

TiCoBasic

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

***TiCoBasic* Version 1.0**

June 2010

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 5 68 19
E-Mail:	info@ADwin.de
Internet	www.ADwin.de



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

Table of contents

Table of contents	III
1 Conventions	2
2 Introduction	3
3 TiCoBasic for ADbasic users	5
4 Development Environment	9
4.1 Basic Steps	9
4.1.1 Starting the Development Environment	9
4.1.2 Check or change TiCoBasic licenses	10
4.1.3 Initializing Communication	12
4.1.4 Basic Elements of the Development Environment	13
4.2 Creating source code	16
4.2.1 Calling online help	16
4.2.2 Context menu in source code window	17
4.2.3 Editor bar	19
4.3 Formatting source code	21
4.3.1 Syntax highlighting	21
4.3.2 Smart formatting	21
4.3.3 Indenting text lines	23
4.3.4 Changing lines into comment	23
4.3.5 Folding text ranges	23
4.4 Searching and replacing	25
4.4.1 Finding text quickly	25
4.4.2 Finding and replacing text	26
Examples - Finding Text	29
Examples - Replacing Text	30
4.4.3 Regular expression	31
4.4.4 Marking control blocks	33
4.4.5 Using bookmarks	34
4.4.6 Jump to a program line	35

4.4.7 Jumping to declaration of instruction or variable	35
4.5 Writing programs with ease	35
4.5.1 Autocomplete for instruction or variable	36
4.5.2 Inserting code snippets	37
4.5.3 Displaying instruction parameters	38
4.5.4 Displaying declaration of instruction or variable	38
4.5.5 Displaying declarations of a file	39
4.5.6 Displaying used global variables and arrays	40
4.6 Transferring a TiCo binary file to TiCo processor	41
4.6.1 Transferring a TiCo binary file	41
4.6.2 Programming the TiCo bootloader	41
4.7 Managing Projects	42
4.8 Menus	43
4.8.1 File Menu	45
4.8.2 Edit Menu	46
4.8.3 View Menu	46
4.8.4 Build Menu	47
4.8.5 Options Menu	48
Compiler Options dialog box	48
Process Options dialog box	50
Settings dialog box	54
4.8.6 Tools Menu	58
4.8.7 Window Menu	59
4.8.8 Help Menu	59
4.9 Windows	61
4.9.1 Toolbox	61
4.9.2 Project Window	61
4.9.3 Parameter Window	63
4.9.4 Process Window	64
4.9.5 Register window	66
4.9.6 Status Bar	67
4.10 Info range	68
4.10.1 Info window	68
4.10.2 ToDo List	70
4.10.3 Global Variables	71
4.10.4 Declarations	72
4.11 ADtools	74

5 Programming Processes	76
5.1 Program Design	76
5.1.1 The Program Sections	78
5.1.2 User defined instructions and variables	78
5.2 Variables and Arrays	80
5.2.1 Overview	80
5.2.2 Data Structures	80
5.2.3 Data Types	82
5.2.4 Entering Numerical Values	82
5.2.5 Global Variables (Parameters)	82
5.2.6 Global Arrays	83
5.2.7 System Variables	85
5.2.8 Local Variables and Arrays	85
5.3 Variables and Arrays – Details.	86
5.3.1 Variables and Arrays in the Data Memory	86
5.3.2 Memory Areas	87
5.3.3 The Data structure Ringbuffer	89
5.4 Expressions	97
5.4.1 Evaluation of Operators	97
5.5 Selection structures, Loops and Modules	99
5.5.1 Subroutine and Function Macros	99
5.5.2 Include-Files	100
5.5.3 Libraries	100
6 Optimizing Processes	102
6.1 Measuring the Processing Time	102
6.2 Useful Information	103
6.2.1 Accessing Hardware Addresses	103
6.2.2 Constants instead of Variables	104
6.2.3 Faster Measurement Function	104
6.2.4 Setting Waiting Times Exactly	105
6.2.5 Using Waiting Times	105
6.2.6 Optimization of memory access	107
7 Processes in the ADwin System	108
7.1 Process Management	110
7.1.1 Timer controlled process	110

7.1.2 Externally controlled process	111
7.1.3 Process without trigger (None)	112
7.2 Time Characteristics of Processes	113
7.2.1 Processdelay	113
7.2.2 Workload of the <i>TiCo</i> processor	114
7.2.3 Different Operating Modes in the Operating System	114
7.3 Communication	116
7.3.1 Data Exchange between Processes	116
7.3.2 Communication with the <i>TiCo</i> processor	116
7.3.3 Communication between ADwin CPU and <i>TiCo</i> Processor 118	
7.3.4 The Device Number	118
8 Instruction Reference	119
8.1 Instruction Syntax	119
8.2 Basic Instructions <i>TiCoBasic</i>	120
8.3 Gold II: <i>TiCo</i> processor	206
8.4 Pro II: <i>TiCo</i> Processor	254
9 How to Solve Problems?	302
Appendix	A-1
A.1 Short-Cuts in <i>TiCoBasic</i>	A-1
A.2 ASCII-Character Set	A-4
A.3 License Agreement	A-5
A.4 Command Line Calling	A-9
A.5 Index	A-17
A.6 Instructions in this manual	A-31

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 3 where differences between *TiCoBasic* and *ADbasic* are listed.

First-time users of a *TiCo* processor and *TiCoBasic* are recommended to read chapters 1 and 5, 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 4 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).

1 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**

2 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.

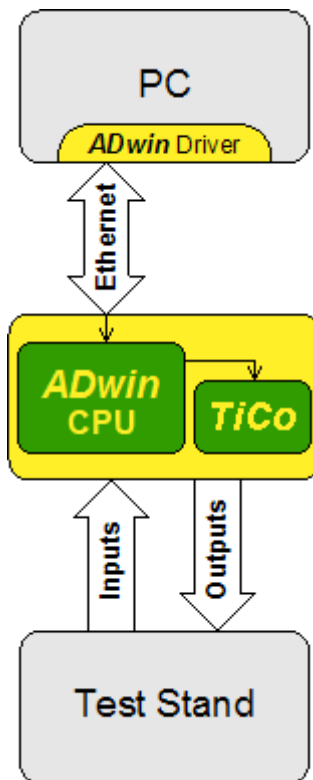
To hit the target of an immediate and efficient start of programming, we first of all would like to shortly explain the concept of the *ADwin* system with the integrated *TiCo* processor:

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. 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 5 explains the design of *TiCoBasic* programs, chapter 7 additionally the action of processes in the operating system.



Mit nur wenigen Programmzeilen können Sie beispielsweise:

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

Die Entwicklungsumgebung *TiCoBasic* unterstützt Sie bei der Umsetzung Ihrer Aufgabe in einen Prozess. Zunächst erstellen Sie den Quelltext in einer erweiterten Basic-Syntax; mit den Befehlen und Funktionen können Sie die Hardware Ihres *ADwin* systems komfortabel programmieren. In chapter 5 ist erklärt, wie Sie Programme aufbauen.



Mit dem integrierten Compiler erzeugen Sie aus dem Quelltext lauffähigen Binärcode, der als Prozess auf das *ADwin* system übertragen und getestet wird. *TiCoBasic* bietet Ihnen auch die Hilfsmittel, mit denen Sie Ihre Prozesse



beobachten, Fehler suchen und die Programme optimieren können (siehe chapter 6).

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

Sie können die Prozesse und Prozessdaten des *TiCo* processors von der *ADwin* CPU aus steuern und beobachten, d.h. Daten lesen und schreiben sowie Prozesse starten, steuern und stoppen.

Obwohl der *TiCo* processor autark arbeitet, können Sie aus von der *ADwin* CPU jederzeit auf globale Variablen und Felder zugreifen, ohne zeitkritische Prozesse zu verzögern. Über die globalen Variablen und Felder können alle Prozesse untereinander oder mit der *ADwin* CPU schnell Daten austauschen.

Die klare Trennung von Echtzeit-Prozessen im *TiCo* processor und im *ADwin* system einerseits und der Bedienoberfläche im PC andererseits garantiert Ihnen höchste Betriebssicherheit und zeitlich nachvollziehbare Abläufe.

3 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

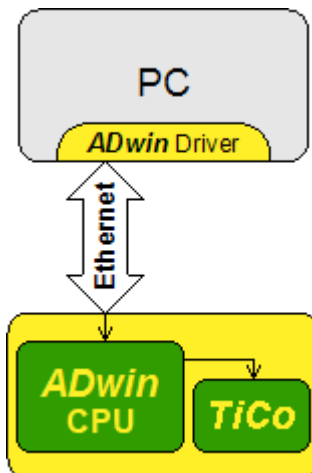
The *TiCo* processor is implemented as soft-core in the FPGA, where to the PC has no direct connection. The data exchange between PC and *TiCo* processor will therefore always need the *ADwin* CPU to forward data.

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	INIT :, EVENT :, FINISH :, REM , : (colon)
Variables	DIM , PAR_1...PAR_80 , DATA_1...DATA_16 Ringbuffer_For_Read Ringbuffer_For_Write Local arrays
Mathematics, operators	+ - * / INC , DEC , SHIFT_LEFT , SHIFT_RIGHT ABS , NOT , OR , XOR , AND
Comparison	= < > OR , AND
Structure	FOR...NEXT , DO...UNTIL IF...THEN , SELECTCASE FUNCTION , SUB , Lib_FUNCTION , Lib_SUB
Processes, system variables	END Processdelay , PROCESS_RUNNING , NWTIME
Pre-Compiler	#IF...#ENDIF , #DEFINE , #INCLUDE
Timing	NOP , NOPS , SLEEP , READ_TIMER
Memory access	IN , OUT

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**.

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 8.3 and 7.2.

Deviations in the development environment

- Actually, the development environment *TiCoBasic* provides neither debug nor timing mode, which are 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.

4 Development Environment

Programs for *TiCo* processors are quickly and easily developed 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.

4.1 Basic Steps

4.1.1 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.

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 4.1.2 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.

4.1.2 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).

