

# ***TiCoBasic***

**Real-Time Development Tool for  
*TiCo* Processors**

***TiCoBasic* Version 1.0**

**Feb. 2017**

**License Key:**

*ADwin* – the fastest real-time systems under Windows

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

Hotline:	+49 6251 96320
Fax:	+49 6251 568 19
E-Mail:	<a href="mailto:info@ADwin.de">info@ADwin.de</a>
Internet	<a href="http://www.ADwin.de">www.ADwin.de</a>



Jäger Com-  
putergesteuerte  
Messtechnik GmbH  
Rheinstraße 2-4  
D-64653 Lorsch  
Germany

## Table of contents

Table of contents . . . . .	III
Conventions . . . . .	2
1 Introduction . . . . .	3
2 TiCoBasic for ADbasic users . . . . .	5
3 Using the Development Environment . . . . .	9
3.1 Basic Steps . . . . .	9
3.1.1 Starting the Development Environment . . . . .	9
3.1.2 Check or change ADbasic licenses . . . . .	10
3.1.3 Initializing Communication . . . . .	12
3.1.4 Basic Elements of the Development Environment . . . . .	12
3.2 Creating source code . . . . .	17
3.2.1 Calling online help . . . . .	17
3.2.2 Context menu in source code window . . . . .	19
3.2.3 Editor bar . . . . .	22
3.3 Compiling source code . . . . .	23
3.4 Formatting source code . . . . .	24
3.4.1 Syntax highlighting . . . . .	24
3.4.2 Smart formatting . . . . .	24
3.4.3 Indenting text lines . . . . .	25
3.4.4 Positioning comments . . . . .	25
3.4.5 Changing lines into comment . . . . .	26
3.4.6 Documenting self-defined instructions and variables . . . . .	26
3.4.7 Defining a foldable text range . . . . .	28
3.4.8 Folding text ranges . . . . .	30
3.5 Searching and replacing . . . . .	31
3.5.1 Finding text quickly . . . . .	31
3.5.2 Finding and replacing text . . . . .	32
Examples – Finding Text . . . . .	35
Examples – Replacing Text . . . . .	36
3.5.3 Regular expression . . . . .	37

3.5.4 Marking control blocks .....	40
3.5.5 Using bookmarks .....	40
3.5.6 Jumping to a program line .....	41
3.5.7 Jumping to declaration of instruction or variable .....	41
3.5.8 Switching between jump targets .....	41
3.6 Writing programs with ease .....	42
3.6.1 Autocomplete for instruction or variable .....	43
3.6.2 Inserting code snippets .....	44
3.6.3 Displaying instruction parameters .....	45
3.6.4 Displaying declaration of instruction or variable .....	46
3.6.5 Displaying declarations of a file .....	46
3.6.6 Displaying used global variables and arrays .....	46
3.6.7 Adding file and folder shortcuts .....	48
3.7 Transferring a TiCo binary file to TiCo processor .....	48
3.7.1 Transferring a TiCo binary file .....	48
3.7.2 Programming the TiCo bootloader .....	49
3.8 Managing Projects .....	50
3.9 Menus .....	53
3.9.1 File Menu .....	54
3.9.2 Edit Menu .....	55
3.9.3 View Menu .....	55
3.9.4 Build Menu .....	56
3.9.5 MakeLibFileOptions Menu .....	57
Compiler Options dialog box .....	57
Process Options dialog box .....	59
Settings dialog box .....	64
Project Settings dialog box .....	68
3.9.6 Debug Menu .....	70
Debug mode Option .....	71
3.9.7 Tools Menu .....	72
3.9.8 Window Menu .....	73
3.9.9 Help Menu .....	73
3.10 Windows .....	75
3.10.1 Toolbox .....	75
3.10.2 Project Window .....	75
3.10.3 Parameter Window .....	77
3.10.4 Process Window .....	79

3.10.5 Register window	81
3.10.6 Status Bar	82
3.11 Info range	83
3.11.1 Info window	83
3.11.2 To-Do List	85
3.11.3 Global Variables	86
3.11.4 Declarations Window	87
3.12 ADtools	89
 4 Programming Processes	 91
4.1 Program Design	91
4.1.1 The Program Sections	93
4.1.2 User defined instructions and variables	93
4.2 Variables and Arrays	95
4.2.1 Overview	95
4.2.2 Data Structures	95
4.2.3 Data Types	96
4.2.4 Entering Numerical Values	97
4.2.5 Global Variables (Parameters)	97
4.2.6 Global Arrays	98
4.2.7 System Variables	99
4.2.8 Local Variables and Arrays	100
4.3 Variables and Arrays – Details	101
4.3.1 Variables and Arrays in the Data Memory	101
4.3.2 Memory Areas	101
4.3.3 Data structure Ringbuffer	103
4.4 Expressions	110
4.4.1 Evaluation of Operators	110
4.5 Selection structures, Loops and Modules	111
4.5.1 Subroutine and Function Macros	112
4.5.2 Include-Files	112
4.5.3 Libraries	113
 5 Optimizing Processes	 115
5.1 Measuring the Processing Time	115
5.2 Useful Information	116
5.2.1 Accessing Hardware Addresses	116

5.2.2 Constants instead of Variables	116
5.2.3 Faster Measurement Function	117
5.2.4 Setting Waiting Times Exactly	117
5.2.5 Using Waiting Times	117
5.2.6 Optimization of memory access	118
5.3 Debugging	119
5.3.1 Finding Run-time Errors (Debug Mode)	119
6 Processes in the ADwin System	121
6.1 Process Management	122
6.1.1 Timer controlled process	122
6.1.2 Externally controlled process	123
6.1.3 Process without trigger (None)	124
6.2 Time Characteristics of Processes	124
6.2.1 Processdelay	124
6.2.2 Workload of the <i>TiCo</i> processor	125
6.2.3 Different Operating Modes in the Operating System	126
6.3 Communication	127
6.3.1 Data Exchange between Processes	127
6.3.2 Communication between PC and <i>TiCo</i> processor	128
6.3.3 Communication between ADwin CPU and <i>TiCo</i> Processor	129
6.3.4 The Device Number	129
7 Instruction Reference	131
7.1 Instruction Syntax	131
7.2 Basic Instructions <i>TiCoBasic</i>	131
7.3 Mathematics Instructions	217
7.4 Gold II: <i>TiCo</i> processor	219
7.5 Pro II: <i>TiCo</i> Processor	263
8 How to Solve Problems?	313
Appendix	A-1
A.1 Short-Cuts in ADbasic	A-1
A.2 ASCII-Character Set	A-4
A.3 License Agreement	A-5

A.4 Command Line Calling . . . . .	A-9
A.5 Instructions for <i>ADwin-Gold II</i> . . . . .	A-16
A.6 Instructions for <i>ADwin-Pro</i> systems. . . . .	A-19
A.7 Index . . . . .	A-21
A.8 Instructions in this manual . . . . .	A-37





**Dear Reader,**

*TiCoBasic* is the programming tool for *TiCo* processors in your *ADwin* system that allows you to create special measurement, open-loop, or closed-loop control application. The purpose of this manual is to: introduce you to the basics of programming real-time processes; and act as a reference manual. the use of the online help with F1 is recommended.

The development environment as well as the programming language *TiCo-Basic* is closely related to *ADbasic*; the change is easy. Please note [chapter 2](#) where differences between *TiCoBasic* and *ADbasic* are listed.

First-time users of a *TiCo* processor and *TiCoBasic* are recommended reading chapters [1](#) and [4](#), in order to get easily into the subject. This manual assumes that the user has some programming experience with Basic or any other language.

Chapter [3](#) describes the development environment and is recommended for all users.

If you have any 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.

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

## Conventions

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



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



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



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



The light bulb icon denotes examples showing practicable solutions.

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

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

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

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

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

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

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

- Hexadecimal notation: **11h**
- Binary notation: **10001b**

## 1 Introduction

The *ADwin* real-time system is responsible for all time-critical tasks in fast dynamic test stands and industrial production facilities. For this task, the *TiCo* processor—being integrated into the real-time system—is set where precise, fine-tuning timing is required.

Alike programming the *ADwin* real-time system with *ADbasic*, you use the language and development environment *TiCoBasic* to program the *TiCo* processor.

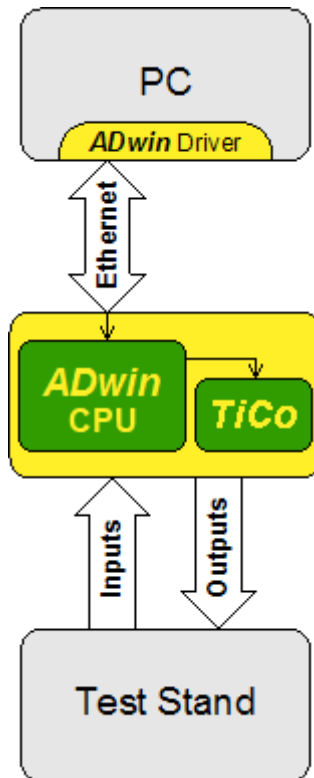
To hit the target of an immediate and efficient start of programming, we first of all would like to shortly explain the concept of the *ADwin* system with the integrated *TiCo* processor:

Each *ADwin* system has a central processing unit, the *ADwin* CPU, which executes all time-critical tasks in real time. In addition, an *ADwin* system (*Pro II* and *Gold II*) can contain one or more *TiCo* processors. Analog and digital inputs and outputs as well as add-ons like counters and bus interfaces are the connection to the test stand.

Inside the *ADwin* system the *TiCo* processor (*Timing Controller*) runs independently as freely programmable co-processor, which has access to all inputs and outputs and fulfills special tasks as filtering, data conversion, communication (SPI), signal generation, control etc. According to the target the *TiCo* processor can support the *ADwin* CPU's task e.g. by preprocessing; or it undertakes a complete task on its own.

The *TiCo* processor is optimized for fast reaction and für schnelle Reaktionszeiten and exact timing in a nanosecond time grid.

Communication between PC and *ADwin* CPU is completely set up via Ethernet. Instead, the *TiCo* processor is implemented as softcore in the FPGA, where to the PC has no direct access. The data exchange between PC and *TiCo* processor will therefore always need the *ADwin* CPU to forward data.



The *TiCo* processor is programmed with the real-time development environment *TiCoBasic*, which enables easy construction of time-critical real-time processes. *TiCoBasic* is an integrated development environment under Windows for the programming language of the same name. The familiar command syntax—very similar to *ADbasic*—allows accessing the inputs and outputs, controlling real-time processes, and preparing the data exchange with the *ADwin* CPU. The [chapter 4](#) explains the design of *TiCoBasic* programs, [chapter 6](#) additionally the action of processes in the operating system.



With only a few *TiCoBasic* program lines, you 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

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

Executable binary code, generated from the source code using the integrated compiler, is transferred to the *TiCo* processor via the *ADwin* CPU and tested. *TiCoBasic* is also a tool, which aids in process monitoring, error detection, and program optimization (see [chapter 3](#)).



*TiCoBasic* environment is no longer needed once the real-time processes are running properly.

From the *ADwin* CPU, you may start, monitor, control, and stop the processes on the *TiCo* processor, and transfer process data from the *TiCo* processor.

Although the *TiCo* processor operates independently, global variables and arrays are accessed through the *ADwin* CPU, without delaying time-critical processes. Using global variables and arrays, all processes and the *ADwin* CPU quickly exchange process data.

Find more information in [chapter 6.3.3 "Communication between ADwin CPU and TiCo Processor"](#) on [page 129](#).

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

## 2 TiCoBasic for ADbasic users


The use of *TiCoBasic* is just like *ADbasic 5*: Most functions and work flows are be present in familiar way, menu entries and buttons remain at the usual spots. Thus, you will have a good start into *TiCoBasic*.


On the other hand, there are some important differences, which you learn to know here. The *TiCo* processor is not simply a second *ADwin* CPU in small format, but is optimized for fast reaction and exact timing in a nanosecond time grid.


### Communication with the TiCo processor

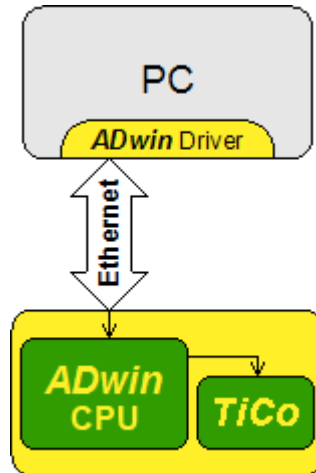
The *TiCo* processor is implemented as soft-core in the FPGA, where to the PC has no direct connection. The data exchange between PC and *TiCo* processor will therefore always need the *ADwin* CPU to forward data.

Please note:

- The main settings under Options > Compiler select the *TiCo* processor and the *ADwin* CPU as well as the *ADwin* system and the Device No.
- Initialize instead of Boot: The button  initializes the selected *ADwin* CPU for the data exchange with the *TiCo* processor.

The *TiCo* processor itself is initialized only, when a process is compiled . In this case, all global variables are set to 0 and the compiled process starts automatically.

- The *TiCo* processor will continue running, even if the *ADwin* CPU is currently not available, e.g. after being booted. In order to reinstall data exchange, you just have to initialize the *ADwin* CPU again with the button .



### TiCo processes as usual

Generally, *TiCo* processes are programmed as *ADbasic* processes.

- The following number and types of processes are available:

Process type	High priority	Low priority
timer-controlled	1	1
externally controlled	1	–
without trigger	1	–

- Please note: Only a *single* process with high priority should be running. ADwin CPU There are 3 program sections: **Init:**, **Event:** and **Finish:**, all running with the same priority. There is no section **LowInit:**.
- The processor clock cycle is 20ns.

### Reduced set of instructions

- The TiCo processor uses data type **Long** only; the data type **Float** is not available.

The basic set of instructions is listed below. In addition, there are instructions to access inputs, outputs and interfaces of the ADwin hardware, which are described in a separate document.

Standard	<b>Init:</b> , <b>Event:</b> , <b>Finish:</b> , <i>REM</i> , : (colon)
Variables	<i>Dim</i> , <i>Par_1...Par_80</i> , <i>Data_1...Data_16</i> <i>Ringbuffer_For_Read</i> <i>Ringbuffer_For_Write</i> local arrays
Mathematics, operators	+ - * / <b>Inc</b> , <b>Dec</b> , <b>Shift_Left</b> , <b>Shift_Right</b> <b>AbsI</b> , <b>Not</b> , <b>Or</b> , <b>XOr</b> , <b>And</b>
Comparison	= < > <b>Or</b> , <b>And</b>
Structure	<i>For...Next</i> , <i>Do...Until</i> <i>If...Then</i> , <i>SelectCase</i> <i>Function</i> , <i>Sub</i> , <i>Lib_Function</i> , <i>Lib_Sub</i>
Processes, system variables	<b>End</b> <b>Processdelay</b> , <b>Process_Running</b> , <b>NwTime</b>
Pre-Compiler	<b>#If...#Endif</b> , <b>#Define</b> , <b>#Include</b>
Timing	<b>NOP</b> , <b>NOPS</b> , <b>Sleep</b> , <b>Read_Timer</b>
Memory access	<b>Peek</b> , <b>Poke</b>

Float calculations like integer powers, trigonometric functions and division with rest are not available.

- Up to 16 global arrays `Data_1...Data_16` can be declared (in *ADbasic* up to `Data_200`). The data structure FIFO is not available.
- There are the new data structures `Ringbuffer_For_Read` and `Ringbuffer_For_Write`.

Ring buffers are used for fast data transfer; the possible applications are mutually exclusive:

- The TiCo process exchanges data with external DRAM in both directions, reading and writing via the same ringbuffer array.
- ADbasic processes (on the ADwin CPU) and TiCoBasic processes exchange data via a ringbuffer. One ringbuffer is required for each direction of data exchange.
- Several processes on the TiCo processor exchange data with each other via ringbuffer. Two ringbuffers are required, one for reading and one for writing.



Using the data structure `Ringbuffer` is not an easy task. Wrongly implemented, there may be errors, which can hardly be tracked. The use of the data structure `Ringbuffer` is therefore reserved to experienced users of *ADbasic* and *TiCoBasic*.

- The instruction `Exit` is replaced by `End`.
- The instructions `In` and `Out` replace `Peek` and `Poke`.
- You access *ADwin* hardware with similar instructions as in *ADbasic*. Please note: Either the *ADwin* CPU (via *ADbasic* instruction) or the *TiCo* processor (via *TiCoBasic* instruction) may access the same *ADwin* hardware (or interface), but not both at the same time.

### ADbasic instructions for control of TiCo

The *ADwin* CPU can directly access data of the *TiCo* processor. There are some *ADbasic* instructions at hand for data exchange and process control, see [chapter 7.4](#) and [chapter 7.5](#).

### **Deviations in the development environment**

- The development environment *TiCoBasic* provides a basic debug but noth the timing mode, which is available in *ADbasic*.
- If several files belong to a project, they will be always compiled together. Compiling a single file is only possible, if it is not part of a project.
- In future, the use of assembler instructions will be possible in *TiCo-Basic*. Therefore, the environment contains a [Register window](#), which shows register values of the *TiCo* processor.



### 3 Using the Development Environment

Programs for *TiCo* processors are developed quickly and easily with the *TiCoBasic* development environment. The *TiCoBasic* compiler works with an enlarged BASIC syntax like *ADbasic*, but with a smaller set of instructions.

After having completed an *TiCoBasic* program, it is compiled to a binary file and transferred—indirectly via *ADwin* CPU—to the *TiCo* processor. In bootloader mode, the program will be started and executed automatically and independently from the development environment.

#### 3.1 Basic Steps

##### 3.1.1 Starting the Development Environment

To start the *TiCoBasic* development environment, do as follows:

1. Start the development environment by selecting **Programs**► **ADwin**► **TiCoBasic** from the Windows start menu.

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

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

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

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

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

Find more information about the *TiCoBasic* license in [chapter 3.1.2 on page 10](#).

3. Enter the settings of the *TiCo* processor to be programmed next in the menu `Options\Compiler`:
  - the type of *ADwin* system and *ADwin* CPU
  - the device no.
  - for a Pro II system: the module with the *TiCo* processor.

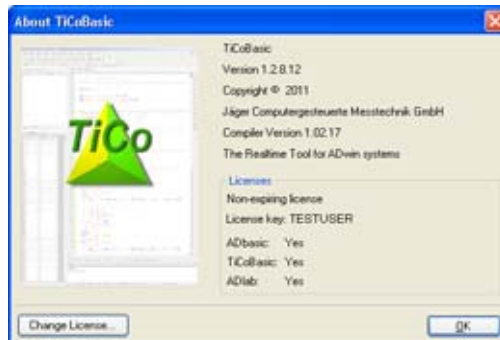
The development environment saves the settings so that upon a new start of *TiCoBasic* they will not need to be entered again, unless a different *TiCo* processor is used.

### 3.1.2 Check or change TiCoBasic licenses

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

1. Select the menu entry `Help►About`.

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



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

The dialog window **License key** opens.

3. Enter your license key.

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



In *TiCoBasic*, the following licenses are available:

- No license (demo mode)

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

- Evaluation license (expiring by date)

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

- Non-expiring license of the Licensee

The following licenses can be enabled:

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

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

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

### 3.1.3 Initializing Communication

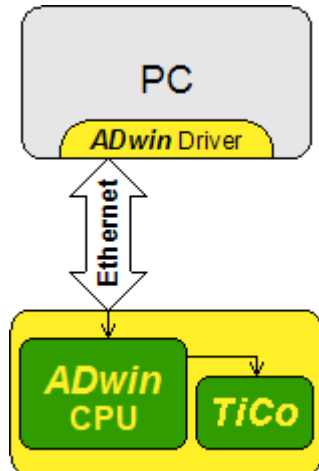
The *TiCo* processor is implemented as softcore in the FPGA, to which the PC has no direct connection. The data exchange between PC and *TiCo* processor will therefore always use the *ADwin* CPU as interstation.

The selected *ADwin* CPU is initialized for data exchange with the *TiCo* processor by clicking the button **I** (= initialize). This will setup a process in the *ADwin* CPU, which organizes the data flow between PC and *TiCo* processor.

The *TiCo* processor itself is only initialized after compiling a process, with the button **C**. Thus, the contents of the program and data memories will be lost, all global *TiCo* parameters set to the value 0 and the compiled process starts automatically. If there is no *TiCo* processor available an error message is launched.

The *TiCo* processor will continue running, even while the *ADwin* CPU is not available, e.g. because of booting. In order to restore the data exchange, it will suffice to initialize the *ADwin* CPU with the **I** (Initialize) button.

You can also stop and reset the *TiCo* processor with the button **R** (Reset). Doing so, you will loose all values in the global *TiCo* variables and the process is deleted.



### 3.1.4 Basic Elements of the Development Environment

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


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

Fig. 1 –

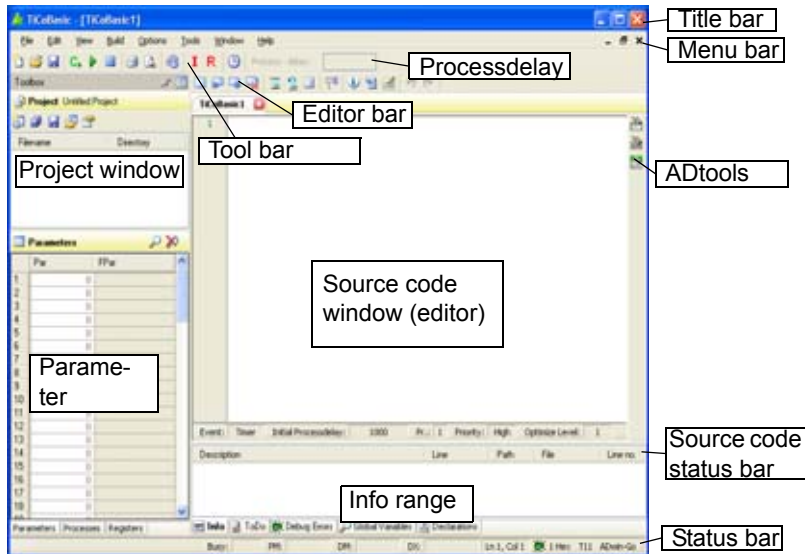


Fig. 2 – Elements of the *TiCoBasic* development environment

The functions of the development environment are called using:

- The tool bar and the editor bar (see [fig. 3](#)).
- The project bar in the [Project Window](#)
- The context menus of the windows (right mouse button).
- The menu bar.
- The [Short-Cuts in ADbasic](#) (see annex).
- The [ADtools](#) bar.

You can add own shortcuts here (see [chapter 3.6.7 on page 48](#)).

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

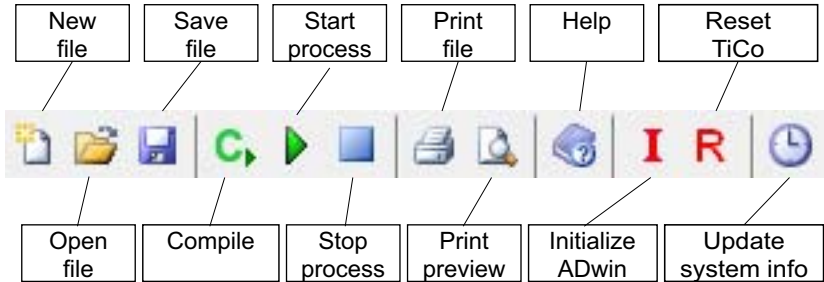


Fig. 3 – The tool bar


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

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

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

A right-click on the bar opens the [Process Options dialog box](#).

- The global parameters used in the source code project are highlighted in the [Parameter Window](#) (see [chapter 3.10.3, page 77](#))

by clicking `ScanGlobal Variables` ; see [Displaying used global variables and arrays](#) on [page 46](#).

- The info range at the bottom displays information in several windows:
  - [Info window](#): The compiler's error messages (highlighted red) and warnings (see [chapter 3.11.1 on page 83](#)).
  - [To-Do List](#): A simple To-Do list from comment lines (see [chapter 3.11.2 on page 85](#)).
  - Search results from a search in all files of a project (see [chapter 3.5.2 on page 32](#)).
  - Debug information if the debug mode is enabled (see [Debug mode Option, page 71](#)).
  - [Global Variables](#): A list of used global variables and arrays (see [chapter 3.11.3 on page 86](#)).
  - [Declarations Window](#): A list of all displays all declarations, related to the source code file (see [chapter 3.11.4 on page 87](#)).

Please note: Editing in the source code window is supported by several tools (see [Creating source code](#) on [page 17](#)).

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

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

- Processdelay (process cycle time) of the process, which has the number as the currently edited source code. Displayed at the right side of the toolbar.
- The values of the global variables in the [Parameter Window](#); a change to one of these values will immediately be transferred to the *ADwin* system.
- The status of running processes in the [Process Window](#) ([page 79](#)).
- The register values of the *TiCo* processor in the [Register window](#) ([page 81](#)).
- Memory usage information in the [Status Bar](#) (see [chapter 3.10.6 on page 82](#)).



### 3.2 Creating source code

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

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

If you use several files for your task, we recommend managing the files in a project file (see [page 50: Managing Projects](#)).

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

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

The source code editor provides several useful tools. Call the tools via [Context menu in source code window](#) ([page 19](#)) or via [Editor bar](#) ([page 22](#)):

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

Find more editor functions here:

- [Formatting source code](#), [page 24](#)
- [Searching and replacing](#), [page 31](#)
- [Writing programs with ease](#), [page 42](#)

#### 3.2.1 Calling online help

The [Help Menu](#) ([page 73](#)) enables to call selected help pages, e.g. table of contents or sorted instruction lists.

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


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

- The text is not an instruction but a user-defined declaration: Variable / array, symbolic name, macro (Sub, Function). For a user define, a help page cannot be provided.

- The instruction is misspelled, e.g. `Digin_Wrod` instead of `Digin_Word`. After being corrected, the instruction will be highlighted correctly.
- The (user-defined) include or library file is missing where the instruction is defined. Please insert the appropriate line at the start of the source code.

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

You can display several help pages side by side:













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

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

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

### 3.2.2 Context menu in source code window

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

	Cut	Ctrl+X
	Copy	Ctrl+C
	Paste	Ctrl+V
<hr/>		
	Comment Block	Ctrl+B
	Uncomment Block	Ctrl+Shift+B
	Create Region from Block	
<hr/>		
	Indent	Ctrl+I
	Outdent	Ctrl+Shift+I
	Mark Controlblock	
	Unmark Controlblock	
<hr/>		
	Add to Project	
<hr/>		
	Declaration Info	F2
	Jump to Declaration	Ctrl+F2
	Codesnippets	Ctrl+K X

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

- Cut: Cut selection and copy into the clipboard.
- Copy: Copy selection into the clipboard.
- Paste: Delete selection and insert text from the clipboard.
- Comment Block, Uncomment Block: [Changing lines into comment, page 26](#).
- Create Region from Block: [Defining a foldable text range, page 28](#).
- Indent, Outdent: [Indenting text lines, page 25](#).
- Mark Control block, Unmark Control block: [Marking control blocks, page 40](#).
- Declaration Info: [Displaying declaration of instruction or variable, page 46](#).
- Jump to Declaration: [Jumping to declaration of instruction or variable, page 41](#).

These functions are available without marking:

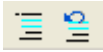
- Add to Project: Add a file to the project.
- Declaration Info: [Displaying declarations of a file, page 46](#).
- Code snippets: [Inserting code snippets, page 44](#).

### 3.2.3 Editor bar

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



[Using bookmarks, page 40.](#)



[Changing lines into comment, page 26.](#)



[Defining a foldable text range, page 28.](#)



[Folding text ranges, page 30.](#)



[Displaying declaration of instruction or variable, page 46.](#)



[Jumping to declaration of instruction or variable, page 41.](#)



[Inserting code snippets, page 44.](#)



[Undo or redo the previous editing action.](#)




[Switching between jump targets, page 41.](#)

### 3.3 Compiling source code

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

You can compile (and build) as follows.

- Test a source code while developing:  
Compile and run project source code files (or a single source code file); button  in the tool bar.
- Save tested source code as binary file:  
Compile project files (or single source code) and save as binary file: Build► Make Bin File.
- Run command lines before and after compiling.  
Command lines are defined in the Project Settings dialog box, [Tabs Prebuild / Postbuild](#) ([page 70](#)).

You can use created binary files for different purposes:

- Transfer and run a binary file from the environment *TiCoBasic* to the *TiCo* processor, see [Transferring a TiCo binary file](#).
- Transfer, run, and control a binary file from an external program—always combined with an *ADbasic* process—to the *TiCo* processor.

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

- Store a binary file permanently in the bootloader of a *TiCo* processor and run it automatically after power-up; see [chapter 3.7.2 "Programming the TiCo bootloader"](#), [page 49](#).

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

### 3.4 Formatting source code

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

- [Syntax highlighting, page 24](#)
- [Smart formatting, page 24](#)
- [Indenting text lines, page 25](#)
- [Positioning comments, page 25](#)
- [Changing lines into comment, page 26](#)
- [Documenting self-defined instructions and variables, page 26](#)
- [Defining a foldable text range, page 28](#)
- [Folding text ranges, page 30](#)

Find more editor functions in the sections:

- [Creating source code, page 17](#)
- [Searching and replacing, page 31](#)
- [Writing programs with ease, page 42](#)

#### 3.4.1 Syntax highlighting

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

The editor divides the character strings you have entered, into several groups of syntax elements being displayed differently. The color design may be changed under `Options ▶ Settings, Editor – Syntax Colors` (see [page 65](#)); the window also shows an overview of syntax groups.

Syntax highlighting requires an active option `Parse Declarations` under `Editor – General` (see [page 64](#)).

#### 3.4.2 Smart formatting

Once a command line is written, the editor will automatically correct the number of spaces, thus giving the line a clear structure. This way



e.g. operators like "=" or keywords like "If" will have a space to left and right.

If you like to format manually you have to switch off smart format under [Editor – General](#), `Smart format` (see [page 64](#)).

### 3.4.3 Indenting text lines

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

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

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

### 3.4.4 Positioning comments

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

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

You can enable or disable automatic positioning under [Editor – General](#), `Align comments` as well as set the comment start position (see [page 64](#)).

#### **3.4.5 Changing lines into comment**

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

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

#### **3.4.6 Documenting self-defined instructions and variables**

You can document self-defined instructions (`Sub`, `Function`), variables, arrays and symbolic names (`#Define`), so that the description text is shown in tooltips and with autocomplete while programming. Find more in the sections "[Autocomplete for instruction or variable](#)" (page 43), "[Displaying instruction parameters](#)" (page 45) and "[Displaying declaration of instruction or variable](#)" (page 46).

## Example

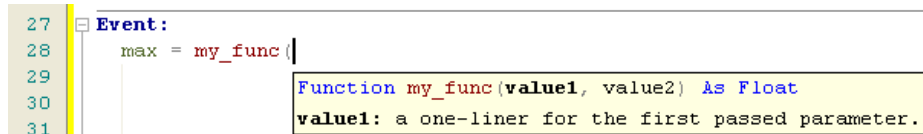
The following examples show how to organize the description text:

```
''' This starts the description of the function my_func;
''' + the first description part refers to the function's
''' + task. All lines beginning with 3 comment chars and a
''' + plus form a description part.
''' val1: a one-liner for the first passed parameter.
''' The second passed parameter val2. Obviously the line
''' may start without the parameter name but clear design
''' should use it anyway.
''' These lines start a new part, which cannot be related
''' to passed parameter and therefore is not used.
Function my_func(val1, val2) As Long
    my_func = (val1 + val2) / 2
EndFunction

''' Now for the description of a variable.
''' Multi-line descriptions do not require the plus char
''' but it does no harm either.
Dim var As Float

''' Description of a symbolic name
#Define max Par_1
```

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



The screenshot shows a code editor with line numbers 27 to 31. Line 27 has a tooltip icon. A tooltip box is open, displaying the function signature and its first description line. The code in the background is:

```
27 Event:
28     max = my_func |
29
30
31
```

The tooltip content is:

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

Please note the rules how to create own descriptions:

- You can document the following self-defined elements
  - Symbolic names (**#Define**)
  - Variables and arrays (**Dim**)
  - Macros (**Sub**, **Function**)
  - Libraries (**Lib\_Sub**, **Lib\_Function**)
- Insert all description text into comment lines, which start with 3 comment chars ' ' '. A line with less than 3 comment chars stops the description range.
- Description lines must be placed directly above the definition line containing one of the keywords given above. If done correctly, the description range is displayed as foldable text range (see [page 28](#)).

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

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

Display of the description texts (e.g. in tooltips) is only available, if the option `Parse Declarations` under **Editor - General** (see [page 64](#)) is active.

### 3.4.7 Defining a foldable text range

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

line of the range. You fold the range with a click on the minus sign in the first line.

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

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

- Mark the source code lines.

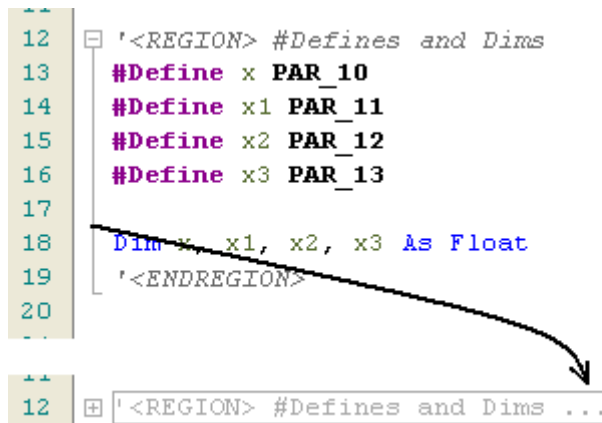
Please make sure to completely enclose control structures within the range or exclude them. Including (or overlapping) another self-defined foldable range is not allowed.


- In the context menu of the source code window, select the menu entry `Create Region from Block`.

A user dialog opens.

- Enter a short description of the foldable range under `Region description` and confirm with `OK`.
- The marked range is now enclosed by the keywords `<Region>` and `<EndRegion>` and can be folded and unfolded.

Alternatively, you can enter the region keywords manually.



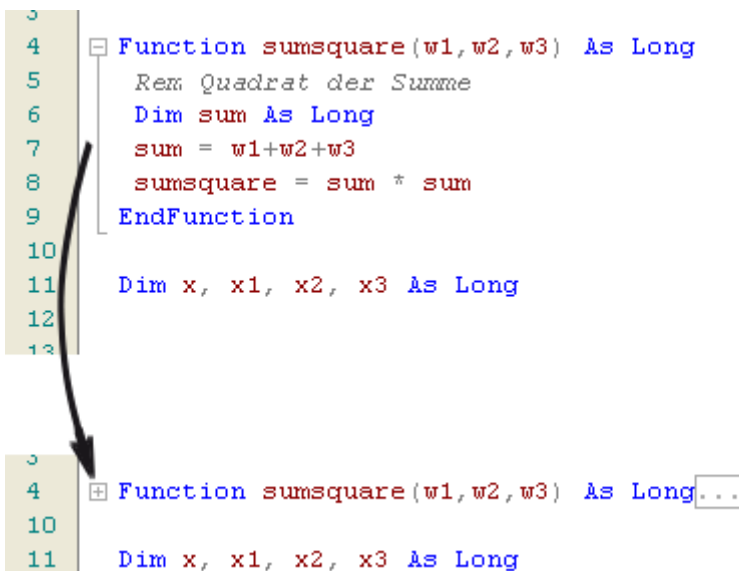
Using the button **Toggle Folding**  all foldable text ranges may be folded or unfolded at once.

Foldable text ranges can be recognized only, if the option `Parse Declarations` under `Editor - General` (see [page 64](#)) is active.

### 3.4.8 Folding text ranges

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

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



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

Foldable text ranges can be recognized only, if the option `Parse Declarations` under `Editor - General` (see [page 64](#)) is active.

### 3.5 Searching and replacing

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

- [Finding text quickly, page 31](#)
- [Finding and replacing text, page 32](#)
- [Regular expression, page 37](#)
- [Marking control blocks, page 40](#)
- [Using bookmarks, page 40](#)
- [Jumping to a program line, page 41](#)
- [Jumping to declaration of instruction or variable, page 41](#)
- [Switching between jump targets, page 41](#)

There are more editor functions:

- [Creating source code, page 17](#)
- [Formatting source code, page 24](#)
- [Writing programs with ease, page 42](#)

#### 3.5.1 Finding text quickly

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

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

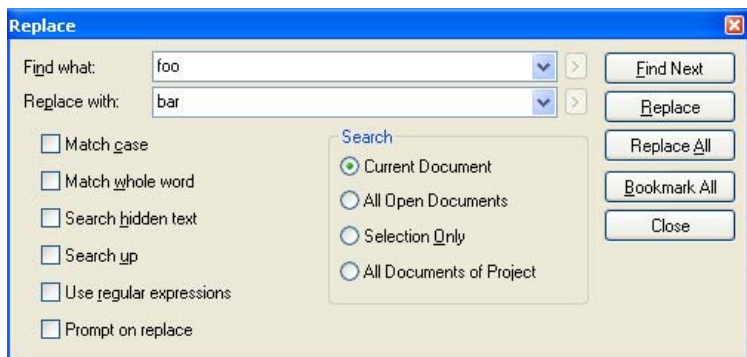
- Uppercase and lowercase letters are of no importance.
- Find text also as part of a word.
- Folded text areas are searched.
- All open documents are searched.

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

### 3.5.2 Finding and replacing text

You can find any occurrence of a combination of any characters, including uppercase and lowercase characters, whole words, parts of words, or regular expressions (see [Regular expression](#) on [page 37](#)).

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



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

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



Option	Description
Search hidden text	<p>The option refers to <a href="#">Folding text ranges</a> (see <a href="#">page 30</a>).</p> <p>Option active: Folded text areas are searched.</p> <p>Option inactive: Folded text areas are skipped.</p>
Search up	<p>Option active: Search in direction to start of file.</p> <p>Option inactive: Search in direction to end of file.</p>
Use regular expressions	<p>Specify that the search string is a <a href="#">Regular expression</a> (see <a href="#">page 37</a>).</p>
Prompt on replace	<p>Option valid with Replace All only.</p> <p>Option active: Each occurrence opens a dialog box to control replacing.</p> <p>Option inactive: All occurrences are replaced without query.</p>

### 5. Set the search range.

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

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

a. Files of the Other Files section are excluded from the search in any case.

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

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

### Notes

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

### Examples – Finding Text

Examples for finding text with [Regular expressions](#).

- Find all spaces or tabs at the end of a line:  
`[ ]+$`  
The search string finds one or more spaces or tabs, being followed by the end of the line.
- Find everything on a line:  
`^.+`  
The search string finds the beginning of a line, followed by one or more of any characters, up to the end of the line.
- Find `$12.34`:  
`\$12\.34`  
Note that `.` and `$` have been escaped using the backslash `\` to hide their regular expression meanings.
- Find a string, which is valid as variable name in *TiCoBasic*:  
`\b[a-z] [_a-z0-9]*`

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

- Find an innermost bracketed expression:

```
\ ( [^\ ( \) ] * \ )
```

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

- Find a repeated expression:

```
( [0-9] + ) - \ 1
```

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

### Examples – Replacing Text

Examples for replacing text with [Regular expressions](#).

- Find two numeric strings separated by one or more spaces:

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

and swap them around, using a colon to separate them:

```
$2:$1
```

- To change simultaneously:

```
from X100000 to X100.000
```

```
from Y100123 to Y100.123
```

```
from Z600 to Z.600
```

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

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

### 3.5.3 Regular expression

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

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

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

Meta char:	Meaning:
.	Any single character. Example: <code>Ma.s</code> matches <code>Mats</code> , <code>Mars</code> and <code>Mads</code> , but not <code>Mas</code> .
[ ]	Any one of the characters 1. given explicitly in brackets, or 2. any of a range of characters separated by a hyphen (-). Examples: <code>h[aeiou][a-z]d</code> matches: <code>hard</code> , <code>head</code> , <code>hand</code> and <code>hold</code> ; <code>[A-Za-z]</code> matches any single letter. The regular expression <code>x[0-9]</code> matches <code>x0</code> , <code>x1</code> , ..., <code>x9</code> .
[^]	Any characters except for those after the caret ^. Example: <code>h[^uo]t</code> matches <code>hat</code> and <code>hit</code> , but not <code>hot</code> or <code>hut</code> .
^	The start of a line (column 1). Example: The search string <code>^start</code> matches <code>start</code> only, when it is the first word on a line.

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

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

### 3.5.4 Marking control blocks

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

Only one control block can be highlighted at a time.

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

The following control block can be highlighted:

- Program sections `Init:`, `Event:`, `Finish:`
- `Do ... Until`
- `For ... Next`
- `If ... EndIf`
- `SelectCase ... EndSelect`
- `Function ... EndFunction`
- `Sub ... EndSub`
- `Lib_Function ... Lib_EndFunction`
- `Lib_Sub ... Lib_EndSub`

All control structures are also foldable text ranges (see [Folding text ranges](#) on page 30).

### 3.5.5 Using bookmarks

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

You can use these actions:

- Set a Bookmark

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

Use `Toggle Bookmark` to remove single bookmarks.

- Go to Next Bookmark



Select the `Next Bookmark` button from the editor bar.

- Go to Previous Bookmark

Select the `Previous Bookmark` button from the editor bar.

- Remove all Bookmarks

Select the `Delete all Bookmark` button from the editor bar.

Use `Toggle Bookmark` to remove single bookmarks.

Bookmarks are saved together with the source code file.

### 3.5.6 Jumping to a program line

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

To show source code line numbers, the option `show linenumbers` under `Editor – General` (see [page 64](#)) must be enabled.

### 3.5.7 Jumping to declaration of instruction or variable

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

To jump to a declaration, you place the cursor on the self-declared name and then either select `Jump to Declaration` from the context menu (right mouse click), or click the `Jump to Declaration` button in the editor bar.

A jump to declaration is only available, when the option `Parse Declarations` under `Editor – General` (see [page 64](#)) is active.

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

### 3.5.8 Switching between jump targets

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



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

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

Used bookmarks are not collected in the jump target history, see [Using bookmarks](#).

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

When you open a project the history is deleted.

