_DIGOUT	198
CNT_CLEAR	223
CNT_CLEARENABLE	225
CNT_ENABLE	227
CNT_GETSTATUS	229
CNT_INPUTMODE	232
CNT_LATCH	234
CNT_MODE	236
CNT_READ	238
CNT_READFLATCH	242
CNT_READLATCH	240

CNT_RESETSTATUS	244
CNT_SET	248
CNT_SE_DIFF	246
CONF_DIO	200
CONF_DIO_E	250
COS	106

D

DAC	202
DATA_n	107
DEC	109
DIGIN	203
DIGIN_WORD	205
DIGIN_WORD1_E	251
DIGIN_WORD2_E	252
DIGOUT_RESET1_E	253
DIGOUT_RESET2_E	254
DIGOUT_SET1_E	255
DIGOUT_SET2_E	256
DIGOUT_WORD	207
DIGOUT_WORD1_E	257
DIGOUT_WORD2_E	258
DIM	112
DO ... UNTIL	114

E-F

END	115
EN_CAN_INTERRUPT	
Gold CAN	280
EN_INTERRUPT	
L16 DIO1	261
EN_RECEIVE	
Gold CAN	281
L16 DIO1	262
EN_TRANSMIT	
Gold CAN	282
L16 DIO1	263
EXIT	116
EXP	117
FIFO	118
FIFO_CLEAR	119
FIFO_EMPTY	121
FIFO_FULL	122
FLOTOSTR	123
FOR ... TO ... {STEP ...}	
NEXT	125
FUNCTION ... END- FUNCTION	127

G-J

GET_CAN_REG	
Gold CAN	283
L16 DIO1	264
GET_RS	297
GLOBALDELAY	130
IF ... THEN ... {ELSE ...}	
ENDIF	132
IMPORT	136
INC	137
INIT_CAN	
Gold CAN	284
L16 DIO1	265

K-L

LIB_FUNCTION ... LIB_ ENDFUNCTION	139
LIB_SUB ... LIB_END- SUB	143
LN	147
LNGTOSTR	148
LOG	150

M-R

NOP	151
NOT	152
OR	153
PEEK	155
POKE	156
PROCESSn_RUNNING	157
READADC	209
READADC12	210
READ_FIFO	298
READ_MSG	
Gold CAN	285
L16 DIO1	266
READ_TIMER	158
REM	159
RS485_SEND	303
RS_INIT	299
RS_RESET	302

S

SELECTCASE	160
SET_CAN_BAUDRATE	
Gold CAN	287
L16 DIO1	268
SET_CAN_REG	
Gold CAN	293
L16 DIO1	274

SET_DIGOUT	212	START_CONV	216	TRACE_MODE_RE-	
SET_MUX	214	START_PROCESS	167	SUME	185
SET_RS	304	STOP_PROCESS	168	TRANSMIT	
SHIFT_LEFT	162	" " (String)	170	Gold CAN	294
SHIFT_RIGHT	163	STRCOMP	172	L16 DIO1	275
SIN	164	STRLEFT	173	VALF	186
SLEEP	165	STRLEN	175	VALI	188
SQRT	166	STRMID	176	WAIT_EOC	218
SSI_MODE	307	STRRIGHT	178	WRITE_FIFO	305
SSI_READ	309	SUB ... ENDSUB	180	XOR	190
SSI_SET_BITS	311	T-Z			
SSI_SET_CLOCK	312	TAN	183		
SSI_START	314	TRACE_MODE_PAUSE			
SSI_STATUS	316	184			