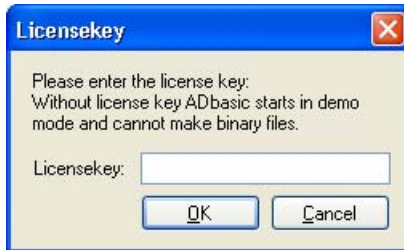


2. In order to enter or change the license key click the button `Change License`.

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 only works 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* 5, works with all *ADwin* processors
- *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 (annex see A-5).

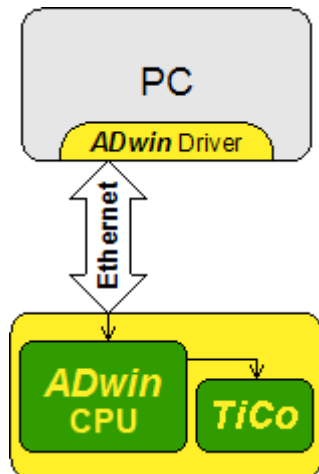
4.1.3 Initializing Communication


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  (Reset). Doing so, you will loose all values in the global *TiCo* variables and the process is deleted.

4.1.4 Basic Elements of the Development Environment

The development environment consists of several bars and windows (see fig. 1); 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.

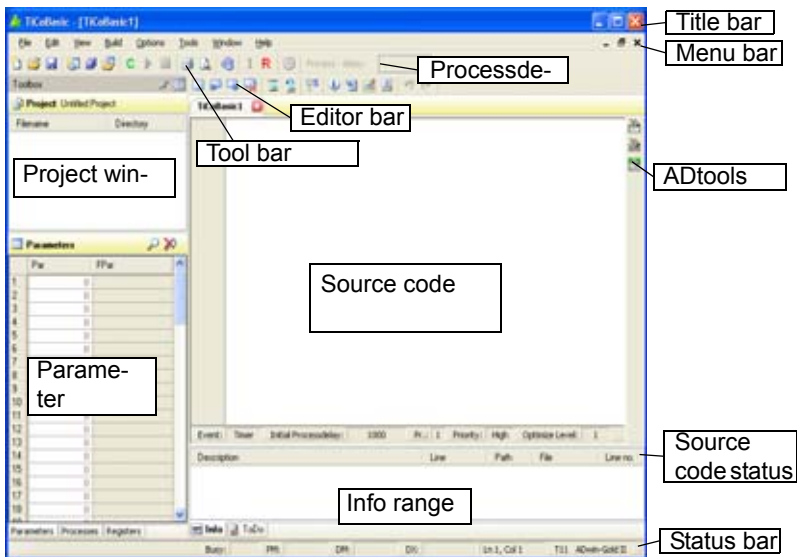


Fig. 1 – Elements of the *TiCoBasic* development environment

The functions of the development environment are called using:

- The tool bar and the editor bar (see fig. 2).
- The context menus of the windows (right mouse button).
- The menu bar.
- The Short-Cuts in *TiCoBasic* (see annex).

While using a function, the function's description is shown at the left of the status bar.

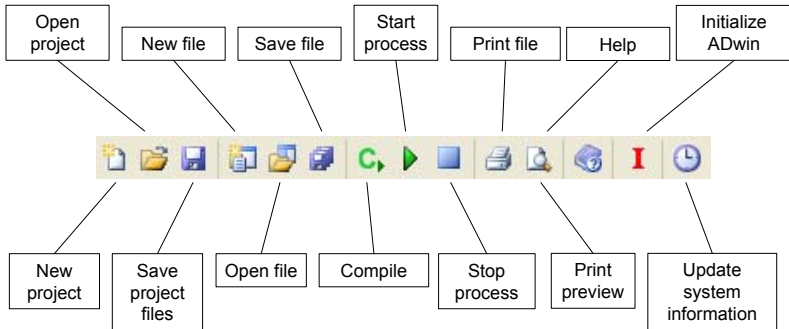


Fig. 2 – The tool bar


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 4.9.3, page 63)

by clicking `Scan Global Variables` ; see Displaying used global variables and arrays on page 40.

- The info range at the bottom displays information in several windows:
 - Info window: The compiler's error messages (highlighted red) and warnings (see chapter 4.10.1 on page 68).
 - ToDo List: A simple ToDo list from comment lines (see chapter 4.10.2 on page 70).
 - Search results from a search in all files of a project (see chapter 4.4.2 on page 26).

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

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

Some data of the *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 64).
- The register values of the *TiCo* processor in the Register window (page 66).
- Memory usage information in the Status Bar (see chapter 4.9.6 on page 67).

4.2 Creating source code

Open a new window for each process source code (using **File ▶ New**).

If you use several files for your task, we recommend to manage the files in a project file (see page 42: 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 17) or via Editor bar (page 19):

Numerical values may be entered into source code in hexadecimal, binary and exponential notation, as well as in decimal (see also chapter 5.2.4).

Find more editor functions here:

- Formatting source code, page 21
- Searching and replacing, page 25
- Writing programs with ease, page 35

4.2.1 Calling online help

The Help Menu (page 59) 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.

4.2.2 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
<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
	Show Declarations	Shift+F2

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

- **Cut:** Cut selection and copy into the clipboard.
- **Copy:** Copy selection into the clipboard.
- **Paste:** Delete selection and insert text from the clipboard.
- **Comment Block, Uncomment Block:** Changing lines into comment, page 23.
- **Indent, Outdent:** Indenting text lines, page 23.
- **Mark Control block, Unmark Control block:** Marking control blocks, page 33.
- **Declaration Info:** Displaying declaration of instruction or variable, page 38.
- **Jump to Declaration:** Jumping to declaration of instruction or variable, page 35.

These functions are available without marking:

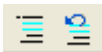
- **Add to Project:** Add a file to the project.
- **Code snippets:** Inserting code snippets, page 37.
- **Show all Declarations:** Displaying declarations of a file, page 39.

4.2.3 Editor bar

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



Using bookmarks, page 34.



Changing lines into comment, page 23.



Folding text ranges, page 23.



Displaying declaration of instruction or variable, page 38.



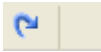
Jumping to declaration of instruction or variable, page 35.



Inserting code snippets, page 37.



Displaying declarations of a file, page 39.



Undo the previous editing action or redo it.

4.3 Formatting source code

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

- Syntax highlighting, page 21
- Smart formatting, page 21
- Indenting text lines, page 23
- Changing lines into comment, page 23
- Folding text ranges, page 23

Find more editor functions in the sections:

- Creating source code, page 16
- Searching and replacing, page 25
- Writing programs with ease, page 35

4.3.1 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 Highlight` (see page 55); the window also shows an overview of syntax groups.

Syntax highlighting requires an active option `Parse Declarations` under `Editor - General` (see page 54).

4.3.2 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 54).

4.3.3 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` oder `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.

4.3.4 Changing lines into comment

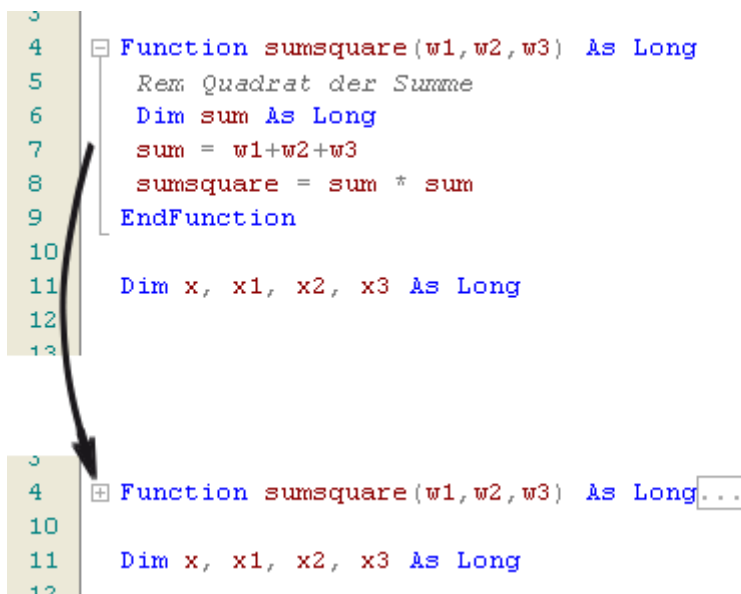
Marked lines may be changed into comment lines in one action by selecting the menu entry `Comment Block` from the source code context menu (right mouse click). The editor will then insert a comment char `'` at every of the marked lines so the compiler will skip these lines.


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

4.3.5 Folding text ranges

The editor recognizes control structures like conditions or loops, program sections, macros and library modules as foldable text ranges. These ranges are marked by a grey 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 Outlining**  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 54) is active.

4.4 Searching and replacing

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

- Finding text quickly, page 25
- Finding and replacing text, page 26
- Regular expression, page 31
- Marking control blocks, page 33
- Using bookmarks, page 34
- Jumping to declaration of instruction or variable, page 35

There are more editor functions:

- Creating source code, page 16
- Formatting source code, page 21
- Writing programs with ease, page 35

4.4.1 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:

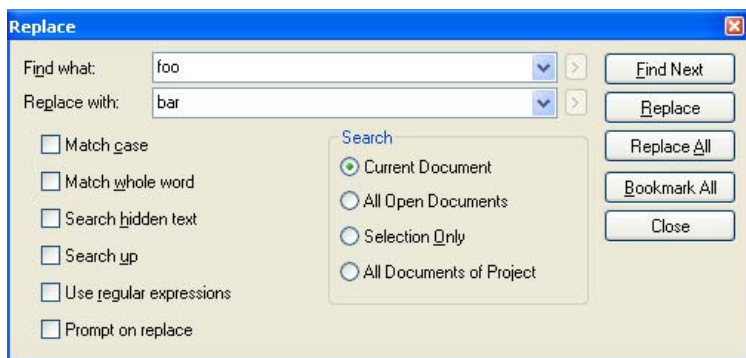
- Uppercase and lowercase letters are of no importance.
- Find text also as part of a word.
- Folded text areas are searched.
- All open documents are searched.