### **3.6 Writing programs with ease**

Be at ease while programming using the following functions:

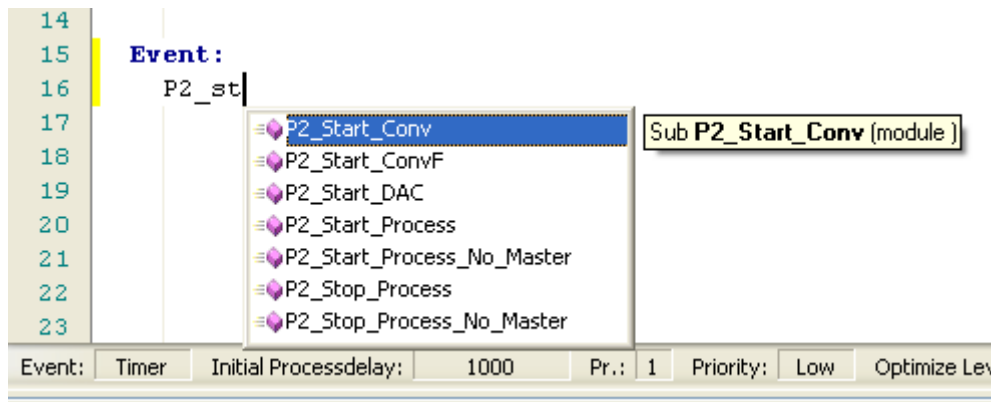
- [Autocomplete for instruction or variable, page 43](#)
- [Inserting code snippets, page 44](#)
- [Displaying declaration of instruction or variable, page 46](#)
- [Displaying declarations of a file, page 46](#)
- [Displaying used global variables and arrays, page 46](#)

Find more editor functions here:

- [Creating source code, page 17](#)
- [Formatting source code, page 24](#)
- [Searching and replacing, page 31](#)

### 3.6.1 Autocomplete for instruction or variable

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



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

Do as follows:

1. Write the first letters of the word and press CTRL-SPACE.  
A drop-down list opens, the entries of which fit to complete the previous letters.  
If you use autocomplete behind a space character, the list will contain all available keywords.
2. Select the desired list entry with mouse or arrow keys.

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

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


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

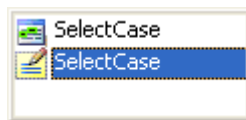
Autocomplete is only available, when the option `Parse Declarations` under [Editor – General](#) (see [page 64](#)) is active.

### 3.6.2 Inserting code snippets

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

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

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



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

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

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

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

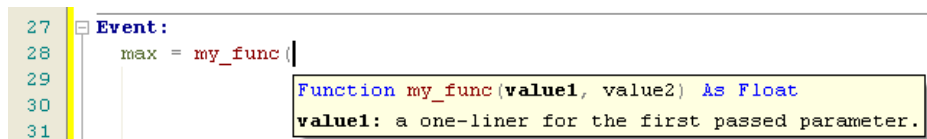
After you have selected a code snippet, the appropriate keyboard shortcut is displayed to the right.

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

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

### 3.6.3 Displaying instruction parameters

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



The tooltip vanishes as soon as the cursor is placed outside the braces around the parameters. You can re-activate the tooltip if you retype the opening brace. Alternatively, you can call the function `Declaration Info` from the context menu or the editor bar to display the complete declaration of the instruction.

Self-defined instructions and their parameters can be documented with description texts being displayed in the same way, see "[Documenting self-defined instructions and variables](#)".

The display of instruction parameters is only available, when the option `Parse Declarations` under [Editor – General](#) (see [page 64](#)) is active.

### 3.6.4 Displaying declaration of instruction or variable

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

- move the mouse over the keyword.

The declaration is displayed only, when the option `Display quick info on mouse over` under **Editor – General** (see [page 64](#)) is active.

- set the cursor on the keyword and press [F2].
- set the cursor on the keyword and select `Declaration Info` in the editor bar or in the context menu.

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


The display of declarations is only available, when the option `Parse Declarations` under **Editor – General** (see [page 64](#)) is active.

### 3.6.5 Displaying declarations of a file

To display all declarations, include and library files referring to a source file, set the **Declarations Window** to the foreground (see [page 87](#)). Declarations of other source code files will not be displayed—even if combined within a project.

The display of declarations is only available, when the option `Parse Declarations` under **Editor – General** (see [page 64](#)) is active.

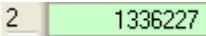


### 3.6.6 Displaying used global variables and arrays

You can display global variables and arrays being used in the active source code and in the appropriate project (if present) by a click on the `Scan Global Variables` button  in the **Parameter Window** (see also [page 77](#)).

This results in two displays:

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

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

- **Green:** Parameter is used in the active source code only.  2 1336227
- **Red:** Parameter is used both in the active source code, and in another source code of the project, too.  1 707279
- **Blue:** Parameter is used in an inactive source code of the project, and not in the active source code.  9 5B12H


Using the [Clear Scan](#) button , both displays are cleared.

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

### 3.6.7 Adding file and folder shortcuts

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

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



- Add file shortcut...
- Add folder shortcut...
- Remove: no file link selected
- Large icons

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

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

## 3.7 Transferring a TiCo binary file to TiCo processor

Using the environment *TiCoBasic*, you can transfer a saved binary file to the *TiCo* processor or into the *TiCo* bootloader.

### 3.7.1 Transferring a TiCo binary file

To transfer a binary file to the *TiCo* processor, do as follows:

- Create a binary file using *Build ► Make Bin File*; see also [page 56](#).

Here, set the process attributes as priority, process number, process type etc.

- Select the menu entry *Load Bin File* in the menu *Tools* and the binary file in the next window.

Confirm the selection with *Open*; now, *TiCoBasic* transfers the binary file as process to the *TiCo* processor which is set in the compiler options.

- The process is started automatically.



### 3.7.2 Programming the TiCo bootloader

The *TiCo* bootloader automatically loads and runs selected *TiCoBasic* processes on start-up of the *ADwin* hardware.

To program the *TiCo* bootloader, do as follows:

- Create a binary file using Build ► Make Bin File; see also [page 56](#).

Here, set the process attributes as priority, process number, process type etc.

- Select the menu entry *Bootloader...* in the menu *Tools*.

The dialog window *Bootloader* opens.



- Click on the button *Load Boot loader* and select the binary file in the next window.
- Enable the bootlader operation with the button *Enable Boot loader*. Up from now, the processes of the binary file are automatically started upon start-up of the *ADwin* hardware.

Using the button `Disable Bootloader`, you can temporarily disable the bootloader operation and enable it later on again.

- Close the dialog window with `Close`.
- After next start-up of the *ADwin* hardware, the process of the binary file is automatically started

### 3.8 Managing Projects

One project can manage many process source codes, include files, library files, and files of other type, for instance when programming an application with several processes. The files are displayed in the [Project Window](#), see [page 75](#). Only one project can be open at a time.


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

The following rules apply to the combination of project files (see also [chapter 6.1 "Process Management"](#)):

- At least one process must have high priority.
- Only a single timer-controller process with high priority is allowed.
- There can be one low priority timer-controller process, if a high priority process belongs to the project, too.

You can combine a timer-controller process and an externally controller process. In this case, please contact our support ([support@adwin.de](mailto:support@adwin.de)); we will inform you about the required precautions.

A project allows the following actions.

- Managing a project
  - Create a new project: File► New Project.
  - Open a saved project: File► Open Project.
  - Save a project: File► Save Project / Save Project As.  
Only project settings and display parameters are saved, but not the source files of the project.
  - Close the project: File► Close Project.  
Closing the project will also close all the files of the project.
  - Change project settings: Options► Project Settings or button Project Settings  in the project bar; see [Project Settings dialog box](#).
- Adding files to a project
  - Add a file to the project: File► Add to Project or Add to Project in the context menu of the source code.
  - Add all open files to the project:  
Add Open Files To Project in the context menu of the [Project Window](#), see [page 75](#).
  - Add a file of other type to the project:  
Add other File to Project in the context menu of the [Project Window](#), see [page 75](#).  
The file is displayed under project category Other Files.

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

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

- With adding a file to a project, the file path is saved relative to the project folder. Only if the file is located on a different drive, an absolute file path is saved. Managing files of a project
  - Open a file of the project as active source code: double click the file in the [Process Window](#), see [page 79](#).
  - Remove selected files from project:  
DEL key or Remove from Project in the context menu of the [Project Window](#), see [page 75](#).
  - Save all files of the project at once:  
Save all Files of Project in the context menu of the [Project Window](#), see [page 75](#).
- Compiling project source code
  - Compile all project files: Build► Compile.
  - Create all binary files of the project:  
Build► Make Bin File.

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

- Searching through all files of a project, including not yet opened files (except files of project category `Other Files`).  
To do so, enable the `All Documents of Project` option in the find window (see [chapter 3.5.2 "Finding and replacing text"](#)). The option is not available for replacing.
- [Displaying used global variables and arrays](#) of a project (see [page 46](#)).
- Opening the Windows Explorer with the path of the selected file, using `Open Path in Explorer Window` from the project window context menu

Project-related capabilities can be accessed via project window context menu (right mouse click, see ["Project Window" on page 75](#)), via project bar in the project window, or in the menu `File` ([chapter 3.9.1](#)).

### 3.9 Menus

The menu bar contains these menus:

- File: Manage files and projects ([page 54](#))
- Edit: Edit source codes ([page 55](#))
- View: Show windows and bars ([page 55](#))
- Build: Tool for generating executable programs ([page 56](#))
- Options: Program settings ([page 57](#))
- Debug: Tools for error detection ([page 70](#))
- Tools: Various help functions ([page 72](#))
- Window: Arrange source code windows ([page 73](#))
- Help: Help, version and license information ([page 73](#))

### 3.9.1 File Menu

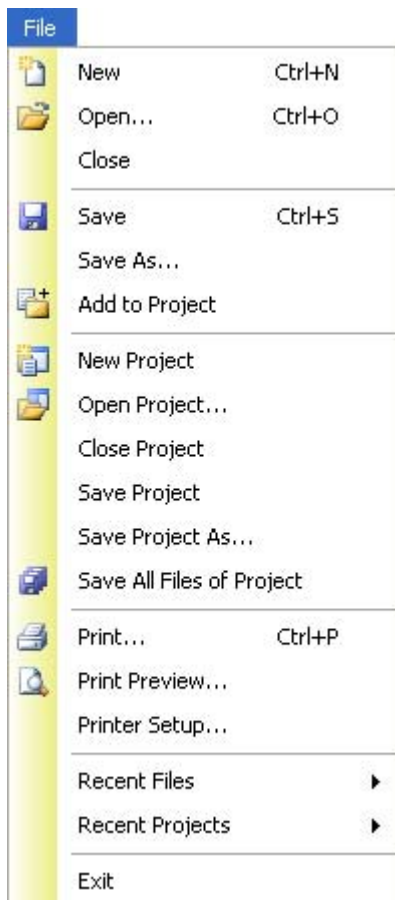
The `File` menu contains instructions for managing files and projects.

Files can be opened, created, saved, or closed. Multiple source code windows may be open simultaneously.

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

The print functions can also be found in the menu.

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










### 3.9.2 Edit Menu

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

Moreover, the menu offers functions for searching (`Find`, `Find Next`) and replacing (`Replace`); see [Finding and replacing text](#) on [page 32](#).

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.

Edit		
	Undo	Ctrl+Z
	Redo	Ctrl+Shift+Z
<hr/>		
	Cut	Ctrl+X
	Copy	Ctrl+C
	Paste	Ctrl+V
<hr/>		
	Select All	Ctrl+A
<hr/>		
	Find...	Ctrl+F
	Find Next	F3
	Replace...	Ctrl+H
	Goto Line..	Ctrl+G

### 3.9.3 View Menu

In the `View` menu, you may open or close

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

You find further information about the process window in [chapter 3.10.4 on page 79](#), about the toolbar see [fig. 3](#).

View	
<input checked="" type="checkbox"/>	Standard Toolbar
<input checked="" type="checkbox"/>	Editor Toolbar
<input checked="" type="checkbox"/>	ADtools Toolbar
<input checked="" type="checkbox"/>	Statusbar
<hr/>	
	Restore Default Layout

With `Restore Default Layout`, the default layout, which was active at the initial start of the *TiCoBasic* program, can be restored with a single mouse-click. This refers also to the [Toolbox](#) settings ([page 75](#)).

### 3.9.4 Build Menu

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

- a process using **Compile**.
- a binary file using **Make Bin File**.
- a library using **Make Lib File**.



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

A change of file may occur by automatic indenting of text lines (see [chapter 3.4.3 on page 25](#)), for example when opening a previously unformatted file.

**Compile** is the most comprehensive instruction: It compiles the whole project (without project: a single source code) and transfers the generated binary file as process to the *TiCo* processor.

The process is automatically started on the *TiCo* processor.

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

**Make Bin File** is only available for licensed *TiCoBasic* users. It compiles the whole project (without project: a single 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 **.TIx**. The **x** denotes the processor type (see [MakeLibFileOptions Menu](#), [Process Options dialog box](#)).



A binary file with the extension **<\*.TI1>** can be transferred to a *TiCo* processor of type 1. A binary file can be transferred to the *TiCo* processor with *TiCoBasic* (see [chapter 3.7.1 "Trans-](#)

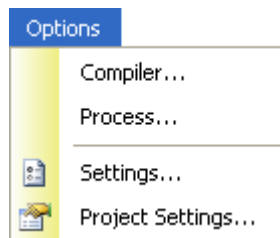


ferring a *TiCo* binary file") or from a development environment such as C or Visual Basic.

`Make Lib File` is available for licensed *TiCoBasic* users only. It compiles the complete project (or a single 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 `.TLx` (where `x` denotes the processor type.) Afterwards the library can be included into other source codes that use their functions and subroutines (see [chapter 4.5.1 on page 112](#)).

### 3.9.5 MakeLibFileOptions Menu

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



#### Compiler Options dialog box

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

To compile source codes for different *TiCo* processors, the parameters need to be set for each *TiCo* processor in the dialog box.



Fig. 4 – The Compiler Options dialog box

- System: Select the *ADwin* system.
- Processor: Select the system's processor type.
- Device No.: Select the device number to access the *ADwin* system.

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

- Module Address (only *ADwin-Pro II* systems): Select the module, where the *TiCo* processor is located.
- Do not access the Device: If inactive, a binary file will be automatically transferred to the hardware after compilation. Thus, the *ADwin* hardware must be connected before compilation.

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

Please note: The menu entries for transfer of bootloader or binary files ([chapter 3.9.7 "Tools Menu"](#), [page 72](#)) are enabled only, when this option is disabled.

- Remember Device No.: Active option saves the last used Device No. (see above) on closing *TiCoBasic*; the next start-up will automatically use the saved number.

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

### Process Options dialog box

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

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

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

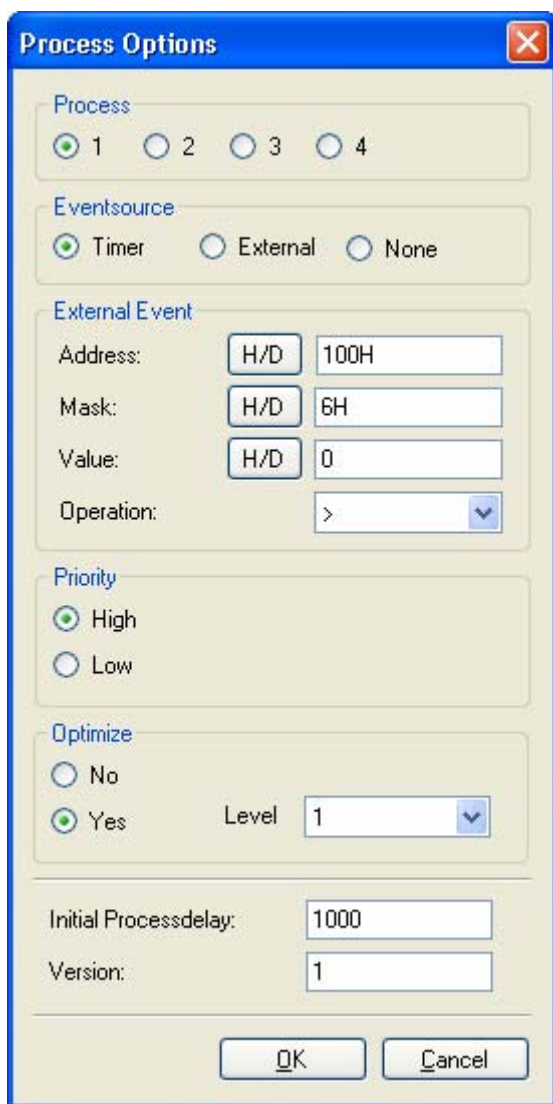


Fig. 5 – The Process Options dialog box

- **Process:** Process number

The number, under which the transferred process is started on the *TiCo* processor .

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

- **Eventsource:** Sets the event source signal, which initiates the **Event:** section of the process.
  - **Timer**  
sets the internal counter as event signal. The system variable **Processdelay** determines the delay, in which the counter creates an event signal.
  - **External**  
sets the (external) signal the event input of the *ADwin* system as event signal. A specified value in a given **Address** is used as event signal. The process always runs with high priority.  
Further settings see below at **External Event**.
  - **None**  
The process type **None** (without event signal) is only used for special applications—mostly programmed in assembler—and excludes other process types. If not programmed differently, the process does not respond to external event signals and is run only once.

- **External Event:** Settings for external event source.

The settings determine the conditions and the hardware which releases an event signal. Any values can be entered hexadecimal or decimal.

An event signal is released as follows: The conjunction of the value in the hardware **Address** and the bit **Mask** is compared to **Value** using the **Operation**. If the comparison is true, an event signal is released.

Hardware addresses, bit masks, and comparison value are different for each *ADwin* hardware. Please note our example pro-

grams under `C:\ADwin\TiCoBasic\`, where the word `Extern_` is part of the file name.



External Event	
Address:	<input type="radio"/> H/D <input type="text" value="70H"/>
Mask:	<input type="radio"/> H/D <input type="text" value="12H"/>
Value:	<input type="radio"/> H/D <input type="text" value="0"/>
Operation:	<input type="text" value="&gt;"/> <input type="button" value="v"/>

Example: The settings above mask the value of address `70h` with `12h`, so only the bits 2 and 5 remain unchanged. If the result is `> 0`, i.e. one of the two bits is set, a process cycle is started with an event signal.

- **Priority:** The priority of the process.

Set the priority the process will be run with in the *ADwin* hardware. For more information, see [chapter 6.1 "Process Management"](#).

- **Optimize:** Status and level of compiler optimization.

Compiler optimization, which may be used optionally, can reduce the execution time of the process by up to 20 percent. A higher setting under `Level` will lead to shorter execution times.

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.

As an alternative, you can set the variable `Processdelay` in the source code.

- `Version`: An integer value for differentiating between several versions of a process.
- `Stacksize`: Processor T12 only. The program's stack size in Bytes.

### Settings dialog box

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

- Editor
  - [Editor – General](#)
  - [Editor – Syntax Colors](#)
  - [Editor – Print Settings](#)
- [Language](#)
- [Directories](#)
- [Help](#)
- [ADtools](#)

### Editor – General

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



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

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

**Autoindent:** Source code is indented automatically. Indent positions are set via `Tabsize`. See also "[Indenting text lines](#)" on [page 25](#).

**Indent TiCoBasic sections:** Program sections are indented by one tab more.

**Smart format:** Format lines automatically, see "[Smart formatting](#)" on [page 24](#).

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



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

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

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

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

[Processor](#)Show line numbers: Line numbers are displayed in the gutter left of the source code. See also "[Jumping to a program line](#)" on [page 41](#).

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

### Editor – Syntax Colors

The editor highlights the syntax elements with different colors; see also [chapter 3.4.1 "Syntax highlighting"](#) on [page 24](#); complete syntax highlighting requires an active option `Parse Declarations` under [Editor – General](#).

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

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

The example text above shows how source code be formatted.

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

You can change the source code font size under [Editor – General](#), `Font size`.

The editor distinguishes the following syntax elements:

- *TiCoBasic*-Syntax (System related):
  - *TiCoBasic* sections: Keywords **Init:**, **Event:** and **Finish:** for program sections.
  - Compiler Directives: Pre-compiler instructions like **#Define**, starting with a **#**.
  - Reserved Keywords: Basic instructions as **Dim** in *TiCoBasic*.
  - Global Variables: Global variables **Par\_1 ... Par\_80** and **Data\_1 ... DATA\_16**.
  - External Keywords: *TiCoBasic* instructions for access to inputs/outputs like **P2\_ADC**. Most of these instructions are declared in the delivered standard include or library files.
  - Symbols: Operators as braces, + or =.
- User related:
  - Defined Names: Symbolic names like **myName**, declared with **#Define**.
  - Local Variables: Variables like **myVar** declared with **Dim**.
  - Sub Names: Names (like **mySub**) of user-defined modules, declared with **Sub** or **Lib\_Sub**.
  - Function Names: Names (like **myFunction**) of user-defined modules, declared with **Function** or **Lib\_Function**.
- Other:
  - Numbers: Numbers in decimal (**15**), hexadecimal (**0Fh**) and binary notation (**1111b**).
  - Strings: Strings in "double quotes".
- Comments: Comments after *Rem* or quote **'**.
- Standard Text: All elements, which do not belong to other groups, e.g. invalid instructions like **Eixt** (instead of **Exit**).

## Editor – Print Settings

The settings refer to printing of source code.

Header refers to the printed header line.

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

**Header text:** The text of the header line.

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

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

**Syntax Highlight:** Syntax highlighting is printed.

**Line numbers:** Line numbers are printed at the left.

**Font size:** Sets the font size of the output.

### Language

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

### Directories

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

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

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

### Help

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

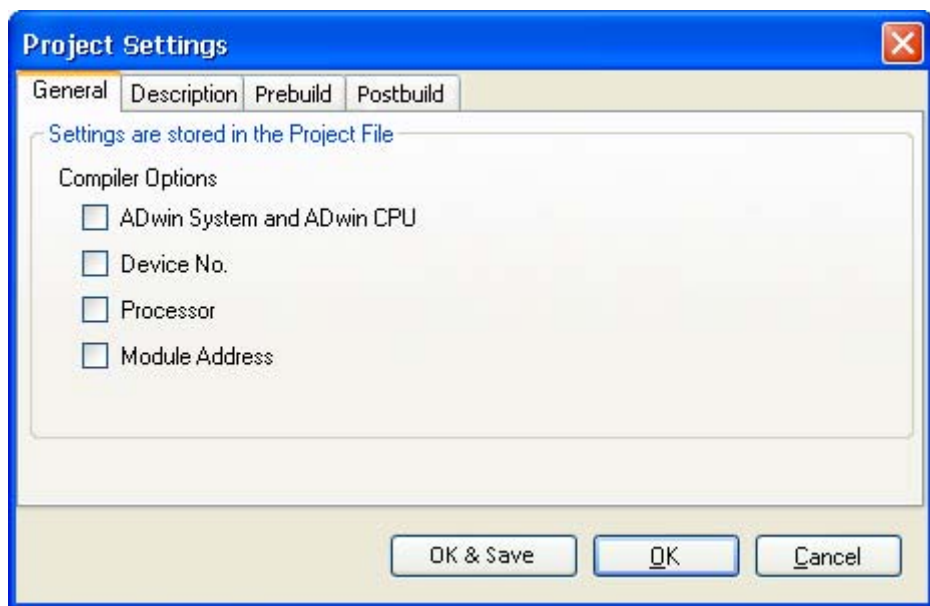
### ADtools

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

### Project Settings dialog box

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

- [Tab General](#)
- [Tab Description](#)
- [Tabs Prebuild / Postbuild](#)



### Tab General

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

- Dialog window Compiler OptionsADwin System and ADwin CPU
  - DeviceNo.
  - Processor
  - Module Address
- Debug menu
- Debug Mode

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

### Tab Description

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

**Tabs** Prebuild / Postbuild

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

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

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

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

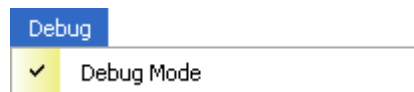
- \$PROJECTNAME: Project file name (without path and file extension).
- \$PROJECTPATH: Path of project directory (without file name).
- \$PROJECT: Projekt file name (without path).
- \$BUILDACTION: String indicating the triggering instruction (see above).
  - C: Menu entry Build► Compile
  - MB: Menu entry Build► Make Bin File
  - ML: Menu entry Build► Make Lib File

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

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

### 3.9.6 Debug Menu

The Debug menu offers the option Debug mode whichs help in debugging a given source code.



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

### Debug mode Option

The **Debug mode** compiler option, when activated, includes additional security queries into the process during the compilation of a source code (see also [chapter 5.3.1 on page 119](#)).

The setting of this compiler option is displayed in the [Status Bar](#), the setting of a running process in the [Process Window](#).

Activation of this option increases program execution time as well as the demand for memory. Therefore, this option should only be used during program development.

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

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

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

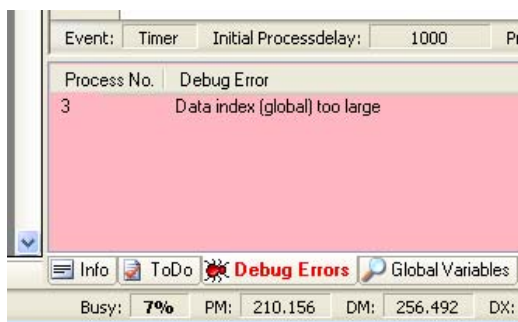


Fig. 6 – The Debug Errors Window

The following table shows, which errors are displayed and which corrections are made.

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.

### 3.9.7 Tools Menu

The `Tools` menu option calls utility programs.

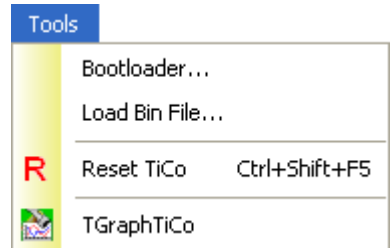
The menu entry `Bootloader...` programs the *TiCo* bootloader (see [chapter 3.7.2 "Programming the TiCo bootloader", page 49](#)). The bootloader can start a process automatically on start-up of the *ADwin* hardware.

The menu entry `Load Bin File...` transfers a saved binary file to the *TiCo* processor (see [chapter 3.7.1 "Transferring a TiCo binary file", page 48](#)).

The menu entry `Reset TiCo` stops and resets the *TiCo* processor; all global *TiCo* variables are set to zero and the process is deleted.

The menu entries `Bootloader...` and `Load Bin File...` are only available when the option `Do not access the device` in the [Compiler Options dialog box](#) is disabled (see [page 57](#)).

The menu entry `TGraphTiCo` starts a utility tool; short description see [chapter 3.12 on page 89](#).



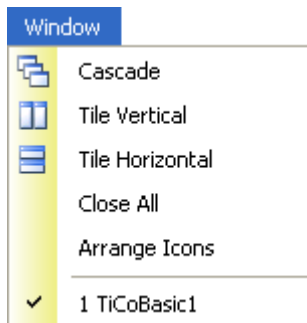


### 3.9.8 Window Menu

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

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

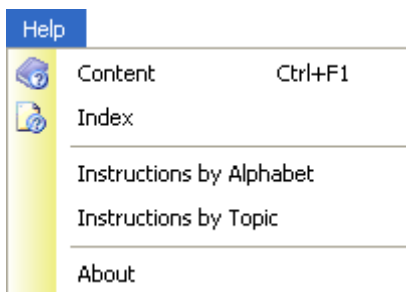
At the bottom of the menu, there is a list of open source codes; by clicking one of these menu items that source code will become the active window. The active source code is checked; in the example at right it is `TiCoBasic1.bas`.




### 3.9.9 Help Menu

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

- Content: Table of contents
- Index: Index directory
- Instructions by:  
Sorted lists of instructions.
  - by Alphabet
  - by Topic



The instruction lists refer to that ADwin system, which is set in the [Compiler Options dialog box](#) (page 57).

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

The About menu entry opens a window that displays the version of the development environment and the License key. The license key can be entered or changed by pressing the Change License button (see also [page 9](#)).

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

### 3.10 Windows

There is separate information about the [Info range](#) on [page 83](#).

#### 3.10.1 Toolbox

The Toolbox is the window range of the environment to the left, where [Project Window](#), [Parameter Window](#), [Register window](#), and [Process Window](#) are displayed.

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

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

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



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

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

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

#### 3.10.2 Project Window

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

The project window is located in the [Toolbox](#) (see [page 75](#)).

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

- Add all open files to the project:  
Select `Add Open Files to Project` from the project window context menu.  
  
Additionally you can add a single file to the project:  
Select `Add to Project` from the source code context menu.
- Add a file of other type to the project:  
Select `Add Other File to Project` from the project window context menu.
- Delete a source code file from the project:  
Highlight the file in the project window, then
  - press the `[DEL]` key or
  - select `Remove from Project` from the context menu.
- Open a source code file and make it the active source code:
  - Double-click the file or
  - Highlight the file in the project window, then select `Open` from the context menu (right mouse button).
- Save all open source code files of the project:  
Select `Save All Files of Project` from the context menu.
- Open the Windows Explorer with the path of the selected file:  
Select `Open Path in Explorer Window` from the context menu.

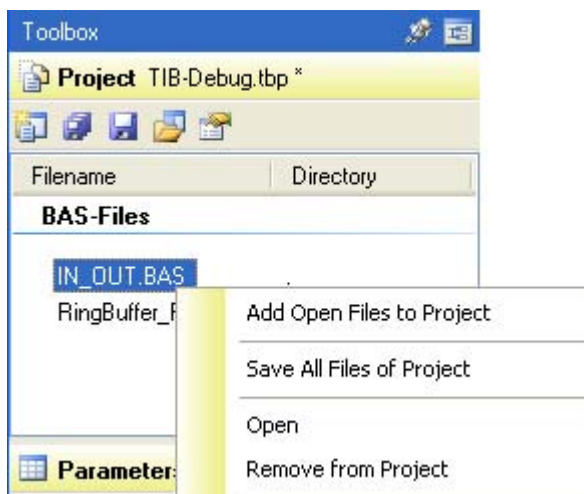


Fig. 7 – The Project Window with the Context Menu

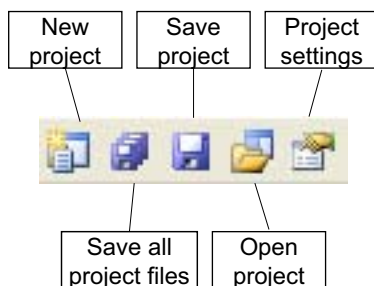



Fig. 8 – The project bar

### 3.10.3 Parameter Window

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

The parameter window is located in the [Toolbox](#) (see [page 75](#)).

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

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

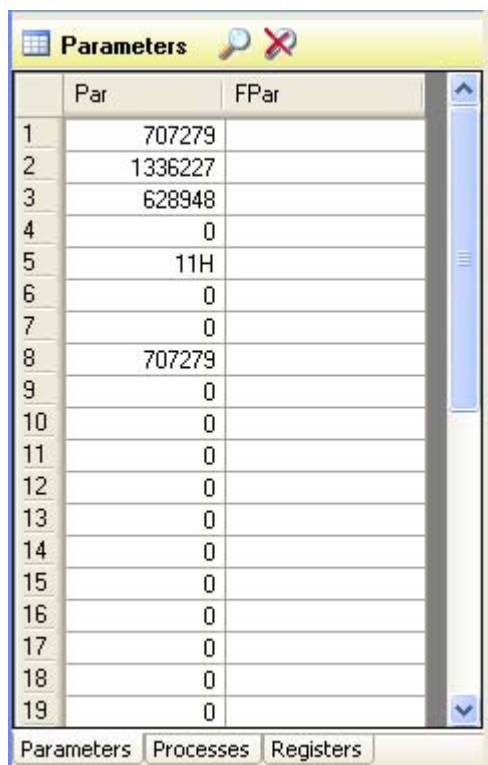




Fig. 9 – The parameter window

To change the display of a parameter's value between decimal and hexadecimal notation (see `Par_5` in [fig. 9](#)), do a mouse click on the number of the variable (left of the table field). A click on the column header changes the display of all parameters at once.

For use of the Scan Global Variables  button, see "Displaying used global variables and arrays" on [page 46](#).

### 3.10.4 Process Window

The process window shows information about the processes 1...4 on the *TiCo* processor, when the communication between the computer and the system is active (icon  in the toolbar). Otherwise the fields are gray.

The process window is located in the [Toolbox](#) (see [page 75](#)). Open the process window with a click on the tab `Processes`.

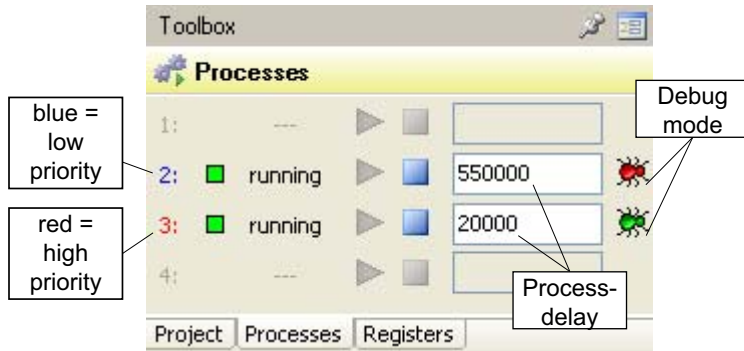


Fig. 10 – The Process Window

For each process, the following information is displayed:

- Process priority



The color of the process number indicates the priority:

- red = high priority
- blue = low priority

The time units and meaning of the process delay are explained in [chapter 6.2.1 "Processdelay"](#), [page 124](#).

- Process status

- `running`: process is running.
- `stopped`: process was stopped.
- `---`: process does not exist.

A process can be stopped with the `Stop` button  and started again with the `Start` button . The buttons of the toolbar have


the same function, but they refer to the process related to the active source code.

- Process delay (process cycle time)

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

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




- Process runs in debug mode 

The icon is displayed if the process runs in debug mode. Find more about debug mode under [Debug mode Option](#).

The compiler setting debug mode is displayed in the [Status Bar](#).

### 3.10.5 Register window

The register window shows the register contents of the *TiCo* processor, if the communication between PC and system is active (button  in the tool bar). Otherwise the fields are disabled.

The register window is located in the [Toolbox](#) (see [page 75](#)). Open the register window with a click on the tab *Registers*.

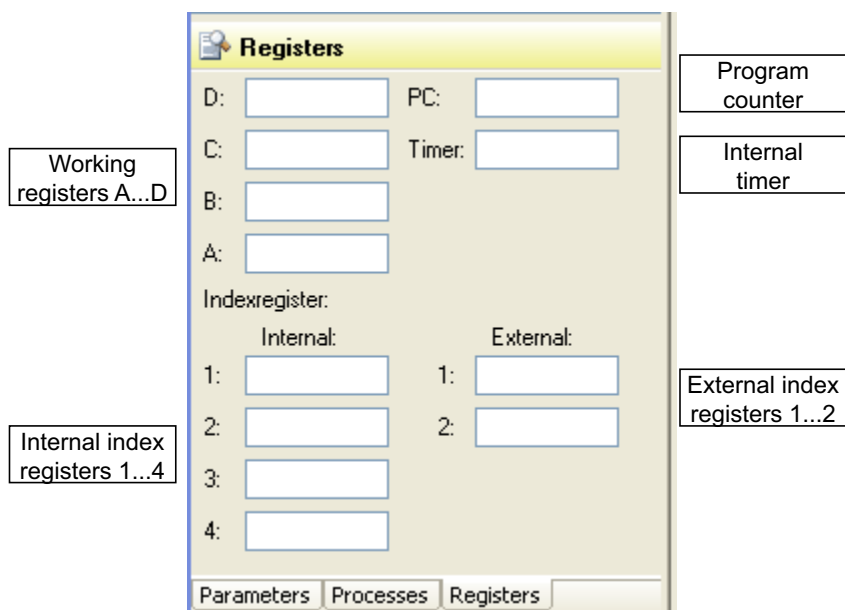


Fig. 11 – Das Register-Fenster

The register contents are useful if you use assembler code in a program. A documentation of assembler instructions is not yet available.

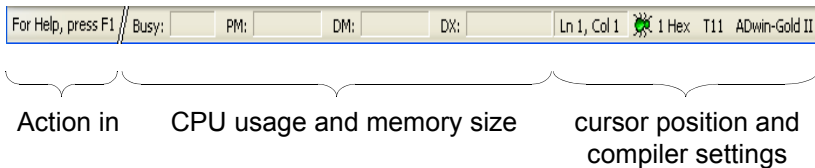
The source code window is the main window of the development environment, where you edit the source codes of *TiCoBasic* programs.

The multitude of functions in the source code window is described in [chapter 3](#). Important sections are the following:

- [Basic Elements of the Development Environment, page 12.](#)
- [Creating source code, page 17.](#)
- [Context menu in source code window, page 19.](#)

### 3.10.6 Status Bar

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



- Left side: Information about the last *TiCoBasic* action.
- Middle: The current workload (during active connection between PC and *TiCo*) and the memory size of the *TiCo* processor.
- Right: The current cursor position in the source code window (line and column); further compiler settings: debug mode, device no., processor, ADwin hardware.

A double click on the compiler settings opens the [Compiler Options dialog box \(page 57\)](#).

The displayed information about the CPU/memory usage:

- **Busy:** the processor workload in percent, calculated as:  
CPU time / (CPU time + idle time).
- **PM:** available program memory in bytes.
- **DM:** up to T11: available internal data memory in bytes.

- DX / SX: up to T11: available external data memory in bytes.

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

### 3.11 Info range

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

- [Info window](#) (see [page 83](#))
- [To-Do List](#) (see [page 85](#))
- The window `Debug Errors` (see [page 71](#))
- [Global Variables](#) (see [page 86](#))
- [Declarations Window](#) (see [page 87](#))

#### 3.11.1 Info window

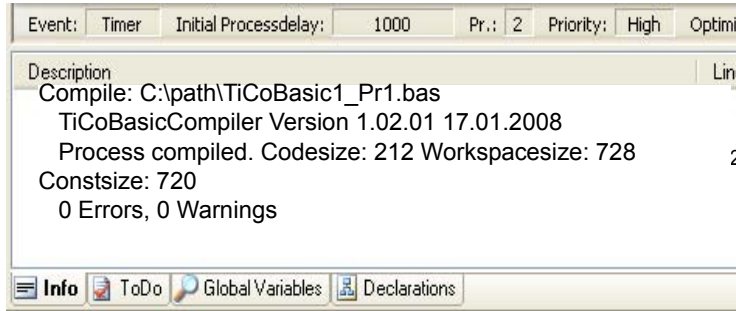
In the info window, the compiler messages concerning the current source code are displayed:

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

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

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

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



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

- **Codesize:** Size of the created binary file in bytes; the file will be stored in the program memory (PM) as process.
- **Workspacesize:** Required memory size in bytes in the local data memory (DM).
  - Compiler table: 720 byte
  - internal purpose: 8 byte
  - temporary compiler variables: size depends on program
  - variables and arrays declared [At DM\\_Local](#):
    - each local variable: 4 byte
    - each local array:  $(\text{array size} + 1) * 4 \text{ Byte}$
    - each global array **Data\_x**:  $(\text{array size} + 6) * 4 \text{ Byte}$
  - Even variables and arrays declared [At DRAM\\_Extern](#) require space in local data memory:
    - each local array: 0 Byte
    - each global array **Data\_x**: 16 Byte
- **Constsize:** Required memory size (DM) for constants in bytes.
- **Stacksize:** Internal stack size (DM) being used for libraries.

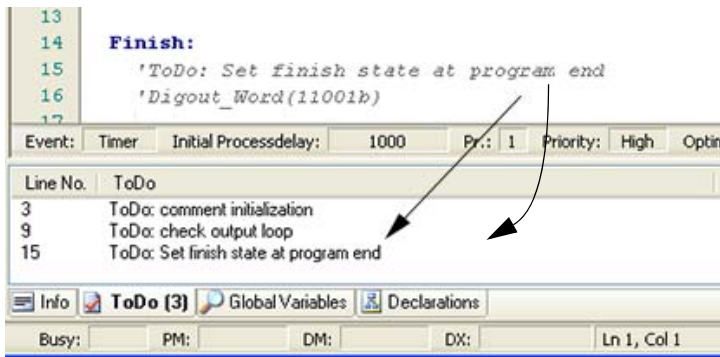
The memory size required in the external data memory (DX) will not be displayed.

### 3.11.2 To-Do List

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

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


The window is part of the [Info range](#) (see [page 83](#)).



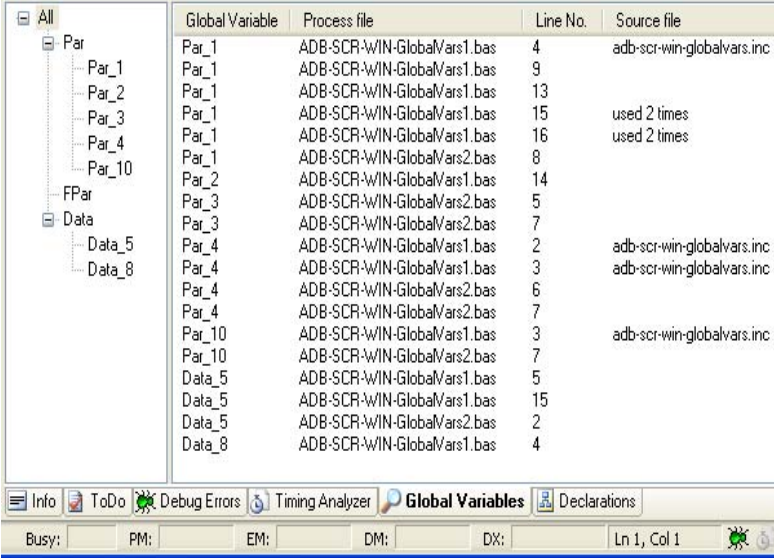
A double click on a To-Do entry positions the cursor in the appropriate line of the source code.

### 3.11.3 Global Variables

The window Global Variables displays, which global variables (Par\_1 ... Par\_80) and arrays (Data\_1 ... Data\_16) are used in a source code or a project.

To start or update the display click the button Scan Global Variables  in the [Parameter Window](#) (see [Displaying used global variables and arrays, page 46](#)).

The window is part of the [Info range](#) (see [page 83](#)).




Global Variable	Process file	Line No.	Source file
Par_1	ADB-SCR-WIN-GlobalVars1.bas	4	adb-scr-win-globalvars.inc
Par_1	ADB-SCR-WIN-GlobalVars1.bas	9	
Par_1	ADB-SCR-WIN-GlobalVars1.bas	13	
Par_1	ADB-SCR-WIN-GlobalVars1.bas	15	used 2 times
Par_1	ADB-SCR-WIN-GlobalVars1.bas	16	used 2 times
Par_1	ADB-SCR-WIN-GlobalVars2.bas	8	
Par_2	ADB-SCR-WIN-GlobalVars1.bas	14	
Par_3	ADB-SCR-WIN-GlobalVars2.bas	5	
Par_3	ADB-SCR-WIN-GlobalVars2.bas	7	
Par_4	ADB-SCR-WIN-GlobalVars1.bas	2	adb-scr-win-globalvars.inc
Par_4	ADB-SCR-WIN-GlobalVars1.bas	3	adb-scr-win-globalvars.inc
Par_4	ADB-SCR-WIN-GlobalVars2.bas	6	
Par_4	ADB-SCR-WIN-GlobalVars2.bas	7	
Par_10	ADB-SCR-WIN-GlobalVars1.bas	3	adb-scr-win-globalvars.inc
Par_10	ADB-SCR-WIN-GlobalVars2.bas	7	
Data_5	ADB-SCR-WIN-GlobalVars1.bas	5	
Data_5	ADB-SCR-WIN-GlobalVars1.bas	15	
Data_5	ADB-SCR-WIN-GlobalVars2.bas	2	
Data_8	ADB-SCR-WIN-GlobalVars1.bas	4	

The marked element of the tree view (on the left) determines, which variables are displayed in the window to the right. After a click on Par the list will display only Par\_1...Par\_80 (as far as used).

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

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

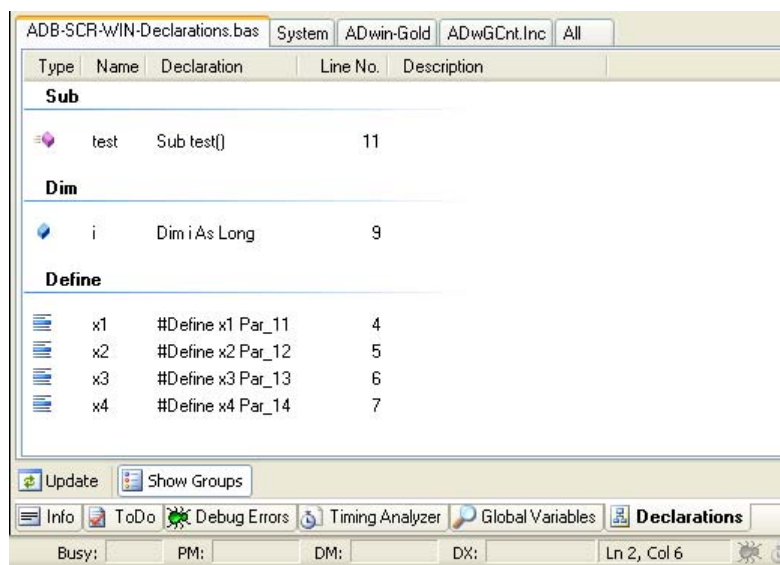
If you change the source code the window is not updated automatically. To do so, use the button `Scan Global Variables`  in the parameter window.

### 3.11.4 Declarations Window

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

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

The window is part of the [Info range](#) (see [page 83](#)).



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

- [file].bas: Declarations within the source file: local variables, arrays, instructions (**Sub**, **Function**) and symbolic names (**#Define**).
- System: System variables and instructions being implemented in *TiCoBasic*, if they fit to the current compiler settings.



Global variables **PAR** are not displayed here. Please note the [Global Variables \(page 86\)](#) and the function "[Displaying used global variables and arrays](#)" ([page 46](#)).

- ADwin-Gold, ADwin-light-16: Instructions for hardware access, which are implemented in *TiCoBasic* and fit to the current compiler settings.
- [file] .inc: Variables and instructions being declared in this include file. Such tabs only show up if there are **#Include** lines in the source code file.
- [file] .lib: Variables and instructions being declared in this library file. Such tabs only show up if there are **Import** lines in the source code file.
- All: All valid declarations of the above sources.

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

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

The display of declarations is only available, when the option **Parse Declarations** under **Editor – General** (see [page 64](#)) is active.

### 3.12 ADtools

*ADtools* is a collection of simple utility programs, which can display data and operation status of an *ADwin* system or a *TiCo* processor. Start one of the *ADtools* simply from the vertical bar at the right.

For *TiCo* processor, only the program **TGraphTiCo** is available at the time; the program can show the values of global arrays (**Data**) in a graph.

Please note: You can add shortcuts to own files or folders to the bar (see [Adding file and folder shortcuts, page 48](#)).

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:



ADtools

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



TGraph-  
TiCo

displays contents of global arrays of a TiCo processor in a graph.

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

## 4 Programming Processes

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


By help of the [Using the Development Environment](#), the completed *TiCoBasic* program is compiled and transferred to the *TiCo* processor. There, the program will be executed as an independently running process.

### 4.1 Program Design

A *TiCoBasic* 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 the *TiCo* processor.

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

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

A program consists of up to 3 sections, which take on different tasks when executed on the *TiCo* processor. [fig. 12](#) outlines the ideal steps for an *TiCoBasic* program. 

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

Exception to standard program design is the [Process without trigger \(None\)](#), which has no defined sections at all. See more on [page 124](#).

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

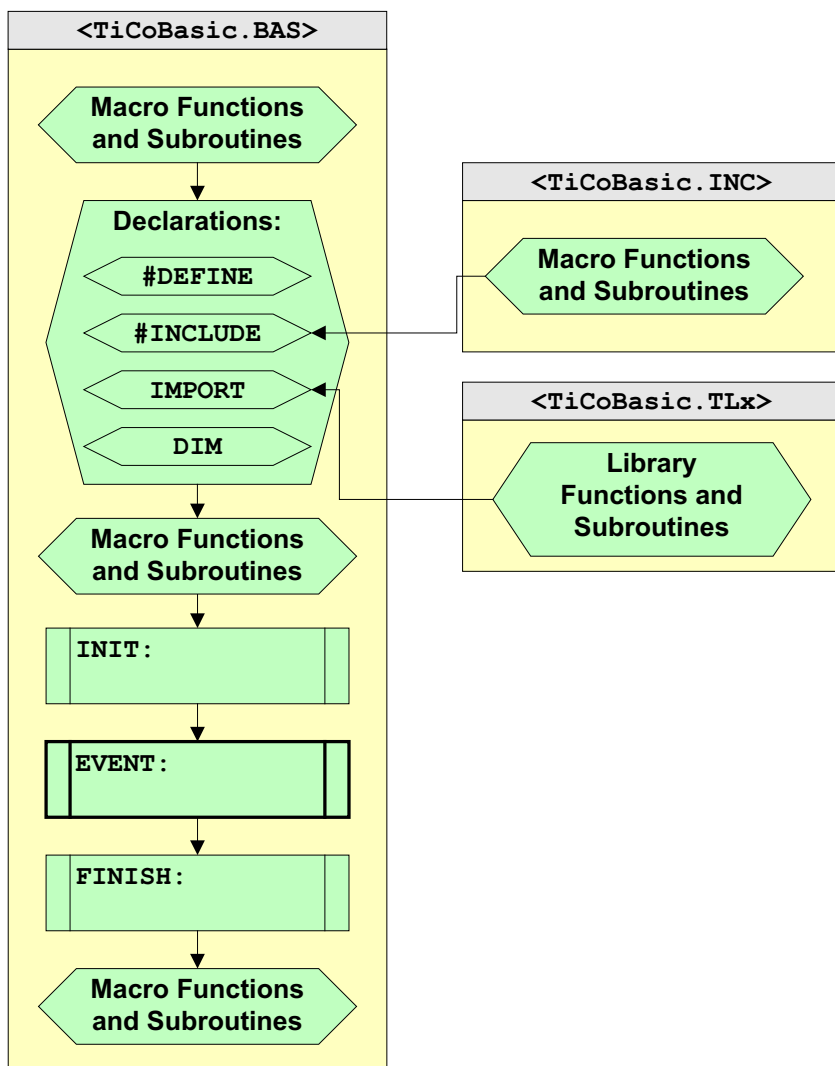


Fig. 12 – Design of an *TiCoBasic* program

### 4.1.1 The Program Sections

Each of the program sections (see [fig. 12](#)) start with a keyword, as described below. All sections have the priority which is set for the process ([Process Options dialog box, page 59](#)).

- **Init:** is the section being executed only once at the start of the process. It is used for initialization e.g. of variables or of data transfer.
- **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.

The **Init:** and **Finish:** sections are optional, while the **Event:** section is not and must be included in your program. Contrary to *ADbasic*, there is no section **LowInit:**.

### 4.1.2 User defined instructions and variables

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

The [Declarations Window](#) (see [page 87](#)) displays an overview of all defined instructions and variables.

All user-defined names (except symbolic names) must begin with a letter and may only consist of alphanumeric characters (a-z, A-Z, or 0-9) or an underscore ("\_")<sup>1</sup>. Special characters like German umlauts (Ä, Ö, Ü) are not allowed and there is no case sensitivity. The name length is only limited by the maximum line length (2048 characters).

#### Symbolic names

The instruction **#Define** defines symbolic names (see [page 147](#)). Group all of these definitions at the beginning of the file and before the start of the program sections.

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

---

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

## Arrays and Local Variables



In a *TiCoBasic* program, the local variables and all arrays must be declared with `Dim` before they can be used (see [page 149](#)). The global variables `Par_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 *TiCo* processor, in order to exchange data.

Global variables and arrays being used in a program can be displayed in the [Global Variables \(page 86\)](#).

## Macros

A macro function `Function ... EndFunction` or subroutine `Sub ... EndSub` call inserts the macro into the program text where it is being used (see also [chapter 4.5.1 on page 112](#)). However, the macro definition cannot be done within the program sections (see [fig. 12 on page 92](#)).

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

## Libraries

Libraries must be included before the program sections that use them. Library functions `Lib_Function ... Lib_EndFunction` and subroutines `Lib_Sub ... Lib_EndSub`, when used more than once within a program, require less memory than similar macro functions or subroutines described above (see also [chapter 4.5.3 on page 113](#)).

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

## 4.2 Variables and Arrays

### 4.2.1 Overview

Data structure	Name	Data type	Notes
Global variables and arrays			
Variable (Scalar)	<code>Par_1...Par_80</code>	<a href="#">Long</a>	
System variable	<code>Processdelay</code>	<a href="#">Long</a>	
	<code>Prozessn_Running</code>	<a href="#">Long</a>	Pre-defined,
One-dimen- sional array (vec- tor)	<code>Data_1 [] ...</code> <code>Data_16 []</code>	<a href="#">Long</a> , <a href="#">Ringbuffer</a>	not declarable Name <code>Data_n</code> not changeable, only dec- laration of array num- ber and dimension.
Local variables and arrays			
Variable (Scalar)	selectable	<a href="#">Long</a>	Must be declared.
One-dimensional array (vector)	selectable	<a href="#">Long</a>	Must be declared.

Variables and arrays are normally stored in the internal memory DM (memory map, see [chapter 4.3.1](#)), if not determined explicitly.

The data type [Long](#) has a length of 32-bit.

### 4.2.2 Data Structures

In *TiCoBasic*, there are two main types of data structures:

- Variables (scalars)

**VAR**

Each variable can store one value only.

- Arrays, one--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 [Ringbuffer](#) (a ring buffer, which works according to the principle: First in, first out, see [chapter 4.3.3 on page 103](#)).

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

The compiler differentiates

- **Global Variables (Parameters)** variables and **Global Arrays** (see [chapter 4.2.5](#) and [chapter 4.2.6](#)):

All *TiCo* processes as well as the *ADwin* CPU can access global variables, for instance to exchange data.

System variables are global variables (see [page 99](#)).

- **Local Variables and Arrays** (see [page 100](#)):

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

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

For easier programming, global variables **Par\_1** ... **Par\_80** are already predefined; thus, global variables do not have to (and cannot) be declared.

The compiler recognizes the declaration of global arrays by the names **Data\_n**, where "**Data\_**" is a fixed text and "**n**" is the array index number (1...16) specified.



After declaration, variables and array elements have an undefined value and thus should be initialized with a useful value (e.g. zero). Exception: With the transfer of a process to the *TiCo* processor all global variables **Par\_1** ... **Par\_80** are automatically initialized with zero.

### 4.2.3 Data Types

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

The compiler processes only data type **Long**; these are 32-bit integer values with the ranges:

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

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



## 4.2.4 Entering Numerical Values

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

1. Decimal notation: `x = 930`

2. Exponential notation: `x = 93E1`

Here `93E1` stands for  $93 \times 10^1$ , where "E" is followed by the exponent to the basis of 10 (max. 2 decimal places).

3. Binary notation: `x = 1110100010b`

4. Hexadecimal notation (an `h` is added): `x = 3A2h`

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

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

## 4.2.5 Global Variables (Parameters)

All running *TiCo* processes and the *ADwin* CPU can access global variables and arrays; therefore, they are ideal for data exchange between the processes or between the processes and the *ADwin* CPU (see also [chapter 6.3.1 "Data Exchange between Processes"](#)). 80 integer variables as well as up to 16 arrays of the `Long` data type are available. All variables and array elements have a length of 32-bit.

[System Variables](#), also globally available, are described on [page 99](#).

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: With the transfer of a process to the *TiCo* processor all global variables `Par_1` ... `Par_80` are automatically initialized with zero.

A list of global variables and arrays being used in the program can be displayed in the [Global Variables](#) (see [page 86](#)).

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.

**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, global variables **Par\_n** must not be declared because they are pre-defined and are already known to the compiler.

**4.2.6 Global Arrays**

Global arrays enable the exchange of data between the processes on the ADwin system or the ADwin CPU (see also [chapter 6.3.1 "Data Exchange between Processes"](#)). Up to 16 arrays of the **Long** data type are available.



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

Global arrays can be declared as standard arrays or as [Data structure Ring-buffer](#) ([page 103](#)).

A list of global arrays and variables being used in the program can be displayed in the [Global Variables](#) (see [page 86](#)).

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...16). The names for **DATA** arrays are:

**Data\_1, Data\_2, ... Data\_16.**

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**

```
REM Declare the array 5 with 20000 elements of the type Long.
Dim Data_5[20000] As Long
```

The maximum size of the array depends on the memory size. For instance on a *TiCo* processor with 256MiB memory, an array of up to 67 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



*Rem The value of the 200th element from array 5 is assigned  
Rem to the global integer variable Par\_1.*

```
Par_1 = Data_5[200]
```

*Rem In this program line, the 345th element from the array  
Rem Data\_5 gets the value 4000.*

```
Data_5[345] = 4000
```

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 *TiCoBasic* 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 is therefore not available. Instead, use the notation `Data_2`. Note the different syntax highlighting of the variables.

### 4.2.7 System Variables

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

#### **Process<sub>n</sub>\_Running**

Returns the status of the process `n` (with `n = 1...10`): the process is running, just being stopped or already stopped (see [page 192](#)). The variable can only be read.

#### **Process\_Error**

Returns the number of the previous error of process `n`, if debug mode is active (with `n = 1...4`, see [page 191](#)). The variable can only be read.

#### **Processdelay**

The nominal time interval, in which time-controlled processes are called by the counter, is the `processdelay` (cycle time). With the system variable `Processdelay` (see also [page 189](#)), you query and set this time, measured in clock cycles of the counter.

You read and write into the variable `Processdelay` in the sections **Init:** and **Event:** only. But writing into the variable is only allowed once per section, because otherwise.



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

#### 4.2.8 Local Variables and Arrays



All local variables and arrays, needed for a process must be declared before the start of the first section of the *TiCoBasic* 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 ("\_")<sup>1</sup>. Special characters like German umlauts (Ä, Ö, Ü) are not allowed and there is no case sensitivity. The length of variable names is only limited by the maximum line length (2048 characters).

Variables (scalars) can be defined as integer values (type `Long`), 32 bits long.



##### Example

```
Rem Define the variable 'value' with data type Long
Dim value As Long
```

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



##### Example

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



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.

---

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

## 4.3 Variables and Arrays – Details

### 4.3.1 Variables and Arrays in the Data Memory

The user can explicitly determine, in which memory area (internal or external) to store arrays and local variables (see below). This allocation is made, in the source code, when the variable is declared using the `Dim` statement using the additions `At DM_Local` or `At DRAM_Extern`.

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

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

The [fig. 13](#) 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)	<code>Dim var As Long</code> or <code>Dim var As Long At DM_Local</code>
	External (DX)	<code>Dim var As Long ... At DRAM_Extern</code>
Array (global/ local)	Internal (DM)	<code>Dim array[5] As Long At DM_Local</code>
	External (DX)	<code>Dim array[5] As Long</code> or <code>Dim array[5] As Long At DRAM_Extern</code>

Fig. 13 – Allocation of the Memory Area with Declarations

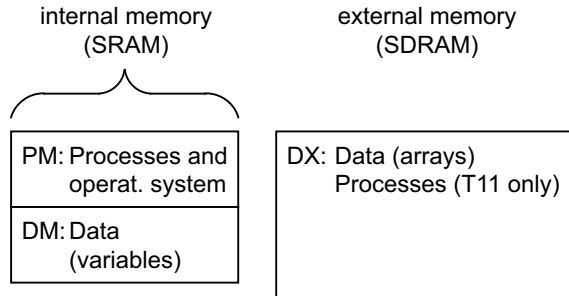
The global variables `Par_1...Par_80` are pre-defined in the internal memory (DM), therefore they cannot be re-declared in the external memory (DX).



### 4.3.2 Memory Areas

The *TiC*oprocessor uses a fast internal memory (SRAM) and—if existing— a huge external memory (SDRAM). Some Pro II modules contain a faster SRAM (SX) instead of an external SDRAM.

Half of internal memory is available as program memory PM and as data memory DM.



- Program memory (PM):  
Program memory occupies half of the internal SRAM and contains the operating system and processes.

- Internal data memory (DM)  
The internal data memory occupies half of the internal SRAM for storing the global and local variables and arrays.
- External data memory (DX, SX)  
The external data memory covers the external SDRAM.

Few Pro II modules (e.g. Pro II-MIO-TiCo) use SRAM instead of SDRAM as external data memory.

Accessing external memory is always combined with a varying waiting time (jitter); an exception is the access with [Data structure Ringbuffer](#) (see [page 103](#)). Please also note [chapter 5.2.6 on page 118](#).



Data in the internal memory (DM) can be accessed faster than data in the external memory (DX). Comparing external SRAM and internal memory, the access speed is nearly equal. The access speed to the memory areas PM, and DM in internal SRAM is equal.

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

### 4.3.3 Data structure Ringbuffer

In order to transfer great data amounts continuously and safely, it is recommended using a `Data_n` global array with the [Ringbuffer](#) data structure: a "First In, First Out" ring buffer.

The basic features of global arrays are described in [chapter 4.2.6 "Global Arrays"](#).

The data structure **FIFO** of the ADwin CPU is quite different from a [Ringbuffer](#). FIFO is described in the *ADbasic* manual.



Ring buffers are useful for several applications; however, the applications are mutually exclusive:

- The *TiCo* process accesses data in the external DRAM in both directions, reading and writing via the same ringbuffer array.
- *ADbasic* processes (on the *ADwin* CPU) and *TiCoBasic* processes exchange data via a ringbuffer. One ringbuffer is required for each direction of data exchange.
- Several processes on the *TiCo* processor exchange data with each other via ringbuffer. Two ringbuffers are required, one for reading and one for writing.



Using the data structure [Ringbuffer](#) is not an easy task. Wrongly implemented, there may be errors which can hardly be tracked. The use of the data structure [Ringbuffer](#) is therefore reserved to experienced users of *ADbasic* and *TiCoBasic*.

### How does a ringbuffer work?

In a ringbuffer, 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).

Since a [Ringbuffer](#) 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.



From the ring structure of the ringbuffer 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 ringbuffer than it is being read out. Subsequently, the earlier stored data will be overwritten and lost.

### Declare and use a ringbuffer

A ringbuffer is declared with [Dim](#):



### Example



```

Rem Read ringbuffer with 103 elements in external memory
Dim DATA_1[103] As Long As Ringbuffer_For_Read At DRAM_Extern
Rem Write ringbuffer with 1000 elements
Rem in internal memory
Dim DATA_2[1000] As Long As Ringbuffer_For_Write At DM_Local
Rem Read and write ringbuffer with 199 elements in
Rem external memory
Dim DATA_3[199] As Long As Ringbuffer_For_Read At DRAM_Extern
Dim DATA_3[199] As Long As Ringbuffer_For_Write At DRAM_Extern

```

Regarding ringbuffer sizes in external memory, please note [page 196](#).

If no memory area is declared, the compiler uses `DM_Local` as default. It is recommended to always specify the memory area on declaration.

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



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

### Example



```

Dim DATA_5[1000] As Long As Ringbuffer_For_Read At DM_Local
Dim DATA_5[1000] As Long As Ringbuffer_For_Write At DM_Local
DATA_5 = 95                                     'Writes the value 95 into the
                                                'DATA_5 ringbuffer array
PAR_7 = DATA_5                                'Reads a value from the ringbuffer and
                                                'stores it in the global variable
                                                'PAR_7

```

To ensure that the ringbuffer is not full, the `Ringbuffer_Empty` function should be used before writing into it. Similarly, the `Ringbuffer_Full` function should be used to check if there are values which have not yet been read, before reading from the ringbuffer.

Referring to the following rules, the external memory `SRAM_Extern` and the internal memory `DM_Local` are regarded as a single memory area. The `SRAM_Extern` replaces the external memory `DRAM_Extern` on certain Pro II modules.

General rules for declaration of ringbuffers:

- Only 2 ringbuffer declarations are acceptable for each memory area.
- In external memory `DRAM_Extern` only one ringbuffer each is allowed for reading and for writing.

In the memory area `DM_Local` + `SRAM_Extern`, combinations of read and write ringbuffers are possible. Thus, you can also use 2 read ringbuffers or 2 write ringbuffers.

- It is forbidden, to declare both, ringbuffers and normal arrays, in external memory `DRAM_Extern`.



### Example

```

Rem 2 ringbuffers in external memory
Rem Normal arrays are forbidden now!
Dim DATA_5[199] as long as Ringbuffer_For_Read at dram_extern
Dim DATA_5[199] as long as Ringbuffer_For_Write at dram_extern

Rem 2 ringbuffers in internal memory
Rem Normal arrays can be declared in addition
Dim DATA_1[200] as long as Ringbuffer_For_Read at dm_local
Dim DATA_2[200] as long as Ringbuffer_For_Read at dm_local
Dim DATA_3[200] as long at dm_local

```

### Accessing external memory (DRAM)

The normal access to global and local arrays in external memory is quite slow. In contrary, fast data exchange is possible with a ringbuffer. Here the *TiCo* process writes and reads data using a single ringbuffer.

### Example



```

Rem Write and read ringbuffer in external memory
Dim DATA_5[199] as long as ringbuffer_for_read at dram_extern
Dim DATA_5[199] as long as ringbuffer_for_write at dram_extern
Rem Normal arrays are forbidden now!
Dim free,used,value1 As Long

Init:
    Rem Initialize read ringbuffer DATA_5
    RingBuffer_Clear(5, PAR_5)

Event:
    Rem Are there elements free for writing?
    free = Ringbuffer_Empty(5,0)
    If (free > 0) Then
        DATA_5 = value1
    EndIf
    Rem Are there used elements to be read?
    used = Ringbuffer_Full(5,0)
    If (used > 0) Then
        PAR_7 = DATA_5
    EndIf

```

After declaration of a read ringbuffer in external memory, the ringbuffer should be initialized with **RingBuffer\_Clear**.

### Data exchange between *TiCo* processes

Two *TiCo* processes in a project (see also [chapter 6.3.1 on page 127](#)) can exchange data with each other via a ringbuffer continuously and fast. The ringbuffer can be declared in the (smaller) internal memory or in the (slower) external memory.

The data exchange works correctly only, if the data flow is unique, i.e. the one process writes into the ringbuffer and the other process reads from the ringbuffer. Even a change of flow is possible as long as the data flow remains unique.

Please note: The declaration of a ringbuffer is valid for the whole project and may therefore be written only in one of the source codes. Nevertheless, all processes of a project can access the declared ringbuffer.

**Example**

Process 1, which writes data:

```
Rem Write and read ringbuffer in internal memory
Dim DATA_5[500] As Long As Long As Ringbuffer_for_Read At DM_Local
Dim DATA_5[500] As Long As Long As Ringbuffer_for_Write At DM_Local
Dim free,value1 As Long
```

**Init:**

```
Rem Initialize read ringbuffer DATA_5
RingBuffer_Clear(5, PAR_5)
```

**Event:**

```
Rem Are there elements free for writing?
free = Ringbuffer_Empty(5,0)
If (free > 0) Then
    DATA_5 = value1
EndIf
```

Process 2, which reads data:

```
Rem No more ringbuffers may be declared here!
Dim used As Long
```

**Event:**

```
Rem Ringbuffer is used, although not declared in this
Rem source code:
Rem Are there used elements to be read?
used = Ringbuffer_Full(5,0)
If (used > 0) Then
    PAR_7 = DATA_5
EndIf
```

**Data exchange with ADbasic processes**

ADbasic processes (on the ADwin CPU) and TiCoBasic processes can exchange data with each other via a ringbuffer continuously and fast. The ringbuffer can be declared in the (smaller) internal memory or in the (slower) external memory.

For each direction of data flow a separate ringbuffer is required, which may be declared in TiCoBasic only. The data exchange works correctly only, if the data flow is unique, i.e. the one process writes into the ringbuffer and the other process reads from the ringbuffer. A change of flow is not possible.

The data exchange uses a global variable **Par\_n** of the TiCo processor for synchronization. In ADbasic, the ringbuffer instruction writes the current value of the write or read pointer—according to the direction of data flow—into the vari-

able. This ensures, that the process in *TiCoBasic* may query the number of data to read or to write.

### Example



*TiCoBasic* process, which writes data

```

Rem Write ringbuffer in internal memory
#Define free PAR_1
Dim DATA_1[500] As Long As Ringbuffer_for_Write at dm_local

Init:
    Rem Initialize write ringbuffer DATA_1
    RingBuffer_Clear(1, PAR_5)

Event:
    Rem Are there elements free in Data_1 for writing?
    Rem Par_5 is set in ADbasic and contains the number
    Rem of free elements.
    free = Ringbuffer_Empty(1, PAR_5)
    If (free > 0) Then
        DATA_1 = PAR_3
    EndIf

```

*ADbasic* process, which reads data (*ADwin-Gold II* here)

```

#include ADwinGoldII.inc
Dim DATA_10[300] As Long 'array for read data
REM define settings array for TiCo
Dim tset[150] As Long 'settings array for TiCo
Dim val As Long 'error code

Init:
    Rem Initialize data exchange to TiCo processor 1
    TDrv_Init(1, tset)

Event:
    Rem Read up to 250 values from TiCo array Data_1 and store
    Rem into Data_10. The number of free elements is written
    Rem into TiCo parameter Par_5.
    val = Get_TiCo_RingBuffer(tset, 1, DATA_10, 1, 250, 1, 5, 0)
    If (val > 0) Then
        Rem Data have been read, and are processed here
    EndIf

```

## 4.4 Expressions

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

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

### 4.4.1 Evaluation of Operators

For the evaluation of an expression, it is important to understand the order, in which the operators are used. The operators are divided into categories, which are resolved according to priorities: A category of higher priority is processed before a category of lower priority (see [fig. 14](#)).

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

Fig. 14 – Priorities of operator categories  
(highest priority on top)

### Example

```
var = Par_1 + Par_2 * Par_1^3 / 4
corresponds to
var = Par_1 + (Par_2 * (Par_1^3) / 4)
```



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

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



### Example

```
var = 1/-x           'not recommended
var = 1 / (-x)       'correct: negative inverse value
```



## 4.5 Selection structures, Loops and Modules

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

- Control structures to help shorten large sections.
  - Loops for sections being frequently repeated:  
`Do ... Until` or  
`For ... Next`.
  - Structures for case-by-case decisions:  
`If ... EndIf`, `#If ... #EndIf` or  
`SelectCase ... EndSelect`.
- **Subroutine and Function Macros** to define frequently used program sections as
  - Subroutine macros with `Sub ... EndSub`
  - Function macros with `Function ... EndFunction`
- **Libraries** of compiled subroutines and functions, which can be included into a user's source code with `Import`:
  - Library subroutines with `Lib_Sub ... Lib_EndSub`
  - Library functions with `Lib_Function ... Lib_EndFunction`
- Collections of source code sections and program modules in **Include-Files**, which can be included into a user's source code using  
`#Include filename.Inc`

More information and examples of instructions can be found in [chapter 7 "Instruction Reference"](#).

### 4.5.1 Subroutine and Function Macros

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

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

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

You will find more information about the structure of macro modules in the instruction reference ([page 157: `Function ... EndFunction`](#); [page 213: `Sub ... EndSub`](#)).

### 4.5.2 Include-Files

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

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

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

Depending on the include file's source, attention must be paid to the position, at which the file is included into another source code file, to maintain a working program structure. If the include-file contains function and subroutine macros, it must be included before the `Init:` section or after the `Finish:` section.

You can also include an include-file into source codes of library files and other include-files (nested include).



Include files installed with *TiCoBasic* 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 92](#)).



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

Math_TiCo.inc	
GoldIITiCo.inc	<i>ADwin-Gold II</i> ; insert text snippet with IG2[TAB].
AInTiCo.inc	<i>ADwin-Pro II</i> : Instructions for several module types.
AOut_TiCo.inc	
ArincTiCo.inc	
CAN_TiCo.inc	
Cnt_TiCo.inc	
DIO32TiCo.inc	
Fieldbus_TiCo.inc	
LSbus_TiCo.inc	
MIO_TiCo.inc	
MIO-D12_TiCo.inc	
RS_LIN_TiCo.inc	
SPI_TiCo.inc	

### 4.5.3 Libraries

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

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

If you compile a source code which imports a library, the binary file contains only library modules being called in the source code. Calling library modules multiple times does not increase the size of the binary file. Compared to macro functions and subroutines, library modules require less memory when they are called more than once. However, additional execution time is needed for calling them (compare to [chapter 4.5.1 "Subroutine and Function Macros"](#)).

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



When nesting libraries (including a library within another library), the source code calling the libraries must include all levels (see [fig. 15](#)), 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 172](#): Lib\_Function ... Lib\_EndFunction; [page 177](#): Lib\_Sub ... Lib\_EndSub).

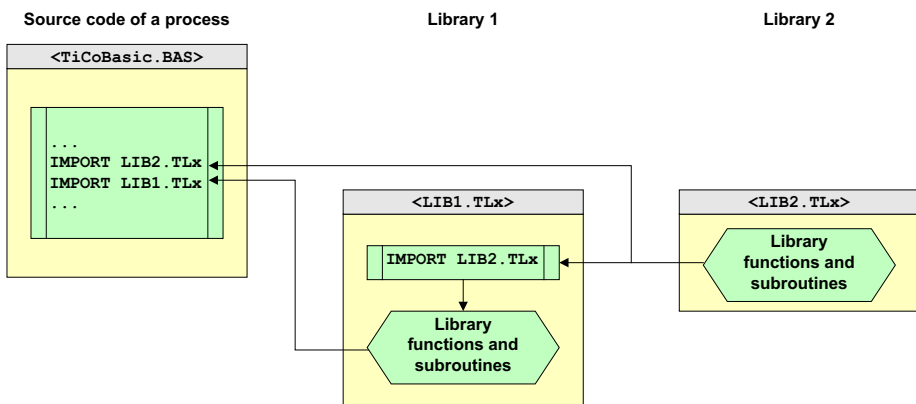


Fig. 15 – Interlaced Libraries

## Macros versus Libraries

What should you do, if you can use a macro as well as a library to solve a task? In this case, you have to weigh execution speed against program size.

A macro will be expanded in the code as often as it is being called in the source code. If your program calls a macro function 100 times, there will be 100 copies of the macro in the resulting program code. On the contrary, the code of a library function will always exist once only. Aiming for a smaller program size, the library function would be the better choice.

If a program calls a library function, there is some processing expense transferring the program execution to the library and later returning to the calling program code. If "calling" a macro, there is no such processing expense since the code is part of the program. Regarding higher execution speed, the macro does it better.

Weighing program size against execution speed is normally irrelevant for beginning programmers. A closer examination of this issue will only gain importance with huge or time-critical applications.

## 5 Optimizing Processes


The *TiCo* processor is designed to quickly and precisely execute control and measurement tasks. Depending on the requirements it may be necessary to optimize your *TiCoBasic* 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.

### 5.1 Measuring the Processing Time

For optimization, it is important to measure the processing time of a process cycle or of a program section. This can be done using the internal counter of the *TiCo* processor.

The *TiCo* processor has an internal counter which is incremented in clock rates of 10ns or 20ns. The current counter value can be read using the `Read_Timer` instruction.

After power-up, the counter is set to the value 0 (zero), then continually incremented in fixed clock pulses. 

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 t1, t2 As Long
```

#### Event:

```
Rem ...
t1 = Read_Timer()
Rem Time-critical section
Rem ...
t2 = Read_Timer()
Par_1 = t2 - t1 - 4          'Process time in clock pulses
                           '(offset = 4 clock pulses)
```

If `Par_1` in the example above equals 37, the time-critical section requires  $37 \times 20\text{ns} = 740\text{ns}$  (with *TiCo1* processor).

It is also possible to measure the time difference between two external events, in an event-driven process. In the following example, the measurement is stored in the global variable `Par_1`.



### Example

```
Dim oldtime, time As Long
```

#### Init:

```
oldtime = Read_Timer()
```

#### Event:

```
time = Read_Timer()
Par_1 = time - oldtime
oldtime = time
```

## 5.2 Useful Information

### 5.2.1 Accessing Hardware Addresses

Many of the *TiCo* processor functions are managed by its control and data registers. These functions can quickly be executed by *directly* accessing the relevant registers with the **InPeek** and **OutPoke** 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.

Please note our example programs under `C:\ADwin\TiCoBasic\`, where the word `Extern_` is part of the file name.

### 5.2.2 Constants instead of Variables

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



### Example

```
Par_1 = Par_2 * Par_2 'with Par_2=17
Par_1 = 17 * 17
```

For the first calculation, the value of the variable `Par_2` must be determined during run-time. The square must then be calculated and assigned to `Par_1`. In the second calculation, the compiler already has determined the value. During run-time it will only be assigned.

### 5.2.3 Faster Measurement Function

With the **ADC** instruction, an analog-to-digital (A/D) conversion for a channel with a specified gain is carried out. In order to make its application easier, the instruction is kept rather simple and combines several sequences (see hardware manual for the ADwin system).

There are different situations resulting in a faster processing when using these individual sequences, compared to using the **ADC** instruction.

For instance, the **ADC** instruction does not consider that the ADwin-Gold II-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 II
REM Set both multiplexers of the ADC to the channel 1
Set_Mux1(000b)
Set_Mux2(000b)
Rem wait for settling time
Rem ...
Start_Conv(11b)           'Start conversion on both ADCs
Wait_EOC(11b)             'Wait for end of conversion
Par_1 = Read_ADC(1)       'Read ADC1
Par_2 = Read_ADC(2)       'Read ADC2
```

### 5.2.4 Setting Waiting Times Exactly

Using a waiting time, you can easily set an exact offset between 2 instructions, for example to bridge a fixed processing time of a hardware component.

The instruction **Sleep** sets the waiting time exactly: The processor stops for the pre-set time, causing the next instruction to be started with appropriate delay.

### 5.2.5 Using Waiting Times

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

The **Set\_Mux1/2** 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 4 times the number of measurements.

The key feature of the example is to carry out the individual steps in the conversion process not sequentially but rather in parallel. The time delay for multiplexer setting is carried out during the A/D conversion of the other channels. Both measurement processes are overlapped: The start of conversion (time 2µs) for the channels 1+2 is followed by setting the multiplexer (time 2µs) for the channels 3+4.



### Example

*REM Example for Gold II*

**#Include** GoldIITiCo.inc

#### Init:

```
Set_Mux1(000b)      'Set Mux1 to channel 1 (and gain 1)
Set_Mux2(000b)      'Set Mux2 to channel 2 (and gain 1)
Sleep(20)            'Wait 2 µs
```

#### Event:

```
Start_Conv(11b)      'Start conversion, channels 1+2
Set_Mux1(001b)      'Set Mux1 to channel 3 (and gain 1)
Set_Mux2(001b)      'Set Mux2 to channel 4 (and gain 1)
Wait_EOC(11b)        'Wait for end of conversion (1+2)
Par_1 = Read_ADC(1)  'Read ADC1, channel 1
Par_2 = Read_ADC(2)  'Read ADC2, channel 2
```

```
Start_Conv(11b)      'Start conversion(channels 3+4)
Set_Mux1(000b)      'Set Mux1 to channel 1 (and gain 1)
Set_Mux2(000b)      'Set Mux2 to channel 2 (and gain 1)
Wait_EOC(11b)        'Wait for end of conversion (3+4)
Par_3 = Read_ADC(1)  'Read ADC1, channel 3
Par_4 = Read_ADC(2)  'Read ADC2, channel 4
```

The **Init:** section sets the multiplexer up for the first measurement so that the A/D is ready the first time the **Event:** section is executed.



It is very important that adequate delay for the multiplexer settling time and A/D conversions be provided, or incorrect measurements or A/D conversion failures may be obtained. There are some hints in [chapter 5.2.4 "Setting Wait-times Exactly"](#).

## 5.2.6 Optimization of memory access

The access to external memory is quite slow, especially with access to single memory address. In a process with low priority, an access to a single address

in external memory can even decrease the reaction time of a process with high priority.


In addition, the access to external memory of the *TiCo* processor is always combined with a variable waiting time ("jitter"). The reason is that the *TiCo* processor assumes an access to random addresses, and therefore organizes the access completely new—with appropriate waiting time.

The above disadvantages are avoided by the use of the data structure [Ringbuffer](#) for the access to data in external memory. Please note: The above disadvantages are avoided only after initialization and after the first access with a [Ringbuffer](#) structure.

Information about the use of the data structure [Ringbuffer](#) is described [chapter 4.3.3 on page 103](#).

## 5.3 Debugging

*TiCoBasic* provides the Debug mode as hands-on tool to find run-time errors. The debug mode is activated via the "Debug" menu (see [chapter 3.9.6, page 42](#)) and adds its helping features to those programs, which are compiled with active mode.

Please note: Activating of the debug mode produces additional program code. Thus, the program will need a longer processing time as well as additional memory—at times at considerable rate. We therefore recommend using the tool for developing and testing of programs only. 

### 5.3.1 Finding Run-time Errors (Debug Mode)

The debug mode is a helping tool to find the following run-time errors in *TiCo-Basic* programs:

- Access to too large / too small element numbers of an array

Without debug mode, these run-time errors are simply ignored, i.e. though the result of the program line is undefined it is nevertheless used for the following program. This may cause, depending on the program, an unwanted behavior, in worst case even the "crash" of the *ADwin* system.

The option "Debug mode" is activated from the "Debug" menu; do then compile the source code to be checked. On occurrence of a run-time error it is automatically displayed in the "Debug Errors" windows (see [Debug mode Option, page 71](#)). As well, the run-time error is being handled to maintain a stable mode of operation.



Errors being found must always be eliminated; the automatic error handling 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 71](#).



## 6 Processes in the ADwin System

An *ADwin* system has the capability to control complex test stands while rapidly executing measurements. Programs using an *TiCoBasic* process are used to provide this capability. Within this process you can specify how analog and digital data is processed within the *TiCo* processor and how it is exchanged with external devices, e.g. to support the *ADwin* CPU or to work independently.

After starting the process, the program<sup>1</sup> in the *TiCo* processor 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.
3. The process type **None** (without event trigger) is only required for special use—mostly programmed in assembler—and excludes any other process type. If not programmed differently, the process does not react to any event signals and is processed only once.

You define the exact function of a process in the *TiCoBasic* source code:

- The initialization in the section **Init:**.
- The actual function of the process cycle in the central **Event:** section (event loop).
- The final processing in the **Finish:** section.

It is possible to control the processes from the *ADwin* CPU that is the processes are started, stopped or their `processdelays` changed. From the PC you can control processes only from the development environment *TiCoBasic*. With the bootloader option, it is also possible to have processes start automatically on power-up of the *ADwin* hardware. For programming the bootloader, see [chapter 3.7.2 "Programming the TiCo bootloader"](#), [page 49](#).

---

1. More precisely: the program section **Event:**.

## 6.1 Process Management

There should be only a single process (with high priority) running on the *TiCo* processor. According to your task one of the following process types will fit:

- **Timer controlled process**

Besides the high priority timer controlled process, a low priority timer controlled process is possible. A low priority process cannot run on its own.

- **Externally controlled process**

The externally controlled process always has high priority.

- **Process without trigger (None)**

For the process without trigger the priority is of no importance.

It is possible to combine a timer controlled process and an externally controlled process. Please contact our support ([support@adwin.de](mailto:support@adwin.de)) for this task, so we can inform you about the required arrangements.

If you want to run more than one process at once, you have to add the source code files to a project (see [chapter 3.10.2 on page 75](#)).

### 6.1.1 Timer controlled process

With a timer controlled process, a process cycle is triggered regularly by a pulse of the internal counter. The time interval between two pulses called cycle time or processdelay and can be set in units of 10ns or 20ns (see also [chapter 4.2.7 on page 99](#)).

Besides the high priority timer controlled process, a low priority timer controlled process is possible. Set the process priority in the menu "Options \ Process Options".