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

4.4.2 Finding and replacing text

You can find each occurrence of a combination of any characters, including uppercase and lowercase characters, whole words, or parts of words, or regular expression (see Regular expression on page 31).

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.

Option	Description
Match whole word	Option active: Find occurrences of the text as whole words. Option inactive: Find text also as part of a word.
Search hidden text	The option refers to Folding text ranges (see page 23). Option active: Folded text areas are searched. Option inactive: Folded text areas are skipped.
Search up	Option active: Search in direction to start of file. Option inactive: Search in direction to end of file.
Use regular expressions	Specify that the search string is a Regular expression (see page 31).
Prompt on replace	Option valid with Replace All only. Option active: Each occurrence opens a dialog box to control replacing. Option inactive: All occurrences are replaced without query.

5. Set the search range.

Option	Description
Current Document	Start search in the current source code at cursor position. If text is selected, the cursor is positioned behind the selection.
All open Documents	All open documents are searched, starting with the current source code.

Option	Description
Selection only	Only the selected range is searched. If no selection is given, search starts at cursor position.
All Documents of Project	All files of the project are searched, not regarding whether the current source code is also part of the project. Cannot be used for replace. The results are shown at the bottom in a window. Double click a result to jump to the appropriate code line or use the arrow buttons.

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 inner-most 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

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

Meta- zeichen :	Bedeutung:
+	Matches one or more of the preceding characters or expressions. Example: <code>ha+d</code> matches <code>had</code> and <code>haad</code> , but not <code>hd</code> .
	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> .

4.4.4 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 23).

4.4.5 Using bookmarks

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

You can use these actions:

- Set a Bookmark

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.

4.4.6 Jump 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 54) must be enabled.

4.4.7 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 54) is active.

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

4.5 Writing programs with ease

Be at ease while programming using the following functions:

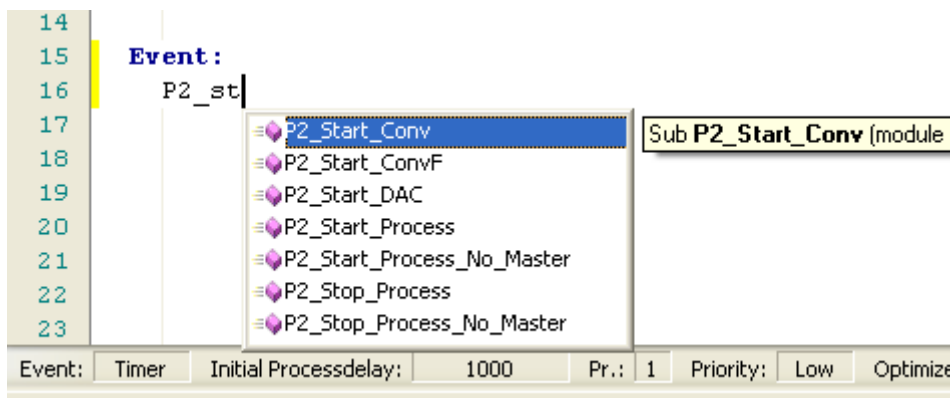
- Autocomplete for instruction or variable, page 36
- Documenting self-defined instructions and variables, page 54
- Inserting code snippets, page 37
- Displaying declaration of instruction or variable, page 38
- Displaying declarations of a file, page 39
- Displaying used global variables and arrays, page 40

Find more editor functions here:

- Creating source code, page 16
- Formatting source code, page 21
- Searching and replacing, page 25

4.5.1 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 will 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" or 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.

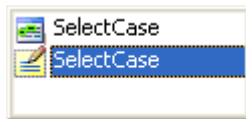
Autocomplete is only available, when the option `Parse Declarations` under Editor - General (see page 54) is active.

4.5.2 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.

4.5.3 Displaying instruction parameters

The passed parameters of an instruction are displayed automatically, 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.

The display of instruction parameters is only available, when the option `Parse Declarations` under Editor - General (see page 54) is active.

4.5.4 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 `Automatic quick info tips` under Editor - General (see page 54) 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 54) is active.

4.5.5 Displaying declarations of a file

To display all declarations, include and library files referring to a source file, set the `Declarations` to the foreground (see page 72). 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 54) is active.

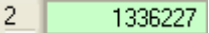


4.5.6 Displaying used global variables and arrays

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 63).

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

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.

4.6 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.

4.6.1 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 47.

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 not started automatically. To do so, click on the button **Start Process** .

4.6.2 Programming the TiCo bootloader

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

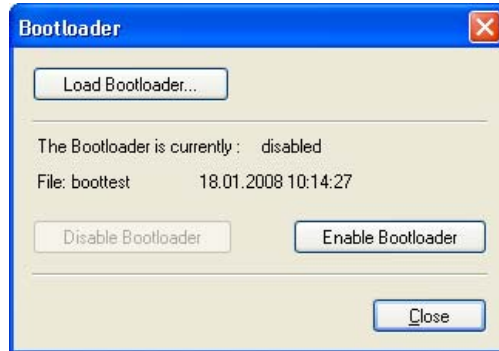
To program the *TiCo* bootloader, do as follows:

- Create a binary file using **Build ▶ Make Bin File**; see also page 47.

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 `LoadBootloader` and select the binary file in the next window.
- Enable the bootlader operation with the button `Enable Bootloader`. 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 onagain.

- Close the dialog window with `Close`.

4.7 Managing Projects

One project can manage many process source codes, include files, and library files, for instance when programming an application with several processes. 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. Thus, with opening a project, the display will be rearranged.

the following rules apply to the combination of project files (see also chapter 7.1 "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); we will inform you about the required precautions.

A project allows the user:

- Displaying used global variables and arrays of a project (see page 40).
- Search through all files of a project, including not yet opened files.
Just enable the `All Documents of Project` option in the find window (see chapter 4.4.2 "Finding and replacing text"). The option is not available for replacing.
- Save all files of project at once, using `Save all Files of Project` from the project window context menu.
- Open 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 61) or in the menu `File`.

4.8 Menus

The menu bar contains these menus:

- `File:` `Manage files and projects` (page 45)

- Edit: Edit source codes (page 46)
)
- View: Show windows and bars (page 46)
)
- Build: Tool for generating executable programs (page 47)
)
- Options Program settings (page 48)
:)
- Tools: Various help functions (page 58)
)
- Window: Arrange source code windows (page 59)
)
- Help: Help, version and license information (page 59)
)

4.8.1 File Menu

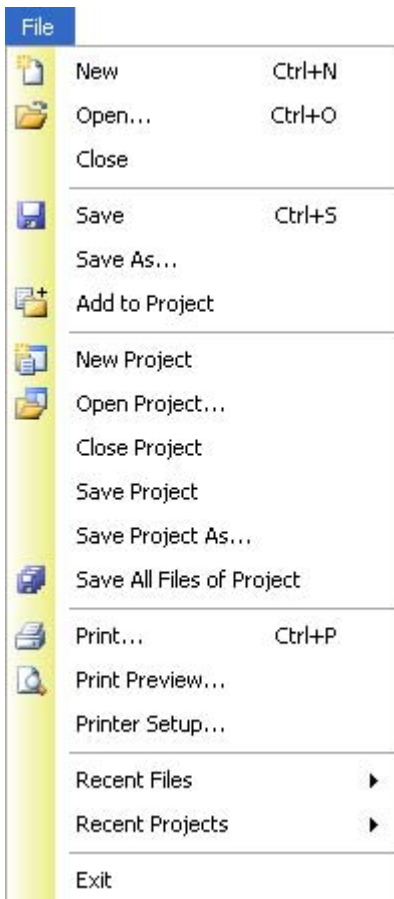
The `File` menu contains instructions for managing files and projects.

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 4.9.2).

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.








4.8.2 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 `Find`-ing and replacing text on page 26.

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

4.8.3 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 4.9.4 on page 64, about the toolbar see fig. 2.

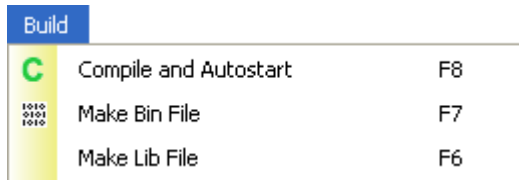
View	
<input checked="" type="checkbox"/>	Standard Toolbar
<input checked="" type="checkbox"/>	Editor Toolbar
<input checked="" type="checkbox"/>	ADtools Toolbar
<input checked="" type="checkbox"/>	Statusbar
<hr/>	
	Restore Default Layout


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 61).

4.8.4 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 4.3.3 on page 23), 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 Options Menu, Process Options dialog box).

A binary file with the extension **<*.TI3>** can be transferred to a *TiCo* processor of type 1. A binary file can be transferred to the *TiCo* processor with *TiCoBasic* (see chapter 4.6.1 "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 5.5.1 on page 99).

4.8.5 Options Menu

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



Compiler Options dialog box

The settings in this dialog box are used in every source code compilation. In particular the information refers to the *ADwin* system and the *TiCo* processor on which 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. 3 – 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 <ADcon-fig.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 4.8.6 "Tools Menu", page 58) 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 quickly open this window do a double click on the source code's status bar.