The process with high priority is processed preferentially:

- The output of signals can be set in intervals of 10ns / 20ns without jitter.
- The maximum latency from the process call by an event signal until execution of the process begins is 120ns, or even 100ns with *TiCo2*.
- A high-priority process cycle cannot be interrupted and is always completely processed. During this time all process cycles with low-priority are blocked.

Even a stop instruction cannot interrupt a running, high-priority process cycle: the system will complete the current high priority process cycle before proceeding.

A low-priority process cycle will be interrupted at the time when a high-priority process cycle is started and as long until it has finished.

Have time-critical tasks run in a high-priority process and other tasks with low priority, so the processor can run time-critical cycles without trouble.



### 6.1.2 Externally controlled process

With an externally controlled process, a process cycle is triggered by an external signal.

The externally controlled process always has high priority:

- The output of signals can be set in intervals of 10 ns / 20 ns without jitter.
- The maximum latency from the process call by an event signal until execution of the process begins is 120 ns.
- A high-priority process cycle cannot be interrupted and is always completely processed.

Even a stop instruction cannot interrupt a running, high-priority process cycle: the system will complete the current high priority process cycle before proceeding.

The calling event signal is set very flexibly via a hardware address. The value in the hardware address and a bit mask are processed in a bitwise AND operation. The result is compared to a fixed value via an operator (<, >, =); if the comparison is true, an event signal is triggered. For the settings see [Process Options dialog box](#) on [page 59](#).

The hardware addresses are different for each ADwin hardware. Please note our example programs under C:\ADwin\TiCoBasic\, where the word `Extern_` is part of the file name.

Example: The above settings masks the value of address 70h with 12h, so only the bits 2 and 5 remain unchanged. If the result is > 0 (operation and value), that is one of the two bits is set, an event signal is triggered and thus

a process cycle started. Be it that the hardware address is a register for digital inputs, any high level pulse on one of the two digital inputs—related to the bits 2 and 5—will trigger a process cycle.

### 6.1.3 Process without trigger (None)

The process type `None` (without trigger) is only required for special use—mostly programmed in assembler—and excludes any other process type. The use is recommended for very experienced users only.

The process does not react to any event signals but starts running as soon as it is transferred to the *TiCo* processor or by the bootloader function. The program is processed only once, it does not repeat processing from the start. In order to run the program more than once, loops can be programmed.

The process type `None` has influence to the operating system. Thus, processes can neither be started or stopped externally, nor can the cycle time of processes be changed.



Please note special characteristics for programming:

- The program has no sections, the key words `Init:`, `Event:` and `Finish:` are therefore invalid and generate a compiler error message.
- The instruction `End` has no function.
- Insert an endless loop at the end of a program. Otherwise unforeseen problems may occur. An endless loop can look like this:

```
Do
Until (1 = 2)
```

The programming with assembler is described in a separate manual.

## 6.2 Time Characteristics of Processes

### 6.2.1 Processdelay

The time interval, in which time-controlled process cycles are called by the counter, which is the cycle time of the `Event:` section of the process. The cycle time is usually measured in clock cycles of the system clock and called *Processdelay*. The process delay of each process is specified by setting the value of the system variable `Processdelay` (see also [page 189](#)).

Also, the time interval between the end of the last instruction in the `Init:` section and the start of the `Event:` section is one *Processdelay* (plus a few clock cycles).

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

Processor	Resolution
TiCo1	20ns
TiCo2	10ns

Fig. 16 – The time resolution of the system clock (units of the processdelay)

For instance, a `processdelay` with the value 1250 means that a process on a TiCo1 processor is called in time intervals of  $1250 \times 20\text{ns} = 25\mu\text{s}$ . You can specify this event interval in the program line:

```
Processdelay = 1250
```

The processing time of a process cycle must not, even under worst case circumstances, be higher than the cycle time, so that each process cycle can be called at the time specified (with `Processdelay`). Differences in the computing time may arise from different program sections, which are run conditionally (If, Case).

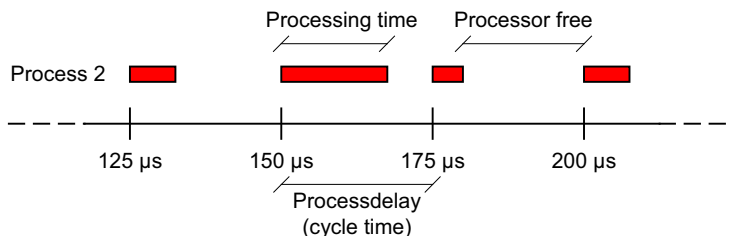


Fig. 17 – Processdelay and processing time

### Example



If an extensive calculation is executed only every, say 1,000 measurements, then the long processing time of this process cycle must be shorter than the cycle time. In order to obtain short process cycles, one alternative is to divide the calculations into small steps and to process a step in each process cycle. Thus, the process cycles have a consistent, short processing time.

## 6.2.2 Workload of the TiCo processor

The workload of the *TiCo* processor is the ratio of the computing time used to the available computing time, indicated in percent.

You can monitor the workload of the processor in the status line display `Busy` within the development environment (see [chapter 3.10.6](#)). This value gives

you an indication if the processor still has enough computing time available to complete all of the required activities.

The workload of the processor should exceed 90 percent only in exceptional cases and must not exceed 100 percent.

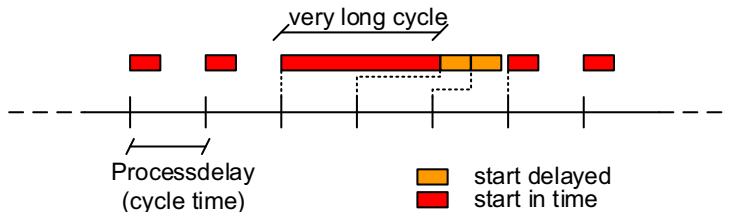
### 6.2.3 Different Operating Modes in the Operating System

The operating system handles the timing of a timer controlled process and an externally controlled process differently. In an externally controlled process single event signals can be lost, in a timer controlled process normally not.

#### Timer controlled process

In a timer controlled process each process cycle is normally called at the pre-defined time (via [Processdelay](#), page 124). Sometimes this timing misses, e.g. because the processing of a process cycle took longer than the set cycle time; in this case the timer event signal "accumulate".

The operating system is making up accumulated timer event signals, by calling process cycles without pause, until the originally set timing pattern is reached. This is also true for a low priority process as long as no high priority process is active.



If accumulated timer event signals have to wait more than  $2^{31}$  clock cycles—with TiCo1: 42.9 seconds, with TiCo2: 21.5 seconds—for being processed, these event signals cannot be made up any more. If a delay of such size appears, you have to check the general timing of the process: Probably the process cycle regularly takes longer than the cycle time. In this case you can enlarge the cycle time or shorten the processing time of the process cycle by suitable programming.

#### Externally Controlled Process

In an externally controlled process incoming event signals are processed very fast, but in special situations single event signals can be lost.

The operating system uses a hardware register to store external event signals. If an event signal has arrived, the operating system immediately starts a process cycle, except a high priority process is currently being processed. In this case the register serves as buffer for the event signal, and the operating system starts the next process cycle immediately after the currently processed cycle has finished.

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

## 6.3 Communication

### 6.3.1 Data Exchange between Processes

Data can be exchanged between different *TiCo* processes via global variables (**Par\_n**) or global arrays (**Data\_n**).

If global arrays are used in several processes, they have to be declared identically in each process. In this case, it is practical to save these declarations of global arrays into an include file and include the file into all of these processes (see also [chapter 4.5.2 "Include-Files"](#)).



This is different for [Data structure Ringbuffer](#) (see [chapter 4.3.3 on page 103](#)); in this case the declaration may only be done once in a project.

Global variables can be used by one process to control a process running simultaneously.

#### Example



Process 1 is a function generator and Process 2 is a controller. The function generator regularly writes the generated value into the global variable **Par\_10**. At every event loop the controller process reads out the global variable **Par\_10** and uses its contents as set point of the control loop.

Thus, the function generator very easily controls the set point of the controller. All *local* variables and arrays of Process 1 are hidden from Process 2 (and vice versa). Take into account that the timing characteristics of both processes must be considered.

### 6.3.2 Communication between PC and TiCo processor

From the PC there is no direct access to the *TiCo* processor, access is only possible from the *ADwin* CPU. The *ADwin* CPU can control processes of the *TiCo* processor as well as read from or write data to the *TiCo*. The *TiCo* processor itself does not communicate actively.

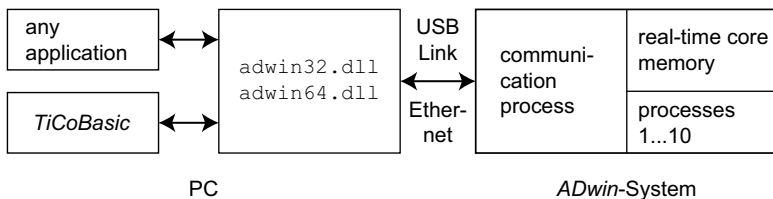
In order to exchange data between PC and *TiCo* processor, the *ADwin* CPU has to be configured as intermediate station. Here, a process on the *ADwin* CPU—which you have to set up on your own—exchanges data and control signals. The same principle is used for the data flow in the development environment *TiCoBasic*.

All data exchange is made via global variables (**Par\_n**, **FPar\_n**) or global arrays (**Data\_n**). This refers also to the [Data Exchange between Processes](#) (see above).

#### Communication between PC and ADwin CPU

The communication to the *ADwin* CPU is managed under Windows with the *ADwin32.dll* (dynamic-link library). In the *ADwin* CPU a communication process is responsible for this task.

If you are working with the ActiveX interface, the latter is responsible for the communication with the *ADwin* CPU. Internally the ActiveX interface transfers or gets the data via the *ADwin32.dll*.



The *ADwin32/64.dll* has the following tasks:

- Communication with the connected *ADwin* system 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.

### 6.3.3 Communication between ADwin CPU and TiCo Processor

The *ADwin* CPU can control processes of the *TiCo* processor as well as read from or write data to the *TiCo*. The *TiCo* processor itself does not communicate actively.

The *ADwin* CPU can access the *TiCo* processor with *ADbasic* instructions and perform the following actions:

- Initialize data access
- Read and write global variables Par\_1...Par\_80.
- Read and write global arrays Data\_1...Data\_16.
- Read and write ringbuffers and query status.
- Set and read processdelay of a *TiCo* process.
- Start and stop *TiCo* processes.
- Start, stop and reset processor.
- Query system information.
- Transfer a binary file.

You find a detailed description of the instructions here:

- *ADwin-Gold II*: [chapter 7.4 on page 219](#).
- *ADwin-Pro II*: [chapter 7.5 on page 263](#).

### 6.3.4 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 (Ethernet). This is the information the *ADwin32.dll* needs in order to being able to communicate with the system.



## 7 Instruction Reference

Below, the available *TiCoBasic* instructions for *TiCo* processors are listed. Find instructions for inputs/outputs in the hardware manual.

The instructions are listed in alphabetical order. In the annex, there are instruction overviews sorted by *ADwin* system and by alphabet.

In [chapter 7.4](#) and [chapter 7.5](#) the *TiCoBasic* instructions are listed which allows the *ADwin* CPU access the *TiCo* processor; the instructions are listed separately for *ADwin-Gold II* and *ADwin-Pro II*.

### 7.1 Instruction Syntax

Please note:

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

**CONST** constant numbers such as **35**, and expressions without variables.

**VAR** variable or array element.

**ARRAY** array, also identified in the command syntax by its brackets [ ] after the array name.

- The expected data type is given for each argument and for a function's return value:

**LONG** | integer number

**LOGIC** | logic expression in a condition

- Some instructions can only be used, when a specific library or include file is included. Under **Syntax**, the relevant include-instruction is indicated (place this command line at the beginning of the source code).

We assume that the necessary library or include file is located in the directory, which is set under the Options Settings menu, Directory item, (see also the instructions [#Include](#) or [Import](#)).

### 7.2 Basic Instructions *TiCoBasic*

The instructions in this section are valid for all *TiCo* processors.

## + Addition

The "+" operator adds two values.

### Syntax

```
ret_val = val_1 + val_2
```

### Parameters

val\_1            Addend 1.

LONG

val\_2            Addend 2.

LONG

### Notes

- / -

### See also

- Subtraction, \* Multiplication, / Division, ^ Power

### Example

```
Par_1 = 9 + 4                    'Par_1 = 13
```

### - Subtraction

The "-" operator subtracts one value from another.

#### Syntax

```
val = val_1 - val_2
```

#### Parameters

val\_1      Minuend.

LONG
------

val\_2      Subtrahend.

LONG
------

#### Notes

- / -

#### See also

+ Addition, \* Multiplication, / Division, ^ Power

#### Example

```
Par_1 = 9 - 4      'Par_1 = 5
```

## \* Multiplication

The "\*" operator multiplies two values.

### Syntax

```
val = val_1 * val_2
```

### Parameters

val\_1                      Multiplicator 1.

LONG

val\_2                      Multiplicator 2.

LONG

### Notes

- / -

### See also

+ Addition, - Subtraction, / Division, ^ Power

### Example

```
Par_1 = 9 * 4                      'Par_1 = 36
```

## / Division

The "/" operator divides one value by another.

### Syntax

```
val = val_1 / val_2
```

### Parameters

val\_1                      Dividend.

LONG
------

val\_2                      Divisor.

LONG
------

### Notes

Please note, that a division is executed without rest.

If the divisor is a variable with a negative sign, you should use braces to ensure you get the expected result (see also [chapter 4.4.1 „Evaluation of Operators“](#) on [page 110](#)).

### See also

[+ Addition](#), [- Subtraction](#), [\\* Multiplication](#), [^ Power](#), [Mod](#)

### Example

```
Par_1 = 36 / 4                      'Par_1 = 9
Par_2 = 2 / 4 * 5                   'Par_2 = 0 ->
                                    'integer calculation
Par_3 = 27 / (-Par_1)               'Par_3 = -3
Rem Please note the braces in the last line
```

## ^ Power

The "^" operator calculates the value of a number raised to a power.

### Syntax

```
val = val_1 ^ val_2
```

### Parameters

val\_1                      Basis.

LONG

val\_2                      Exponent.

LONG

### Notes



If the basis and/or the exponent are a variable with a negative sign, you should use braces to ensure the sign will be considered upon exponentiation (see also [chapter 4.4.1 „Evaluation of Operators“](#)). This is not necessary with constants.



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



Polynoms are calculated quicker, if you reduce powers by factoring out receiving a multiplication.

```
y = a + b*x + c*x^2 + d*x^3 + e*x^4 'slower version
y = a + x*(b + x*(c + x*(d + x*e))) 'quicker version
```

### See also

[+ Addition](#), [- Subtraction](#), [\\* Multiplication](#), [/ Division](#)

### Example

```
Par_1 = 9 ^ 4           'Par_1 = 6561
```



## #..., Compiler Statement

An *TiCoBasic* instruction beginning with the "#" sign instructs the compiler to treat the following source code differently before creating binary code.

The following compiler statements are available:

<b>#Define</b>	Definition of symbolic constants: Search and replace character strings in the source code with other character strings.
<b>#Include</b>	Include a file: Insert a file (with source code) into the source code.
<b>#If</b> ...	Conditional compilation: If the condition is true the corresponding code lines are compiled, otherwise deleted.
<b>#EndIf</b>	
<b>#Begin_ Debug_ Mode_ Disable</b>	
<b>#End_ Debug_ Mode_ Disable</b>	

## : Colon

The sign ":" separates more than one instruction within a single line.

### Syntax

```
[Step_1] : [Step_2] { : [Step_3] ... }
```

### Notes

[Step\_n] refers to any program instruction as is otherwise indicated in one individual program line.

We recommend using this instruction only when it makes the source code more clearly structured.

### 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

```
var = expr
```

### Parameters

`var` Variable or array.

**VAR**LONG

`expr` Expression.

LONG

### Notes

- / -

### Example

```
Dim val_1, val_2 As Long 'Declaration
```

```
Init:
```

```
    val_1 = 69 'Assigning a constant
```

```
Event:
```

```
    val_2 = val_1 * 2 'Assigning an expression
```

## < = > Comparison

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

### Syntax

```
If (val_1 > val_2) Then
```

### Parameters

val\_1            Operand.

LONG

val\_2            Operand.

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
```

## AbsI

**AbsI** provides the absolute value of a long variable.

### Syntax

```
ret_val = AbsI(value)
```

### Parameters

value	Argument: $-(2^{31}-1) \dots +2^{31}-1$ .	LONG
ret_val	Absolute value of the argument ( $0 \dots +2^{31}-1$ ).	LONG

### Notes

The smallest negative integer value  $-2^{31}$  has no positive counterpart in *TiCoBasic*; the absolute value of  $-2^{31}$  is therefore undefined.

### See also

[Mod](#)

```
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, 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](#)

```
Rem Bitwise operator of long variables
Dim val_1, val_2, val3 As Long
val_1 = 0100b      '= 4
val_2 = 0110b      '= 6
val3 = val_1 And val_2 'bitwise operator
Rem Result: val3 = 0100b = 4
```

Or:

*Rem Boolean operation of Boolean expressions*

```
Dim val_1, val4 As Long
```

```
val_1 = 314
```

*Rem Boolean operation: (true And true) = true*

```
If ((val_1 < 910) And (val_1 > 310)) Then
```

```
    val4 = 1
```

```
Else
```

```
    val4 = 0
```

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

## #Begin\_Debug\_Mode\_Disable

**#Begin\_Debug\_Mode\_Disable** disables the creation of debug code if debug mode is enabled.

### Syntax

```
#Begin_Debug_Mode_Disable
```

### Parameters

```
- / -
```

### Notes

**#Begin\_Debug\_Mode\_Disable** is a pre-processor statement and has only a function if the process is compiled with [Debug mode Option](#) enabled. The debug mode is used to recognize run-time errors, see [chapter 5.3.1 on page 119](#).

The debug mode interruption is cancelled with **#End\_Debug\_Mode\_Disable**.

Debug mode interruptions cannot be nested i.e. each **#End\_Debug\_Mode\_Disable** will cancel the interruption in any case. Pay attention especially to interruptions in macros and libraries.

### See also

[#End\\_Debug\\_Mode\\_Disable](#), [#Include](#)

#### Event:

```
#Begin_Debug_Mode_Disable
Rem uncritical source code being not controlled
Rem ...
#End_Debug_Mode_Disable
Rem critical source code, which is controlled
Rem ...
```



## Data\_n

The `Dim Data_n[...]` `As ...` instruction dimensions a global `Data` array.

More information about dimensioning see [page 149](#).

### Syntax

```
Dim Data_n[dim1] As Long {At <mem_type>}
```

### Parameters

<code>Data_n</code>	Name of the declared <code>Data</code> array ( <code>n</code> : 1...16).
<code>dim1, dim2</code>	Array size: Number ( $1 \dots 2^{31}$ ) of the array elements of the type <code>Arr_type</code> . <span style="float: right;"><b>CONST</b> LONG</span>
<code>&lt;mem_type&gt;</code>	Memory, where the array elements are stored: <code>DRAM_Extern</code> : external data memory. <code>SRAM_Extern</code> : external data memory in Pro II modules (instead of DRAM). <code>DM_Local</code> : internal data memory (default).

### Notes

You can access the array elements  $1 \dots \text{dim}$ . 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 *TiCo* processor.

### See also

[Dim](#), [Ringbuffer](#), „Global Arrays“ on [page 98](#), „Variables and Arrays in the Data Memory“ on [page 101](#)

```
Rem Declare the global array Data_15 with
Rem 1000 long elements
Dim Data_15[1000] As Long
```

## Declare Declare 1000Dec

**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

**Dec** (var) provides the same result as the program line: **val=val-1** and it may have shorter execution time.

### See also

[Inc, - Subtraction](#)

```
Dim index As Long
Dim Data_1[1000] As Long

Init:
index=1000

Event:
  DAC(1,Data_1[index]) 'Output the value on DAC1
  Dec(index)           'Decrement the index by 1
  If (index<1) Then
    index=1000         'Start again after 1000 outputs
  EndIf
```

## #Define

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

### Syntax

```
#Define name expression
```

### Parameters

<code>name</code>	Symbolic name, <i>without</i> quotation marks. Special chars are not allowed, only alphanumeric characters (a...z, A...Z, 0...9) and the underscore (_).	<b>CONST</b> <code>STRING</code>
<code>expression</code>	Expression for the symbolic name, <i>without</i> quotation marks. All characters are allowed.	<b>CONST</b> <code>STRING</code>

### Notes

Place this instruction at the beginning of a source code. The replacement is done in the following source code, starting from the next line.



The instruction **#Define** is a preprocessor instruction that means the replacement is made when you compile the source code (even before the compiler generates the program). Use **#Define** in order to use names that are more descriptive in the source code instead of constants, parameters or expressions.

The first string up to a blank is interpreted as symbolic name, the following text until the carriage return is interpreted as an expression to be inserted<sup>1</sup>. The expression is inserted exactly as you have defined it; variable names in the expression are not replaced by their value, but as a character string.

Neither `name` nor `expression` are case-sensitive.

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

---

1. Text behind a comment char " / " will be ignored by the compiler.

**See also**[#Include](#)

```
#Define setpoint Par_1 'Comments like this are ignored  
#Define measured Data_1  
#Define const 12441223
```

With these instructions, you can use the names `setpoint`, `measured` and `const` in the source code instead of `Par_1`, `Data_1` and the number.

```
#Define setpoint (13 + 4^3)  
Par_1 = 2 * setpoint    '= 2 * (13 + 4^3)
```

Without the parentheses in the `#Define` expression, you would get the value "90" instead of the expected "154".

## Dim

`Dim` declares one or more

- *local* variables
- *local* one-dimensional arrays
- *global* one-dimensional arrays `Data_n[n]` (also ringbuffer arrays)

Information about variables and data types can be found in [chapter 4.2.3](#), information about ringbuffer arrays under the heading [Ringbuffer](#) on [page 196](#).

### Syntax

```
Dim var1 {, var2, ...} As Long
Dim array1[dim1] As Long
    {At <mem_type>}
Dim Data_n[dim1] As Long
    {As Ringbuffer_For_Read / Ringbuffer_For_Write}
    {At <mem_type>}
```

### Parameters

<code>var1, var2</code>	Names of the declared variables.
<code>array1</code> , <code>Data_n</code>	Names of the declared arrays. For <code>Data_n</code> , you can select <code>n</code> from 1...16.
<code>dim1, dim2</code>	Array size: Number ( $1 \dots 2^{31}$ ) of the array elements of the type <code>Long</code> . <span style="float: right;"><b>CONST</b> LONG  </span>
<code>&lt;mem_type&gt;</code>	Memory where the variables are stored: <code>DRAM_Extern</code> : external memory. <code>SRAM_Extern</code> : external data memory in Pro II modules (instead of DRAM). <code>DM_Local</code> : local memory (default).

### Notes

The global variables `Par_n` must not be declared, because they are predefined.

If you want to access data from the *ADwin* CPU or from several processes, you can only do this by using *global* variables and arrays.

A fast data exchange is enabled using *RingBuffer*, either between *ADwin* CPU and *TiCo* processor or between *TiCo* processor and external memory.



Using the data structure *Ringbuffer* is not an easy task. Wrongly implemented, there may be errors which can hardly be tracked. The use of the data structure *Ringbuffer* is therefore reserved to experienced users of *ADbasic* and *TiCoBasic*.

Please note the hints in [chapter 4.3.3](#) on [page 103](#).

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 *TiCo* processor.

You find notes about the use of memory areas via the cross-references below.

## See also

[Data\\_n](#), [Event:](#), [Ringbuffer](#), [Finish:](#), [Init:](#), „[Variables and Arrays in the Data Memory](#)” on [page 101](#), „[Memory Areas](#)” on [page 101](#)

```
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 1007 Long elements as ringbuffer for read
Dim Data_20[1007] AS Long AS Ringbuffer_For_Read
```

## Do ... Until

**Do...Until** repeatedly executes a block of instructions until the Exit condition evaluates to "true". The block is executed at least one time.

### Syntax

```
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.

```
Dim count As Long
Dim Data_1[103] AS Long AS Ringbuffer_For_Write

Init:
    count = 1

Event:
    Do                               'Start loop
        Data_1 = ADC(1,4)           'Read measurement value
        Inc count                   'Increase count variable
    Until (count > 103)             '100 measurements done?
```

## End

**End** ends a process.

### Syntax

**End**

### Notes

**End** stops the processing of a section immediately. **End** is valid in all program sections.

If used in the **Event:** section, it starts processing the section **Finish:** (if existing). Any instructions in the **Event:** section following the **End** instruction are not processed.

### See also

[ProcessN\\_Running](#)

#### **Event:**

```
If (ADC(1) > 3000) Then 'Measure and compare
    Rem End Event: process, but execute Finish:
    End
EndIf
```

#### **Finish:**

```
Set_Digout(1)          'Set digital output 1
```



## #End\_Debug\_Mode\_Disable

**#End\_Debug\_Mode\_Disable** cancels the interruption of the debug mode.

### Syntax

```
#End_Debug_Mode_Disable
```

### Parameters

- / -

### Notes

**#End\_Debug\_Mode\_Disable** is a preprocessor instruction. It has only a function if the process was compiled with enabled [Debug mode Option](#) (see [page 71](#)) and the debug mode has been interrupted with **#Begin\_Debug\_Mode\_Disable**. The debug mode is used to recognize run-time errors, see [chapter 5.3.1 on page 119](#).

The debug mode can be interrupted with **#Begin\_Debug\_Mode\_Disable**.

You cannot nest debug mode interruptions. Pay attention especially to interruptions inside macros or libraries.

### See also

[#Begin\\_Debug\\_Mode\\_Disable](#), [#Include](#)

#### Event:

```
#Begin_Debug_Mode_Disable
```

```
Rem uncritical source code being not controlled
```

```
Rem ...
```

```
#End_Debug_Mode_Disable
```

```
Rem critical source code, which is controlled
```

```
Rem ...
```

## Event:

The keyword **Event:** marks the start of the main program section, which is called every Event signal.

### Syntax

**Event:**

### Parameters

- / -

### Notes

See also overview of program sections in [chapter 4.1.1](#) on [page 93](#).

The program section **Event:** is the central functional section, which in a process is called in (typically) regular intervals, until it is stopped. Depending on the settings the call is triggered by a cyclic timer Event signal or by an external Event signal. See more in [chapter 6 „Processes in the ADwin System“](#).

### See also

[Dim](#), [Init:](#), [Finish:](#)

```
Dim val_1 As Long
```

**Event:**

```
val_1 = -5
```

## Finish:

The key word **Finish:** marks the start of the finishing program section.

### Syntax

```
Finish:
```

### Parameters

```
- / -
```

### Notes

See also overview of program sections in [chapter 4.1.1](#) on [page 93](#).

The program section **Finish:** is run once as soon as the process is stopped.

After having processed the last instruction in the **Finish:** section, there will be a certain delay until the process status "stopped" is valid.

In contrary to *ADbasic*, the program section **Finish:** has the priority, which is selected for the process.

### See also

[Dim](#), [Init](#), [Event](#), [ProcessN\\_Running](#)

```
Dim val_1 As Long
```

```
Finish:
```

```
    val_1 = -5
```

## For ... To ... {Step ...} Next

The `For...Next` instruction creates a program loop, which executes a specified number of times.

### Syntax

```
For i = X To Y {Step Z}
    ...
Next i
```

*'instruction block'*

### Parameters

<code>i</code>	Count variable.	LONG
<code>X</code>	Start value of the run variable.	LONG
<code>Y</code>	End value of the run variable.	LONG
<code>Z</code>	Step length ( $\geq 1$ ) 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 *TiCo* 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`

- / -

## Function ... EndFunction

**Function...EndFunction** is used to define a function macro with passed and returned values.

### Syntax

```
Function macro_name ({val_1, val_2, ...})
    As Long

    {Dim var As Long}

    ...                'instruction block

    macro_name = ... 'assign return value
EndFunction
```

### Parameters

<b>macro_name</b>	Name of the function and of the return value, data type <b>Long</b> .
<b>val_1,</b> <b>val_2</b>	Names of passed parameters; for arrays use the syntax with dimension brackets: <b>array[]</b> or <b>Data_n[]</b> .

LONG |

### Notes

You will find general information about macros in [chapter 4.5.1 on page 112](#).

This instruction defines a function macro, which means that the whole instruction block between **Function** and **EndFunction** is inserted any place where the macro is called.

Functions help to make your source code more clearly structured. Please note that each function call will increase the size of the compiled file.

You may insert functions at the following 3 locations:

1. Before the section **Init:**
2. After the section **Finish:**
3. In a separate file, which you include with **#Include** (only in locations described in 1. and 2.).

Please note the following when defining functions:

- No process sections such as **Init:**, **Event:**, or **Finish:** can be defined.
- Local variables can be defined at the beginning, which are only available in the function and for the processing period. This is true even when a variable has the same name as a variable outside of the function.
- A value should be assigned to the function name, which will be the returned value for the function in the source code.

A function is called with its name and with the arguments, which you have defined; the function must be used as argument in the calling program line, e.g. in an assignment (see example). All expression types (including one- and two-dimensional arrays) are allowed as arguments, as long as they have the appropriate data type.

If you don't define arguments you nevertheless have to use the (empty) braces for the function's call: `name()`.

If an array is used as a passed parameter, the syntax is different for call and definition:

- call of function *without* dimension brackets:  
`ret_val=name(array_pass)`
- definition of function *with* dimension brackets:  
`Function name(array_def[]) ...`

Values are assigned to elements of passed arrays as usual:

```
array_def[2] = value
```



If a value is assigned to a passed parameter **x** within the function, the function's call must not use a constant **x**, but a variable or a single array element. If so, a passed parameter can be used to hold a return value.

Passed parameters in symbolic names (**#Define**) lead to compiler problems. Therefore, use definitions without passed parameters.

```
Rem not suitable: passed parameter 'array_def' in a Define
#Define pointer1 array_def[2]
Rem suitable alternative
#Define pointer2 2
Function name(array_def[])
  If (pointer1 > 0) Then 'line triggers compiler error
  If (array_def[pointer2] > 0) Then 'this line works fine
  ...
EndFunction
```

If a passed parameter is part of an expression inside a function, the parameter should be set in braces. This avoids problems with the order of operator evaluation.

### See also

**#Include, Sub ... EndSub**

```
Function sumsquare(w1, w2, w3) AS Long
Rem The function calculates the square of the sum of the
Rem values w1, w2 and w3
  Dim sum AS Long
  sum = w1 + w2 + w3
  sumsquare = sum * sum
EndFunction
```

Calling the function e.g. is done by the following program lines:

```
Event:
  x = sumsquare(x1, x2, x3)
  DAC(1, sumsquare(x1, x2, x3))
```

The same function with an array as passed parameter:

```
Function sumsquare_array(array[]) AS Long
  sum = array[1] + array[2] + array[3]
  sumsquare_array = sum * sum
EndFunction
```

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

```
Dim my_array[3] As Long
Dim Data_1[3] As Long
Dim x As Long
Event:
  x = sumsquare_array(my_array)
  DAC(1, sumsquare_array(Data_1))
```

For **array**, you can indicate a global array (as **Data\_1**) or a local array (as **my\_array**). Enter the array name only, without element number and brackets.



## If ... Then ... {Else ...} EndIf

The **If...Then** control structure is used to conditionally execute a single instruction (**If...Then...**) or a block of instructions (**If ... Then ... Else ... EndIf**).

### Syntax

```

If (condition) Then
    ...                               'Instruction block
{Else                               'the Else-block is optional
    ...                               '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: put the condition, which has most common state, in the **Else** statement, for instance when you check if limit values are exceeded.

In the single-line version, the instruction cannot call a subroutine macro (**Sub**) nor a function macro (**Function**).

**See also**

[< = > Comparison](#), [And](#), [Or](#), [Do ... Until](#), [SelectCase](#)

```
Dim val As Long           'Declaration

Event:
    val = ADC(1)           'Acquire measurement value

If (val > 3000) Then       'Limit value is exceeded:
    Clear_Digout(1)        'Reset DIGOUT 1
    Set_Digout(0)          'Set DIGOUT 0
Else                       'Limit value not exceeded
    Clear_Digout(0)        'Reset DIGOUT 0
    Set_Digout(1)          'Set DIGOUT 1
EndIf                     'End of control structure
```

## #If ... Then ... {#Else ... } #EndIf

This preprocessor structure is used to conditionally compile a block of instructions (**#If...Then...#Else...#EndIf**).

### Syntax

```
#If <SYSPAR> = value Then
...
                                'instruction block
{#Else
...
                                'the Else-block is optional
                                'instruction block}
#EndIf
```

### Parameters

**<SYSPAR>** = Boolean condition (no braces or quotation marks) **LOGIC** |  
**value** with the equal operator =.

If the condition is "true", the instructions after **Then** are executed.

The system parameter **<SYSPAR>** and the corresponding **value** are shown in the table below:

<b>&lt;SYSPAR&gt;</b>	<b>value</b>	<b>Meaning</b>
ADwin_ SYSTEM	ADWIN_GOLDII ADWIN_PROII	System setting in the window Compiler Options.
Processor	TICO1 TICO2	Processor setting in the window Compiler Options.

### Notes

The condition may only use the operator "="; neither Boolean conditions using **And** and **Or** nor bracing is allowed. You can nest **#If** structures repeatedly; only limited by the available memory.

There is no single-line version as with **If...Then**.

When calling the compiler via **Command Line Calling** (see [page A-9](#)) the system parameters refer to the command line options /Sx and /Px.

**See also**

[< = > Comparison, If ... Then ... {Else ...} EndIf](#)

```
Rem set Processdelay to 800µs
#If Processor = TiCol Then
    Rem 800µs = 40000 x 20ns
    Processdelay = 40000
#EndIf
```

## Import

**Import** includes functions and subroutines from the specified library file during compilation.

### Syntax

```
Import {path}file
```

### Parameters

file	File name of the library file <i>without</i> quotes. The file extension is .TL1 for TiCo1, .TL2 for TiCo2..	<b>CONST</b> <u>STRING</u>
path	Path name of the library file (with drive), without quotes.	<b>CONST</b> <u>STRING</u>

### Notes

General information about include files to be found in [chapter 4.5.3 on page 113](#).

Insert **Import** instructions at the beginning of your source code (before you declare the variables). If you import library files into the source code of a library A, you also have to import the other library files in the source code where library A is called.

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

If the path name misses, only the standard directory is searched (see [MakeLibFileOptions Menu](#), [Directories](#), [page 67](#)). Use the back slash "\" in the path name to separate directory names.

The base directory for relative paths is—if the source code is member of a project—the directory of the project file, otherwise the directory of the source code file.

At the moment, *TiCoBasic* is delivered without library files.

The following library files are delivered with *TiCoBasic*:

### See also

[#Include](#)

```
Rem import a user library for the TiCo1 processor  
Import C:\MyFiles\ADwinLibs\dig2volt.TL1
```

## In

**In** returns the content of a specified memory location of the I/O address range of a *TiCo* processor.

### Syntax

```
ret_val = In(addr)
```

### Parameters

**addr** Address of the memory location to be read out.

LONG

**ret\_val** Contents of the memory location.

LONG

### Notes

Normally, there is no need to use the instruction **In**. Use the instructions of the include files instead to control the *TiCo* processor. **In** is provided for special tasks only which are developed in combination with our support. The documentation will therefore not contain register addresses.

If a project contains an externally triggered process, **In** may only be used within a high-priority process.

### See also

[Out](#)

### Example

*Rem The example shows the use of the instructions In  
Rem and Out symbolically. Do not execute this program!*

#### Event:

```
If (In(100h) <> 0) Then 'read address 100h
    Out (0, 255)         'set address 0 to value 255
Else
    Out (0, 0)           'set address 0 to value 0
EndIf
```

## Inc

**Inc** increments the value of a local or global integer variable by one.

### Syntax

**Inc** (var)

### Parameters

**var**                      Name of a local or global Long-variable.

**VAR**

**CONST**

**LONG**

### Notes

**Inc** (val) is equivalent the program line: **val=**val+1 and it may have shorter execution time.

### See also

Dec, + Addition

```
Dim index As Long
Dim Data_1[1000] As Long
```

```
Init:
    index=1
```

```
Event:
    Rem Transfer measurement value into the array
    Data_1[index] = ADC(1)
    Inc(index)           'Increment index by 1
    Rem End program after 1000 measurements
    If (index>1000) Then End
```



## #Include

**#Include** includes all the contents of an include file into the source code.

### Syntax

```
#Include {path}filename
```

### Parameters

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

### Notes

You find general information about include files in [chapter 4.5.2 on page 112](#).

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](#).

If the path name misses, only the standard directory is searched (see [MakeLibFileOptions Menu Directories, page 67](#)). Use the back slash "`\`" in the path name to separate directory names.

The base directory for relative paths is—if the source code is member of a project—the directory of the project file, otherwise the directory of the source code file.

To include any of the include files delivered with *TiCoBasic*—the files contain instruction to access hardware I/Os—you enter the first characters of the instruction **#Include**, press [CTRL][SPACE] and select the required include file from the list.

The following include files are delivered with *TiCoBasic* and contain the instructions to access the hardware I/Os:

GoldIITiCo.inc	<i>ADwin-Gold II</i> : All instructions; insert text snippet with IG[TAB].
AInTiCo.inc	<i>ADwin-Pro II</i> : Instructions for several module types; insert text snippet with IP[TAB].
AOutTiCo.inc	
AOut1TiCo.inc	
ArincTiCo.inc	
CAN_TiCo.inc	
Cnt_TiCo.inc	
DIO32TiCo.inc	
MIO_TiCo.inc	
MIO-D12_TiCo.inc	
Fieldbus_TiCo.inc	
RS_LIN_TiCo.inc	
SPI_TiCo.inc	
Math_TiCo.inc	
	Special mathematics instructions.

**See also**

[#Define, Import, Function ... EndFunction, Sub ... EndSub](#)

```
Rem find file in the given directory
```

```
#Include C:\Test\demofunc.Inc
```

```
Rem find file in standard directory
```

```
#Include demofunc.Inc
```

```
Rem relative path.
```

```
Rem The base directory is relative to the directory of
```

```
Rem the project file (if the source file is member of
```

```
Rem a project).
```

```
Rem If the source code is not a project member, the base
```

```
Rem directory is the directory of the source file.
```

```
#Include .\demofunc.Inc
```

## Init:

The keyword **Init:** marks the start of the initializing program section.

### Syntax

```
Init:
```

### Parameters

```
- / -
```

### Notes

See also overview of program sections in [chapter 4.1.1](#) on [page 93](#).

The program section **Init:** is run once as soon as the process is started. The delay between having processed the last instruction of the **Init:** section and starting the **Event:** section is somewhat more than  $2 \times \text{Processdelay}$ .

The program section has the priority as set for the process (menu entry "Options / Process"). With high priority, the section cannot be interrupted and should then be as short as possible.

### See also

[Dim](#), [Event:](#), [Finish:](#), [Processdelay](#)

```
Dim val_1 As Long
```

```
Init:  
    val_1 = -5
```

## Lib\_Function ... Lib\_EndFunction

With `Lib_Function...Lib_EndFunction`, a function with passed and return parameters is defined in a library file.

### Syntax

```
Lib_Function lib_name(<lib_par1> {, <lib_par2>, ...} )
  As <fct_type>

  {Dim var As <var_type>}

  {#Define name expression}

  ...                               'Instruction block

  name = ...

Lib_EndFunction
```

Syntax of passed parameters `<lib_par>`:

```
<by_type> var_name As <var_type> {At <mem_type>}
```

**Parameters**

<code>lib_name</code>	Name of the library function and of the return value; data type <code>&lt;fct_type&gt;</code> .
<code>&lt;fct_type&gt;</code>	Data type: <code>Float32</code> , <code>Float64</code> , <code>Float</code> (up to T11), <code>Long</code> .
<code>var_name</code>	Name of a passed parameter inside of library function; for arrays use the syntax with dimension brackets: <code>array[]</code> or <code>Data_n[]</code> .
<code>&lt;by_type&gt;</code>	Methods for the transfer of parameters: <code>ByRef</code> : pass reference (pointer) to variable or array. <code>Byval</code> : pass value only.
<code>&lt;var_type&gt;</code>	Data type: <code>Float32</code> , <code>Float64</code> , <code>Float</code> (up to T11), <code>Long</code> , <code>String</code> .

**Notes**

You will find general information about library files in [chapter 4.5.3 on page 113](#).

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.  
Declare variables always at the beginning of the subroutine, but never outside.
- use global variables and arrays, which are passed as parameters.
- assign a value to the function name, which will be the value returned for the function in the source code.

In a library function, you *cannot*

- define process sections such as `Init:`, `Event:`, or `Finish:`.
- call a library function or subroutine from the same library file.  
If necessary, you have to put the function, which is to be called, into a new library file and Import it from there.
- use `SelectCase`.
- declare symbolic names using `#Define`.
- access any hardware like analog or digital inputs or outputs.

There are 2 differing methods for passing parameters:

- `ByRef`: The library function can change the parameter; the changed value is available in the program (the address of the parameter is transferred).
- `ByVal`: The library function can only access the value of the parameter, but cannot change it. Thus, the parameter remains the same for the program that calls the function.



Passed parameters should always be declared `At <mem_type>`, to save valuable processor time (<mem\_type> must fit with the declaration of the passed parameters in the calling program, see `Dim`). If not, the library function has to detect the parameter's memory type at run time.

If an array is passed as parameter, the syntax for definition and call differs:

- Definition of the library function's parameter *with* brackets:  
`Lib_Function 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 `Ringbuffer` arrays as passed parameters.