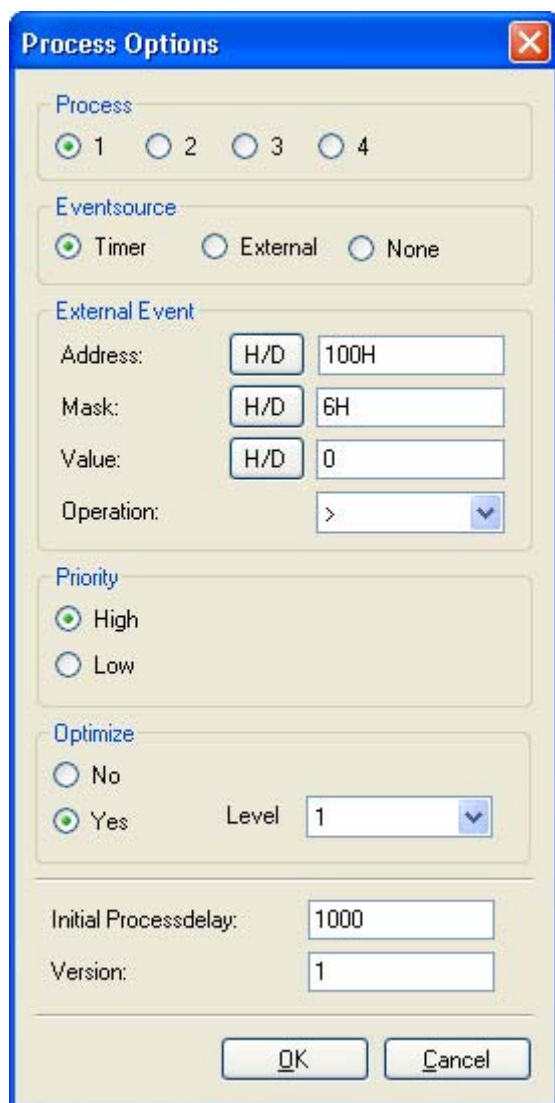


Fig. 4 – The Process Options dialog box

- **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.

The hardware addresses are different for each *ADwin* hardware.

External Event		
Address:	H/D	70H
Mask:	H/D	12H
Value:	H/D	0
Operation:		> ▼

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 7.1 "Process Management" chapter 6.1.1 "Types of Processes".

- **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.
- **Version:** An integer value for differentiating between several versions of a process.

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 Highlight
 - Editor - Print Settings
- Language
- Directory
- ADtools

Editor - General

`Parse and Indent`: 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 information found is the base for more comfortable functions like Autocomplete for instruction or variable, Displaying declarations of a file or Documenting self-defined instructions and variables.



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

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

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

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

`Smart format`: Format lines automatically, see "Smart formatting" on page 21.

`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 as before.

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

Show line numbers: Line numbers are displayed in the gutter left of the source code. See also „Jump to a program line“ on page 35.

Column mark, visible: A grey 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 Highlight

The editor highlights the syntax elements with different colors; see also chapter 4.3.1 "Syntax highlighting" on page 21; complete syntax highlighting requires an active option `Parse Declarations` under Editor - General.

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

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

The example text above shows how source code be formatted.

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

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.

Syntax Highlight: Syntax highlighting is printed.

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

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*.

Directory

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.

It is recommended that default directories for BTL, Include and Library be not changed. To include library and include files from other directories, type the full or relative path name with the instruction.

ADtools

The *ADtools* (description see chapter 4.11) can be started from the ADtools bar. If the appropriate option is active, the tool is displayed in the bar.

4.8.6 Tools Menu

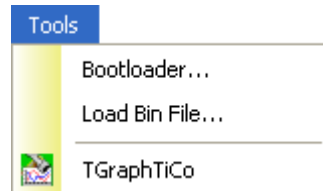
The `Tools` menu option calls utility programs.

The menu entry `Bootloader...` programs the *TiCo* bootloader (see chapter 4.6.2 "Programming the *TiCo* bootloader", page 41). 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 4.6.1 "Transferring a *TiCo* binary file", page 41).

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 48).

The menu entry `TGraphTiCo` starts a utility tool; short description see chapter 4.11 on page 74.

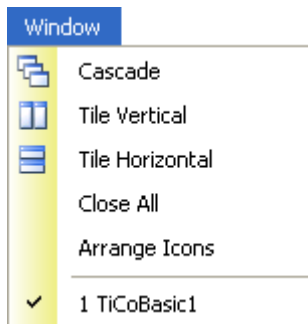


4.8.7 Window Menu

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.

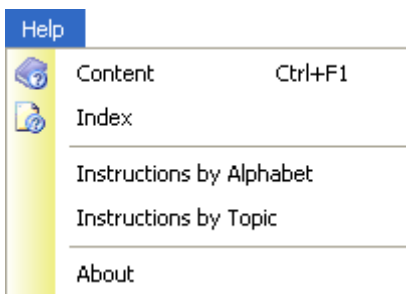
At the bottom of the menu, there is a list of open source codes; by clicking one of these menu items that source code will become the active window. The active source code is checked; in the example at right it is `TiCoBasic1.bas`.




4.8.8 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.



The instruction lists refer to the *ADwin* system, which is set in the `Compiler Options` dialog box on page 48.

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.

4.9 Windows

4.9.1 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.

4.9.2 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 61).

In the project window the following actions may be executed:

- Add a source code or include file to the project:
Select `Add to Project` from the source code context menu.
- Add all open files to the project:
Select `Add Open Files 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.

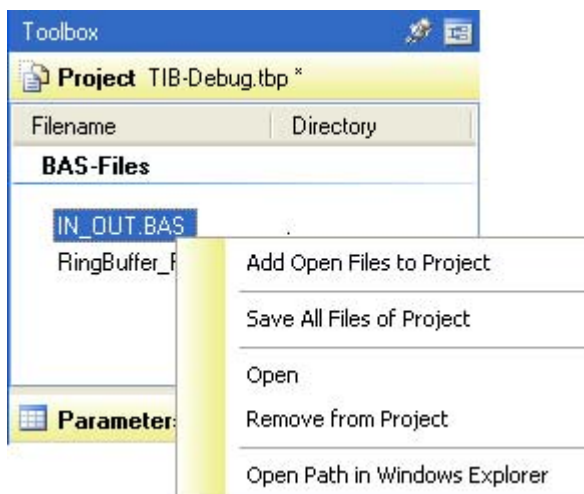



Fig. 5 – The Project Window with the Context Menu

4.9.3 Parameter Window

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

The parameter window is located in the Toolbox (see page 61).

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 grey background color when the communication is inactive.

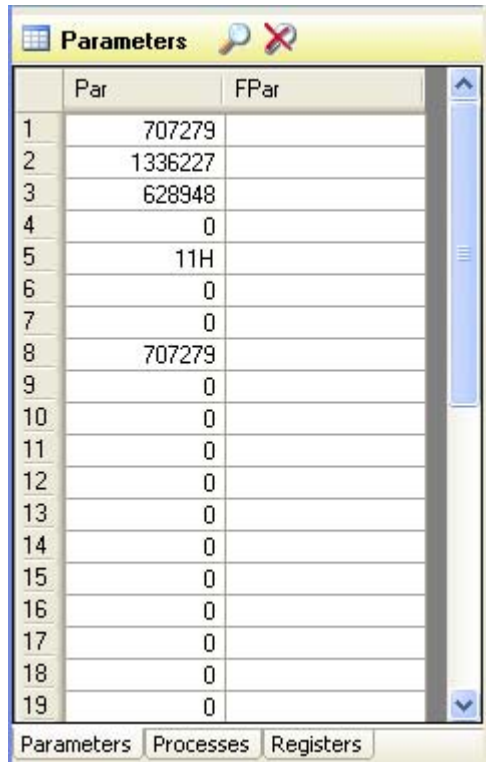



Fig. 6 – The parameter window

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 40.

4.9.4 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 grey.

The process window is located in the Toolbox (see page 61). Open the process window with a click on the tab `Processes`.

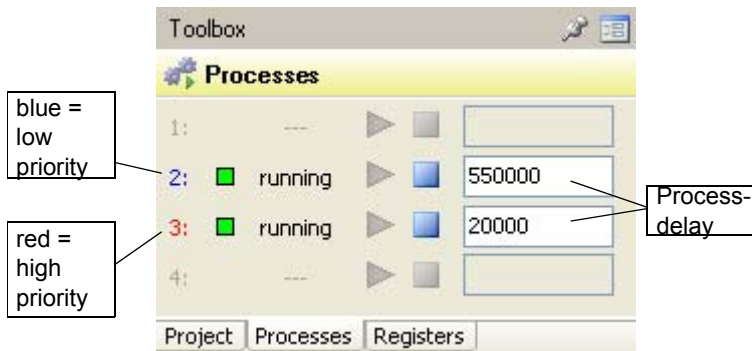




Fig. 7 – The Process Window

For each process the following information is displayed:

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

A process can be stopped with  button and started again with  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 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 7.2.1 "Processdelay", page 113.

- Process runs in debug mode
- Process runs in timing mode

4.9.5 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 61). Open the register window with a click on the tab `Registers`.

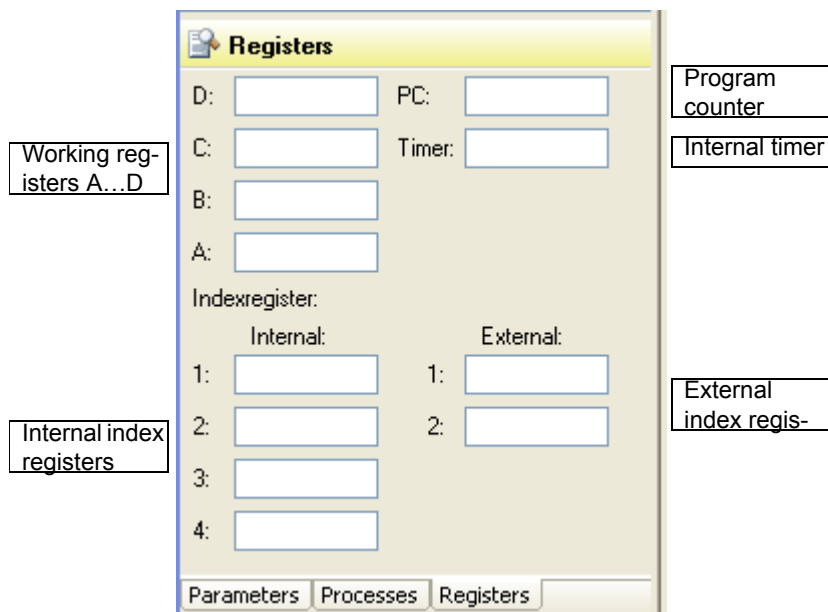
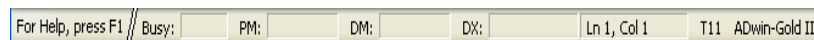


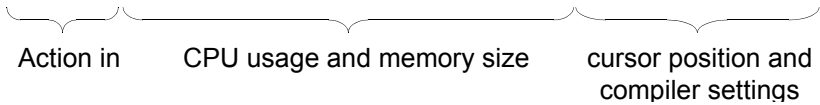
Fig. 8 – Das Register-Fenster

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

4.9.6 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 (bei aktiver Verbindung zwischen PC und *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, timing mode, device no., processor, ADwin hardware).