### See also

`Lib_Sub ... Lib_EndSub`, `Import`, `Function ... EndFunction`, `Sub ... EndSub`

```
Rem ----- Calculate a mean value -----
Rem save binary file as MEAN.TL1
Rem (extension according to processor)
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.tll
#Define cnt 10           'Number of the samples
#Define samples Data_1 'Number of measm. values
#Define filtered Data_2 'Number of filtered measm. values
#Define length 1000     'Length of the array
Dim samples[length] As Long 'Source array
Dim filtered[length] As Long 'Destination array
Dim i As Long           'Count variable

Init:
    i = 1                'Initialize counter
    Processdelay = 40000 'Measurement with 1 kHz

Event:
    samples[i] = ADC(1)   'Measure and save analog values
    Inc i                'Increment counter
    If (i > length) Then End '1000 measurements completed?
                                'If yes: process Finish

Finish:
    For i = 1 To (length - cnt) 'For all measm. values
        Rem Call library function "average"
        filtered[i + cnt] = average(samples,i,cnt)
        Rem Note the call with the passed array 'samples'
        Rem *without* dimension brackets
    Next i

```



## Lib\_Sub ... Lib\_EndSub

The `Lib_Sub...Lib_EndSub` is used to define a subroutine with passed parameters in a library file.

### Syntax

```
Lib_Sub lib_name(<lib_par1> {, <lib_par2>, ...})
    {Dim var as <var_type>}
    {#Define name expression}
    ...
    'Instruction block
Lib_EndSub
```

Syntax of passed parameters `<lib_par>`:

```
<by_type> var_name As <var_type>
```

### Parameters

<code>lib_name</code>	Name of the library subroutine.
<code>var_name</code>	Name of a passed parameter inside of library Sub; for arrays use the syntax with dimension brackets: <code>array[]</code> or <code>Data_n[]</code> .
<code>&lt;by_type&gt;</code>	Methods for the transfer of parameters: <code>ByRef</code> : pass reference (pointer) to variable and array. <code>Byval</code> : pass value only.
<code>&lt;var_type&gt;</code>	Data types: <code>Float</code> , <code>Long</code> , <code>String</code> .

### Notes

You will find general information about library files in [chapter 4.5.3 on page 113](#).

Generate library subroutines (and library functions) in a separate source code file. The compilation with "Build/Make lib file" creates the library file. With `Import`, those library modules are included into a process, which are being called in the process.

In a library subroutine, you can

- declare and use local variables and arrays.  
Declare variables always at the beginning of the subroutine, but never outside.
- use global variables and arrays, which are passed as parameters.

In a library subroutine, you *cannot*

- define process sections such as `Init:`, `Event:`, or `Finish:`.
- call a library function or subroutine from the same library file.  
If necessary, you have to put the function, which is to be called, into a new library file and Import it from there.
- use `SelectCase`.
- declare symbolic names using `#Define`.
- access any hardware like analog or digital inputs or outputs.

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

- `ByRef`: The library function can change the parameter; the changed value is available in the program (the method transfers the address of the parameter).
- `ByVal`: The library function can only access the value of the parameter, but cannot change it. Thus, the parameter remains the same for the program that calls the function.

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 `Ringbuffer` arrays as passed parameters.

### See also

`Lib_Function ... Lib_EndFunction`, `Import`, `Function ... EndFunction`,  
`Sub ... EndSub`

:

```
Rem save binary file as DIG2VOLT.TL1
Rem (extension according to processor)
Rem Measurement value conversion from Digits(0...65535)
Rem to Volt(±10V)
Lib_Sub dig2volt(ByRef digit[] As Long,
    ByVal ptr As Long, ByVal cnt As Long,
    ByVal gain As Long, ByRef volt[] As Float)
    Dim i As Long
    For i = ptr To (ptr + cnt)
        volt[i] = ((digit[i] * 20 / 65536) - 10) / gain
    Next i
Lib_EndSub
```

**Library call**

Calling the library function `dig2volt` is illustrated in the following example, a conversion of measurement values:

```

Rem The library 'DIG2VOLT' is imported
Import C:\MyFiles\ADwinLibs\DIG2VOLT.tl1

#Define cnt 1000           'Number of the samples
#Define ptr 1              'Start point of samples
                           'which are to be converted
#Define gain 1             'Gain of the PGA
#Define samples Data_1     'Memory for meas. values
#Define scaled Data_2      'Memory for converted meas. values
#Define length 1000        'array length

Dim samples[length] As Long 'Source array
Dim scaled[length] As Long  'Destination array
Dim i As Long               'Counter

Init:
  i = 1                     'Initialize counter
  Processdelay = 40000      'Measurement with 1 kHz

Event:
samples[i] = ADC(1)         'Measure and save analog values
  Inc i                     'Increment counter
  If (i > length) Then End  '1000 measurements done?
                           'If yes: process Finish

Finish:
  Rem Convert measurement values by
  Rem calling the library subroutine 'dig2volt'
  dig2volt(samples,ptr,cnt,gain,scaled)
  Rem Note the call with the passed array 'samples'
  Rem *without* dimension brackets

```

## Max\_Long

**Max\_Long** returns the greater of 2 integer values.

### Syntax

```
ret_val = Max_Long(val1, val2)
```

### Parameters

**val\_1**            Compared value 1

LONG
------

**val\_2**            Compared value 2

LONG
------

**ret\_val**          The greater of both values.

LONG
------

### Notes

- / -

### See also

[Absl](#), [Min\\_Long](#)

#### Event:

```
Par_10 = Max_Long(Par_1, Par_2)
```

## Min\_Long

**Min\_Long** returns the smaller of 2 integer values.

### Syntax

```
ret_val = Min_Long(val1, val2)
```

### Parameters

val\_1                      Compared value 1

LONG
------

val\_2                      Compared value 2

LONG
------

ret\_val                    The smaller of both values.

LONG
------

### Notes

- / -

### See also

[Absl](#), [Max\\_Long](#)

### Event:

```
Par_10 = Min_Long(Par_1, Par_2)
```

## NOP

[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:

TiCo1	20ns
TiCo2	10ns

With this instruction, you can delay for a necessary waiting period (e.g. after [Set\\_Mux1](#)) if there is no other use of processing time. With T11, please note [chapter 5.2.4 on page 117: Setting Waiting Times Exactly](#).

### See also

[NOPs](#), [Sleep](#)

## NOPs

**NOPS** causes the processor to wait for several processor cycles. The waiting time refers to a multiple of **NOP** instructions (assembler: No Operation).

### Syntax

**NOPS** (*number*)

### Parameters

*number*      Number ( $\geq 1$ ) of NOP instructions to insert. With constants, 32767 is the maximum value.

LONG


### Notes

Use **NOPS** in high-priority processes only. For low-priority processes, **Sleep** is the better alternative, especially for a high *number* value which could else cause unexpectedly long delays.

The use of **NOPS** takes less program memory than the appropriate number of **NOP** instructions.

If *number* is a constant, **NOPS** will replace the appropriate number of **NOP** instructions exactly. The use of a variable takes 2...4 clock cycles in addition, as does the access to external memory or additional calculations.

Under special circumstances, the compiler will add an additional **NOP** instruction behind **NOPS**. If an exact waiting time is required, decrement *number* by 1 and insert the **NOP** instruction (behind **NOPS**) manually into the program.

If the execution of **NOPS** takes very long, the variable *number*  may have a negative value. Use a positive constant value instead.

### See also

**NOP**, **Sleep**

### Example

- / -





## 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           '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. For bit wise combining of integer values, parentheses are not required.

### See also

**And, If ... Then ... {Else ...} EndIf, Not, XOr**

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 < 3) Or (x > 9)) Then
    val4 = 1
Else
    val4 = 0
EndIf
```

'Result: val4 = 1

## Out

**Out** writes a value into a specified memory location into the I/O memory range of the *TiCo* processor.

### Syntax

```
Out (addr, value)
```

### Parameters

<b>addr</b>	Address of the memory location to be written.	LONG
<b>value</b>	Value to be written.	LONG

### Notes

Normally, there is no need to use the instruction **Out**. Use the instructions of the include files instead to control the *TiCo* processor. **Out** is provided for special tasks only which are developed in combination with our support. The documentation will therefore not contain register addresses.

### See also

[In](#)

### Example

*Rem The example shows the use of the instructions In  
Rem and Out symbolically. Do not execute this program!*

#### Event:

```
If (In(100h) <> 0) Then 'read address 100h
Out (1, 12)             'set address 1 to value 12
EndIf
```

## Processdelay

The system variable **Processdelay** defines the process delay (cycle time) of a process.

**Processdelay** replaces the system variable **Globaldelay**, which is still valid for reasons of compatibility.

### Syntax

```
ret_val = Processdelay
```

or

```
Processdelay = expr
```

### Parameters

ret_val	Current cycle time in clock cycles.	LONG
expr	Cycle time to be set: Number ( $\geq 1$ ) of clock cycles.	LONG

### Notes

In a time-controlled process, the section **Event:** is called repeatedly and in fixed time intervals by the internal counter. The time interval between two cyclic calls is called process delay and is counted in clock cycles.

The time interval of **Processdelay** is 20ns with TiCo1, 10ns with TiCo2.

With high-priority processes, select a sufficiently large process delay to avoid overloading the *TiCo* processor. As a rule of thumb the processor workload (display field: "Busy x%" in the status bar) should be under 90 percent and must not exceed 100 percent.

If the time needed for processing the section **Event:** is larger than the process delay, the next counter call and following will be delayed.

You may set a constant process delay by assigning a value to the variable **Processdelay** in the section **Init:**. You will then overwrite the default value you have set in the dialog window "Options / Process" under "Initial Processdelay".

You can set the variable only once in a section.

If the parameter **Processdelay** is changed in a process cycle in the section **Event:**, the cycle time (processdelay) will be changed imme-

diately. This may be critical especially when the cycle time has been shortened: Make sure that the execution time of the program remains less than the newly set cycle time.

### See also

[Read\\_Timer](#)

#### Init:

```
Rem Set cycle time to 800µs  
Processdelay = 40000
```

If you need a longer cycle time than may be set with **Processdelay**, you can use an auxiliary variable:

#### Init:

```
Rem Set max. cycle time of about 42.9 s with TiCo1  
Processdelay = 2147483647  
Rem initialize auxiliary variable  
Par_1 = 0
```

#### Event:

```
Inc Par_1  
Rem use 100fold cycle time  
IF (Par_1 = 100) Then  
  Par_1 = 0  
  Rem run program  
EndIf
```

## Process\_Error

[Process\\_Error](#) returns the previously occurred error of the current process.

### Syntax

```
ret_val = Process_Error
```

### Parameters

<code>ret_val</code>	Number of the previously occurred error in the process. Some error numbers: 0: no error 10: Accessing a too high element number of a global array. 11: Accessing a too small element number ( $\leq 0$ ) of a global array. 12: Accessing a too high element number of a local array. 13: Accessing a too small element number ( $\leq 0$ ) of a local array.	<code>LONG</code>
----------------------	--	-------------------

### Notes

The return value is defined only if debug mode is enabled (see [Debug mode Option, page 71](#)). The variable is read-only.

### See also

- / -

## ProcessN\_Running

The system variable `Processn_Running` returns the current status of the specified process.

### Syntax

```
ret_val = Processn_Running
```

### Parameters

`n`                      Number of the requested process (0...4).

**CONST**

LONG

`ret_val`                Process status:  
                           1 Process is running.  
                           0 Process is stopped.  
                          -1 Process is being stopped.

LONG

### Notes

The system variable is read only.

### See also

[End](#)

### Event:

*Rem Get the status of process 2*

`Par_2 = Process2_Running`



## Read\_Timer

**Read\_Timer** returns the current counter value of the timer.

### Syntax

```
ret_val = Read_Timer()
```

### Parameters

**ret\_val**      Current counter value in units of 20 ns.

LONG
------

### Notes

The counter value cannot be written.

The length of a clock cycle refers to the processor type:

processor	clock cycle
TiCo1	20ns
TiCo2	10ns

You may determine a time interval from the difference of 2 timer values. Please note that any read timer value will be reached again after a certain time interval, which depends on the units of time given above:

processor	duration
TiCo1	42.9s
TiCo2	21.5s

### See also

[Processdelay](#)

```
Dim timervalue As Long
```

#### Event:

```
timervalue = Read_Timer()
```

See related example `seconds_timer_TiCo.bas` in folder

`C:\ADwin\TiCoBasic\samples_ADwin.`

**See also**

[Processdelay](#), [Read\\_Timer](#)

```
Dim timervalue As Long
```

**Event:**

```
timervalue = Read_Timer_Sync()
```

## 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

<i>comment</i>	Any character strings.
<i>instr</i>	TiCoBasic 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.

You can set the horizontal position of a comment after an instruction; find more under [Positioning comments \(page 25\)](#).

```
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
```

## Ringbuffer

Using `Dim Data_n As Ringbuffer_For_x`, a global `Data` array is dimensioned as ringbuffer for write or for read.

### Syntax

```
Dim Data_n[length] As Long As Ringbuffer_For_Read
    {At <Mem_Type>}

Dim Data_n[length] As Long As Ringbuffer_For_Write
    {At <Mem_Type>}
```

### Parameters

<code>Data_n</code>	Name of the declared <code>Data</code> array (n: 1...16).
<code>length</code>	Array size: Number ( $\geq 1$ ) of the array elements of the type <code>Long</code> . <span style="float: right;"><b>CONST</b> <code>LONG</code>  </span>
<code>&lt;mem_type&gt;</code>	memory, where the array elements are stored: <code>DRAM_Extern</code> : external data memory. Here the value range for <code>length</code> is to be set in steps of 8: $length = 8 \times a + 7; a \geq 0$ <code>SRAM_Extern</code> : external data memory in Pro II modules (instead of DRAM). <code>DM_LOCAL</code> : internal data memory (default).

### Notes



Using the data structure `Ringbuffer` is not an easy task. Wrongly implemented, there may be errors which can hardly be tracked. The use of the data structure `Ringbuffer` is therefore reserved to experienced users of *ADbasic* and *TiCoBasic*. Please note the hints in [chapter 4.3.3 on page 103](#).

Only `Data` arrays may be used as ringbuffers. If so, a ringbuffer arrays may not be used as "normal" array.

After dimensioning, a ringbuffer should be initialized with `Ringbuffer_Clear` in the `Init`: section.

For a ringbuffer array in external memory please note:

- If an invalid array size is set with `length`, the ringbuffer array is automatically dimensioned with the next higher valid array size. For example the compiler changes an array size `[1000]` automatically into `[1007]`.
- In a read ringbuffer, the data should be updated after dimensioning with the instruction `Refresh_RingBuffer` in the `Init:` section.

If you write data into a ringbuffer array faster than you read it, previously stored data will be overwritten and are lost. To avoid this you can use the instructions `Ringbuffer_Empty` and `Ringbuffer_Full` to determine the amount of space in the array.

### See also

[Dim](#), [Data\\_n](#), [Refresh\\_RingBuffer](#), [Ringbuffer\\_Empty](#), [RingBuffer\\_Full](#), „Global Arrays“ on page 98, „Data structure Ringbuffer“ on page 103

### Example

```
Rem Dimension the global array Data_15 with
Rem 1007 Long elements as read ringbuffer
Dim Data_15[1007] As Long As Ringbuffer_For_Read
```

## Refresh\_RingBuffer

**Refresh\_RingBuffer** updates the data in a read ringbuffer in the external memory.

### Syntax

```
Refresh_RingBuffer (data_no)
```

### Parameters

<code>data_no</code>	Number (1...16) of the ringbuffer array <code>Data_x</code> .	<code>LONG</code>
----------------------	---	-------------------

### Notes

For a ringbuffer in internal memory or a write ringbuffer in external memory no update with **Refresh\_RingBuffer** is required.

For a read ringbuffer in external memory data update (before reading) is required in following cases:

- The read ringbuffer contains data and a value will be read for the first time.
- After previously reading values, the ringbuffer contained less than 8 values and afterwards new data has been written into the ringbuffer.

In other words: You can avoid a regular update with **Refresh\_RingBuffer**, if after each reading step the ringbuffer holds 8 values or more.

The update will not do harm in any case, i.e. values cannot be read double and cannot be lost. In case of doubt it is better to update once too much with **Refresh\_RingBuffer** than missing an update.

### See also

[Ringbuffer](#) (declaration), [RingBuffer\\_Clear](#), [Ringbuffer\\_Empty](#), [RingBuffer\\_Full](#), „Data structure Ringbuffer“ on page 103

**Example**

```
Rem Use global array Data_12 as ringbuffer
Dim Data_12[999] As Long As Ringbuffer_For_Read At DRAM_Extern
```

**Init:**

```
Rem initialize ringbuffer
RingBuffer_Clear(12, Par_1)
Rem wait until ringbuffer contains more than 7 values
Do
Until (RingBuffer_Full(12, Par_1) > 7)
Rem refresh ringbuffer data
Refresh_RingBuffer(12)
```

**Event:**

```
Rem read 500 ringbuffer values, but always have more
Rem than 7 values left in it
If (RingBuffer_Full(12, Par_1) > 507)
For i = 1 To 500
Par_10 = Data_12
DAC(2, Par_10) 'do something with Par_10
Next i
EndIf
```

**Finish:**

```
For i = 1 To RingBuffer_Full(12, Par_1)
Par_10 = Data_12
Next i
```

## RingBuffer\_Clear

**RingBuffer\_Clear** initializes the write or read pointer of a ringbuffer.

### Syntax

```
RingBuffer_Clear (data_no, Par_x)
```

### Parameters

<b>data_no</b>	Number (1...16) of the ringbuffer array <b>Data_x</b> .	LONG
<b>Par_x</b>	For data exchange <i>ADwin</i> CPU / <i>TiCo</i> only: Global variable ( <b>Par_1</b> ... <b>Par_80</b> ), which contains a copy of the write or read pointer to the ringbuffer.	LONG

### Notes

The initialization of a ringbuffer is useful in 2 cases:

- Before first access to the ringbuffer.  
You should initialize in the **Init:** section in any case, since the ringbuffer pointers are not initialized during dimensioning.
- While the program is active, if you want to discard all data contained in the ringbuffer (e.g. because of a measuring error).

Pointer initialization will not change the values in the ringbuffer.

The global variable **Par\_x** is only required for data exchange between *ADwin* CPU (e.g. T11) and *TiCo* processor. The variable will then contain a copy of the read or write pointer to the ringbuffer.

If for example you write data into the ringbuffer with **Set\_TiCo\_RingBuffer** (in *ADbasic*), the *ADwin* CPU updates the write pointer with every write access and copies the value into the global variable **Par\_x** on the *TiCo* processor. Using the value of **Par\_x**, **RingBuffer\_Full** can calculate the number of used elements in the ringbuffer in *TiCoBasic*.



For a data exchange between *ADwin* CPU and *TiCo* processor, you have to follow these steps to initialize the ringbuffer:

1. Initialize the ringbuffer in *TiCoBasic* with **Ringbuffer\_Clear**.
2. Make sure, that neither in *ADbasic* nor in *TiCoBasic* data is written into the ringbuffer or read from it.
3. Re-initialize the data exchange in *ADbasic* with **TDrv\_Init**.
4. Now you can resume to work with the ringbuffer.

### See also

[Ringbuffer](#) (declaration), [Refresh\\_RingBuffer](#), [Ringbuffer\\_Empty](#), [RingBuffer\\_Full](#), „Data structure Ringbuffer” on page 103

### Example

```
REM 1007 LONG elements as write ringbuffer
Dim Data_11[1007] As Long As Ringbuffer_For_Write

Init:
  Rem initialize read pointer of Data_11 (using Par_4)
  Ringbuffer_Clear(11, Par_4)
```

## Ringbuffer\_Empty

**RingBuffer\_Empty** returns the number of free elements in a write ringbuffer array.

### Syntax

```
ret_val = RingBuffer_Empty (data_no, Par_x)
```

### Parameters

<b>data_no</b>	Number (1...16) of the ringbuffer array <b>Data_x</b> .	LONG
<b>Par_x</b>	For data exchange <i>ADwin</i> CPU / <i>TiCo</i> only: Global variable ( <b>Par_1...Par_80</b> ), which contains a copy of the read pointer to the ringbuffer. For data exchange with external memory: 0.	LONG
<b>ret_val</b>	Number of the free array elements.	LONG

### Notes

Initialize the write pointer of the ringbuffer with **Ringbuffer\_Clear** before accessing the ringbuffer the first time.

If you want to write data into a ringbuffer array, you can use **RingBuffer\_Empty**, to determine if the ringbuffer still has enough empty elements.

Please note dimensioning in steps of 8 (see [page 196](#)).

The global variable **Par\_x** is only required for data exchange between *ADwin* CPU (e.g. T11) and *TiCo* processor. The variable will then contain a copy of the read or write pointer to the ringbuffer.

If for example you read data from the ringbuffer with **Get\_TiCo\_RingBuffer** (in *ADbasic*), the *ADwin* CPU updates the read pointer with every read access and copies the value into the global variable **Par\_x** on the *TiCo* processor. Using the value of **Par\_x**, **RingBuffer\_Empty** can calculate the number of free elements in the ringbuffer in *TiCoBasic*.

### See also

[Ringbuffer](#) (declaration), [RingBuffer\\_Full](#), [Get\\_TiCo\\_RingBuffer](#) (*ADbasic*), „Data structure Ringbuffer“ on [page 103](#)

### Example

```
REM 1007 LONG elements as write ringbuffer  
Dim Data_11[1007] As Long As Ringbuffer_For_Write  
  
Init:  
    Ringbuffer_Clear(11, Par_4)  
  
Event:  
    Rem read number of unused elements in Data_11,  
    Rem using Par_4 as read pointer  
    Par_1 = RingBuffer_Empty(11, Par_4)
```

## RingBuffer\_Full

**RingBuffer\_Full** returns the number of used elements in a read ringbuffer array.

### Syntax

```
ret_val = RingBuffer_Full (data_no, Par_x)
```

### Parameters

<b>data_no</b>	Number (1...16) of the ringbuffer array <b>Data_x</b> .	LONG
<b>Par_x</b>	For data exchange <i>ADwin</i> CPU / <i>TiCo</i> only: Global variable ( <b>Par_1...Par_80</b> ), which contains a copy of the write pointer to the ringbuffer. For data exchange with external memory: 0.	VAR LONG
<b>ret_val</b>	Number of used array elements.	LONG

### Notes

Initialize the write pointer of the ringbuffer with **Ringbuffer\_Clear** before accessing the ringbuffer the first time.

Before reading data from the ringbuffer array, you should use **RingBuffer\_Full** to check if there is data in the ringbuffer. If there is no data, an undefined value is returned from the ringbuffer array.

Please note dimensioning in steps of 8 (see [page 196](#)).

The global variable **Par\_x** is only required for data exchange between *ADwin* CPU (e.g. T11) and *TiCo* processor. The variable will then contain a copy of the read or write pointer to the ringbuffer.

If for example you write data into the ringbuffer with **Set\_TiCo\_RingBuffer** (in *ADbasic*), the *ADwin* CPU updates the write pointer with every write access and copies the value into the global variable **Par\_x** on the *TiCo* processor. Using the value of **Par\_x**, **RingBuffer\_Full** can calculate the number of used elements in the ringbuffer in *TiCoBasic*.

### See also

[Ringbuffer](#) (declaration), [Ringbuffer\\_Empty](#), [Set\\_TiCo\\_RingBuffer](#) (*ADbasic*), „Data structure Ringbuffer“ on [page 103](#)

### Example

```
REM 1007 LONG elements as read ringbuffer  
Dim Data_12[1007] As Long As RingBuffer_For_Read  
  
Init:  
    Ringbuffer_Clear(12, Par_3)  
  
Event:  
    Rem read number of used elements in Data_12,  
    Rem using Par_3 as write pointer  
    Par_1 = RingBuffer_Full(12, Par_3)
```

## 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

This control structure cannot be used within a library function or sub-routine.

You may nest several `SelectCase` structures; the only limit is the memory size.

Depending on the argument you can replace multiple nested `IF` structures with `SelectCase` so that they will be more clearly structured; another benefit is this structure is executed faster than several consecutive `If` structures.

If the argument to be evaluated does not correspond to one of the `Case` constants, only the `CaseElse` instruction block is executed (if

there is any). This is also true when the argument to be evaluated is beyond the value range of the constant.

**CCase** means "Continue Case": If a **Case** or **CCase** instruction block has been executed, then a directly following **CCase** instruction block is executed, too.

In the example below, not only **ADC (5)**, but also **ADC (7)** are executed. However, if **Par\_1=3**, then only **ADC (7)** will be executed.

If you change variables in the instruction blocks in such a manner that the value of the argument is changed, this will only be considered at the next **SelectCase** query.

The **SelectCase** structure creates an internal branch table located in the data memory (DM), whose memory requirements correspond to the greatest used **Case**-/**CCase**-constant. In order to limit the memory requirements to a minimum, the value range of constants is restricted to 0...255. There is:

Memory requirement in bytes = [ (greatest constant value)+1 ] 4

As an example the memory requirement with a max. **Case** constant **200** is  $(200 + 1) \cdot 4 = 804$  Bytes; the maximum possible memory requirement is 1 KiB.

### See also

[Do ... Until, For ... To ... {Step ...} Next, If ... Then ... {Else ...} EndIf](#)

```
Event:
  Par_1=2
  SelectCase Par_1      'Evaluate Par_1
    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 zeros.

### 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

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](#)

```
Dim val1, val2 As Long
```

#### Event:

```
val1 = 1024  
val2 = Shift_Left(val1, 2) 'Result: val2=4096
```

## Shift\_Right

The **Shift\_Right** instruction shifts all bits of a value by a specified number of places to the right. The empty bits at the left are filled with zeros.

### Syntax

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

### Parameters

val	Argument.	LONG
num	Number of places, which are shifted (0...31).	LONG
ret_val	Argument with shifted bits or. 0 for (num<0) and for (num>31).	LONG

### Notes

If the argument **val** is a positive number, shifting it **num** 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](#)

```
Dim val1, val2 As Long
```

#### Event:

```
val1 = 1024
val2 = Shift_Right(val1, 3) 'Result: val2=128
```

## Sleep

**Sleep** causes the processor to wait for a certain time.

### Syntax

```
Sleep(val)
```

### Parameters

val

Number ( $\geq 1$ ) of time units to wait in 100ns.

LONG

### Notes

Since **Sleep** is executed as a count loop, it cannot be interrupted in high-priority process.

If possible, use a constant as argument. If the argument **val** requires a calculation, it requires additional time; this time interval is constant and takes a few clock cycles.

The following conditions require a calculation:

- The argument is an expression with variables or array elements.
- The variable in the argument is declared in the memory area [DRAM\\_Extern](#).
- The argument is an array.

### See also

[NOP](#), [NOPs](#)

### Example

```
Event:
SET_MUX(0)           'Set multiplexer
Sleep(25)            'Wait 2.5  $\mu$ s (=25*100ns) = settling
                     'time of the MUX
START_CONV(1)        'Start conversion
```

With an invalid argument, the return value is undefined.

## Sub ... EndSub

The `Sub...EndSub` commands are used to define a subroutine macro with passed parameters.

### Syntax

```
Sub macro_name({val_1, val_2, ...})
    {Dim var As <var_type>}
    ...
    'Instruction block
EndSub
```

### Parameters

<code>macro_name</code>	Name of the subroutine.	
<code>val_1,</code> <code>val_2</code>	Name of the passed parameter; for arrays use the syntax with dimension brackets: <code>array[]</code> or <code>Data_n[]</code> .	<code>LONG</code>

### Notes

You will find general information about macros in [chapter 4.5.1 on page 112](#).

This instruction defines a subroutine-macro, which means the whole instruction block between `Sub` and `EndSub` 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. Before section `Init:`
2. After section `Finish:`
3. In a separate file, which you include with `#Include` (only at the locations 1. and 2.).

Be aware that in subroutines:

- no process sections such as **Init:**, **Event:**, or **Finish:** can be defined,
- local variables can be defined at the beginning, which are only available in the function and for the processing period.  
This is true even when a variable has the same name as a variable outside the function.

If a passed parameter is part of an expression inside a subroutine, the parameter should be set in braces. This avoids problems with precedence rules (e.g. BODMAS).

A subroutine is called with its name and with all its arguments, which you have defined. Valid arguments include every expression (also arrays), as long as it has the appropriate data type.

If you do not define arguments, you have to use the empty parentheses when calling the subroutine: `name()`.

If an array (not an array element) is used as a passed parameter, the syntax is different for call and definition:

- Subroutine call *without* dimension brackets:  
`subname(array_pass)`
- Subroutine definition *with* dimension brackets:  
`Sub subname(array_def []) ...`

Values are assigned to elements of passed arrays as usual:

```
array_pass[2] = value
```



If a value is assigned to a passed parameter `x` within the subroutine, the subroutine's call must not use a constant `x`, but a variable or a single array element. If so, a passed parameter can be used to hold a return value.

### See also

[#Include, Function ... EndFunction](#)

- / -



## 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
------

### See also

[And](#), [Not](#), [Or](#)

```
Dim value As Long
Event:
value = 0100b XOr 0110b
Rem Result: value = (4 XOr 6) = 0010b = 2
```



### 7.3 Mathematics Instructions

The include file `math.inc` contains additional mathematics instructions, which are not part of the instruction set of the *ADbasic* compiler.

#### Mathematics instructions

Name	Function
<code>Mod</code>	Mod returns the integer remainder of an integer division.

## Mod

**Mod** returns the integer remainder of an integer division.

### Syntax

```
#Include Math_TiCo.inc
val = Mod(x_param, y_param)
```

### Parameters

<code>x_param</code>	Dividend.	LONG
<code>y_param</code>	Divisor.	LONG
<code>val</code>	Remainder of the division <code>x_param / y_param</code> .	LONG

### Notes

The remainder calculation performs the truncated division, where the quotient is defined by truncation. With this definition, the quotient is rounded towards zero and the remainder has the same sign as the dividend.

The integer remainder of a division by zero equals the dividend:

**Mod**(`x`, 0) = `x`.

up to 0.86µs with a TiCo-1 (high priority).

### See also

[/ Division](#), [Absl](#)

### Example

```
#Include Math_TiCo.inc
Par_1 = Mod(17, 3)      'Par_1 = 2
Par_2 = Mod(-9, 5)     'Par_2 = -4
Par_3 = Mod(72, Par_2) 'Par_3 = 3
```

## 7.4 Gold II: TiCo processor

This section describes *ADbasic* instructions, which allow the *ADwin* CPU (T11) to access the *TiCo* processor.

Initialize	<a href="#">TDrv_Init</a>
Global variables	<a href="#">Get_Par</a> , <a href="#">Get_Par_Block</a> <a href="#">Set_Par</a> , <a href="#">Set_Par_Block</a>
Global arrays	<a href="#">GetData_Long</a> , <a href="#">SetData_Long</a>
Ringbuffer	<a href="#">Get_TiCo_RingBuffer</a> , <a href="#">RingBuffer_Empty</a> <a href="#">Set_TiCo_RingBuffer</a> , <a href="#">RingBuffer_Full</a>
Processdelay	<a href="#">TiCo_Get_Processdelay</a> , <a href="#">TiCo_Set_Processdelay</a>
Debug error	<a href="#">TiCo_Get_Process_Error</a>
Process control	<a href="#">TiCo_Start_Process</a> , <a href="#">TiCo_Stop_Process</a> <a href="#">TiCo_Stop</a> , <a href="#">TiCo_Start</a> , <a href="#">TiCo_Restart</a> , <a href="#">TiCo_Reset_Mode</a>
System information	<a href="#">Get_TiCo_Status</a> , <a href="#">Process_Status</a> , <a href="#">Workload</a>
Data transfer	<a href="#">TiCo_Flash</a> , <a href="#">TiCo_Load</a>

Please note: Most instructions must be initialized using **TDrv\_Init** before data transfer.

## Get\_Par

**Get\_Par** returns the value of the global variable **Par\_x** of a *TiCo* processor.

### Syntax

```
#Include ADwinGoldII.Inc  
  
ret_val = Get_Par(tico_no, par_no)
```

### Parameters

<code>tico_no</code>	Number (1) of <i>TiCo</i> processor.	LONG
<code>par_no</code>	Number (1...80) of global variable.	LONG
<code>ret_val</code>	Value ( $-2^{31} \dots +2^{31}-1$ ) of global variable.	LONG

### Notes

Several values are read more quickly using **Get\_Par\_Block**.

The number `tico_no` is not to be confused with the type of the *TiCo* processor as *TiCo1* or *TiCo2* and must have the value 1 at the time. The parameter is planned for later developments, when *ADwin* hardware contains several *TiCo* processors.

### See also

[Get\\_Par\\_Block](#), [GetData\\_Long](#), [Set\\_Par](#), [Set\\_Par\\_Block](#), [SetData\\_Long](#), [TDrv\\_Init](#)

### Valid for

Gold II

### Pin assignments

### Example

```
#Include ADwinGoldII.Inc
#Define tico_no 1           'TiCo no.
Dim tset[150] As Long      'settings array for data
                           'transfer

Init:
    TDrv_Init(tico_no,tset)

Event:
    REM read Par_1 from TiCo and write value to Par_2 of ADwin
    REM CPU
    Par_2 = Get_Par(tico_no,1)
```

## Get\_Par\_Block

**Get\_Par\_Block** reads a number of global variables **Par\_x** of the *TiCo* processor and writes the values into an array.

### Syntax

```
#Include ADwinGoldII.Inc

Get_Par_Block(tico_no, dest_array[],
              dest_array_idx, par_no, par_count)
```

### Parameters

<code>tico_no</code>	Number (1) of <i>TiCo</i> processor.	LONG
<code>dest_array[]</code>	Destination array, into which values are transferred.	LONG
<code>dest_array_idx</code>	Destination start index (1...n): array element, where the first value is stored.	LONG
<code>par_no</code>	Index (1...80) of the first global variable, that is read.	LONG
<code>par_count</code>	Number (1...80) of variables to be read.	LONG

### Notes

- / -

### See also

[Get\\_Par](#), [GetData\\_Long](#), [Set\\_Par](#), [Set\\_Par\\_Block](#), [SetData\\_Long](#), [TDrv\\_Init](#)

### Valid for

Gold II

### Pin assignments

### Example

```
#Include ADwinGoldII.Inc
#Define tico_no 1           'TiCo no.
Dim Data_1[80] As Long
Dim tset[150] As Long      'settings array for data
                           'transfer

Init:
    TDrv_Init(tico_no,tset)

Event:
    REM read Par_1...Par_80 from TiCo and write values to
    REM Data_1[1]...Data_1[80] of ADwin CPU
    Get_Par_Block(tico_no,Data_1,1,1,80)
```

## Get\_TiCo\_RingBuffer

**Get\_TiCo\_RingBuffer** reads values from a ringbuffer of a *TiCo* processor and writes the values into an array of *ADwin* CPU.

### Syntax

```
#Include ADwinGoldII.Inc

ret_val = Get_TiCo_RingBuffer(
    tdrv_datatable[], src_array_no,
    dest_array[], dest_array_idx, maxcount,
    flowrate, tico_par, struct)
```

### Parameters

<b>tdrv_datatable[]</b>	Array holding settings for data transfer, e.g. <i>TiCo</i> number.	LONG
<b>src_array_no</b>	Number (1...16) of write ringbuffer <i>Data_n</i> on the <i>TiCo</i> processor.	FLOAT
<b>dest_array[]</b>	Destination array, into which values are to be transferred.	LONG
<b>dest_array_idx</b>	Index (1...n) of the first element in <i>dest_array[]</i> to be written.	LONG
<b>maxcount</b>	Max. number (1...n) of transferred values.	LONG
<b>flowrate</b>	Evaluation for low priority processes only: Code for data flow rate 1: slow. 2: medium. 3: fast.	LONG
<b>tico_par</b>	Code number to transfer the read pointer to the <i>TiCo</i> processor. 1...80: Number (1...80) of global variable <i>Par_n</i> of the <i>TiCo</i> processor, into which the current read pointer value is written. 0: Read pointer value is not transferred.	LONG



<code>struct</code>	Code for measurement records: 0: Read arbitrary number of values. >0: The number of transferred values must be a multiple of <code>struct</code> .	LONG
<code>ret_val</code>	Success status of instruction: -1: Error: The <i>TiCo</i> write ringbuffer is dimensioned wrongly. ≥0: Successful. The return value equals the number of transferred values.	LONG

### Notes

The instruction does not read more than `maxcount` values. If the ringbuffer holds less values, all ringbuffer values are transferred.

While reading from a ringbuffer, `Get_TiCo_RingBuffer` stores the last reading position, the read pointer, into the array `tdrv_datatable[]`. If the instruction `RingBuffer_Empty` of the *TiCo* processor is to run correctly, `tico_par` must hold the number of a global variable. Thus, the read pointer value is transferred into the global variable of the *TiCo* processor.

### See also

[Set\\_TiCo\\_RingBuffer](#), [TDrv\\_Init](#)

### Valid for

Gold II

### Pin assignments

### Example

- / -

## Get\_TiCo\_Status

**Get\_TiCo\_Status** returns, whether the *TiCo* processor is active.

### Syntax

```
#Include ADwinGoldII.inc  
ret_val = Get_TiCo_Status()
```

### Parameters

<b>ret_val</b>	Status of <i>TiCo</i> processor:	LONG
	0: Processor is stopped.	
	1: Processor is running.	
	-3: Error, no <i>TiCo</i> processor available.	

### Notes

- / -

### See also

[Process\\_Status](#), [Workload](#)

### Valid for

Gold II

### Pin assignments

### Example

- / -

## GetData\_Long

**GetData\_Long** reads values from a global array of a *TiCo* processor and writes the values into a specified array.

### Syntax

```
#Include ADwinGoldIII.Inc

GetData_Long(tdrv_datatable[], src_array_no,
             src_array_idx, count, dest_array[],
             dest_array_idx, flowrate)
```

### Parameters

<code>tdrv_datatable[]</code>	Array holding settings for data transfer, e.g. <i>TiCo</i> number.	LONG
<code>src_array_no</code>	Number (1...16) of global array <code>Data_x</code> on the <i>TiCo</i> processor.	LONG
<code>src_array_idx</code>	Index (1...n) of the first element, to be read from the global array <code>src_array_no</code> .	LONG
<code>count</code>	Number (1...n) of transferred values.	LONG
<code>dest_array[]</code>	Destination array, into which values are transferred.	LONG
<code>dest_array_idx</code>	Destination start index (1...n): Array element, which is written first.	LONG
<code>flowrate</code>	Evaluation for low priority processes only: Code for data flow rate 1: slow. 2: medium. 3: fast.	LONG

**Notes**

It is to be assured that

- the global array `src_array_no` on the *TiCo* processor is already dimensioned and
- the array `dest_array` has at least `count` elements.

**See also**

[Get\\_Par](#), [Set\\_Par](#), [SetData\\_Long](#), [TDrv\\_Init](#)

**Valid for**

Gold II

**Pin assignments****Example**

```
#Include ADwinGoldII.Inc
#Define tico_no 1           'TiCo no.
Dim tset_41[150] As Long   'settings array for data
                             'transfer

Dim Data_4[200] As Long
```

**Init:**

```
REM initialize data transfer ADwin CPU <-> TiCo
TDrv_Init(tico_no,tset_41)
```

**Event:**

```
REM read 120 values from TiCo Data_2 (starting from
REM Data_2[20]) into ADwin CPU Data_4 (starting
REM from index 5). flowrate is high.
GetData_Long(tset_41,2,20,120,Data_4,5,3)
```

## Process\_Status

**Process\_Status** returns the status of a process on a *TiCo* processor.

### Syntax

```
#Include ADwinGoldIII.Inc  
  
ret_val = Process_Status(tico_no,  
                        process_no_no)
```

### Parameters

<code>tico_no</code>	Number (1) of <i>TiCo</i> processor.	LONG
<code>process_no_no</code>	Number (1...4) of <i>TiCo</i> process.	LONG
<code>ret_val</code>	Process status: ≠1: Process is running. 0: Process does not run, i.e. it is not loaded, not started or stopped.	LONG

### Notes

- / -

### See also

[TiCo\\_Get\\_Processdelay](#), [TiCo\\_Set\\_Processdelay](#), [TiCo\\_Start\\_Process](#), [TiCo\\_Stop\\_Process](#), [Workload](#)

### Valid for

Gold II

### Pin assignments

**Example**

```
#Include ADwinGoldII.Inc
#Define tico_no 1           'TiCo no.
Dim Data_4[200] As Long
Dim tset[150] As Long      'settings array for data
                           'transfer

Init:
    TDrv_Init(tico_no,tset)

Event:
    REM read status of TiCo process 1
    Par_1 = Process_Status(tico_no,1)
    REM if process is running, read TiCo Par_5
    If (Par_1 = 1) Then
        Par_2 = Get_Par(tico_no,5)
    EndIf
```

## RingBuffer\_Empty

**RingBuffer\_Empty** returns the number of free elements in a write ringbuffer on a *TiCo* processor.

### Syntax

```
#Include ADwinGoldIII.Inc  
ret_val = RingBuffer_Empty(tdrv_datatable[],  
    Data_no)
```

### Parameters

<code>tdrv_datatable</code>	Array holding settings for data transfer, e.g. <i>TiCo</i> number.	LONG
<code>Data_no</code>	Number (1...16) of read ringbuffer <code>Data_n</code> on the <i>TiCo</i> processor.	LONG
<code>ret_val</code>	Number (0...n) of free elements in the write ringbuffer.	LONG

### Notes

For use of **Get\_TiCo\_RingBuffer**, a previous query with **Ringbuffer\_Empty** is not required.

### See also

[Get\\_TiCo\\_RingBuffer](#), [RingBuffer\\_Full](#), [TDrv\\_Init](#)

### Valid for

Gold II

### Pin assignments

### Example

- / -

## RingBuffer\_Full

**RingBuffer\_Full** returns the number of used elements in a read ringbuffer on a *TiCo* processor.

### Syntax

```
#Include ADwinGoldII.Inc

ret_val = RingBuffer_Full(tdrv_datatable[],
    Data_no)
```

### Parameters

<b>tdrv_datatable</b>	Array holding settings for data transfer, e.g. <i>TiCo</i> number.	LONG
<b>Data_no</b>	Number (1...16) of read ringbuffer <b>Data_n</b> on the <i>TiCo</i> processor.	LONG
<b>ret_val</b>	Number (0...n) of used elements in the read ringbuffer.	LONG

### Notes

For use of **Set\_TiCo\_RingBuffer**, a previous query with **Ringbuffer\_Full** is not required.

### See also

[Set\\_TiCo\\_RingBuffer](#), [RingBuffer\\_Empty](#), [TDrv\\_Init](#)

### Valid for

Gold II

### Pin assignments

### Example

- / -



## Set\_Par

**Set\_Par** sets the value of a global variable **Par\_x** on a *TiCo* processor.

### Syntax

```
#Include ADwinGoldIII.Inc  
Set_Par(tico_no, par_no, value)
```

### Parameters

<code>tico_no</code>	Number (1) of <i>TiCo</i> processor.	LONG
<code>par_no</code>	Number (1...80) of global variable.	LONG
<code>value</code>	Value ( $-2^{31} \dots +2^{31}-1$ ) of global variable.	LONG

### Notes

**Set\_Par** sets the value of the global variable, not regarding whether a *TiCo* process is running. Since *ADwin* CPU and *TiCo* processes do not run synchronously, changing the value of a global variable during run-time may be totally unexpected to the *TiCo* process.

If needed, we recommend synchronizing *ADwin* CPU and *TiCo* processes with aid of a software handshake.

### See also

[Get\\_Par](#), [Get\\_Par\\_Block](#), [GetData\\_Long](#), [Set\\_Par\\_Block](#), [SetData\\_Long](#), [TDrv\\_Init](#)

### Valid for

Gold II

### Pin assignments

**Example**

```
#Include ADwinGoldII.Inc
#Define tico_no 1           'TiCo no.
Dim tset[150] As Long      'settings array for data
                           'transfer

Init:
    TDrv_Init(tico_no,tset)
    Par_5=200

Event:
    REM write value of Par_5 (ADwin CPU) to TiCo Par_2
    Set_Par(tico_no,2,Par_5)
```

## Set\_Par\_Block

**Set\_Par\_Block** writes values from an array into a number of global variables **Par\_x** of the *TiCo* processor.

### Syntax

```
#Include ADwinGoldIII.Inc

Set_Par_Block(tico_no, src_array[],
              src_array_idx,
              par_no, par_count)
```

### Parameters

<code>tico_no</code>	Number (1) of <i>TiCo</i> processor.	LONG
<code>src_array[]</code>	Source array, from which values are to be transferred.	LONG
<code>src_array_idx</code>	Index of the array element, starting from which values are read from <code>src_array[]</code> .	LONG
<code>par_no</code>	Index (1...80) of the first global <i>TiCo</i> variable to be written.	LONG
<code>par_count</code>	Number of transferred values.	LONG

### Notes

**Set\_Par\_Block** sets values of global variables, not regarding whether a *TiCo* process is running or not. Since *ADwin* CPU and *TiCo* processes do not run synchronously, changing the value of a global variable during run-time may be totally unexpected to the *TiCo* process.

If needed, we recommend synchronizing *ADwin* CPU and *TiCo* processes with aid of a software handshake.

### See also

[Get\\_Par](#), [Get\\_Par\\_Block](#), [GetData\\_Long](#), [Set\\_ParSetData\\_Long](#), [TDrv\\_Init](#)

**Valid for**

Gold II

**Pin assignments****Example**

```
#Include ADwinGoldII.Inc
#Define tico_no 1           'TiCo no.
Dim tset[150] As Long 'array for data transfer settings
Dim Data_1[40] As Long
Dim i As Long
```

**Init:**

```
TDrv_Init(tico_no,tset)
For i = 1 To 40
    Data_1[i] = i*10
Next
Par_15=1111
Par_16=2222
Par_17=3333
Par_18=4444
Par_19=5555
```

**Event:**

```
REM write 40 values from Data_1 (starting from Data_1[1])
REM into Par_1...Par_40 (TiCo)
Set_Par_Block(tico_no,Data_1,1,1,40)
```

### Set\_TiCo\_RingBuffer

**Set\_TiCo\_RingBuffer** writes values from an *ADwin* CPU array into a ringbuffer on the *TiCo* processor.

#### Syntax

```
#Include ADwinGoldIII.Inc

ret_val =
    Set_TiCo_RingBuffer(tdrv_datatable[],
        dest_array_no, src_array[], src_array_idx,
        maxcount, flowrate, tico_par, struct)
```

#### Parameters

<code>tdrv_datatable[]</code>	Array holding settings for data transfer, e.g. <i>TiCo</i> number.	LONG
<code>dest_array_no</code>	Number (1...16) of read ringbuffer <i>Data_n</i> on the <i>TiCo</i> processor.	FLOAT
<code>src_array[]</code>	Source array, from which values are transferred.	LONG
<code>src_array_idx</code>	Index (1...n) of the first element in <code>src_array[]</code> to be read.	LONG
<code>maxcount</code>	Max. number (1...n) of transferred values.	LONG
<code>flowrate</code>	Evaluation for low priority processes only: Code for data flow rate 1: slow. 2: medium. 3: fast.	LONG
<code>tico_par</code>	Code number to transfer the write pointer to the <i>TiCo</i> processor. 1...80: Number (1...80) of global variable <i>Par_n</i> of the <i>TiCo</i> processor, into which the current write pointer value is written. 0: Write pointer value is not transferred.	LONG

<b>struct</b>	Code for measurement records: 0: Read arbitrary number of values. >0: The number of transferred values must be a multiple of <b>struct</b> .	LONG
<b>ret_val</b>	Success status of instruction: -1: Error: The <i>TiCo</i> read ringbuffer is dimensioned wrongly. ≥0: Successful. The return value equals the number of transferred values.	LONG

### Notes

The instruction does not write more than **maxcount** values. If the ringbuffer has less empty elements, only the empty ringbuffer elements are filled.

While writing into a ringbuffer, **Set\_TiCo\_RingBuffer** stores the last writing position, the write pointer, into the array **tdrv\_datatable[]**. If the instruction **RingBuffer\_Full** of the *TiCo* processor is to run correctly, the number of a global variable has to be set in **tico\_par**. Thus, the write pointer value is transferred into the global variable of the *TiCo* processor.

### See also

[Get\\_TiCo\\_RingBuffer](#), [TDrv\\_Init](#)

### Valid for

Gold II

### Pin assignments

### Example

- / -

## SetData\_Long

**SetData\_Long** reads values from an array and writes them into a global array of a *TiCo* processor.

### Syntax

```
#Include ADwinGoldIII.Inc

SetData_Long(tdrv_datatable[], dest_array_no,
             dest_array_idx, count, src_array[],
             src_array_idx, flowrate)
```

### Parameters

<code>tdrv_datatable[]</code>	Array holding settings for data transfer, e.g. <i>TiCo</i> number.	LONG
<code>dest_array_no</code>	Number (1...16) of global array <code>Data_x</code> on the <i>TiCo</i> processor.	LONG
<code>dest_array_idx</code>	Index (1...n) of the first element, to be written in the global array <code>dest_array</code> .	LONG
<code>count</code>	Number (1...n) of transferred values.	LONG
<code>src_array[]</code>	Source array, from where values are read.	LONG
<code>src_array_idx</code>	Source start index (1...n): Array element, which is read first.	LONG
<code>flowrate</code>	Evaluation for low priority processes only: Code for data flow rate 1: slow. 2: medium. 3: fast.	LONG

**Notes**

It is to be assured that

- the global array `dest_array_no` on the *TiCo* processor is already dimensioned and
- the array `src_array` has at least `count` elements.

**See also**

[Get\\_Par](#), [GetData\\_Long](#), [Set\\_Par](#), [TDrv\\_Init](#)

**Valid for**

Gold II

**Pin assignments****Example**

```
#Include ADwinGoldII.Inc
#Define tico_no 1           'TiCo no.
Dim tset[150] As Long      'settings array for data
                           'transfer

Dim Data_1[150] As Long
Dim tset_41[150] As Long
Dim i As Long
```

**Init:**

```
REM initialize data transfer ADwin CPU <-> TiCo
TDrv_Init(tico_no,tset_41)
For i = 1 To 80
    Data_1[i] = i
Next
```

**Event:**

```
REM read 120 values from TiCo Data_2 (starting from
REM Data_2[1]) into ADwin CPU Data_1 (starting from
REM Data_1[5]). flowrate is high.
SetData_Long(tset_41,2,1,120,Data_1,5,3)
```



## TDrv\_Init

**TDrv\_Init** initializes the data transfer between ADwin CPU and a *TiCo* processor.

### Syntax

```
#Include ADwinGoldIII.Inc  
  
REM define settings array for TiCo x  
Dim tdrv_datatable[150] As Long  
  
TDrv_Init(tico_no, tdrv_datatable[])
```

### Parameters

<code>tico_no</code>	Number (1) of <i>TiCo</i> processor.	LONG
<code>tdrv_</code> <code>datatable</code>	Array holding settings for data transfer, e.g. <i>TiCo</i> number <i>x</i> .	LONG
<code>[]</code>		

### Notes

The instruction must be processed before data transfer between ADwin CPU and *TiCo* processor. The instruction should be set into the **Init:** section.

Most instructions accessing a *TiCo* processor require initialization of data transfer.

Initialization must be run for each *TiCo* processor separately. For each processor an array `tdrv_datatable[]` with 150 elements has to be dimensioned, too.

The array `tdrv_datatable[]` is used by **GetData\_Long**, **SetData\_Long** and **Workload**.

### See also

[Get\\_Par](#), [Get\\_Par\\_Block](#), [TiCo\\_Get\\_Processdelay](#), [GetData\\_Long](#), [Set\\_Par](#), [Set\\_Par\\_Block](#), [TiCo\\_Set\\_Processdelay](#), [SetData\\_Long](#)

### Valid for

Gold II

## Pin assignments

### Example

```
#Include ADwinGoldII.Inc
#Define tico_no 1           'TiCo no.
Dim Data_4[200] As Long
Dim Data_5[1000] As Long
Dim tset[150] As Long
```

### Init:

```
REM initialize data transfer ADwin CPU <-> TiCo
TDrv_Init(tico_no,tset)
```

### Event:

```
REM read 120 values from TiCo Data_2 (starting from
REM Data_2[20]) into ADwin CPU Data_4 (starting from
REM index 5).
REM flowrate is 3 (high)
GetData_Long(tset,2,20,120,Data_4,5,3)
REM read 800 values from TiCo Data_5 (starting from
REM Data_7[9]) into Data_5 of ADwin CPU (starting from
REM index 1).
REM Flowrate is 3 (high)
GetData_Long(tset,5,9,800,Data_7,1,3)
```

## TiCo\_Flash

**TiCo\_Flash** transfers a *TiCoBasic* binary file from an array into the flash memory of a *TiCo* processor.

### Syntax

```
#Include ADwinGoldIII.inc  
  
ret_val = TiCo_Flash(tico_no, array[])
```

### Parameters

<code>tico_no</code>	Number (1) of <i>TiCo</i> processor.	LONG
<code>array[]</code>	Array, which holds the <i>TiCoBasic</i> binary file to be transferred.	LONG
<code>ret_val</code>	Status of data transfer: 0: Data transfer successful. -1: Error: Instruction may only be used with low priority. 2: Error: Program memory too small. 3: Error: Data memory too small. 4: Error: Program and data memory too small. 5: Error: Wrong password. 6: Error: External memory memory too small. 7: Error: Flash memory too small. 12: Error: Binary file invalid for this <i>TiCo</i> processor.	LONG

### Notes

Use **TiCo\_Flash** only in low priority processes.

The instruction **TiCo\_Flash** is used to transfer a *TiCoBasic* binary file—a compiled *TiCo* program—into the flash memory of a *TiCo* processor. The following steps are required:

- Create a binary file in the development environment *TiCoBasic* using **Build** ► **Make Bin File**.
- Transfer the binary file into a global array of the *ADwin* CPU. A useful means is the function **File2Data** (or **LoadFromFile**) which is provided in several programming languages.
- Transfer the data of the global **array []** into the flash memory using **TiCo\_Flash**.
- If the bootloader option is enabled, the process of the *TiCoBasic* binary file is started automatically after the next start-up of the *ADwin* hardware.

If **array []** does not contain a *TiCoBasic* binary file the return value is invalid.

#### See also

[TiCo\\_Load](#)

#### Valid for

Gold II

#### Pin assignments

#### Example

- / -

## TiCo\_Get\_Processdelay

**TiCo\_Get\_Processdelay** returns the **Processdelay** (cycle time) of a process on a *TiCo* processor.

### Syntax

```
#include ADwinGoldIII.Inc

ret_val =
    TiCo_Get_Processdelay(tdrv_datatable[],
        process_no)
```

### Parameters

<code>tdrv_datatable</code>	Array holding settings for data transfer, e.g. <i>TiCo</i> number.	LONG
<code>process_no</code>	Number (1...4) of <i>TiCo</i> process.	LONG
<code>ret_val</code>	Currently set cycle time ( $-2^{31} \dots +2^{31}-1$ ) of <i>TiCo</i> processor in clock cycles. One clock cycle takes 20ns.	LONG

### Notes

Using a timer controlled process, the **Event:** section is triggered by the internal counter cyclical and with fixed time interval. The time interval between 2 trigger signals, called cycle time or **Processdelay**, is measured in counter clock cycles.

### See also

[Process\\_Status](#), [TiCo\\_Set\\_Processdelay](#), [TDrv\\_Init](#)

### Valid for

Gold II

### Pin assignments

**Example**

```
#Include ADwinGoldII.Inc
#Define tico_no 1           'TiCo no.
Dim tset[150] As Long      'settings array for data
                             'transfer

Init:
    TDrv_Init(tico_no,tset)

Event:
    REM read processdelay of process 1 into Par_1
    Par_1 = TiCo_Get_Processdelay(tset,1)
```

## TiCo\_Get\_Process\_Error

**TiCo\_Get\_Process\_Error** returns the most recently occurred error of the selected process of a *TiCo* processor.

### Syntax

```
#Include ADwinGoldIII.inc  
  
ret_val = TiCo_Get_Process_Error(tdrv_datatable[],  
                                process_no)
```

### Parameters

<code>tdrv_datatable</code>	Array holding settings for data transfer, e.g. <i>TiCo</i> number.	LONG
<code>process_no</code>	Number (1...4) of <i>TiCo</i> process.	LONG
<code>ret_val</code>	Number of the previously occurred error in the process: 0: no error 10: Accessing a too high element number of a global array. 11: Accessing a too small element number ( $\leq 0$ ) of a global array. 12: Accessing a too high element number of a local array. 13: Accessing a too small element number ( $\leq 0$ ) of a local array.	LONG

### Notes

The return value is defined only if debug mode is enabled (see [Debug mode Option](#), [page 71](#)). The variable is read-only.

### See also

[Process\\_Status](#), [TiCo\\_Get\\_Processdelay](#), [TDrv\\_Init](#)

**Valid for**

Gold II

**Pin assignments****Example**

```
#Include ADwinGoldII.INC
#Define tico_no 1           'TiCo no.
Dim tset[150] As Long      'settings array for data
                           'transfer

Init:
    TDrv_Init(tico_no, tset)

Event:
    REM read recent process error of process 1 into Par_1
    Par_1 = TiCo_Get_Process_Error(tset, 1)
```



## TiCo\_Load

**TiCo\_Load** transfers a *TiCoBasic* binary file from an array into the memory of a *TiCo* processor and starts the process.

### Syntax

```
#Include ADwinGoldIII.inc  
ret_val = TiCo_Load(tico_no, array[])
```

### Parameters

<code>tico_no</code>	Number (1) of <i>TiCo</i> processor.	LONG
<code>array[]</code>	Array, which holds the <i>TiCoBasic</i> binary file to be transferred.	LONG
<code>ret_val</code>	Status of data transfer: 0: Data transfer successful. -1: Error: Instruction may only be used with low priority. 1: Error: Invalid processor number. 2: Error: Program memory too small. 3: Error: Data memory too small. 4: Error: Program and data memory too small. 5: Error: Wrong password. 6: Error: External memory too small. 10: Error: Required external SRAM too small. 11: Error: Array contains no valid <i>TiCoBasic</i> binary file. 12: Error: Binary file invalid for this <i>TiCo</i> processor.	LONG

### Notes

Use **TiCo\_Load** only in low priority processes.

The instruction **TiCo\_Load** is used to transfer a *TiCoBasic* binary file—a compiled *TiCo* program—into the memory of a *TiCo* processor. The following steps are required:

- Create a binary file in the development environment *TiCoBasic* using Build ► Make Bin File.
- Transfer the binary file into a global array of the *ADwin* CPU. A useful means is the function `File2Data` (or `LoadFromFile`), which is provided in several programming languages.
- Transfer the data of the global `array[]` into the memory using **TiCo\_Load**.
- The process of the *TiCoBasic* binary file is automatically started.

If `array[]` does not contain a *TiCoBasic* binary file the return value is invalid.

### See also

[TiCo\\_Flash](#)

### Valid for

Gold II

### Pin assignments

### Example

- / -

## TiCo\_Restart

**TiCo\_Restart** stops the *TiCo* processor and restarts it afterwards.

### Syntax

```
#Include ADwinGoldII.Inc  
TiCo_Restart()
```

### Parameters

- / -

### Notes

Stopping immediately interrupts any running process as well as counters on the TiCo processor. Data are not changed by stopping.

### See also

[TiCo\\_Stop](#), [TiCo\\_Start](#)

### Valid for

Gold II

### Pin assignments

### Example

- / -

## TiCo\_Reset\_Mode

**TiCo\_Reset\_Mode** sets whether booting the *ADwin* CPU (T11) will reset the *TiCo* processor or not.

### Syntax

```
#Include ADwinGoldII.inc  
  
TiCo_Reset_Mode(mode)
```

### Parameters

<b>mode</b>	Select reset mode while booting the T11.	LONG
	0: Operating status of the <i>TiCo</i> processor remains unchanged. Default.	
	1: The <i>TiCo</i> processor is stopped and restarted.	

### Notes

The reset **mode**=1 is only functional if the *TiCo* processor is already running.

### See also

[TiCo\\_Restart](#), [TiCo\\_Stop](#), [TiCo\\_Start](#)

### Valid for

Gold II

### Pin assignments

### Example

```
#Include ADwinGoldII.inc  
INIT:  
    TiCo_Reset_Mode(1)      'reset TiCo processor with T11  
                             'boot
```

## TiCo\_Set\_Processdelay

**TiCo\_Set\_Processdelay** sets the **Processdelay** (cycle time) of a *TiCo* process.

### Syntax

```
#Include ADwinGoldIII.Inc

TiCo_Set_Processdelay(tdrv_
    datatable[], process_no,
    value)
```

### Parameters

<code>tdrv_</code>	Array holding settings for data transfer, e.g. <i>TiCo</i>	
<code>datatable</code>	number.	LONG
<code>process_</code>	Number (1...4) of <i>TiCo</i> process.	LONG
<code>no</code>		
<code>value</code>	Value to be set for <b>Processdelay</b> .	LONG

### Notes

- / -

### See also

[TiCo\\_Get\\_Processdelay](#), [Process\\_Status](#), [TDrv\\_Init](#)

### Valid for

Gold II

### Pin assignments

**Example**

```
#Include ADwinGoldII.Inc
#Define tico_no 1           'TiCo no.
Dim tset[150] As Long      'settings array for data
                             'transfer

Init:
  TDrv_Init(tico_no,tset)
  Processdelay = 6000       'set cycle time

Event:
  REM set TiCo processdelay to run with same cycle time as
  REM the
  REM T12 process
  TiCo_Set_Processdelay(tset,1,Processdelay/6)
```

## TiCo\_Start

**TiCo\_Start** starts the *TiCo* processor.

### Syntax

```
#Include ADwinGoldII.Inc  
TiCo_Start()
```

### Parameters

- / -

### Notes

- / -

### See also

[TiCo\\_Stop](#), [TiCo\\_Restart](#)

### Valid for

Gold II

### Pin assignments

### Example

- / -

## TiCo\_Start\_Process

**TiCo\_Start\_Process** starts a process on a *TiCo* processor.

### Syntax

```
#include ADwinGoldII.Inc

TiCo_Start_Process(tdrv_datatable[],
                  process_no)
```

### Parameters

<code>tdrv_datatable</code>	Array holding settings for data transfer, e.g. <i>TiCo</i> number.	LONG
<code>process_no</code>	Number (1...4) of <i>TiCo</i> process.	LONG

### Notes

The process must already be loaded to the *TiCo* processor.

### See also

[TiCo\\_Get\\_Processdelay](#), [Process\\_Status](#), [TiCo\\_Set\\_Processdelay](#),  
[TiCo\\_Start\\_Process](#), [TiCo\\_Stop\\_Process](#), [Workload](#)

### Valid for

Gold II

### Pin assignments



### Example

```
#Include ADwinGoldII.INC
#Define tico_no 1           'TiCo no.
Dim tset[150] As Long      'settings array for data
                           'transfer

Init:
  Par_1 = 0
  Processdelay = 100000
  REM start TiCo process in parallel to ADwin CPU process
  TDrv_Init(tico_no,tset)
  TiCo_Start_Process(tset,1)

Event:
  REM ... program

Finish:
  REM stop TiCo process
  TiCo_Stop_Process(tset,1)
```

## TiCo\_Stop

**TiCo\_Stop** stops the *TiCo* processor.

### Syntax

```
#Include ADwinGoldII.Inc  
TiCo_Stop()
```

### Parameters

- / -

### Notes

Stopping immediately interrupts any running process as well as counters on the *TiCo* processor. Data are not changed by stopping.

### See also

[TiCo\\_Start](#), [TiCo\\_Restart](#)

### Valid for

Gold II

### Pin assignments

### Example

- / -

### TiCo\_Stop\_Process

**TiCo\_Stop\_Process** stops a process on a *TiCo* processor.

#### Syntax

```
#Include ADwinGoldIII.Inc  
  
TiCo_Stop_Process (tdrv_datatable[] ,  
                  process_no)
```

#### Parameters

<code>tdrv_datatable</code>	Array holding settings for data transfer, e.g. <i>TiCo</i> number.	LONG
<code>process_no</code>	Number (1...4) of <i>TiCo</i> process.	LONG

#### Notes

The process must already be loaded to the *TiCo* processor.

#### See also

[TiCo\\_Get\\_Processdelay](#), [Process\\_Status](#), [TiCo\\_Set\\_Processdelay](#),  
[TiCo\\_Start\\_Process](#), [Workload](#)

#### Valid for

Gold II

#### Pin assignments

**Example**

```
#Include ADwinGoldIII.INC
#Define tico_no 1           'TiCo no.
Dim tset[150] As Long      'settings array for data
                           'transfer

Init:
  Par_1 = 0
  Processdelay = 100000
  REM start TiCo process in parallel to ADwin CPU process
  TDrv_Init(tico_no,tset)
  TiCo_Start_Process(tset,1)

Event:
  REM ... program

Finish:
  REM stop TiCo process
  TiCo_Stop_Process(tset,1)
```

## Workload

**Workload** returns the workload of a *TiCo* processor.

### Syntax

```
#Include ADwinGoldII.Inc  
ret_val = Workload(tdrv_datatable[])
```

### Parameters

<code>tdrv_datatable</code>	Array holding settings for data transfer, e.g. <i>TiCo</i> number.	LONG
<code>ret_val</code>	Processor workload in percent (0.0 ... 100.0) or error value: <0: <i>TiCo</i> processor is stopped or no <i>TiCo</i> processor available.	FLOAT

### Notes

The return value is the average processor workload in the time between the previous and the current call of **Workload**. Therefore, the return value of the first call in a program is invalid.

The shortest time between two instruction calls should be at least 100 times of **Processdelay**. Otherwise, the return value may have an error of more than 1%.

If the previous call of **Workload** dates back more than 85 seconds, the return value is invalid. Simply call the instruction a second time to receive a valid return value.

### See also

[TiCo\\_Get\\_Processdelay](#), [Process\\_Status](#), [TiCo\\_Set\\_Processdelay](#),  
[TiCo\\_Stop\\_Process](#), [TDrv\\_Init](#)

### Valid for

Gold II

## Pin assignments

### Example

```
#Include ADwinGoldII.Inc
#Define tico_no 1           'TiCo no.
Dim tset[150] As Long      'settings array for data
                           'transfer

Init:
    TDrv_Init(tico_no,tset)

Event:
    REM read TiCo workload
    FPar_1 = Workload(tset)
```

### 7.5 Pro II: TiCo Processor

This section describes instructions, which allow the *ADwin* CPU to access the *TiCo* processor on a Pro II module.

Initialize	P2_TDrv_Init, P2_Get_TiCo_Status
Global variables	P2_Get_Par, P2_Get_Par_Block P2_Set_Par, P2_Set_Par_Block
Global arrays	P2_GetData_Long, P2_SetData_Long
Ring buffer	P2_Get_TiCo_RingBuffer, P2_Ringbuffer_Empty P2_Set_TiCo_RingBuffer, P2_Ringbuffer_Full
Processdelay	P2_TiCo_Get_Processdelay, P2_TiCo_Set_Processdelay
Debug error	P2_TiCo_Get_Process_Error
Process control	P2_TiCo_Start_Process, P2_TiCo_Stop_Process P2_TiCo_Stop, P2_TiCo_Start, P2_TiCo_Restart
System information	P2_Get_TiCo_Status, P2_Process_Status P2_Get_TiCo_Bootloader_Status, P2_Workload
Data transfer	P2_TiCo_Flash, P2_TiCo_Load

Please note: Most instructions must be initialized using **P2\_TDrv\_Init** before data transfer.

## P2\_Get\_Par

**P2\_Get\_Par** returns the value of the global variable **Par\_x** of a *TiCo* processor of the specified module.

### Syntax

```
#Include ADwinPro_All.Inc

ret_val = P2_Get_Par(module, tico_no, par_no)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>tico_no</b>	Number (1) of <i>TiCo</i> processor on the module.	LONG
<b>par_no</b>	Number (1...80) of the global variable.	LONG
<b>ret_val</b>	Value ( $-2^{31} \dots +2^{31}-1$ ) of the globale variable.	LONG

### Notes

Several values are read more quickly using **P2\_Get\_Par\_Block**.

### See also

[P2\\_Get\\_Par\\_Block](#), [P2\\_GetData\\_Long](#), [P2\\_Set\\_Par](#), [P2\\_Set\\_Par\\_Block](#), [P2\\_SetData\\_Long](#), [P2\\_TDrv\\_Init](#)

### Valid for

Aln-32/18-D-TiCo Rev. E, Aln-8/18-TiCo Rev. E, AOut-1/16 Rev. E, AOut-4/16-TiCo Rev. E, AOut-8/16-TiCo Rev. E, ARINC-429 Rev. E, CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, DIO-32-TiCo Rev. E, DIO-32-TiCo2 Rev. E, DIO-32/1-TiCo Rev. E, DIO-8-D12 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E, RSxxx-2 Rev. E, RSxxx-4 Rev. E, SPI-2-D Rev. E, SPI-2-T Rev. E



### Example

```
#Include ADwinPro_All.Inc
#Define module 4           'module address
#Define tico_no 1          'TiCo no.
Dim tset[150] As Long      'settings array for data
                           'transfer

Init:
    P2_TDrv_Init(module,tico_no,tset)

Event:
    REM read Par_1 from TiCo and write value to Par_2 of ADwin
    REM CPU
    Par_2 = P2_Get_Par(module,tico_no,1)
```

## P2\_Get\_Par\_Block

**P2\_Get\_Par\_Block** reads a number of global variables **Par\_x** of the *TiCo* processor on the specified module and writes the values into an array.

### Syntax

```
#Include ADwinPro_All.Inc

P2_Get_Par_Block(module, tico_no,
    dest_array[], dest_array_idx, par_no,
    par_count)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>tico_no</b>	Number (1) of <i>TiCo</i> processor on the module.	LONG
<b>dest_array[]</b>	Destination array, into which values are transferred.	LONG
<b>dest_array_idx</b>	Destination start index (1...n): Array element, from which values are stored.	LONG
<b>par_no</b>	Index (1...80) of the first global variable to be read.	LONG
<b>par_count</b>	Number (1...80) of variables to be read.	LONG

### Notes

- / -

### See also

[P2\\_Get\\_Par](#), [P2\\_GetData\\_Long](#), [P2\\_Set\\_Par](#), [P2\\_Set\\_Par\\_Block](#), [P2\\_SetData\\_Long](#), [P2\\_TDrv\\_Init](#)

### Valid for

Aln-32/18-D-TiCo Rev. E, Aln-8/18-TiCo Rev. E, AOut-1/16 Rev. E, AOut-4/16-TiCo Rev. E, AOut-8/16-TiCo Rev. E, ARINC-429 Rev. E, CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, DIO-32-

TiCo Rev. E, DIO-32-TiCo2 Rev. E, DIO-32/1-TiCo Rev. E, DIO-8-D12 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E, RSxxx-2 Rev. E, RSxxx-4 Rev. E, SPI-2-D Rev. E, SPI-2-T Rev. E

### Example

```
#Include ADwinPro_All.Inc
#Define module 4           'module address
#Define tico_no 1          'TiCo no.
Dim Data_1[80] As Long
Dim tset[150] As Long      'settings array for data
                           'transfer
```

### Init:

```
P2_TDrv_Init(module,tico_no,tset)
```

### Event:

```
REM read Par_1...Par_80 from TiCo and write values to
REM Data_1[1]...Data_1[80] of ADwin CPU
P2_Get_Par_Block(module,tico_no,Data_1,1,1,80)
REM read Par_20...Par_29 from TiCo and write values to
REM Par_5...Par_14 of ADwin CPU
P2_Get_Par_Block(module,tico_no,PAR,5,20,10)
```

## P2\_Get\_TiCo\_Bootloader\_Status

**P2\_Get\_TiCo\_Bootloader\_Status** returns the *TiCo* bootloader status on the specified module.

### Syntax

```
#Include ADwinPRO_ALL.inc

ret_val = P2_Get_TiCo_Bootloader_Status(module)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>ret_val</b>	<i>TiCo</i> bootloader status: 0: Bootloader is disabled. 1: Bootloader is enabled. -1: Error; instruction used in low priority process. -2: Error; modul cannot be accessed (timeout). -3: Error; no <i>TiCo</i> processor on the module.	LONG

### Notes

The instruction can only be processed in a low priority process.

### See also

[P2\\_TDrv\\_Init](#)

### Valid for

Aln-32/18-D-TiCo Rev. E, Aln-8/18-TiCo Rev. E, AOut-1/16 Rev. E, AOut-4/16-TiCo Rev. E, AOut-8/16-TiCo Rev. E, ARINC-429 Rev. E, CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, DIO-32-TiCo Rev. E, DIO-32-TiCo2 Rev. E, DIO-32/1-TiCo Rev. E, DIO-8-D12 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E, RSxxx-2 Rev. E, RSxxx-4 Rev. E, SPI-2-D Rev. E, SPI-2-T Rev. E

### Example

- / -

### P2\_Get\_TiCo\_RingBuffer

**P2\_Get\_TiCo\_RingBuffer** reads values from a ringbuffer of a *TiCo* processor and writes the values into an array of *ADwin* CPU.

#### Syntax

```
#Include ADwinPro_All.Inc  
ret_val =  
    P2_Get_TiCo_RingBuffer(tdrv_datatable[],
```

```
src_array_no, dest_array[], dest_array_idx,
maxcount, flowrate, tico_par, struct)
```

### Parameters

<code>tdrv_</code>	Array holding settings for data transfer, e.g.	
<code>datatable</code>	module address and <i>TiCo</i> number.	LONG
<code>src_array_no</code>	Number (1...16) of write ringbuffer <i>Data_n</i> on the <i>TiCo</i> processor.	FLOAT
<code>dest_array[]</code>	Destination array, into which values are to be transferred.	LONG
<code>dest_array_idx</code>	Index (1...n) of the first element in <code>dest_array[]</code> to be written.	LONG
<code>maxcount</code>	Max. number (1...n) of transferred values.	LONG
<code>flowrate</code>	Evaluation for low priority processes only: Code for data flow rate 1: slow. 2: medium. 3: fast.	LONG
<code>tico_par</code>	Code number to transfer the read pointer to the <i>TiCo</i> processor. 1...80: Number (1...80) of global variable <i>Par_n</i> of the <i>TiCo</i> processor, into which the current read pointer value is written. 0: Read pointer value is not transferred.	LONG
<code>struct</code>	Code for measurement records: 0: Read arbitrary number of values. >0: The number of transferred values must be a multiple of <code>struct</code> .	LONG
<code>ret_val</code>	Success status of instruction: -1: Error: The <i>TiCo</i> write ringbuffer is dimensioned wrongly. ≥0: Successful. The return value equals the number of transferred values.	LONG

**Notes**

The instruction does not read more than `maxcount` values. If the ringbuffer holds less values, all ringbuffer values are transferred.

While reading from a ringbuffer, `P2_Get_TiCo_RingBuffer` stores the last reading position, the read pointer, into the array `tdrv_data-table[]`. If the instruction `RingBuffer_Empty` of the *TiCo* processor is to run correctly, this read pointer value must be transferred into the appropriate global variable of the *TiCo* processor.

**See also**

[P2\\_Set\\_TiCo\\_RingBuffer](#), [P2\\_TDrv\\_Init](#)

**Valid for**

Aln-32/18-D-TiCo Rev. E, Aln-8/18-TiCo Rev. E, AOut-1/16 Rev. E, AOut-4/16-TiCo Rev. E, AOut-8/16-TiCo Rev. E, ARINC-429 Rev. E, CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, DIO-32-TiCo Rev. E, DIO-32-TiCo2 Rev. E, DIO-32/1-TiCo Rev. E, DIO-8-D12 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E, RSxxx-2 Rev. E, RSxxx-4 Rev. E, SPI-2-D Rev. E, SPI-2-T Rev. E

**Example**

- / -

## P2\_Get\_TiCo\_Status

**P2\_Get\_TiCo\_Status** returns, whether the *TiCo* processor is active on the specified module.

### Syntax

```
#Include ADwinPRO_ALL.inc

ret_val = P2_Get_TiCo_Status(module)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>ret_val</b>	Status of <i>TiCo</i> processor. -3: Error, no <i>TiCo</i> processor available. -2: Error, not a <i>Pro II</i> module. 0: Processor is stopped. 1: Processor is running.	LONG

### Notes

- / -

### See also

[P2\\_Process\\_Status](#), [P2\\_Workload](#)

### Valid for

Aln-32/18-D-TiCo Rev. E, Aln-8/18-TiCo Rev. E, AOut-1/16 Rev. E, AOut-4/16-TiCo Rev. E, AOut-8/16-TiCo Rev. E, ARINC-429 Rev. E, CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, DIO-32-TiCo Rev. E, DIO-32-TiCo2 Rev. E, DIO-32/1-TiCo Rev. E, DIO-8-D12 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E, RSxxx-2 Rev. E, RSxxx-4 Rev. E, SPI-2-D Rev. E, SPI-2-T Rev. E

### Example

- / -



## P2\_GetData\_Long

**P2\_GetData\_Long** reads values from a global array of a *TiCo* processor and writes the values into a specified array.

**Syntax**

```
#Include ADwinPro_All.Inc

P2_GetData_Long(tdrv_datatable[], src_array_no,
                src_array_idx, count, dest_array[],
                dest_array_idx, flowrate)
```

**Parameters**

<code>tdrv_</code>	Array holding settings for data transfer, e.g.	
<code>datatable</code>	module address and <i>TiCo</i> number.	LONG
<code>src_</code>		
<code>array_no</code>	Number (1...16) of global array <code>Data_x</code> on the <i>TiCo</i> processor.	LONG
<code>src_</code>		
<code>array_idx</code>	Index (1...n) of the first element, which is read from the global array <code>src_array_no</code> .	LONG
<code>count</code>	Number (1...n) of values to be transferred.	LONG
<code>dest_</code>		
<code>array[]</code>	Destination array where values are stored.	LONG
<code>dest_</code>		
<code>array_idx</code>	Destination start index (1...n): Array element, from which values are stored.	LONG
<code>flowrate</code>	Evaluation for low priority processes only: Code for data flow rate	LONG
	1: slow.	
	2: medium.	
	3: fast.	

**Notes**

It is to be assured that

- the global array `src_array_no` on the *TiCo* processor is already dimensioned and
- the array `dest_array` has at least `count` elements.

**See also**

[P2\\_Get\\_Par](#), [P2\\_Set\\_Par](#), [P2\\_SetData\\_Long](#), [P2\\_TDrv\\_Init](#)

### Valid for

Aln-32/18-D-TiCo Rev. E, Aln-8/18-TiCo Rev. E, AOut-1/16 Rev. E, AOut-4/16-TiCo Rev. E, AOut-8/16-TiCo Rev. E, ARINC-429 Rev. E, CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, DIO-32-TiCo Rev. E, DIO-32-TiCo2 Rev. E, DIO-32/1-TiCo Rev. E, DIO-8-D12 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E, RSxxx-2 Rev. E, RSxxx-4 Rev. E, SPI-2-D Rev. E, SPI-2-T Rev. E

### Example

```
#Include ADwinPro_All.Inc
#Define module 4           'module address
#Define tico_no 1          'TiCo no.
Dim tset_41[150] As Long   'settings array for data
                             'transfer

Dim Data_4[200] As Long
```

### Init:

```
REM initialize data transfer ADwin CPU <-> TiCo
P2_TDrv_Init(module,tico_no,tset_41)
```

### Event:

```
REM read 120 values from TiCo Data_2 (starting from
REM Data_2[20])
REM into ADwin CPU Data_4 (starting from index 5).
REM flowrate is high
P2_GetData_Long(tset_41,2,20,120,Data_4,5,3)
```

## P2\_Process\_Status

**P2\_Process\_Status** returns the status of a process on a *TiCo* processor of the specified module.

### Syntax

```
#Include ADwinPro_All.Inc

ret_val = P2_Process_Status(module, tico_no,
                             process_no)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>tico_no</b>	Number (1) of <i>TiCo</i> processor on the module.	LONG
<b>process_no</b>	Number (1...4) of <i>TiCo</i> process.	LONG
<b>ret_val</b>	Process status: ≠1: Process is running. 0: Prozess does not run, i.e. it is not loaded, not started or stopped.	LONG

### Notes

- / -

### See also

[P2\\_TiCo\\_Get\\_Processdelay](#), [P2\\_TiCo\\_Set\\_Processdelay](#), [P2\\_TiCo\\_Start\\_Process](#), [P2\\_TiCo\\_Stop\\_Process](#), [P2\\_Workload](#)

### Valid for

Aln-32/18-D-TiCo Rev. E, Aln-8/18-TiCo Rev. E, AOut-1/16 Rev. E, AOut-4/16-TiCo Rev. E, AOut-8/16-TiCo Rev. E, ARINC-429 Rev. E, CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, DIO-32-TiCo Rev. E, DIO-32-TiCo2 Rev. E, DIO-32/1-TiCo Rev. E, DIO-8-D12 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E, RSxxx-2 Rev. E, RSxxx-4 Rev. E, SPI-2-D Rev. E, SPI-2-T Rev. E

### Example

```
#Include ADwinPro_All.Inc
#Define module 4           'module address
#Define tico_no 1          'TiCo no.
Dim Data_4[200] As Long
Dim tset[150] As Long      'settings array for data
                           'transfer
```

### Init:

```
P2_TDrv_Init(module,tico_no,tset)
```

### Event:

```
REM read status of TiCo process 1
Par_1 = P2_Process_Status(module,tico_no,1)
REM if process is running, read TiCo Par_5
If (Par_1 = 1) Then
    Par_2 = P2_Get_Par(module,tico_no,5)
EndIf
```

## P2\_Ringbuffer\_Empty

**P2\_Ringbuffer\_Empty** returns the number of free elements in a write ringbuffer on a *TiCo* processor

### Syntax

```
#Include ADwinPro_All.Inc

ret_val =
    P2_Ringbuffer_Empty(tdrv_datatable[],
        data_no)
```

### Parameters

<b>tdrv_</b>	Array holding settings for data transfer, e.g.	
<b>datatable</b>	module address and <i>TiCo</i> number.	LONG
<b>data_no</b>	Number (1...16) of write ringbuffer <b>Data_n</b> on the <i>TiCo</i> processor.	LONG
<b>ret_val</b>	Number (0...n) of free elements in the write ringbuffer.	LONG

### Notes

For use of **P2\_Get\_TiCo\_RingBuffer**, a previous query with **P2\_Ringbuffer\_Empty** is not required.

### See also

[P2\\_Get\\_TiCo\\_RingBuffer](#), [P2\\_Ringbuffer\\_Full](#), [P2\\_TDrv\\_Init](#)

### Valid for

Aln-32/18-D-TiCo Rev. E, Aln-8/18-TiCo Rev. E, AOut-1/16 Rev. E, AOut-4/16-TiCo Rev. E, AOut-8/16-TiCo Rev. E, ARINC-429 Rev. E, CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, DIO-32-TiCo Rev. E, DIO-32-TiCo2 Rev. E, DIO-32/1-TiCo Rev. E, DIO-8-D12 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E, RSxxx-2 Rev. E, RSxxx-4 Rev. E, SPI-2-D Rev. E, SPI-2-T Rev. E

## Example

- / -

## P2\_Ringbuffer\_Full

**P2\_Ringbuffer\_Full** returns the number of used elements in a read ringbuffer on a *TiCo* processor.

### Syntax

```
#Include ADwinPro_All.Inc

ret_val = P2_Ringbuffer_Full(tdrv_datatable[],
                             data_no)
```

### Parameters

<b>tdrv_</b>	Array holding settings for data transfer, e.g.	
<b>datatable</b>	module address and <i>TiCo</i> number.	LONG
<b>data_no</b>	Number (1...16) of read ringbuffer <b>Data_n</b> on the <i>TiCo</i> processor.	LONG
<b>ret_val</b>	Number (0...n) of used elements in the read ringbuffer.	LONG

### Notes

For use of **P2\_Set\_TiCo\_RingBuffer**, a previous query with **P2\_Ringbuffer\_Full** is not required.

### See also

[P2\\_Set\\_TiCo\\_RingBuffer](#), [P2\\_Ringbuffer\\_Empty](#), [P2\\_TDrv\\_Init](#)

### Valid for

Aln-32/18-D-TiCo Rev. E, Aln-8/18-TiCo Rev. E, AOut-1/16 Rev. E, AOut-4/16-TiCo Rev. E, AOut-8/16-TiCo Rev. E, ARINC-429 Rev. E, CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, DIO-32-TiCo Rev. E, DIO-32-TiCo2 Rev. E, DIO-32/1-TiCo Rev. E, DIO-8-D12 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E, RSxxx-2 Rev. E, RSxxx-4 Rev. E, SPI-2-D Rev. E, SPI-2-T Rev. E

### Example

- / -



## P2\_Set\_Par

**P2\_Set\_Par** sets the value of a global variable **Par\_x** on a *TiCo* processor of the specified module.

### Syntax

```
#Include ADwinPro_All.Inc  
  
P2_Set_Par(module, tico_no, par_no, value)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>tico_no</b>	Number (1) of <i>TiCo</i> processor on the module.	LONG
<b>par_no</b>	Number (1...80) of the global variable.	LONG
<b>value</b>	Value ( $-2^{31} \dots +2^{31}-1$ ) of the globale variable.	LONG

### Notes

**P2\_Set\_Par** sets the value of the global variable, not regarding whether a *TiCo* process is running. Since *ADwin* CPU and *TiCo* processes do not run synchronously, changing the value of a global variable during run-time may be totally unexpected to the *TiCo* process.