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:** available internal data memory in bytes.
- **DX / SX:** available external data memory in bytes.

4.10 Info range

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

- Info window
- ToDo List
- Global Variables
- Declarations

4.10.1 Info window

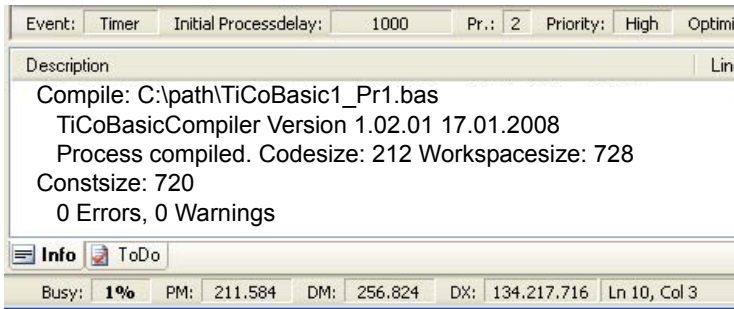
In the info window the compiler messages concerning the current source code are displayed:

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

The 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:



The values be used as hints about the required memory:

- **Codesize:** Size of the created binary file in bytes; the file will be stored in the program memory (PM) as process.
- **Workspacesize:** Required memory size in bytes in the local data memory (DM), being used for
 - local variables and arrays
 - internal purpose (2×4 byte)

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

- Each global array requires about fourty byte in the local data memory (internal purpose).
- Each element of a global array requires 4 byte (in the external data memory; if the array be declared [At DM_Local](#), the elements are stored in the local data memory).
- **Constsize:** Benötigter Speicherplatz für Konstanten in Bytes.
- **Stacksize:** Internal stack size, which is used for libraries.

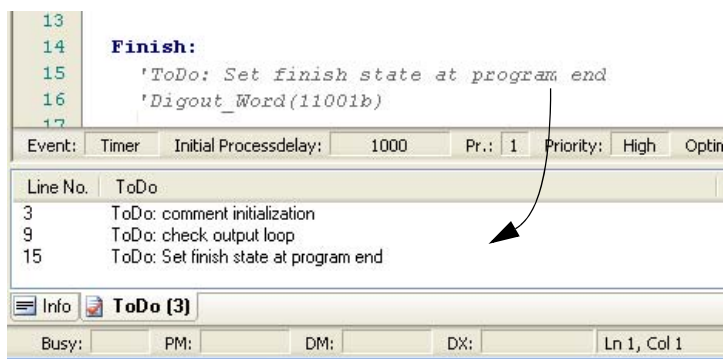
The memory size required in the external data memory (DX) will not be displayed.

4.10.2 ToDo List

The `ToDo` window serves as a simple `ToDo` 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 68).



A double click on a `ToDo` entry positions the cursor in the appropriate line of the source code.







4.10.3 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 40).

The window is part of the Info range (see page 68).

Global Variable	Processfile	Line No.	Comment
Par_1	ADB-SCR-WIN-GlobalVars1.bas	4	ADB-SCR-WIN-GLOBALVARS.INC
Par_1	ADB-SCR-WIN-GlobalVars1.bas	10	
Par_1	ADB-SCR-WIN-GlobalVars1.bas	14	
Par_1	ADB-SCR-WIN-GlobalVars1.bas	16	used 2 times
Par_1	ADB-SCR-WIN-GlobalVars1.bas	17	used 2 times
Par_1	ADB-SCR-WIN-GlobalVars2.bas	8	
Par_2	ADB-SCR-WIN-GlobalVars1.bas	15	
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	6	
Data_5	ADB-SCR-WIN-GlobalVars1.bas	16	
Data_5	ADB-SCR-WIN-GlobalVars2.bas	2	
Data_8	ADB-SCR-WIN-GlobalVars1.bas	5	

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


Busy: PM: DM: DX: Ln 3, Col 1 

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

- the name of the scanned file
- the line number where the variable is called or used.

If the comment contains a file name, the line number refers to this file, else to the scanned file.

- a comment, if
 - the variable is used more than once in the line
 - the variable is used only indirectly.
This case happens if e.g. a function of an include or a library file uses a global variable. The function call in the source code thus uses the global variable indirectly, even though it does not show up in the calling line.

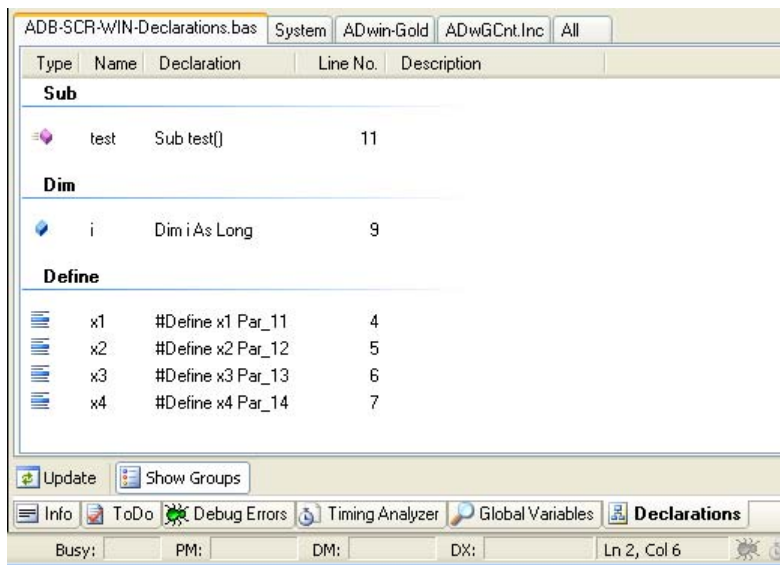
If you change the source code the window is not updated automatically. To do so, use the button `Scan Global Variables`  in the parameter window.

4.10.4 Declarations

The `Declarations` window displays all declarations, include and library files related to a source code file. 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 68).



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 71) and the function "Displaying used global variables and arrays" (page 40).

- ADwin-Gold, ADwin-light-16: Instructions for hardware access, which are implemented in *TiCoBasic* und 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 54) is active.

4.11 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.

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

The following *ADtools* are available:



TDigit

Global variable and array values can be displayed and adjusted.



TGraph

Global array contents can be displayed in a graph.



TButton

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



TLed

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



TMeter

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



TPoti

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



TProcess

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



TPar_FPar

All or selected global variables can be displayed or entered.



TFIFO

Save FIFO array data into a file..



TBin

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



TString

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



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.

5 Programming Processes

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

5.1 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.

jThe source code consists of any number of command lines; each containing an instruction or assignment (exception see : Colon), with up to 255 (ASCII-) characters in one line.

TiCoBasic accepts instructions and variable names in lower and upper case 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 112.

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

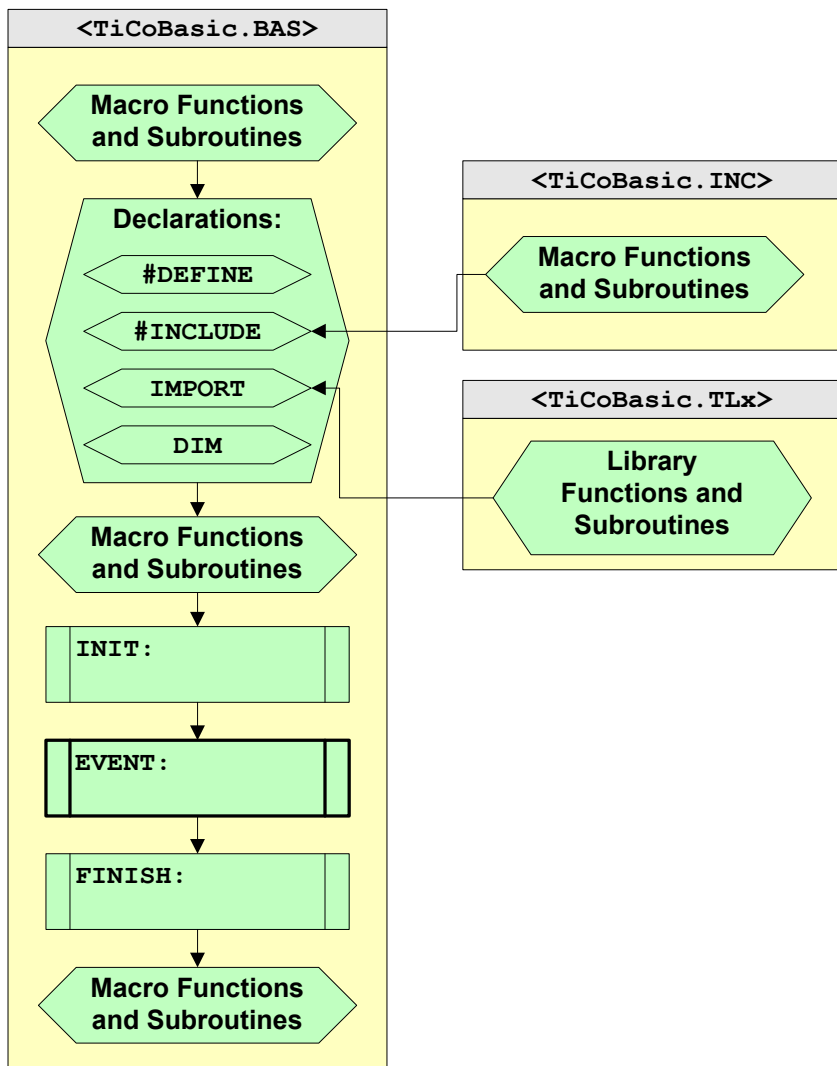


Fig. 9 – Design of an *TiCoBasic* program

5.1.1 The Program Sections

Each of the program sections (see fig. 9) start with a keyword, as described below. All sections have the priority which is set for the process (Process Options dialog box, page 50).

- **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:**.

5.1.2 User defined instructions and variables

Symbolic names

The instruction **#Define** defines symbolic names (see page 135). Group all of these definitions at the beginning of the file and before the start of the program sections.

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

Arrays and Local Variables



In an *TiCoBasic* program the local variables and all arrays must be declared with **Dim** before they can be used (see page 137). 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.

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 5.5.1 on page 99). However, the macro definition cannot be done within the program sections. (see fig. 9 on page 77).

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 5.5.3 on page 100).

5.2 Variables and Arrays

5.2.1 Overview

Data structure	Name	Data type	Notes
Global variables and arrays			
Variable (Scalar)	Par_1...Par_80	Long	
System variable	Processdelay	Long	Pre-defined, not declarable, memory area DM
	Prozessn_Running	Long	
One-dimensional array (vector)	Data_1[]...Data_16[]	Long, Ringbuffer	Name Data_ not changeable, only declaration of array number and dimension.
Local variables and arrays			
Variable (Scalar)	selectable	Long	must be declared
One-dimensional array (vector)	selectable	Long	must be declared

Variables and arrays are normally stored in the internal memory DM (memory map, see chapter 5.3.1), if not determined explicitly.

The data type **Long** has a length of 32-bit.

5.2.2 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 FIFO (a ring buffer which works according to the principle: First in, first out, see chapter 5.3.3 on page 89).

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 5.2.5 and chapter 5.2.6):

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 85).

- Local Variables and Arrays (see page 85):

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 pre-defined; thus, global variables don't 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.



5.2.3 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.

5.2.4 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 93×10^1 , 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.

5.2.5 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 7.3.1 "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. The System Variables, also globally available, are described on page 85.

The global variables can be used anywhere in a program without being declared. Since the variables have an undefined value at program start they should be initialized with a useful value (e.g. zero). Exception: With the transfer of a process to the TiCo processor all global variables **Par_1** ... **Par_80** are automatically initialized with zero. The global variables are also termed parameters and have the names **Par_1**, **Par_2**, ..., **Par_80** with the **Long** data type for 32-bit integer values.

Example

Par_5 = 700	<i>'Parameter 5 contains the</i>	
value 700.		
Par_72 = ADC (1)	<i>'The voltage at the analog</i>	
input 1	<i>'is measured and stored</i>	
into	<i>'parameter 72.</i>	

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

5.2.6 Global Arrays

The global arrays enable the exchange of data between the processes on the *ADwin* system or the *ADwin* CPU (see also chapter 7.3.1 "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).

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

5.2.7 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.

Prozess_n_Running

Returns the status of the process `n` (with `n` = 1...10): the process is running, just being stopped or already stopped (see page 183). 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 180) 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.



5.2.8 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 ("_"). Special characters like german umlauts (Ä, Ö, Ü) are not allowed and there is no case sensitivity. The length of variable names is only limited by the maximum line length (255 characters).

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.

5.3 Variables and Arrays – Details

5.3.1 Variables and Arrays in the Data Memory

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

It is recommended that the internal memory be used 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. 10 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. 10 – Allocation of the Memory Area with Declarations

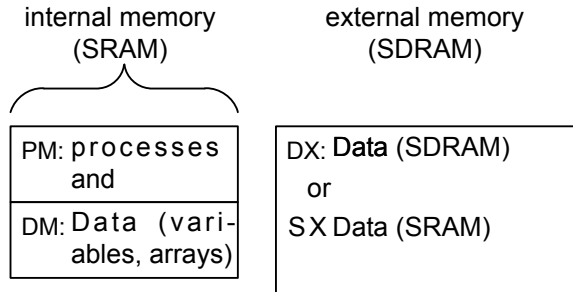
The global variables `Par_1...Par_80` are pre-defined in the internal memory (DM), therefore they cannot be re-declared in the external memory (DX).



5.3.2 Memory Areas

The *TiCopro*cessor uses a fast internal memory (SRAM) and—if existing—a huge external memory (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 The Data structure Ringbuffer (see page 89). Please also note chapter 6.2.6 on page 107.


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.

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 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.


5.3.3 The 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 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 ele-

ment 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`:

```

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 186.

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)

```

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 7.3.1 on page 116) 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 ringbuffer_for_read at
dm_local
Dim DATA_5[500] as long as ringbuffer_for_write at
dm_local
Dim free,value1 As Long
```

Init:

```
Rem Initialize read ringbuffer DATA_5
RingBuffer_Clear(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 variable. 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
Dim DATA_1[500] as long as ringbuffer_for_Write at
dm_local
Dim free,used,value1 As Long

Init:
  Rem Initialize write ringbuffer DATA_1
  RingBuffer_Clear(1)

Event:
  Rem Are there elements free in Data_1 for writing?
  Rem Par_5 contains the number of free elements and is
  set
  Rem in ADbasic
  free = Ringbuffer_Empty(1,5)
  If (free > 0) Then
    DATA_5 = value1
  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)
Rem Data has been read, and are processed here
EndIf
```

5.4 Expressions

5.4.1 Evaluation of Operators

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

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

For the evaluation of an expression, it is important to understand the order in which the operators are used. The operators are divided into categories, which are resolved according to priorities: A category of higher priority is processed before a category of lower priority (see fig. 10).

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

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

5.5 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` 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 8 "Instruction Reference".

5.5.1 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 146: `Function ... EndFunction`; page 202: `Sub ... EndSub`).

5.5.2 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 77).


5.5.3 Libraries

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

The library modules are similar to the subroutine and function macros. They are created in a source code file using the `Lib_Sub ...`

`Lib_EndSub` and/or `Lib_Function ... Lib_EndFunction` instructions. The library file is then compiled using the `Build / Make lib file` menu option.

Also, calling library modules several times does not increase the size of the binary file. Compared to macro functions and subroutines, library modules require less memory when they are called more than once. However, additional execution time is needed for calling them (compare to chapter 5.5.1 "Subroutine and Function Macros").

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

When interlacing libraries (including a library within another library), the source code calling the libraries must include all levels (see fig. 12), 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 161: `Lib_Function ... Lib_EndFunction`; page 166: `Lib_Sub ... Lib_EndSub`).

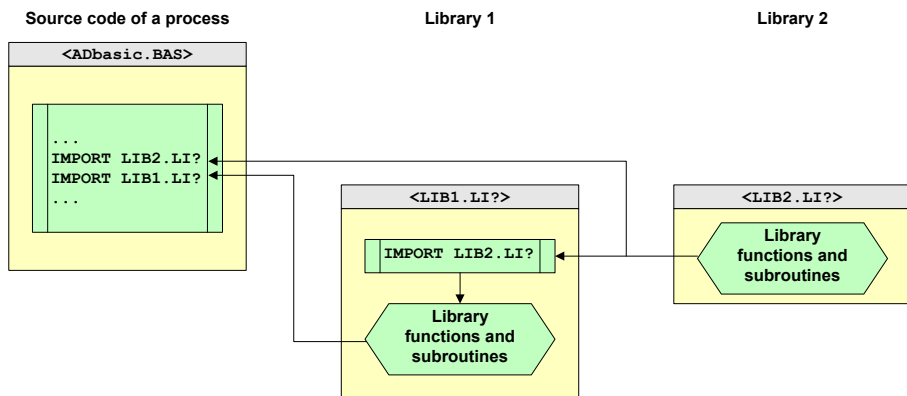


Fig. 12 – Interlaced Libraries

6 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.

6.1 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 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 $37 \times 20\text{ns} = 740\text{ns}$.

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
```

6.2 Useful Information

6.2.1 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.

The addresses for the control and data registers can be found in the relevant hardware manual.

6.2.2 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.

6.2.3 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 ADC (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 = ReadADC(1)      'Read out ADC1
Par_2 = ReadADC(2)      'Read out ADC2

```

6.2.4 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.

6.2.5 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 multiplexe setting is carried out during the A/D conversion of the other channels. Both measurement processes are overlapped: The start of conversion for the channels 1+2 is followed by setting the multiplexer for the channels 3+4.



Example

REM Example for Gold Rev. B

Init:

```
Set_Mux(000000b)      'Set Mux for first
measurement,           'channels 1+2

Sleep(140)             'Wait 14 µs
```

Event:

```
Start_Conv(11b)        'Start conversion
(channels 1+2)

Set_Mux(001001b)       'Set Mux, channels 3+4
Wait_EOC(11b)          'Wait for end of
conversion              ' (channels 1+2)

Par_1 = ReadADC(1)      'Read out ADC1, channel 1
Par_2 = ReadADC(2)      'Read out ADC2, channel 2

Start_Conv(11b)        'Start
conversion(channels 3+4)

Set_Mux(000000b)       'Set Mux, channels 1+2
Wait_EOC(11b)          'Wait for end of
conversion              ' (channels 3+4)