If needed, we recommend synchronizing *ADwin* CPU and *TiCo* processes with aid of a software handshake.

### See also

[P2\\_Get\\_Par](#), [P2\\_Get\\_Par\\_Block](#), [P2\\_GetData\\_Long](#), [P2\\_Set\\_Par\\_Block](#), [P2\\_SetData\\_Long](#), [P2\\_TDrv\\_Init](#)

### Valid for

Aln-32/18-D-TiCo Rev. E, Aln-8/18-TiCo Rev. E, AOut-1/16 Rev. E, AOut-4/16-TiCo Rev. E, AOut-8/16-TiCo Rev. E, ARINC-429 Rev. E, CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, DIO-32-TiCo Rev. E, DIO-32-TiCo2 Rev. E, DIO-32/1-TiCo Rev. E, DIO-8-D12 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E, RSxxx-2 Rev. E, RSxxx-4 Rev. E, SPI-2-D Rev. E, SPI-2-T Rev. E

**Example**

```
#Include ADwinPRO_ALL.INC
#Define module 4          'module address
#Define tico_no 1         'TiCo no.
Dim tset[150] As Long     'settings array for data
                           'transfer

Init:
  P2_TDrv_Init(module,tico_no,tset)
  Par_5=200

Event:
  REM write value of Par_5 (ADwin CPU) to TiCo Par_2
  P2_Set_Par(module,tico_no,2,Par_5)
```

## P2\_Set\_Par\_Block

**P2\_Set\_Par\_Block** writes values from an array into a number of global variables **Par\_x** of the *TiCo* processor on the specified module.

### Syntax

```
#Include ADwinPro_All.Inc  
  
P2_Set_Par_Block(module, tico_no, src_array[],  
                 src_array_idx, par_no, par_count)
```

### Parameters

<code>module</code>	Specified module address (1...15).	LONG
<code>tico_no</code>	Number (1) of <i>TiCo</i> processor on the module.	LONG
<code>src_array[]</code>	Source array, from which values are to be transferred.	LONG
<code>src_array_idx</code>	Index of the array element, starting from which values are read from <code>array[]</code> .	LONG
<code>par_no</code>	Index (1...80) of the first global <i>TiCo</i> variable to be written.	LONG
<code>par_count</code>	Number of transferred values.	LONG

### Notes

**P2\_Set\_Par\_Block** sets values of global variables, not regarding whether a *TiCo* process is running or not. Since *ADwin* CPU and *TiCo* processes do not run synchronously, changing the value of a global variable during run-time may be totally unexpected to the *TiCo* process.

If needed, we recommend synchronizing *ADwin* CPU and *TiCo* processes with aid of a software handshake.

### See also

[P2\\_Get\\_Par](#), [P2\\_Get\\_Par\\_Block](#), [P2\\_GetData\\_Long](#), [P2\\_Set\\_ParP2\\_SetData\\_Long](#), [P2\\_TDrv\\_Init](#)

**Valid for**

Aln-32/18-D-TiCo Rev. E, Aln-8/18-TiCo Rev. E, AOut-1/16 Rev. E, AOut-4/16-TiCo Rev. E, AOut-8/16-TiCo Rev. E, ARINC-429 Rev. E, CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, DIO-32-TiCo Rev. E, DIO-32-TiCo2 Rev. E, DIO-32/1-TiCo Rev. E, DIO-8-D12 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E, RSxxx-2 Rev. E, RSxxx-4 Rev. E, SPI-2-D Rev. E, SPI-2-T Rev. E

**Example**

```
#Include ADwinPro_All.Inc
#Define module 4           'module address
#Define tico_no 1          'TiCo no.
Dim tset[150] As Long      'settings array for data
                             'transfer

Dim Data_1[40] As Long
Dim i As Long
```

**Init:**

```
P2_TDrv_Init(module,tico_no,tset)
For i = 1 To 40
    Data_1[i] = i*10
Next
Par_15=1111
Par_16=2222
Par_17=3333
Par_18=4444
Par_19=5555
```

**Event:**

```
REM write 40 values from Data_1 (starting from Data_1[1])
REM into Par_1...Par_40 (TiCo)
P2_Set_Par_Block(module,tico_no,Data_1,1,1,40)
REM write Par_15...Par_19 (ADwin CPU) into Par_50...Par_54
REM (TiCo)
P2_Set_Par_Block(module,tico_no,PAR,15,50,5)
```

### P2\_Set\_TiCo\_RingBuffer

**P2\_Set\_TiCo\_RingBuffer** writes values from an *ADwin* CPU array into a ringbuffer on the *TiCo* processor.

#### Syntax

```
#Include ADwinPro_All.Inc  
ret_val =  
    P2_Set_TiCo_RingBuffer(tdrv_datatable[ ],
```

```
dest_array_no, src_array[], src_array_idx,
maxcount, flowrate, tico_par, struct)
```

### Parameters

<code>tdrv_</code>	Array holding settings for data transfer, e.g.	
<code>datatable</code>	module address and <i>TiCo</i> number.	LONG
<code>dest_array_no</code>	Number (1...16) of read ringbuffer <code>Data_n</code> on the <i>TiCo</i> processor.	FLOAT
<code>src_array[]</code>	Source array, from which values are transferred.	LONG
<code>src_array_idx</code>	Index (1...n) of the first element in <code>src_array[]</code> to be read.	LONG
<code>maxcount</code>	Max. number (1...n) of transferred values.	LONG
<code>flowrate</code>	Evaluation for low priority processes only: Code for data flow rate 1: slow. 2: medium. 3: fast.	LONG
<code>tico_par</code>	Code number to transfer the write pointer to the <i>TiCo</i> processor. 1...80: Number (1...80) of global variable <code>Par_n</code> of the <i>TiCo</i> processor, into which the current write pointer value is written. 0: Write pointer value is not transferred.	LONG
<code>struct</code>	Code for measurement records: 0: Read arbitrary number of values. >0: The number of transferred values must be a multiple of <code>struct</code> .	LONG
<code>ret_val</code>	Success status of instruction: -1: Error: The <i>TiCo</i> read ringbuffer is dimensioned wrongly. ≥0: Successful. The return value equals the number of transferred values.	LONG

**Notes**

The instruction does not write more than `maxcount` values. If the ringbuffer has less empty elements, only the empty ringbuffer elements are filled.

While writing into a ringbuffer, `P2_Set_TiCo_RingBuffer` stores the last writing position, the write pointer, into the array `tdrv_data-table[]`. If the instruction `RingBuffer_Full` of the *TiCo* processor is to run correctly, the number of a global variable has to be set in `tico_par`. Thus, the write pointer value is transferred into the global variable of the *TiCo* processor.

**See also**

[P2\\_Get\\_TiCo\\_RingBuffer](#), [P2\\_TDrv\\_Init](#)

**Valid for**

Aln-32/18-D-TiCo Rev. E, Aln-8/18-TiCo Rev. E, AOut-1/16 Rev. E, AOut-4/16-TiCo Rev. E, AOut-8/16-TiCo Rev. E, ARINC-429 Rev. E, CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, DIO-32-TiCo Rev. E, DIO-32-TiCo2 Rev. E, DIO-32/1-TiCo Rev. E, DIO-8-D12 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E, RSxxx-2 Rev. E, RSxxx-4 Rev. E, SPI-2-D Rev. E, SPI-2-T Rev. E

**Example**

- / -

## **P2\_SetData\_Long**

**P2\_SetData\_Long** reads values from an array and writes them into a global array of a *TiCo* processor on the specified module.



## Syntax

```
#Include ADwinPro_All.Inc

P2_SetData_Long(tdrv_datatable[],
    dest_array_no, dest_array_idx, count,
    src_array[], src_array_idx, flowrate)
```

## Parameters

<code>tdrv_datatable</code>	Array holding settings for data transfer, e.g. module address and <i>TiCo</i> number.	LONG
<code>dest_array_no</code>	Number (1...16) of global array <code>Data_x</code> on the <i>TiCo</i> processor.	LONG
<code>dest_array_idx</code>	Index (1...n) of the first element, to be written in the global array <code>dest_array_no</code> .	LONG
<code>count</code>	Number (1...n) of transferred values.	LONG
<code>src_array</code>	Source array, from where values are read.	LONG
<code>src_array_idx</code>	Source start index (1...n): Array element, which is read first.	LONG
<code>flowrate</code>	Evaluation for low priority processes only: Code for data flow rate 1: slow. 2: medium. 3: fast.	LONG

## Notes

It is to be assured that

- the global array `dest_array_no` on the *TiCo* processor is already dimensioned and
- the array `src_array` has at least `count` elements.

## See also

[P2\\_Get\\_Par](#), [P2\\_GetData\\_Long](#), [P2\\_Set\\_Par](#), [P2\\_TDrv\\_Init](#)

**Valid for**

Aln-32/18-D-TiCo Rev. E, Aln-8/18-TiCo Rev. E, AOut-1/16 Rev. E, AOut-4/16-TiCo Rev. E, AOut-8/16-TiCo Rev. E, ARINC-429 Rev. E, CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, DIO-32-TiCo Rev. E, DIO-32-TiCo2 Rev. E, DIO-32/1-TiCo Rev. E, DIO-8-D12 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E, RSxxx-2 Rev. E, RSxxx-4 Rev. E, SPI-2-D Rev. E, SPI-2-T Rev. E

**Example**

```
#Include ADwinPro_All.Inc
#Define module 4           'module address
#Define tico_no 1          'TiCo no.
Dim tset[150] As Long      'settings array for data
                             'transfer

Dim tset_41[150] As Long
Dim Data_1[150] As Long
Dim i As Long
```

**Init:**

```
REM initialize data transfer ADwin CPU <-> TiCo
P2_TDrv_Init(module,tico_no,tset_41)
For i = 1 To 80
    Data_1[i] = i
Next
```

**Event:**

```
REM write 120 values
REM from Data_1 of ADwin CPU, starting from Data_1[5]
REM into TiCo Data_2, starting from Data_2[1]
REM flowrate is high
P2_SetData_Long(tset_41,2,1,120,Data_1,5,3)

-/-
-/-
```

## P2\_TDrv\_Init

**P2\_TDrv\_Init** initializes the data transfer between ADwin CPU and a *TiCo* processor.

### Syntax

```
#Include ADwinPro_All.Inc

REM define settings array for TiCo y on module x
Dim tdrv_datatable[150] As Long

P2_TDrv_Init(module, tico_no,tdrv_datatable[])
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>tico_no</b>	Number (1) of <i>TiCo</i> processor on the module.	LONG
<b>tdrv_datatable</b>	Array holding settings for data transfer, e.g. module address and <i>TiCo</i> number.	LONG
<b>[]</b>		

### Notes

The instruction must be processed before data transfer between ADwin CPU and *TiCo*. The instruction should be set into the **Init:** section.

Most instructions accessing a *TiCo* processor require initialization of data transfer.

Initialization must be run for each *TiCo* processor separately. For each processor an array `tdrv_datatable[]` with 150 elements has to be dimensioned, too.

The array `tdrv_datatable[]` is used by **P2\_GetData\_Long**, **P2\_SetData\_Long**, **P2\_Workload**.

### See also

[P2\\_Get\\_Par](#), [P2\\_Get\\_Par\\_Block](#), [P2\\_TiCo\\_Get\\_Processdelay](#), [P2\\_GetData\\_Long](#), [P2\\_Set\\_Par](#), [P2\\_Set\\_Par\\_Block](#), [P2\\_TiCo\\_Set\\_Processdelay](#), [P2\\_SetData\\_Long](#)

**Valid for**

Aln-32/18-D-TiCo Rev. E, Aln-8/18-TiCo Rev. E, AOut-1/16 Rev. E, AOut-4/16-TiCo Rev. E, AOut-8/16-TiCo Rev. E, ARINC-429 Rev. E, CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, DIO-32-TiCo Rev. E, DIO-32-TiCo2 Rev. E, DIO-32/1-TiCo Rev. E, DIO-8-D12 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E, RSxxx-2 Rev. E, RSxxx-4 Rev. E, SPI-2-D Rev. E, SPI-2-T Rev. E

**Example**

```
#Include ADwinPro_All.Inc
#Define module_a 4           'address of module a
#Define tico_a 1             'TiCo no.
#Define module_b 7           'address of module b
#Define tico_b 1             'TiCo no.
Dim Data_4[200] As Long
Dim Data_5[1000] As Long
Dim tset_41[150] As Long
Dim tset_71[150] As Long
```

**Init:**

```
REM initialize data transfer ADwin CPU <-> TiCo
P2_TDrv_Init(module_a,tico_a,tset_41)
P2_TDrv_Init(module_b,tico_b,tset_71)
```

**Event:**

```
REM read 120 values from module a, TiCo Data_2 (starting
REM from
REM Data_2[20]) into ADwin CPU Data_4 (starting from
REM index 5).
REM flowrate is high
P2_GetData_Long(tset_41,2,20,120,Data_4,5,3)
REM read 800 values from module b, TiCo Data_7 (starting
REM from
REM Data_7[9]) into ADwin CPU Data_5 (starting from index
REM 1).
REM flowrate is high
P2_GetData_Long(tset_71,7,9,800,Data_5,1,3)
```

## P2\_TiCo\_Get\_Processdelay

**P2\_TiCo\_Get\_Processdelay** returns the **Processdelay** (cycle time) of a process on a *TiCo* processor on the specified module.

### Syntax

```
#Include ADwinPro_All.Inc

ret_val = P2_TiCo_Get_Processdelay (
    tdrv_datatable[], process_no)
```

### Parameters

<b>tdrv_</b>	Array holding settings for data transfer, e.g.	
<b>datatable</b>	module address and <i>TiCo</i> number.	LONG
<b>process_</b>	Number (1...4) of <i>TiCo</i> process.	LONG
<b>no</b>		
<b>ret_val</b>	Current cycle time ( $-2^{31} \dots +2^{31}-1$ ) of the <i>TiCo</i> processor in clock cycles. One clock cycle takes 20ns.	LONG

### Notes

Using a timer controlled process, the **Event:** section is triggered by the internal counter cyclical and with fixed time interval. The time interval between 2 trigger signals, called cycle time or **Processdelay**, is measured in counter clock cycles.

### See also

[P2\\_Process\\_Status](#), [P2\\_TiCo\\_Set\\_Processdelay](#), [P2\\_TDrv\\_Init](#)

### Valid for

Aln-32/18-D-TiCo Rev. E, Aln-8/18-TiCo Rev. E, AOut-1/16 Rev. E, AOut-4/16-TiCo Rev. E, AOut-8/16-TiCo Rev. E, ARINC-429 Rev. E, CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, DIO-32-TiCo Rev. E, DIO-32-TiCo2 Rev. E, DIO-32/1-TiCo Rev. E, DIO-8-D12 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E, RSxxx-2 Rev. E, RSxxx-4 Rev. E, SPI-2-D Rev. E, SPI-2-T Rev. E

**Example**

```
#Include ADwinPro_All.Inc
#Define module 4           'module address
#Define tico_no 1          'TiCo no.
Dim tset[150] As Long      'settings array for data
                           'transfer

Init:
    P2_TDrv_Init(module,tico_no,tset)

Event:
    REM read processdelay of process 1 into Par_1
    Par_1 = P2_TiCo_Get_Processdelay(tset,1)
```

## P2\_TiCo\_Get\_Process\_Error

**P2\_TiCo\_Get\_Process\_Error** returns the most recently occurred error of the selected process of a *TiCo* processor.

### Syntax

```
#Include ADwinPro_All.Inc

ret_val = P2_TiCo_Get_Process_Error(
    tdrv_datatable[], process_no)
```

### Parameters

<b>tdrv_</b>	Array holding settings for data transfer, e.g.	
<b>datatable</b>	module address and <i>TiCo</i> number.	LONG
<b>process_</b>	Number (1...4) of <i>TiCo</i> process.	LONG
<b>no</b>		
<b>ret_val</b>	Number of the previously occurred error in the process:	LONG
	0: no error	
	10: Accessing a too high element number of a global array.	
	11: Accessing a too small element number ( $\leq 0$ ) of a global array.	
	12: Accessing a too high element number of a local array.	
	13: Accessing a too small element number ( $\leq 0$ ) of a local array.	

### Notes

The return value is defined only if debug mode is enabled (see [Debug mode Option](#), page 71). The variable is read-only.

### See also

[P2\\_Process\\_Status](#), [P2\\_TiCo\\_Set\\_Processdelay](#), [P2\\_TDrv\\_Init](#)

**Valid for**

- / -

**Example**

```
#Include ADwinPro_All.Inc
#Define module 4           'module address
#Define tico_no 1          'TiCo no.
Dim tset[150] As Long      'settings array for data
                           'transfer

Init:
    P2_TDrv_Init(module,tico_no,tset)

Event:
    REM read recent process error of process 1 into Par_1
    Par_1 = P2_TiCo_Get_Process_Error(tset,1)
```



## P2\_TiCo\_Flash

**P2\_TiCo\_Flash** transfers a *TiCoBasic* binary file from an array into the flash memory of a *TiCo* processor.

### Syntax

```
#Include ADwinPRO_ALL.inc  
  
ret_val = P2_TiCo_Flash(module, tico_no,  
                        array[])
```

### Parameters

module	Specified module address (1...15).	LONG
tico_no	Number (1) of <i>TiCo</i> processor on the module.	LONG
array[]	Array holding the data to be transferred.	LONG
ret_val	Status of data transfer: 0: Data transfer successful. -2: no module at this address or module has no <i>TiCo</i> processor. -1: Error: Instruction may only be used with low priority. 1: Error: Invalid procoessor number. 2: Error: Program memory too small. 3: Error: Data memory too small. 4: Error: Program and data memory too small. 5: Error: Wrong password. 6: Error: External memory memory too small. 7: Error: Flash memory too small. 12: Error: Binary file invalid for this <i>TiCo</i> processor.	LONG

### Notes

Use **P2\_TiCo\_Flash** only in low priority processes.

The instruction **P2\_TiCo\_Flash** is used to transfer a *TiCoBasic* binary file—a compiled *TiCo* program—into the flash memory of a *TiCo* processor. The following steps are required:

- Create a binary file in the development environment *TiCoBasic* using Build► Make Bin File.
- Transfer the binary file into a global array of the *ADwin* CPU. A useful means is the function `File2Data` (or `LoadFromFile`), which is provided in several programming languages.
- Transfer the data of the global `array []` into the flash memory using **P2\_TiCo\_Flash**.
- If the bootloader option is enabled, the process of the *TiCoBasic* binary file is started automatically after the next start-up of the *ADwin* hardware.

If `array []` does not contain a *TiCoBasic* binary file the return value is invalid.

#### See also

[P2\\_TiCo\\_Load](#)

#### Valid for

Aln-32/18-D-TiCo Rev. E, Aln-8/18-TiCo Rev. E, AOut-1/16 Rev. E, AOut-4/16-TiCo Rev. E, AOut-8/16-TiCo Rev. E, ARINC-429 Rev. E, CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, DIO-32-TiCo Rev. E, DIO-32-TiCo2 Rev. E, DIO-32/1-TiCo Rev. E, DIO-8-D12 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E, RSxxx-2 Rev. E, RSxxx-4 Rev. E, SPI-2-D Rev. E, SPI-2-T Rev. E

#### Example

- / -

## P2\_TiCo\_Load

**P2\_TiCo\_Load** transfers a *TiCoBasic* binary file from an array into the memory of a *TiCo* processor and starts the process.

### Syntax

```
#Include ADwinPRO_ALL.inc  
ret_val = P2_TiCo_Load(module,tico_no,array[])
```

### Parameters

module	Specified module address (1...15).	LONG
tico_no	Number (1) of <i>TiCo</i> processor on the module.	LONG
array[]	Array holding the data to be transferred.	LONG
ret_val	Status of data transfer: 0: Data transfer successful. -1: Error: Instruction may only be used with low priority. -2: no module at this address or module has no <i>TiCo</i> processor. 1: Error: Invalid processor number. 2: Error: Program memory too small. 3: Error: Data memory too small. 4: Error: Program and data memory too small. 5: Error: Wrong password. 6: Error: External memory too small. 10: Error: Required external SRAM too small. 11: Error: Array contains no valid <i>TiCoBasic</i> binary file. 12: Error: Binary file invalid for this <i>TiCo</i> processor.	LONG

### Notes

Use **P2\_TiCo\_Load** only in low priority processes.

The instruction **P2\_TiCo\_Load** is used to transfer a *TiCoBasic* binary file—a compiled *TiCo* program—into the memory of a *TiCo* processor. The following steps are required:

- Create a binary file in the development environment *TiCoBasic* using Build ► Make Bin File.
- Transfer the binary file into a global array of the *ADwin* CPU. A useful means is the function `File2Data` (or `LoadFromFile`), which is provided in several programming languages.
- Transfer the data of the global `array[]` into the memory using **P2\_TiCo\_Load**.
- The process of the *TiCoBasic* binary file is automatically started.

### See also

[P2\\_TiCo\\_Flash](#)

### Valid for

Aln-32/18-D-TiCo Rev. E, Aln-8/18-TiCo Rev. E, AOut-1/16 Rev. E, AOut-4/16-TiCo Rev. E, AOut-8/16-TiCo Rev. E, ARINC-429 Rev. E, CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, DIO-32-TiCo Rev. E, DIO-32-TiCo2 Rev. E, DIO-32/1-TiCo Rev. E, DIO-8-D12 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E, RSxxx-2 Rev. E, RSxxx-4 Rev. E, SPI-2-D Rev. E, SPI-2-T Rev. E

### Example

see also C:\ADwin\ADbasic\samples\_ADwin\_Proll

## P2\_TiCo\_Restart

**P2\_TiCo\_Restart** stops the *TiCo* processors on the specified modules and restarts them afterwards.

### Syntax

```
#Include ADwinPro_All.Inc
P2_TiCo_Restart (module_pattern)
```

### Parameters

**module\_pattern** Bit pattern to address the modules, where *TiCo* processors are to be reset: LONG  
 Bit = 0: Ignore module.  
 Bit = 1: Reset *TiCo* processor on the module.

Bits in <b>module_pattern</b>	31:15	14	13	...	01	00
Module address	–	15	14	...	2	1

### Notes

Stopping immediately interrupts any running process as well as counters on the addressed *TiCo* processors. Data are not changed by stopping.

The *TiCo* processors on the addressed modules are being started at the same time. Thus, the instruction can be used to synchronize the addressed *TiCo* processors.

### See also

[P2\\_TiCo\\_Stop](#), [P2\\_TiCo\\_Start](#)

### Valid for

Aln-32/18-D-TiCo Rev. E, Aln-8/18-TiCo Rev. E, AOut-1/16 Rev. E, AOut-4/16-TiCo Rev. E, AOut-8/16-TiCo Rev. E, ARINC-429 Rev. E, CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, DIO-32-TiCo Rev. E, DIO-32-TiCo2 Rev. E, DIO-32/1-TiCo Rev. E, DIO-8-D12 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E, RSxxx-2 Rev. E, RSxxx-4 Rev. E, SPI-2-D Rev. E, SPI-2-T Rev. E

**Example**

- / -

## P2\_TiCo\_Set\_Processdelay

**P2\_TiCo\_Set\_Processdelay** sets the **Processdelay** (cycle time) of a *TiCo* process on the specified module.

### Syntax

```
#Include ADwinPro_All.Inc

P2_TiCo_Set_Processdelay(tdrv_datatable[],
    process_no, value)
```

### Parameters

<b>tdrv_</b>	Array holding settings for data transfer, e.g.	
<b>datatable</b>	module address and <i>TiCo</i> number.	LONG
<b>process_</b>	Number (1...4) of <i>TiCo</i> process.	LONG
<b>no</b>		
<b>value</b>	Value to be set for <b>Processdelay</b> .	LONG

### Notes

- / -

### See also

[P2\\_TiCo\\_Get\\_Processdelay](#), [P2\\_Process\\_Status](#), [P2\\_TDrv\\_Init](#)

### Valid for

Aln-32/18-D-TiCo Rev. E, Aln-8/18-TiCo Rev. E, AOut-1/16 Rev. E, AOut-4/16-TiCo Rev. E, AOut-8/16-TiCo Rev. E, ARINC-429 Rev. E, CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, DIO-32-TiCo Rev. E, DIO-32-TiCo2 Rev. E, DIO-32/1-TiCo Rev. E, DIO-8-D12 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E, RSxxx-2 Rev. E, RSxxx-4 Rev. E, SPI-2-D Rev. E, SPI-2-T Rev. E

**Example**

```
#Include ADwinPro_All.Inc
#Define module 4          'module address
#Define tico_no 1         'TiCo no.
Dim tset[150] As Long     'settings array for data
                           'transfer

Init:
  P2_TDrv_Init(module,tico_no,tset)
  Processdelay = 6000 'set cycle time

Event:
  REM set TiCo processdelay to run with same cycle time as
  REM the T11 process
  P2_TiCo_Set_Processdelay(tset,1,Processdelay/6)
```



## P2\_TiCo\_Start

**P2\_TiCo\_Start** starts the *TiCo* processors on the specified modules.

### Syntax

```
#Include ADwinPro_All.Inc
P2_TiCo_Start(module_pattern)
```

### Parameters

**module\_pattern** Bit pattern to address the modules, where *TiCo* processors are to be started: **LONG**  
 Bit = 0: Ignore module.  
 Bit = 1: Start *TiCo* processor on the module.

Bits in <b>module_pattern</b>	31:15	14	13	...	01	00
Module address	—	15	14	...	2	1

### Notes

The *TiCo* processors on the addressed modules are being started at the same time. Thus, the instruction can be used to synchronize the addressed *TiCo* processors.

### See also

[P2\\_TiCo\\_Stop](#), [P2\\_TiCo\\_Restart](#)

### Valid for

Aln-32/18-D-TiCo Rev. E, Aln-8/18-TiCo Rev. E, AOut-1/16 Rev. E, AOut-4/16-TiCo Rev. E, AOut-8/16-TiCo Rev. E, ARINC-429 Rev. E, CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, DIO-32-TiCo Rev. E, DIO-32-TiCo2 Rev. E, DIO-32/1-TiCo Rev. E, DIO-8-D12 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E, RSxxx-2 Rev. E, RSxxx-4 Rev. E, SPI-2-D Rev. E, SPI-2-T Rev. E

### Example

- / -

## P2\_TiCo\_Start\_Process

**P2\_TiCo\_Start\_Process** starts a process on a *TiCo* processor.

### Syntax

```
#Include ADwinPro_All.Inc

P2_TiCo_Start_Process (tdrv_datatable[],
                      process_no)
```

### Parameters

<b>tdrv_</b>	Array holding settings for data transfer, e.g.	
<b>datatable</b>	module address and <i>TiCo</i> number.	LONG
<b>process_</b>	Number (1...4) of <i>TiCo</i> process.	LONG
<b>no</b>		

### Notes

The process must already be loaded to the *TiCo* processor.

### See also

[P2\\_TiCo\\_Get\\_Processdelay](#), [P2\\_Process\\_Status](#), [P2\\_TiCo\\_Set\\_Processdelay](#), [P2\\_TiCo\\_Start\\_Process](#), [P2\\_TiCo\\_Stop\\_Process](#), [P2\\_Workload](#)

### Valid for

Aln-32/18-D-TiCo Rev. E, Aln-8/18-TiCo Rev. E, AOut-1/16 Rev. E, AOut-4/16-TiCo Rev. E, AOut-8/16-TiCo Rev. E, ARINC-429 Rev. E, CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, DIO-32-TiCo Rev. E, DIO-32-TiCo2 Rev. E, DIO-32/1-TiCo Rev. E, DIO-8-D12 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E, RSxxx-2 Rev. E, RSxxx-4 Rev. E, SPI-2-D Rev. E, SPI-2-T Rev. E

### Example

```
#Include ADwinPro_All.Inc
#Define module 4           'module address
#Define tico_no 1          'TiCo no.
Dim tset[150] As Long      'settings array for data
                           'transfer

Init:
  Par_1 = 0
  Processdelay = 100000
  P2_TDrv_Init(module,tico_no,tset)
  REM start TiCo process in parallel to ADwin CPU process
  P2_Tico_Start_Process(tset,1)

Event:
  REM ... program

Finish:
  REM stop TiCo process in parallel to ADwin CPU process
  P2_Tico_Stop_Process(tset,1)
```

## P2\_TiCo\_Stop

**P2\_TiCo\_Stop** stops the *TiCo* processors on the specified modules.

### Syntax

```
#Include ADwinPro_All.Inc
P2_TiCo_Stop (module_pattern)
```

### Parameters

**module\_** Bit pattern to address the modules, where *TiCo* LONG  
**pattern** processors are to be stopped:  
 Bit = 0: Ignore module.  
 Bit = 1: Stop *TiCo* processor on the module.

Bits in <b>module_pattern</b>	31:15	14	13	...	01	00
Module address	–	15	14	...	2	1

### Notes

Stopping immediately interrupts any running process as well as counters on the addressed *TiCo* processors. Data are not changed by stopping.

### See also

[P2\\_TiCo\\_Start](#), [P2\\_TiCo\\_Restart](#)

### Valid for

Aln-32/18-D-TiCo Rev. E, Aln-8/18-TiCo Rev. E, AOut-1/16 Rev. E, AOut-4/16-TiCo Rev. E, AOut-8/16-TiCo Rev. E, ARINC-429 Rev. E, CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, DIO-32-TiCo Rev. E, DIO-32-TiCo2 Rev. E, DIO-32/1-TiCo Rev. E, DIO-8-D12 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E, RSxxx-2 Rev. E, RSxxx-4 Rev. E, SPI-2-D Rev. E, SPI-2-T Rev. E

### Example

- / -

## P2\_TiCo\_Stop\_Process

**P2\_TiCo\_Stop\_Process** stops a process on a *TiCo* processor.

### Syntax

```
#Include ADwinPro_All.Inc  
  
P2_TiCo_Stop_Process(tdrv_datatable[],  
                    process_no)
```

### Parameters

<code>tdrv_</code>	Array holding settings for data transfer, e.g.	
<code>datatable</code>	module address and <i>TiCo</i> number.	LONG
<code>process_</code>	Number (1...4) of <i>TiCo</i> process.	LONG
<code>no</code>		

### Notes

The process must already be loaded to the *TiCo* processor.

### See also

[P2\\_TiCo\\_Get\\_Processdelay](#), [P2\\_Process\\_Status](#), [P2\\_TiCo\\_Set\\_Processdelay](#), [P2\\_TiCo\\_Start\\_Process](#), [P2\\_Workload](#)

### Valid for

Aln-32/18-D-TiCo Rev. E, Aln-8/18-TiCo Rev. E, AOut-1/16 Rev. E, AOut-4/16-TiCo Rev. E, AOut-8/16-TiCo Rev. E, ARINC-429 Rev. E, CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, DIO-32-TiCo Rev. E, DIO-32-TiCo2 Rev. E, DIO-32/1-TiCo Rev. E, DIO-8-D12 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E, RSxxx-2 Rev. E, RSxxx-4 Rev. E, SPI-2-D Rev. E, SPI-2-T Rev. E

**Example**

```
#Include ADwinPro_All.Inc
#Define module 4           'module address
#Define tico_no 1          'TiCo no.
Dim tset[150] As Long      'settings array for data
                           'transfer

Init:
  Par_1 = 0
  Processdelay = 100000
  P2_TDrv_Init(module,tico_no,tset)
  REM start TiCo process in parallel to ADwin CPU process
  P2_Tico_Start_Process(tset,1)

Event:
  REM ... program

Finish:
  REM stop TiCo process in parallel to ADwin CPU process
  P2_Tico_Stop_Process(tset,1)
```

## P2\_Workload

**P2\_Workload** returns the workload of a *TiCo* processor on the specified module.

### Syntax

```
#Include ADwinPro_All.Inc  
ret_val = P2_Workload(tdrv_datatable[])
```

### Parameters

<code>tdrv_datatable</code>	Array holding settings for data transfer, e.g. module address and <i>TiCo</i> number.	LONG
<code>ret_val</code>	Processor workload in percent (0.0 ... 100.0) or error value: <0: <i>TiCo</i> processor is stopped or module has no <i>TiCo</i> processor or no module at the given module address.	FLOAT

### Notes

The return value is the average processor workload in the time between the previous and the current call of **Workload**. Therefore, the return value of the first call in a program is invalid.

The shortest time between two instruction calls should be at least 100 times of **Processdelay**. Otherwise, the return value may have an error of more than 1%.

If the previous call of **Workload** dates back more than 85 seconds, the return value is invalid. Simply call the instruction a second time to receive a valid return value.

### See also

[P2\\_TiCo\\_Get\\_Processdelay](#), [P2\\_Process\\_Status](#), [P2\\_TiCo\\_Set\\_Processdelay](#), [P2\\_TiCo\\_Stop\\_Process](#), [P2\\_TDrv\\_Init](#)

**Valid for**

Aln-32/18-D-TiCo Rev. E, Aln-8/18-TiCo Rev. E, AOut-1/16 Rev. E, AOut-4/16-TiCo Rev. E, AOut-8/16-TiCo Rev. E, ARINC-429 Rev. E, CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, DIO-32-TiCo Rev. E, DIO-32-TiCo2 Rev. E, DIO-32/1-TiCo Rev. E, DIO-8-D12 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E, RSxxx-2 Rev. E, RSxxx-4 Rev. E, SPI-2-D Rev. E, SPI-2-T Rev. E

**Example**

```
#Include ADwinPro_All.Inc
#Define module 4           'module address
#Define tico_no 1          'TiCo no.
Dim tset[150] As Long      'settings array for data
                           'transfer
```

**Init:**

```
P2_TDrv_Init(module,tico_no,tset)
```

**Event:**

```
REM read TiCo workload
FPar_1 = P2_Workload(tset)
```



### 8 How to Solve Problems?

If problems already occur during installation, please refer to the documentation for your *ADwin* system. Make sure all settings have been carried out properly and completely. Also check if the base address, the processor type, etc. are set correctly in the menu `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.




## Appendix

### A.1 Short-Cuts in TiCoBasic

To display short-cuts of code snippets, open <TiCoBasicCS.xml> in the folder C:\ADwin\TiCoBasic\Common\ with a browser.

Short cut key	Function	Matching menu item
F1	Show help topic for marked instruction.	
CTRL-F1	Show online help content.	Help►Content
F2	Show declaration of marked instruction.	
CTRL-F2	Jump to declaration of marked instruction.	
F3	Find next forward.	Edit►Find Next
SHIFT-F3	Find next backwards.	
CTRL-F3	Find Text at cursor position forward.	
CTRL-SHIFT-F3	Find Text at cursor position backwards.	
CTRL-F5	Initialize <i>ADwin</i> -CPU for communication with <i>TiCo</i> processor.	
CTRL-SHIFT-F5	Stop and reset <i>TiCo</i> processor at once.	
F6	Create library.	Build►Make Lib File
F7	Create binary file.	Build►Make Bin File
F8	Compile source code.	Build►Compile
CTRL-F8	Start process.	
F9	Stop process.	
CTRL-SPACE	Insert or complete a declaration.	

Short cut key	Function	Matching menu item
CTRL-SHIFT-SPACE	Show parameters of a sub / function.	
CTRL-A	Select all.	Edit ► Select All
CTRL-B	Comment marked lines	Source context menu: Comment Block
CTRL-SHIFT-B	Uncomment marked lines	Source context menu: Uncomment Block
CTRL-C	Copy.	Edit ► Copy
CTRL-F	Find text.	Edit ► Find
CTRL-G	Jump to a line.	
CTRL-H	Replace text.	Edit ► Replace
CTRL-I	Indent marked lines	Source context menu: Indent
CTRL-SHIFT-I	Outdent marked lines	Source context menu: Outdent
CTRL-N	New source code file.	File ► New
CTRL-O	Open source code file.	File ► Open
CTRL-P	Print source code file.	File ► Print
CTRL-R	Colour mark used parameters	Parameter window: Icon 
CTRL-S	Save source code file.	File ► Save
CTRL-U	Lower case.	
CTRL-SHIFT-U	Upper case.	
CTRL-V	Paste.	Edit ► Paste
CTRL-X	Cut.	Edit ► Cut
CTRL-Z	Undo input.	Edit ► Undo
CTRL-SHIFT-Z	Redo input.	Edit ► Redo
CTRL-K + K	Insert / delete bookmark.	
CTRL-K + N	Jump to next bookmark.	
CTRL-K + P	Jump to previous bookmark.	

Short cut key	Function	Matching menu item
CTRL-K + X	Insert a code snippet.	

Legend:

A-B: Press keys A and B at the same time.

A+B: Press key A first, release and then press key B.

## A.2 ASCII-Character Set

<b>NUL</b>	<b>SOH</b>	<b>STX</b>	<b>ETX</b>	<b>EOT</b>	<b>ENQ</b>	<b>ACK</b>	<b>BEL</b>
00h 0	01h 1	02h 2	03h 3	04h 4	05h 5	06h 6	07h 7
<b>BS<sup>1</sup></b>	<b>TAB<sup>2</sup></b>	<b>LF<sup>3</sup></b>	<b>VT</b>	<b>FF</b>	<b>CR<sup>4</sup></b>	<b>SO</b>	<b>SI</b>
08h 8	09h 9	0Ah 10	0Bh 11	0Ch 12	0Dh 13	0Eh 14	0Fh 15
<b>DLE</b>	<b>DC1</b>	<b>DC2</b>	<b>DC3</b>	<b>DC4</b>	<b>NAK</b>	<b>SYN</b>	<b>ETB</b>
10h 16	11h 17	12h 18	13h 19	14h 20	15h 21	16h 22	17h 23
<b>CAN</b>	<b>EM</b>	<b>SUB</b>	<b>ESC</b>	<b>FS</b>	<b>GS</b>	<b>RS</b>	<b>US</b>
18h 24	19h 25	1Ah 26	1Bh 27	1Ch 28	1Dh 29	1Eh 30	1Fh 31
<b>SPC<sup>5</sup></b>	<b>!</b>	<b>"</b>	<b>#</b>	<b>\$</b>	<b>%</b>	<b>&amp;</b>	<b>'</b>
20h 32	21h 33	22h 34	23h 35	24h 36	25h 37	26h 38	27h 39
<b>(</b>	<b>)</b>	<b>*</b>	<b>+</b>	<b>,</b>	<b>-</b>	<b>.</b>	<b>/</b>
28h 40	29h 41	2Ah 42	2Bh 43	2Ch 44	2Dh 45	2Eh 46	2Fh 47
<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>
30h 48	31h 49	32h 50	33h 51	34h 52	35h 53	36h 54	37h 55
<b>8</b>	<b>9</b>	<b>:</b>	<b>;</b>	<b>&lt;</b>	<b>=</b>	<b>&gt;</b>	<b>?</b>
38h 56	39h 57	3Ah 58	3Bh 59	3Ch 60	3Dh 61	3Eh 62	3Fh 63
<b>@</b>	<b>A</b>	<b>B</b>	<b>C</b>	<b>D</b>	<b>E</b>	<b>F</b>	<b>G</b>
40h 64	41h 65	42h 66	43h 67	44h 68	45h 69	46h 70	47h 71
<b>H</b>	<b>I</b>	<b>J</b>	<b>K</b>	<b>L</b>	<b>M</b>	<b>N</b>	<b>O</b>
48h 72	49h 73	4Ah 74	4Bh 75	4Ch 76	4Dh 77	4Eh 78	4Fh 79
<b>P</b>	<b>Q</b>	<b>R</b>	<b>S</b>	<b>T</b>	<b>U</b>	<b>V</b>	<b>W</b>
50h 80	51h 81	52h 82	53h 83	54h 84	55h 85	56h 86	57h 87
<b>X</b>	<b>Y</b>	<b>Z</b>	<b>[</b>	<b>\</b>	<b>]</b>	<b>^</b>	<b>_</b>
58h 88	59h 89	5Ah 90	5Bh 91	5Ch 92	5Dh 93	5Eh 94	5Fh 95
<b>`</b>	<b>a</b>	<b>b</b>	<b>c</b>	<b>d</b>	<b>e</b>	<b>f</b>	<b>g</b>
60h 96	61h 97	62h 98	63h 99	64h 100	65h 101	66h 102	67h 103
<b>h</b>	<b>i</b>	<b>j</b>	<b>k</b>	<b>l</b>	<b>m</b>	<b>n</b>	<b>o</b>
68h 104	69h 105	6Ah 106	6Bh 107	6Ch 108	6Dh 109	6Eh 110	6Fh 111
<b>p</b>	<b>q</b>	<b>r</b>	<b>s</b>	<b>t</b>	<b>u</b>	<b>v</b>	<b>w</b>
70h 112	71h 113	72h 114	73h 115	74h 116	75h 117	76h 118	77h 119
<b>x</b>	<b>y</b>	<b>z</b>	<b>{</b>	<b> </b>	<b>}</b>	<b>~</b>	<b>␣</b>
78h 120	79h 121	7Ah 122	7Bh 123	7Ch 124	7Dh 125	7Eh 126	7Fh 127

<sup>1</sup> Backspace, <sup>2</sup> Tabulator, <sup>3</sup> Linefeed,  
<sup>4</sup> Carriage Return, <sup>5</sup> Space

### A.3 License Agreement

Between the buyer of *TiCoBasic*—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 *TiCoBasic* (hereinafter termed *TiCoBasic* software) as well as the printed user manual "*TiCoBasic: 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 *TiCoBasic* software only on a single computer and only in one single location. The Licensee may transfer the *TiCoBasic* 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 *TiCoBasic* 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 *TiCoBasic* software without prior written consent of Jaeger Messtechnik GmbH,
- 4. electronically transfer the *TiCoBasic* software from one computer to another over a network or a data transfer channel,
- 5. change or modify, translate, reverse engineer, decompile or disassemble the *TiCoBasic* software without prior written consent of Jaeger Messtechnik GmbH.

## 6. OWNERSHIP

- 6.1 Upon purchasing the product, only title to the physical storage device, where the *TiCoBasic* software has been stored, is passed to the Licensee. No title to the rights of the *TiCoBasic* software itself is passed to the Licensee.
- 6.2 Jaeger Messtechnik GmbH reserves all rights for publication, copying, processing and commercialization of the *TiCoBasic* software.

## 7. COPYRIGHTS

- 7.1 The *TiCoBasic* software and the printed materials are protected by copyright.

For backup purposes, the Licensee may generate a single copy of the *TiCoBasic* software. He must reproduce the copyright notice of Jaeger Messtechnik GmbH on the copy. The copyright notice on the *TiCoBasic* software must not be removed.

- 7.2 It is expressly not permitted to fully or partially copy or reproduce the *TiCoBasic* software as well as the printed materials in its original or modified form or merged or included in other software.

## 8. GRANT OF LICENSE

- 8.1 The right to use the *TiCoBasic* software can only be granted to a third party with prior written consent of Jaeger Messtechnik GmbH. The Licensee must then completely delete the software, which he has installed and pass it to the third party. (The transfer has to include the original data carrier with the documentation, backup version included). The license may furthermore only be transferred to a third party, if the latter agrees for the benefit of Jaeger Messtechnik GmbH to the terms and conditions of this License Agreement and to the General Conditions of the company Jaeger Messtechnik GmbH.



8.2 You must not rent, lease or lend the *TiCoBasic* software.

## 9. PERIOD OF AGREEMENT

9.1 The period of the License Agreement is unlimited.

9.2 The right of the Licensee for using the *TiCoBasic* 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 *TiCoBasic* software, possible modified copies included, as well as the printed materials.

## 10. CLAIM FOR DAMAGES AND PENALTY UPON VIOLATION OF THE CONTRACT

10.1 If the Licensee violates conditions of this License Agreement he must pay damages.

10.2 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.

10.3 The title to omission on completion of the contract is not influenced by the claim for damages and the penalties.

## 11. MODIFICATIONS AND UPDATES

Jaeger Messtechnik GmbH is entitled to update the *TiCoBasic* software upon its own discretion. Jaeger Messtechnik GmbH is not obliged to have updates of the *TiCoBasic* software available for the Licensee.

For extensive updates, Jaeger Messtechnik GmbH reserves the right to charge an additional fee.

## 12. WARRANTY AND LIABILITY OF JAEGER MESSTECHNIK GMBH

- a) Jaeger Messtechnik GmbH assumes warranty to the Licensee that at the moment of delivery the data medium, on which the *TiCoBasic* software is stored, is error-free in accordance with the accompanying

materials, when applied under normal operating conditions and under normal maintenance conditions.

13. If the data medium is faulty, the Licensee is granted a replacement within the warranty period of 6 months from the date of delivery. He must return the data medium as well as a copy of the invoice to Jaeger Messtechnik GmbH or to the distributor from whom he has purchased the product.
14. If a fault as described in Section 10 b) is not eliminated within an adequate period of time by replacement of the product, the Licensee may choose between either allowance (price reduction) or conversion (rescission of the License Agreement). The Licensee is not entitled to any further claims.
15. For the reasons mentioned in Section 1.2, Jaeger Messtechnik GmbH does not assume liability for the absence of defects with regards to the *TiCoBasic* software. In particular, Jaeger Messtechnik GmbH does not assume warranty for the fact that the *TiCoBasic* 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 *TiCoBasic* 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 *TiCoBasic* software.
16. Jaeger Messtechnik does not assume liability for damages, unless Jäger Messtechnik GmbH has caused damages by intention or by gross negligence. Liability because of properties assured by Jaeger Messtechnik GmbH remains unaffected. Liability is excluded for consequential damages, which are not part of the assurance given above.
17. Jaeger Messtechnik GmbH does not assume liability for damages caused by viruses, which are passed on by the data medium. The Licensee is hold responsible for checking the data medium for viruses, before installing the *TiCoBasic* software on his computer.
18. FINAL CONDITIONS

The invalidity of some individual conditions does not affect the validity of the License Agreement.

In addition to the conditions of this License Agreement, the General Terms and Conditions of Jaeger Messtechnik GmbH apply.

### A.4 Command Line Calling

The *TiCoBasic* 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 *TiCoBasic*.



Please note the general hints about [Command line calls in Windows](#) on [page 14](#).

#### A.4.1 Syntax

There are command line calls to create binary files (main option `/M`) and to create a library file (main option `/L`).

You add command line options, beginning with a slash `/`, some of which have optional parameters. If an option is missing, the compiler will use a default setting; nevertheless, we recommend typing all options to avoid ambiguities<sup>1</sup>.

While creating a binary file, more than one source code files may be compiled. Thus, some options are specified globally for all files, while other options are specified for each file separately (see option `/PROCESS`).

As an alternative, options of a single call may be written into a `make-file` and the compiler called with main option `/MAKE`.

At last there are the main options `/H` to display a short help text, and `/VER` to display the compiler version number.

The command line call is entered in a single line; option letters are case sensitive.

1. As an example, a call with all options given remains correct, even when a default setting is being changed.

**Syntax**