Par_3 = ReadADC(1)      'Read out ADC1, channel 3
Par_4 = ReadADC(2)      'Read out ADC2, channel 4
```

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



It is very important that adequate delay for the multiplexer settling time and A/D conversions be provided or incorrect measurements or A/D conversion failures may be obtained. There are some hints in chapter 6.2.4 "Setting Waiting Times Exactly".

6.2.6 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 5.3.3 on page 89.

7 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 program¹ 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 environ-

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

ment *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 4.6.2 "Programming the TiCo bootloader", page 41.

7.1 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) 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 4.9.2 on page 61).

7.1.1 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 20ns (see also chapter 5.2.7 on page 85).

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 20ns without jitter.
- The maximum latency from the process call by an event signal until execution of the process begins is 120ns.
- 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.



7.1.2 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:

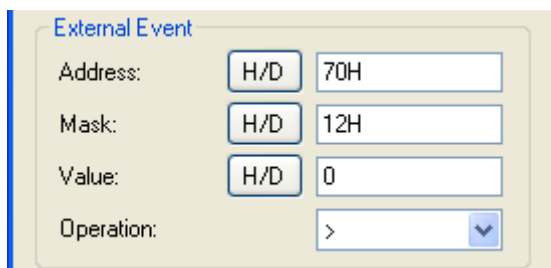
- The output of signals can be set in intervals of 20ns without jitter.
- The maximum latency from the process call by an event signal until execution of the process begins is 120ns.
- A high-priority process cycle cannot be interrupted and is always completely processed.

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 50.

The hardware addresses are different for each *ADwin* hardware.



External Event		
Address:	H/D	70H
Mask:	H/D	12H
Value:	H/D	0
Operation:		>

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.

7.1.3 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.

7.2 Time Characteristics of Processes

7.2.1 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. It 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 180).

One timer interval has 20 ns.

For instance, a processdelay with the value 1250 means that a process is called in time intervals of $1250 \times 20\text{ ns} = 25\text{ }\mu\text{s}$. 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).

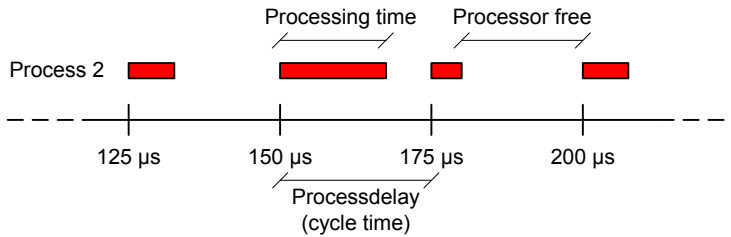


Fig. 13 – Processdelay and processing time



Example

If an extensive calculation is executed only every, say 1000 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.

7.2.2 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 4.9.6). 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.

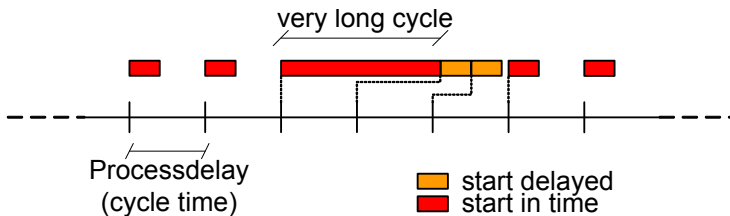
7.2.3 Different Operating Modes in the Operating System


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 predefined time (via `Processdelay`, page 113). 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 42.9 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:**).

7.3 Communication

7.3.1 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 5.5.2 "Include-Files"). This is different for The Data structure Ringbuffer (see chapter 5.3.3 on page 89); 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 setpoint of the control loop.

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

7.3.2 Communication with the *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 pro-

cesses 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.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.

7.3.3 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 8.3 on page 206.
- *ADwin-Pro II*: chapter 8.4 on page 254.

7.3.4 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.

8 Instruction Reference

Below, the available *TiCoBasic* instructions for *TiCo* processors are listed. Instructions for inputs/outputs be found 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 8.3 and chapter 8.4 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*.

8.1 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**).

8.2 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 5.4.1 "Evaluation of Operators" on page 97).

See also

+ Addition, - Subtraction, * Multiplication, ^ Power

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 5.4.1 "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
```

#..., Preprocessor Statement

An *TiCoBasic* instruction beginning with the "#" sign instructs the preprocessor to treat the following source code differently. The output of the preprocessor is further processed by the compiler.

The following preprocessor statements are available:

- | | |
|---------------------|---|
| #Define | Definition of symbolic constants: Search and replace character strings in the source code with other character strings. |
| #Include | Include a file: Insert a file (with source code) into the source code. |
| #If...#EndIf | Conditional compilation: If the condition is true the corresponding code lines are compiled, otherwise deleted. |

: Colon

The sign ":" separates more than one instruction within a single line.

Syntax

```
[Step_1] : [Step_2] {: [Step_3] ...}
```

Notes

[Step_n] refers to any program instruction as is otherwise indicated in one individual program line.

A program line must not be longer than 255 characters (exception see **#Include** on page 158).

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

Example

```
Inc Par_1 : Inc Par_2  
'Increase Par_1 and Par_2 in *one* line
```

=, Assignment

The operator "=" assigns the result of the expression on the right side of the operator to the variable or the array element on the left side of the operator.

Syntax

```
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 'Assignment of a constant
```

Event:

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

< = > Comparison

The operators "<", "=" and ">" are used to compare two values. In *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

- / -

Example

```
Dim val_1, val_2 AS Long
```

Event:

```
val_1 = -5
```

```
val_2 = AbsI(val_1)
```

```
'Result: val_2 = 5
```


And

The operator `And` combines two integer values bit by bit or two Boolean expressions as Boolean operator.

Syntax

```
var = val_1 And val_2           'bitwise  
operator  
  
If ((expr1) And (expr2)) Then  'Boolean  
operator
```

Parameters

`val_1, val_2` Integer value.

LONG

`expr1, expr2` Boolean operator with the value "true" or "false".

LOGIC

Notes

With `And` you can only combine expressions of the same type (integer or Boolean) with each other, mixing them is not possible.

You can use Boolean operators only in statements such as `If ... Then ... Else` or `Do ... Until` (variables cannot have Boolean values).

If you use several Boolean operators in one line, you have to put each operation into separate parentheses. This is not necessary for combining integer values.

See also

Not, Or, XOr

Example

```
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
```

Data_n

The `Dim Data_n[...]` `AS ...` instruction dimensions a global **DATA** array.

More information about dimensing see page 137.

Syntax

```
Dim Data_n[dim1] AS Long
    {AT <mem_type>}
```

Parameters

Data_n	Name of the declared DATA array (n: 1...16).	
dim1, dim2	Array size: Number (≥ 1) of the array elements of the type <code>Arr_type</code> .	CONST LONG
<mem_type>	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...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 83, "Variables and Arrays in the Data Memory" on page 86

Example

```
Rem Dimension the global array Data_15 with
Rem 1000 long elements
Dim Data_15[1000] AS Long
```

Dec

Dec decrements the value of aLong-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

Example

```
Dim index AS Long
```

```
Dim Data_1[1000] AS Long
```

```
Init:
```

```
index=1000
```

```
Event:
```

```
DAC(1,Data_1[index])
```

'Output the value on DAC1

```
Dec(index)
```

'Decrement the index by 1

```
If (index<1) Then
```

```
index=1000
```

'Start 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

name	Symbolic name, <i>without</i> quotation marks.	CONST
	Special chars are not allowed, only alphanumeric characters (a...z, A...Z, 0...9) and the underscore (_).	STRING
expression	Expression for the symbolic name, <i>without</i> quotation marks.	CONST
n	All characters are allowed.	STRING

Notes

Place this instruction at the beginning of a source code.



The function **#Define** is a preprocessor instruction, that means the replacement is made when you compile the source code (even before the compiler generates the program). Use this function in order to use more descriptive names in the source code instead of constants, parameters or expressions.

The first string up to a blank is interpreted as symbolic name, the following text until the carriage return is interpreted as an expression to be inserted¹. The expression is inserted exactly as you have defined it; variable names in the expression are not replaced by their value, but as a character string.

Neither **name** nor **expression** are case-sensitive.

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

1. Text behind a comment char " " will be ignored by the compiler.

See also

#Include

Example

```
#Define setpoint Par_1           'Comments like this are
ignored
#Define measured Data_1
#Define 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 5.2.3, information about ringbuffer arrays under the heading Ringbuffer on page 186.

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) of the array elements of the type <code>Long</code> . CONST
<code><mem_type></code>	Memory where the variables are stored: DRAM_EXTERN : external memory. SRAM_Extern : external data memory in Pro II modules (instead of DRAM). DM_LOCAL : local memory (default). <code>LONG</code>

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 ringbuffers, 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 5.3.3 on page 89.

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.

See also

Data_n, Event:, Ringbuffer, Finish:, Init:, "Variables and Arrays in the Data Memory" on page 86

Example

```
Rem Dimension var1 as long variable
```

```
Dim var1 AS Long
```

```
Rem Dimension the local array "array1" with 1000 long elements
```

```
Dim array1[1000] AS Long
```

```
Rem Dimension the global array Data_20 with
```

```
Rem 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
    ...
Until (condition)
```

'Instruction block'

Parameters

`condition` Boolean abort condition with the operators `<`, `>`, `=`, `LOGIC` **AND** and **OR**.

See also

`< = >` Comparison, And, Or, For ... To ... {Step ...} Next, Select-Case

Notes

You can nest `Do...Until` loops repeatedly; only the available memory size will limit the number of nested loops.

Avoid loops with long execution times in high-priority processes, because they cannot be interrupted.

Example

```
Dim count AS Long
Dim DATA_1[103] AS Long AS Ringbuffer_For_Write

Init:
    count = 1

Event:
    Do                                     'Start loop
        Data_1 = ADC(1,4)                 'Read out measurement
value
        Inc count                         'Increase count variable
    Until (count > 103)                   'Are 100 measurements
being made?
```

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

Example

```
Event:
  If (ADC(1) > 3000) Then 'Measure and compare
    End                      'End process, but execute
Finish:
  EndIf

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

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 5.1.1 on page 78.

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 7 "Processes in the ADwin System".

With processor module Pro-CPU T11, the memory area can only be set starting with revision E04.

See also

Dim, Init:, Finish:

Example

```
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 5.1.1 on page 78.

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.

With processor module Pro-CPU T11, the memory area can only be set starting with revision E04.

See also

Dim, Init:, Event:, ProcessN_Running

Example

```
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 (≥ 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

Example

- / -

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

<code>macro_name</code>	Name of the function and of the return value, data type <code>Long</code> .
<code>val_1, val_2</code>	Names of passed parameters; for arrays use the syntax with dimension brackets: <code>array[]</code> or <code>Data_n[]</code> .

LONG

Notes

You will find general information about macros in chapter 5.5.1 on page 99.

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 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 va-



riable or a single array element. If so, a passed parameter can be used to hold a return value.