```

TiCoBasicCompiler /M [/A"dest"] [/IP"path"]
[/LP"path"] [/Lx] [/Sx] [/P1] /PROCESS
src.bas [/ET | /EA | /EN | /EE /EEAx
/EEEx /EEVx /EEOx] [/PNx] [/PH | /PL] [/PDx]
[/Ox] [/Vx]

TiCoBasicCompiler /L src.bas [/A"dest"]
[/IP"path"]
[/LP"path"] [/Lx] [/Sx] [/P1] [/Ox]

TiCoBasicCompiler /MAKE"makefile"

TiCoBasic /H

TiCoBasic /VER

```

Please note: With processor type T12 (option /P12), you have to write `ADbasic_C` instead of `TiCoBasicCompiler`.

Optional settings are given in brackets []. The character | separates options, which are mutually exclusive.

File names can be written without, with relative or with absolute path names. The base directory for a file name without or with relative path name is the working directory, from which the command line is called.

### Main Options

<code>/M</code>	Generate a binary file with the extension <code>.TIn</code> .  <code>n</code> Process number; see option <code>/PNx</code> .
<code>/L</code>	Generate a library file with the extension <code>.TLx</code> .  <code>x</code> Processor type; see option <code>/Px</code> .
<code>/MAKE</code>	Read main option, file name and other options of a single call from the <code>makefile</code> .  The text in the <code>makefile</code> may be written using several lines. Options outside the <code>makefile</code> are not permitted
<code>/H</code>	Display a short help text.
<code>/VER</code>	Display compiler version number.

### Options

<code>src.bas</code>	File name of the source code to be compiled; type with suffix <code>.bas</code> . With main option <code>/M</code> , specify after <code>/PROCESS</code> .  Compiler warnings are written into the file <code>src.wrn</code> , error messages into the file <code>src.err</code> .
<code>/A"dest"</code>	[Path and] name of the binary or library file <code>&lt;dest&gt;</code> , which is to be generated, without suffix. The default is the file name <code>src</code> .  The file suffix <code>.TIn</code> (binary file) or <code>.TLx</code> (library file) is attached automatically.
<code>/IP"path"</code>	Directory, where include files are searched.  This setting overwrites the <i>TiCoBasic</i> standard directory and should thus be used with caution.
<code>/LP"path"</code>	Directory, where library files are searched.  This setting overwrites the <i>TiCoBasic</i> standard directory and should thus be used with caution.

/Lx	Language for warnings and error messages. /LE English. Default. /LG German
/Sx	Hardware, for which the file is compiled: /SGII Gold II //SPII Pro II; Default
/Px	Processor type, for which the file is compiled: /P1 <i>TiCo</i> processor 1
/PROCESS	Keyword for options of the following source code file. Has to be repeated for each source code file. Use only in combination with main option /M.
/ET	Create timer-triggered process, see also <a href="#">chapter 6.1.1 on page 122</a> . Default. Excludes /EE and /EN.
/EE	Create externally triggered process, see also <a href="#">chapter 6.1.2 on page 123</a> ; requires options /EEA, /EEM, /EEV, /EEO. Excludes /ET and /EN.
/EEAn	Hardware address <i>n</i> (decimal), which is evaluated for the event signal.
/EEMn	Mask value <i>n</i> (decimal), which is used for OR-disjunction with the address.
/EEVn	Campring value <i>n</i> (decimal).
/EEOx	Comparing operator <i>x</i> : 1: < smaller than 2: = equal 3: <= smaller than or equal 4: > greater than 6: >= greater than or equal 8: <> not equal
/EN	Create process without trigger, see <a href="#">chapter 6.1.3 on page 124</a> . Excludes /EE and /ET.
/PNx	Number <i>x</i> (1...4) of the process. Default: 1.

/PH	Create process with high priority. Default. See also <a href="#">chapter 6.1 on page 122</a> .
/PL	Create process with low priority (time triggered process only). See also <a href="#">chapter 6.1 on page 122</a> .
/PDx	Set cycle time (Processdelay) of the process to x. Default: 3000. See also <a href="#">chapter 6.2.1 on page 124</a> .
/Ox	Set optimize level x (0, 1, 2) of the compiler, see also <a href="#">Process Options dialog box (page 59)</a> .
/O0	Optimize level 0 (=don't optimize)
/O1	Optimize level 1 (Default)
/O2	Optimize level 2
/Vx	Set process version x, see <a href="#">Process Options dialog box (page 59)</a> . Default: 1.

### A.4.2 Notes

The order of options is arbitrary. Command line calls are case sensitive.

If option /A is not used, the generated binary or library file is saved in the same directory, as the source code.

If warnings or errors occur during compilation, they are saved in the files <src.WRN> and <src.ERR>. The error messages are the same as those that *TiCoBasic* displays in the info window (see [chapter 3.11.1](#)).

The files <src.WRN> and <src.ERR> are saved in the same directory, as the source code. If you use the option /A, the files are saved in the directory where the binary or library file is created.

We recommend deleting the files containing the warnings and error messages before compilation, so that you can very easily check if the compilation has proceeded without any errors.

### A.4.3 Examples

```
C:\ADwin\TiCoBasic\TiCoBasiccompiler.exe /L
Z:\Myfiles\test.bas
```



This command line compiles the source code <test.bas> and generates the library file <test.TL1> in the directory <Z:\Myfiles\>.

Since nothing else is indicated, the default setting is used:

- save generated file in the directory of the source code file.
- use English warnings and error messages.
- Hardware: *ADwin-Pro II*.
- *TiCo* processor 1.
- Optimize level: 1.

If you do the call from the directory <C:\ADwin\TiCoBasic>, you can shorten this line to:

```
TiCoBasicCompiler.exe /L Z:\Myfiles\test.bas
```

The shortest version is when the source code is stored in the directory <C:\ADwin\TiCoBasic> (here without file name extension):

```
TiCoBasicCompiler /L test.bas
```

Anyway, we recommend the complete version—at least for automation of the call:

```
TiCoBasiccompiler /L test.bas /A"test" /LE /SPII /P1 /O1
```



```
TiCoBasicCompiler /M /LE /SGII /P1 /PROCESS  
- TiCo_inc_Par_1.bas /ET /PN3 /PH /O1
```

Compiles the demo file <TiCo\_inc\_Par\_1.bas> into a binary file for a *Gold II* system with *TiCo* processor. It is a process with number 3 and high priority.



```
TiCoBasicCompiler /M /A"Y:\somewhere\your_file" /LE /SGII  
/P1 /PROCESS C:\user\my_file.bas /ET /PN3 /PH /O1
```

The binary file now is saved as <Y:\somewhere\your\_file.TL1>; It is a process with number 3 and high priority.

#### A.4.4 Command line calls in Windows

The term and functionality "command line call" originates from DOS, where commands to the operating system (DOS) had to be entered in command lines. Entering such command lines is still possible under Windows.



There are several ways to enter commands under Windows:

- Open a Command Prompt window (from Windows start menu, directory `Programs / Accessories`).

The compiler call needs the Windows environment anyway. Thus, the call works only from the Command Prompt window, not from original DOS-mode.



- Select `Run` in the start menu and enter a command line in the input window.
- For frequently needed command lines, create an icon on the desktop. When you generate an icon, enter the command line directly.

One or more command lines can be combined in one batch file `<*.bat>`, for example in order to compile several source code files of a project with only one call.

When you call a command line, you have to transfer the relevant options and parameters.

**A.5 Instructions for ADwin-Gold II**

Below are listed the *TiCoBasic* instructions for *TiCo* processors. Instructions for the access to inputs/outputs and interfaces be found in the hardware documentation.

+ (Addition)  
 - (Subtraktion)  
 \* (multiplication)  
 / (Division)  
 ^ (power)  
 = (assignment)  
 : Colon  
 #Begin\_Debug\_Mode\_Disable  
 #Define  
 #End\_Debug\_Mode\_Disable  
 #If ... Then ... {#Else ...} #EndIf  
 #Include  
 #..., compiler statement

**A**

Absl  
 And

**D**

Data\_n  
 Dec  
 Dim  
 Do ... Until

**E**

End  
 Event:

**F**

Finish:  
 For ... To ... {Step ...} Next  
 Function ... EndFunction

**G-L**

If ... Then ... {Else ...} EndIf  
 Import  
 In  
 Inc  
 Init:  
 Lib\_Function ... Lib\_EndFunction  
 Lib\_Sub ... Lib\_EndSub

**M-Q**

Max\_Long  
 Min\_Long  
 Mod  
 NOP  
 NOPs  
 Not  
 Or  
 Out  
 Processdelay  
 Processn\_Running  
 Process\_Error

**R**

Read\_Timer  
 Refresh\_RingBuffer  
 Rem  
 Ringbuffer  
 RingBuffer\_Clear  
 RingBuffer\_Empty  
 RingBuffer\_Full

**S**

SelectCase  
 Shift\_Left

Shift\_Right  
Sleep  
Sub ... EndSub

XOr

### Symbols

< = > (comparison)

**T-Z**



### A.6 Instructions for ADwin-Pro systems

Below, the available *TiCoBasic* instructions for *TiCo* processors are listed. Instructions for inputs/outputs be found in the hardware manual.

+ (Addition)	Import
- (Subtraktion)	In
* (multiplication)	Inc
/ (Division)	Init:
^ (power)	
= (assignment)	<b>L</b>
: Colon	Lib_Function ... Lib_EndFunction
#Begin_Debug_Mode_Disable	Lib_Sub ... Lib_EndSub
#Define	
#End_Debug_Mode_Disable	<b>M</b>
#If ... Then ... {#Else ...} #EndIf	Max_Long
#Include	Min_Long
#..., compiler statement	Mod
<b>A</b>	
Absl	
And ,	<b>N-P</b>
<b>D</b>	NOP
Data_n	NOPs
Dec	Not
Dim	Or
Do ... Until	Out
	Processdelay
	Processn_Running
	Process_Error
<b>E</b>	
End	<b>R</b>
Event:	Read_Timer
	Refresh_RingBuffer
<b>F</b>	Rem
Finish:	Ringbuffer
For ... To ... {Step ...} Next	RingBuffer_Clear
Function ... EndFunction	RingBuffer_Empty
	RingBuffer_Full
<b>I</b>	
If ... Then ... {Else ...} EndIf	<b>S</b>
	SelectCase

Shift\_Left  
Shift\_Right  
Sleep  
Sub ... EndSub

**V-Z**  
XOr

**Symbols**  
< = > (comparison)

## A.7 Index

### Symbols

- · 133
- # · 137
- #Begin\_Debug\_Mode\_Disable · 144
- #Define · 147
- #Else · 163
- #End\_Debug\_Mode\_Disable · 153
- #EndIf · 163
- #If · 163
- #Include · 169
- \* · 134
- + · 132
- .gotolink TiCoBasic\_Gold2-Thema\_eng.fm Gold2T · 16
- .gotolink TiCoBasic\_Pro-ModulUebers\_eng.fm ProTM · 19
- .gotolink TiCoBasic\_Pro-Thema\_eng.fm ProTT · 19
- / · 135
- : · 138
- < = > · 140
- = · 139
- ^ · 136
- ' (Rem) · 195

### Numerics

150h, *see* device no.

### A

Absl · 141

- absolute value
  - integer number · 141
- ActiveX
  - communication to ADwin system · 128
- ADbasic
  - demo mode · 10
  - license agreement · 5
  - license key · 10
  - start · 9
- ADbasicCompiler, command line · 9
- ADconfig · 129
- add file/folder shortcut · 48
- Add Open Files to Project · 75
- Add to Project
  - context menu · 19
  - project window · 75
- addition · 132
- ADtools
  - overview · 89
  - set bar · 68
- ADWIN\_GOLDII · 163
- ADWIN\_PROII · 163
- ADWIN\_SYSTEM · 163
- analyze
  - run-time error · 119
- And · 142
- arithmetic functions
  - · 133
  - \* · 134
  - + · 132
  - / · 135
  - ^ · 136
  - Dec · 146
  - Inc · 168
- Array-Index (local) too large / <1, *see* run-time error

- arrays
  - allocate memory area · 101
  - Data\_n · 145
  - global · 98
    - first element · 98
  - initialize · 94
  - local · 100
    - first element · 100
  - overview · 95
- (Dim) AS · 149
- ASCII-character set · 4
- assign a value · 97
- assignment (=) · 139
- (Dim ...) AT · 149
- autocomplete, instruction or variable · 43
- autoindent · 64
- autoSave · 56
- autostart · 56

## B

- bar
  - ADtools · 89
  - editor · 22
  - menu · 53
  - status bar · 82
- Begin\_Debug\_Mode\_Disable *see* #Begin\_Debug\_Mode\_Disable
- binary file
  - see also* library
  - create · 56
    - from ADbasic · 56
    - from command line · 9
  - transfer to TiCo processor · 48
- binary notation · 97
- bit shifting
  - left · 209
  - right · 210

- bookmark · 40
- Bootloader
  - menu entry · 72
- bootloader
  - programming · 49
- break, *see* stop process
- BTL file
  - directory settings · 67
- busy display · 82
- bypass waiting time · 183

## C

- case sensitivity · 17
- Case, CCase, CaseElse (Select-Case ...) · 206
- change license key · 10
- change to TiCoBasic · 5
- clear parameter scan · 46
- code size · 83
- code snippets · 44
- color settings · 65
- command line
  - call · 9
  - line length
    - standard · 91
  - upper case / lower case · 91
- comment
  - comment Block · 26
  - position · 25
  - see* remarks
- communication
  - between ADwin CPU and TiCo processor · 129
  - between processes · 127
  - with the TiCo processor · 128
- compare macros to libraries · 114
- comparison
  - < = > · 140



- compiler
    - [autoSave](#) · 56
    - [call](#) · 56
    - [command line call](#) · 9
    - [compiler message, error / status](#) · 83
    - [prebuild, postbuild](#) · 70
    - [set options](#) · 57
    - [store project options](#) · 68
  - compiler instructions
    - [#Begin\\_Debug\\_Mode\\_Disable](#) · 144
    - [#Define](#) · 147
    - [#End\\_Debug\\_Mode\\_Disable](#) · 153
    - [#If ... Then](#) · 163
    - [#Include](#) · 169
    - [overview](#) · 137
  - conditional jump
    - [If ... Then](#) · 161
    - [SelectCase](#) · 206
  - constant · 93
  - context menu
    - [project window](#) · 75
    - [source code window](#) · 19
  - control block
    - [context menu](#) · 19
    - [mark](#) · 40
  - control structures
    - [overview](#) · 111
    - [toggle folding](#) · 30
  - counter
    - [internal, clock cycle](#) · 125
    - [read](#) · 193
  - [cursor position](#) · 82
  - [cycle time](#) · 124
- D**
- data exchange
    - [between processes](#) · 127
    - [with the TiCo processor](#) · 128
  - data loss
    - [on initialize](#) · 12
    - [on reset](#) · 12
    - [ringbuffer](#) · 104
  - data memory
    - [see also memory](#)
    - [additional demand by debug mode](#) · 119
    - [allocate](#) · 101
    - [overview, internal, external](#) · 101
  - data structures
    - [global arrays](#) · 98
    - [global variables](#) · 97
    - [local variables and arrays](#) · 100
    - [overview](#) · 95
    - [Ringbuffer](#) · 103
  - data types
    - [overview](#) · 96
  - data word, numbering of bits · 2
  - [Data\\_n](#) · 98
    - [dimensioning](#) · 149
    - [overview](#) · 145
  - [Data-Index \(global\) too large / <1,](#)
    - [see run-time error](#)
  - debug
    - [general](#) · 119
    - [debug mode](#) · 119
    - [menu](#) · 70
  - [debug errors](#) · 71
  - [debug mode](#) · 71
  - [Dec](#) · 146
  - [decimal notation](#) · 97
  - [decimal separator](#) · 97
  - declaration
    - [display all](#) · 87
    - [jump to](#) · 41
    - [see dimensioning](#)
    - [show all](#) · 46
    - [show single info](#) · 46
  - [decrement](#) · 146

- Define, *see* #Define
- defining foldable text range · 28
- definition of macros, position in the program · 94
- demo mode · 10
- design of an ADbasic program · 91
- Deutsch · 67
- development environment
  - bars and windows · 12
  - directory settings · 67
  - short-cuts · 1
  - start · 9
- device no.
  - definition · 129
  - set · 58
- Dim · 149
- dimensioning
  - instruction Dim · 149
  - memory area · 101
  - position in the program · 94
- directory
  - settings · 67
- display
  - all declarations · 46
  - current information · 16
  - declarations · 87
  - memory usage: CPU, PM, DM, DX · 82
  - passed parameters · 45
  - single declaration info · 46
  - syntax highlighting · 24
- division
  - by 2 · 210
  - remainder · 218
  - simple · 135
- DM, *see* memory
- DM\_LOCAL
  - Dim · 149
- Do ... Until · 151

- documenting self-defined instructions/variables · 26

- DRAM\_Extern

- Dim · 149

- DX, *see* memory

## E

- editor

- bar · 22

- general · 64

- print settings · 66

- syntax colors · 65

- Else (If ... Then) · 161

- End · 152

- End\_Debug\_Mode\_Disable *see* #End\_Debug\_Mode\_Disable

- EndFunction · 157

- EndIf (If ... Then) · 161

- EndSelect (SelectCase ...) · 206

- EndSub · 213

- enter license key · 10

- equal to = · 140

- error

- see also* run-time error

- forced by Cut&Paste · 55

- run-time · 71

- try lower optimization level · 62

- error message, compiler · 83

- Ethernet · 128

- evaluate

- operators · 110

- event

- external signal · 121

- externally controlled · 123

- lost signal

- externally controlled

- process · 126
- timer-controlled process · 126
- measure time difference · 116
- set signal source · 61
- timer controlled
  - 122
- without trigger · 124
- Event, program section · 154
- exclusive Or operation · 216
- exponential notation · 97
- expressions
  - evaluate · 110
  - symbolic names · 93
- external data memory (DX) · 103
- external event signal · 121
- external memory (SDRAM) · 101

## F

- F1: call help · 17
- Felder
  - Ringbuffer · 196
- FIFO · 103
- file name
  - binary file · 56
  - library · 57
- file, add shortcut · 48
- find
  - declaration of
    - instruction/variable · 41
  - examples · 35
  - regular expressions · 37
  - text · 32
  - text quickly · 31
- Finish: · 155
- fold text ranges · 30
- foldable text range, define · 28
- folder, add shortcut · 48
- font settings · 65
- For ... Next · 156
- formatting, smart · 24
- Function · 157

## function

- documenting · 26
- general features · 112
- library
  - definition · 172
  - general · 113
- macro · 157
- position in the program · 94
- valid characters · 93

## G

- german · 67
- Get\_Par · 220
- Get\_Par · 220
- Get\_Par\_Block · 222
- Get\_Par\_Block · 222
- Get\_TiCo\_RingBuffer · 224
- Get\_TiCo\_RingBuffer · 224
- Get\_TiCo\_Status · 226
- Get\_TiCo\_Status · 226
- GetData\_Long · 227
- GetData\_Long · 227
- global arrays, *see* arrays, global
- global variables, *see* variables, global
- global variables, window · 86
- Globaldelay · 189
- Gold2cess\_Status · 229
- goto line · 41
- greater than >, >= · 140

## H

- halt
  - TiCo processor · 12
- halt, *see* stop process
- Hardware access
  - read · 167
  - write · 188
- header, print · 66

## help

- call selected · 17

- F1 · 17

- stay on top · 68

- hexadecimal notation · 97

## I

- If · 161

- see *also* #If · 163

- Import · 165

- In · 167

- Inc · 168

- Include · 169

- include

- directory settings · 67

- include a file: #Include · 169

- include a library: Import · 165

- include-file, general · 112

- increment · 168

- indent

- ADbasic sections · 64

- comments · 25

- lines · 25

- info range · 83

- info window · 83

- Init: · 171

- initialize · 12, 93

- input license key · 10

- insert code snippets · 44

- instruction

- autocomplete · 43

- declaration info · 46

- display passed parameters · 45

- documenting self-defined ~ · 26

- jump to declaration · 41

- measure processing time · 115

- separator (:) · 138

- instruction reference · 131

- instructions

- TiCo processor · 219

- integer numbers

- value range · 96

- internal counter

- clock cycle · 125

- internal memory

- data (DM) · 103

- SRAM · 101

- interrupt, see stop process

## J

- Jump forward/backward · 41

- jump to declaration · 41

- jump to program line · 41

- jump, conditional

- If ... Then · 161

- SelectCase · 206

## K

- keyboard, settings display · 82

## L

- language · 67

- layout, print · 66

- less than <, <= · 140

- Lib\_EndFunction · 172

- Lib\_EndSub · 177

- Lib\_Function · 172

- Lib\_function

- documenting · 26

- Lib\_Sub · 177

- Lib\_sub

- documenting · 26

## library

## create

[from ADbasic · 57](#)[from command line · 9](#)[directory settings · 67](#)[function · 172](#)[general · 113](#)[Import · 165](#)[position in the program · 94](#)[subroutine · 177](#)[toggle folding · 30](#)[valid characters · 93](#)[versus macros · 114](#)

## library file

[create · 56](#)[license agreement · 5](#)[license key · 10](#)[line comment, indent · 25](#)[line length, max.](#)[standard · 91](#)

## lines

[change to comment · 26](#)[indenting · 25](#)[jump to · 41](#)[numbering · 64](#)[smart format · 24](#)[Load Bin File · 72](#)

## logic functions

[And · 142](#)[Not · 185](#)[Or · 186](#)[Shift\\_Left · 209](#)[Shift\\_Right · 210](#)[XOr · 216](#)[long, see integer numbers](#)[lost events \(T12\) · 125](#)

## M

## macro

[documenting · 26](#)[function · 157](#)[general features · 112](#)[position in the program · 94](#)[toggle folding · 30](#)[valid characters · 93](#)[versus library · 114](#)[Make Bin File, Make Lib File · 56](#)[manual indenting · 25](#)[mark Control block · 40](#)

## Matlab

[license key · 10](#)[Max\\_Long · 181](#)

## maximum

[integer values · 181](#)[maximum line length](#)[standard · 91](#)[measure processing time · 115](#)[measurement graph · 89](#)

## memory

[see also data memory](#)[additional demand by](#)[debug mode · 119](#)[allocate · 101](#)[areas \(PM, DM, DX\) · 101](#)[calculate need of · 83](#)[workload · 82](#)

## menu

[bar · 53](#)[build · 56](#)[debug · 70](#)[edit · 55](#)[file · 54](#)[help · 73](#)[options · 57](#)[select · 14](#)[tools · 72](#)[view · 55](#)[window · 73](#)

Min\_Long · 182

minimum

integer values · 182

Mod · 218

multiplication

by 2 · 209

simple · 134

## N

names

arrays and local variables · 100

macros, libraries · 93

negative sign · 111

new in TiCoBasic · 5

Next (For ...) · 156

none: without event trigger · 124

NOP · 183

NOPs · 184

Not · 185

not equal to <> · 140

notation of numbers · 97

notes, *see* remarks

number, *see* device no.

numerical values, notation · 97

## O

Hypertext · 16, 19

operating system

directory settings · 67

load, *see* initialize

operators

And · 142

evaluate · 110

negative sign · 111

Or · 186

priority · 110

XOr · 216

Optimierung

Speicherzugriff · 118

optimize

calculate polynoms quickly · 136

constants instead of

variables · 116

general · 115

measure faster · 117

measure processing time · 115

register access · 116

run-time error · 119

setting waiting time · 117

use waiting times · 117

options setting

ADtools · 68

compiler · 57

directory · 67

editor · 64

general · 64

help · 68

language · 67

print · 66

process · 59

project · 68

description · 69

general · 69

prebuild, postbuild · 70

syntax colors · 65

Or · 186

Out · 188

outdent lines · 25

overload of processor · 125

## P

P2\_Get\_Par · 264

P2\_Get\_Par\_Block · 266

P2\_Get\_TiCo\_Bootloader\_Status ·  
268

P2\_Get\_TiCo\_RingBuffer · 269

P2\_Get\_TiCo\_Status · 272

P2\_GetData\_Long · 273

P2\_Process\_Status · 276

P2\_Ringbuffer\_Empty · 278

- P2\_Ringbuffer\_Full · 280
- P2\_Set\_Par · 281
- P2\_Set\_Par\_Block · 283
- P2\_Set\_TiCo\_RingBuffer · 285
- P2\_SetData\_Long · 288
- P2\_TDrv\_Init · 291
- P2\_TiCo\_Flash · 297
- P2\_TiCo\_Get\_Process\_Error · 295
- P2\_TiCo\_Get\_Processdelay · 293
- P2\_TiCo\_Load · 299
- P2\_TiCo\_Restart · 301
- P2\_TiCo\_Set\_Processdelay · 303
- P2\_TiCo\_Start · 305
- P2\_TiCo\_Start\_Process · 306
- P2\_TiCo\_Stop · 308
- P2\_TiCo\_Stop\_Process · 309
- P2\_Workload · 311
- Par\_n · 97
- parameter scan · 46
- parameter window · 77
- parameters, *see* variables, global
- parse and indent · 64
- passed parameters, display · 45
- PM, *see* memory
- polynoms, calculate quickly · 136
- position, comments · 25
- postbuild · 70
- power · 136
  - replace in polynom · 136
- prebuild · 70
- pre-processor instructions
  - #Begin\_Debug\_Mode\_Disable · 144
  - #Define · 147
  - #End\_Debug\_Mode\_Disable · 153
  - #If ... Then · 163
  - #Include · 169
- pre-processor, *see* compiler instructions · 137
- print settings · 66
- priority
  - operators · 110
- problems
  - slow editor · 64
- process
  - autostart · 56
  - communication · 127
  - externally controlled · 123
  - operating modes for timing · 126
  - options, show · 14
  - processing time · 125
  - query status · 192
  - read out error · 191
  - setting options · 59
  - stop, *see* stop process
  - time characteristic · 124
  - timer controlled
    - low priority · 122
    - high priority · 122
    - without event trigger · 124
- process control
  - End · 152
  - Process\_Error · 191
  - ProcessN\_Running · 192
- process cycle
  - call
    - by event · 121
    - time interval · 125
- process optimization, *see* optimize
- Process\_Error · 191
- Process\_Status · 229
- Process\_Status · 229
- Processdelay · 124
  - syntax · 189
  - time resolutions · 124
- Processn\_Running · 192
- Processor · 163

- program architecture
  - jump
    - If ... Then · 161
    - SelectCase · 206
  - library
    - function · 172
    - Lib\_Sub · 177
  - loop
    - Do ... Until · 151
    - For ... Next · 156
  - modules
    - function · 157
    - subroutine Sub · 213
    - remarks Rem · 195
  - program design · 91
  - program improvement, *see* optimize
  - program line, jump to · 41
  - program memory · 103
    - additional demand by
      - debug mode · 119
  - program section
    - Event: · 93
    - Finish: · 93
    - Init: · 93
    - overview · 93
  - program structure
    - overview · 111
    - include-file · 112
    - library · 113
    - module (macro) · 112
    - toggle folding · 30
  - project
    - general · 50
    - highlight used parameters · 46
    - options setting
      - description · 69
      - general · 69
      - prebuild, postbuild · 70
    - window · 75

- Prozessn\_Running · 192

## R

- Read\_Timer · 193
- redo · 22
- Refresh\_RingBuffer · 198
- register access · 116
- regular expressions · 37
- Rem · 195
- remainder of integer division · 218
- remarks · 195
- replace
  - examples · 36
  - regular expressions · 37
  - text · 32
- reset
  - TiCo processor · 12
- Ringbuffer
  - dimensioning · 149
  - overview · 196
- ringbuffer
  - check number of elements · 105
  - data loss · 104
  - design of data structure · 103
- RingBuffer\_Clear · 200
- RingBuffer\_Empty · 202, 231
- Ringbuffer\_For\_Read · 196
- Ringbuffer\_For\_Write · 196
- RingBuffer\_Full · 204, 232
- run-time error
  - see also* debug mode
  - display · 71
  - find · 119

## S

- Save All Files of Project · 75
- SDRAM, *see* memory



- search · 32
  - declaration of
    - instruction/variable · 41
  - examples · 35
  - regular expressions · 37
- SelectCase · 206
- self-defined instructions/variables
  - documenting · 26
- separator : · 138
- Set\_Par · 233
- Set\_Par · 233
- Set\_Par\_Block · 235
- Set\_Par\_Block · 235
- Set\_TiCo\_RingBuffer · 237
- Set\_TiCo\_RingBuffer · 237
- SetData\_Long · 239
- SetData\_Long · 239
- settings
  - ADtools · 68
  - compiler · 57
  - directory · 67
  - editor · 64
  - general · 64
  - help · 68
  - print · 66
  - process · 59
  - syntax colors · 65
- Shift\_Left · 209
- Shift\_Right · 210
- (bit) shifting
  - left · 209
  - right · 210
- short-cuts · 1
- show
  - declarations of a file · 46
  - declarations window · 87
  - line numbers · 64
  - process options window · 14
- Sleep · 211
- smart format · 24
- snippets · 44
- source code
  - change tab order · 17
  - creating · 17
  - editor window · 82
  - formatting · 23, 24
  - information · 14
  - structured display · 24
  - to do's · 85
  - use in a project · 75
- source code status bar · 14
- special char, find · 37
- springen
  - anderes Sprungziel · 41
- Sprungziel, wechseln zwischen · 41
- SRAM, see memory
- SRAM\_EXTERN
  - Dim · 149
- stack size
  - until T11 · 83
- starting ADbasic · 9
- status bar · 82
- status bar of source code
  - window · 14
- status message, compiler · 83
- stay on top, help window · 68
- Step (For ...) · 156
- stop
  - TiCo processor · 12
- stop process
  - itself
    - in Event: · 152
- structure
  - Coloured display of source code · 24
  - indent lines · 25
  - program sections · 111
  - toggle folding · 30
- Sub · 213

- subroutine
  - documenting · 26
  - general features · 112
  - library
    - definition (Lib\_Sub) · 177
    - general · 113
    - macro · 213
    - position in the program · 94
    - valid characters · 93
  - subtraction · 133
  - symbolic names · 93
  - syntax
    - colors · 65
    - highlighting · 24
  - system variable
    - overview · 99
    - Process\_Error · 191
    - Processdelay · 189
    - ProcessN\_Running · 192
- T**
  - T12
    - clock cycle · 125
    - Stack size · 63
  - tab
    - size · 64
  - tab order, change · 17
  - TCP/IP
    - see Ethernet
  - TDrv\_Init · 241
  - TDrv\_Init · 241
  - terminate, see stop process
  - text
    - define foldable range · 28
    - find And replace · 32
    - find quickly · 31
    - fold ranges · 30
    - indenting · 25
    - smart format · 24
  - Then (If ... Then) · 161
  - TiCo bootloader
    - menu entry · 72
  - TiCo bootloader, programming · 49
  - TiCo processor
    - reset / stop · 12
  - TiCo\_Flash · 243
  - TiCo\_Flash · 243
  - TiCo\_Get\_Process\_Error · 247
  - TiCo\_Get\_Process\_Error · 247
  - TiCo\_Get\_Processdelay · 245
  - TiCo\_Get\_Processdelay · 245
  - TiCo\_Load · 249
  - TiCo\_Load · 249
  - TiCo\_Reset\_Mode · 252
  - TiCo\_Restart · 251
  - TiCo\_Restart · 251
  - TiCo\_Set\_Processdelay · 253
  - TiCo\_Set\_Processdelay · 253
  - TiCo\_Start · 255
  - TiCo\_Start · 255
  - TiCo\_Start\_Process · 256
  - TiCo\_Start\_Process · 256
  - TiCo\_Stop · 258
  - TiCo\_Stop · 258
  - TiCo\_Stop\_Process · 259
  - TiCo\_Stop\_Process · 259
  - TICO1 · 163
  - TICO2 · 163
  - TiCoBasic
    - license key · 10
  - TiCoBasic: differences to ADbasic · 5
  - time
    - cycle time · 124

## time saving

- constants instead of variables · 116
- measure faster · 117
- register access · 116
- setting waiting time · 117
- use waiting times · 117

timer event · 121

timer, *see* counter

## timing

*see* optimize

changed by

debug mode · 119

operating modes

externally controlled

process · 126

general · 126

timer-controlled process · 126

To (For ...) · 156

to do list · 85

toggle folding · 30

tool bar · 14

toolbox · 75

## Tools

bootloader · 72

load binary file · 72

TGraphTiCo · 72

## tools

TBin · 89

TButton · 89

TDigit · 89

TFifo · 89

TGraph · 89

TLed · 89

TMeter · 89

TPar\_FPar · 89

TPoti · 89

TProcess · 89

## U

uncomment Block · 26

undo · 22

unmark Control block · 40

Until (Do ...) · 151

upper / lower case letters · 17

USB · 128

user defined instructions and variables · 93

user surface · 12

utility programs, *see* ADtools

## V

valid characters

macros, libraries · 93

variables · 100

value range · 96

## variables

autocomplete · 43

declaration info · 46

display · 77

global · 97

highlight used · 46

name · 95

initialize · 12, 94

jump to declaration · 41

local · 100

allocate memory area · 101

name length · 100

overview · 95

switch hex/decimal display · 78

symbolic names · 93

valid characters · 93, 100

*see also* system variable

variables, documenting · 26

Verlauf der Sprungziele · 41

## view

to do list · 85

**W**

## wait

- NOP · 183
- NOPs · 184
- setting waiting time exactly · 117
- Sleep · 211

## window

- compiler options · 57
- debug errors · 71
- declarations · 87
- global variables · 86
- info range · 83
- info window · 83
- overview · 12
- parameter · 77
- process Options · 59
- project · 75
- project options · 68
  - description · 69
  - general · 69
  - prebuild, postbuild · 70
- source code · 82
- source code information · 14
- source code status bar · 14
- status bar · 82
- to do list · 85
- toolbox · 75

without event trigger · 124

Workload · 261

Workload · 261

## workload

- definition · 125
- display · 82
- workspace size · 83

**X**

XOr · 216

## Symbols

< = > (comparison)	162
+ (addition)	151
+ (String addition)	152
- (subtraction)	154
* (multiplication)	155
/ (division)	156
^ (power)	157
= (assignment)	161
: colon	160
" " (String)	281
#Begin_Debug_Mode_	
Disable	172
#Define	184
#End_Debug_Mode_Dis-	
able	191
#If ... Then ... {#Else ...}	
#EndIf	213
#Include	218
#..., compiler statement	
159	

## A-B

Abs	163
AbsF	164
AbsI	165
And	166
ArcCos	168
ArcSin	169
ArcTan	170
Asc	171

## C

Cast_Float32ToLong	175
Cast_FloatToLong	173
Cast_LongToFloat	174
Cast_LongToFloat32	176
Chr	177
Cos	178
CPU_Sleep	179

## D

Data_n	181
Dec	183
Dim	186

Do ... Until	189
--------------	-----

## E-F

End	190
Event:	192
Exit	193
Exp	194
FIFO	195
FIFO_Clear	197
FIFO_Empty	199
FIFO_Full	200
Finish:	201
Flo40ToStr	204
FloToStr	202
For ... To ... {Step ...}	
Next	206
Function ... EndFunction	
208	

## G-J

If ... Then ... {Else ...} En-	
dlf	211
Import	215
Inc	217
Init:	220
IO_Sleep	222

## K-L

Lib_Function ... Lib_End-	
Function	224
Lib_Sub ... Lib_EndSub	
229	
LN	233
LngToStr	234
Log	236
LowInit:	237

## M-O

Max_Float	239
Max_Long	241
Min_Float	240
Min_Long	242
NOP	245
Not	246
Or	247

## P

P1_Sleep	249
P2_Sleep	251
Peek	253
Poke	254
Processdelay	255
Processn_Running	259
Process_Error	258

## R

Read_Timer	260
Read_Timer_Sync	262
Rem	263
Reset_Event	264
Restart_Process	265
Round	266

## S

SelectCase	267
Shift_Left	270
Shift_Right	271
Sin	272
Sleep	273
Sqrt	275
Start_Process	276
Stop_Process	279
" " (String)	281
StrComp	283
StrLeft	284
StrLen	286
StrMid	287
StrRight	289
Sub ... EndSub	291

## T-Z

Tan	294
User_Version	295
ValF	296
Vall	298
XOr	300