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

Example

```
Function sumsquare(w1, w2, w3) AS Long
  Rem The function calculates the square of the sum of the
  Rem values w1, w2 und 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:

```
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):

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

For `array` you can indicate a global or a local 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 <, >, =, **AND** LOGIC and **OR**.
If the condition is "true" the instructions after **Then** are executed.

instr Instruction (corresponds to an instruction line).

Notes

You can nest **If** structures repeatedly; only limited by the available memory.

The instruction block after **Else** (if there is one) is executed faster than the one after **If...Then**. This can be used to speed up the total execution time of the **Event**: section, by putting the condition that has most common state, into the **Else** statement, for instance when you check if limit values are exceeded.

In the single-line version, the instruction cannot call a subroutine macro (**Sub**) nor a function macro (**Function**).

See also

< = > Comparison, And, Or, Do ... Until, SelectCase

Example

```
Dim val AS Long                                'Declaration

Event:
    val = ADC(1)                                'Acquire measurement
    value

    If (val > 3000) Then                          'Limit value is exceeded:
        CLEAR_DIGOUT(1)                          'Reset DIGOUT 1
        SET_DIGOUT(0)                            'Set DIGOUT 0
    Else                                          'Limit value is not
exceeded:
        CLEAR_DIGOUT(0)                          'Reset DIGOUT 0
        SET_DIGOUT(1)                            'Set DIGOUT 1
    EndIf                                        'End of control structure
```

#If ... Then ... {#Else ... } #EndIf

This preprocessor structure is used to conditionally compile a block of instructions (**#If...Then...#Else...#EndIf**).

Syntax

```
#If condition Then
    ...
    'instruction block
{#Else
    'the Else-block is optional
    ...
    'instruction block}
#EndIf
```

Parameters

condition Boolean condition (no braces or quotation marks) LOGIC
of the form:

<SYSPAR> = value

If the condition is "true" the instructions after **Then** are executed.

The system parameter <SYSPAR> and the corresponding value are shown in the table below:

<SYSPAR>	value	Meaning
ADwin_ SYSTEM	ADWIN_GOLDII ADWIN_PROII	"System" setting in the window "Compiler Options".
Processor	TiCol	"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*

Example

```
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.	CONST STRING
path	Path name of the library file (with drive), without quotes.	CONST STRING

Notes

General information about include files to be found in chapter 5.5.2 on page 100.

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

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

If the path name misses, only the standard directory is searched (see Options Menu, Directory, page 57). Use the back slash "\" in the path name to separate directory names.

The base directory for relative paths is—if the source code is member of a project—the directory of the project file, otherwise the directory of the source code file.

The following library files are delivered with *TiCoBasic*:

math.tl1	Special mathematics instructions.
----------	-----------------------------------

See also

#Include

Example

```
Import math.tl1           'import mathe library
Rem import a user library
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)
```

You will find an overview of the register addresses (*Gold* and *Light-16*) in your hardware documentation.

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

Example

```
Dim index AS Long
Dim Data_1[1000] AS Long

Init:
    index=1

Event:
    Data_1[index] = ADC(1) 'Transfer the measurement value
into
                                'the array
    Inc(index)                'Increment index by 1
    If (index>1000) Then End 'End the program after
                                '1000 measurements
```

#Include

#Include includes all the contents of an include file into the source code.

Syntax

```
#Include {path}filename
```

Parameters

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

Notes

You find general information about include files in chapter 5.5.2 on page 100.

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 Options Menu Directory, page 57). 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.

Please note: A program line with an **#Include** instruction should not exceed 136 characters (maximum length for other lines see page 127). Any further character of this line will not be processed by the compiler.



See also

#Define, Import, Function ... EndFunction, Sub ... EndSub

Example

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 the

Rem project file (if the source file is member of 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 5.1.1 on page 78.

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.

With processor module Pro-CPU T11, the memory area can only be set starting with revision E04.

See also

Dim, Event:, Finish:

Example

```
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><FCT_type></code> .
<code><FCT_TYPE></code>	Data type: <code>Float</code> , <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><BY_TYPE></code>	Methods for the transfer of parameters: <code>Byref</code> : pass reference (pointer) to variable or array. <code>Byval</code> : pass value only.
<code><VAR_TYPE></code>	Data type: <code>Float</code> , <code>Long</code> , <code>String</code> .
<code><mem_type></code>	Useful for processor T10 only: Type of memory, where the passed parameters are stored; to be used only with arrays: <code>DRAM_EXTERN</code> : external memory. <code>DM_LOCAL</code> : local memory.

Notes

You will find general information about library files in [chapter 5.5.3 on page 100](#).

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 methods for passing parameters that differ as follows:

- **BYREF**: The library function can change the parameter, so that the changed value is available in the program (the address of the parameter is transferred).
- **BYVAL**: The library function can only access the value of the parameter, but cannot change it. Thus, the parameter remains the same for the program that calls the function.

Passed parameters should always be declared `AT <mem_type>`, to save valuable processor time (<mem_type> must fit with the declaration of the passed parameters in the calling program, see `Dim`). If not, the library function has to detect the parameter's memory type at run time.



If an array is passed as parameter, the syntax for definition and call differs:

- Definition of the library function's parameter *with* brackets:
`Lib_Function funcname (... array[] ...)`
- Call with the parameter *without* brackets:
`ret_val=funcname(... array ...)`

If arrays are used as passed parameters always define them as `Byref` and without indicating any array size. You cannot use FIFO arrays as passed parameters.

See also

`Lib_Sub ... Lib_EndSub`, `Import`, `Function ... EndFunction`, `Sub ... EndSub`

Example

```
'----- Calculate a mean value -----  
Lib_Function average(BYREF array[] AS Long, BYVAL ptr  
AS Long,  
    BYVAL cnt AS Long) AS Long  
    Dim i AS Long  
    average = 0  
    If (cnt > 0) Then  
        FOR i = ptr TO (ptr + cnt)  
            average = average + array[i]  
        NEXT i  
        average = average / cnt  
    EndIf  
Lib_EndFunction
```

Calling the library function **average** is illustrated in the following example, a "moving average filter":

```

Rem Import the library 'MEAN'
Import C:\MyFiles\ADwinLibs\MEAN.tl1
#Define cnt 10 'Number of the samples
#Define samples Data_1 'Number of measm. values
#Define filtered Data_2 'Number of filtered measm.
                        'values
#Define length 1000 'Length of the array
Dim samples[length] AS Long 'Source array
Dim filtered[length] AS Long 'Destination array
Dim i AS Long 'Count variable

Init:
    i = 1 'Initialize the count
variable
    Processdelay = 40000 'Measurement with 1 kHz

Event:
    samples[i] = ADC(1) 'Measure and save analog
values
    Inc i 'Increment count variable
    If (i > length) Then End 'Are 1000 measurements
complete?
                        'If yes: process Finish

Finish:
    FOR i = 1 TO (length - cnt) 'For all measm. values
        Rem Call library function "average"
        filtered[i + cnt] = average(samples,i,cnt)
        Rem Note the call with the passed array 'samples'
        Rem *without* dimension brackets
    NEXT i

```

Lib_Sub ... Lib_EndSub

The `Lib_Sub...Lib_EndSub` is used to define a subroutine with passed parameters in a library file.

Syntax

```
Lib_Sub 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><By_type></code>	Methods for the transfer of parameters: <code>Byref</code> : pass reference (pointer) to variable and array. <code>Byval</code> : pass value only.
<code><var_type></code>	Data types: <code>Float</code> , <code>Long</code> , <code>String</code> .

Notes

You will find general information about library files in [chapter 5.5.3 on page 100](#).

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, so that the changed value is available in the program (the method transfers the address of the parameter).
- `BYVAL`: The library function can only access the value of the parameter, but cannot change it. Thus, the parameter remains the same for the program that calls the function.

If an array is passed as parameter, the syntax for definition and call differs:

- Definition of the library subroutine's parameter *with* brackets: `Lib_Sub subname (... array[] ...)`
- Call with the parameter *without* brackets:
`subname (... array ...)`

If arrays are used as passed parameters always define them as `Byref` and without indicating any array size. You cannot use FIFO arrays as passed parameters.

See also

Lib_Function ... Lib_EndFunction, Import, Function ... End-Function, Sub ... EndSub

Example:

```
Rem Measurement value conversion from Digits(0...65535)  
Rem to Volt(±10V)  
Lib_Sub dig2volt(BYREF digit[] AS Long, BYVAL ptr AS  
Long,  
    BYVAL cnt AS Long, BYVAL gain AS Long,  
    BYREF volt[] AS Float)  
Dim i AS Long  
FOR i = ptr TO (ptr + cnt)  
    volt[i] = ((digit[i] * 20 / 65536) - 10) / gain  
NEXT i  
Lib_EndSub
```

Calling the library function `dig2volt` is illustrated in the following example, a conversion of measurement values:

```
Rem The library 'DIG2VOLT' is imported
Import C:\MyFiles\ADwinLibs\DIG2VOLT.tl1

#Define cnt 1000                'Number of the samples
#Define ptr 1                   'Start point of the
samples which are              'to be converted

#Define gain 1                  'Gain of the PGA
#Define samples Data_1          'Memory for measurement
values
#Define scaled Data_2           'Memory for converted
measurement                    'values
#Define length 1000             'Length of the array

Dim samples[length] AS Long    'Source array
Dim i AS Long                  'Count variable

Init:
    i = 1                      'Initialize the count
variable
    Processdelay = 40000       'Measurement with 1 kHz

Event:
samples[i] = ADC(1)            'Measure and save analog
values
    Inc i                      'Increment count variable
    If (i > length) Then End    'Are 1000 measurements being
made?
                                'If yes: process Finish

Finish:
    Rem Convert the measurement values by
    Rem calling the library subroutine 'dig2volt'
    dig2volt(samples,ptr,cnt,gain,scaled)
    Rem Note the call with the passed array 'samples'
    Rem *without* dimension brackets
```

With processor module Pro-CPU T11, the memory area can only be set starting with revision E04.

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

Example

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

<code>val_1</code>	Compared value 1	LONG
<code>val_2</code>	Compared value 2	LONG
<code>ret_val</code>	The smaller of both values.	LONG

Notes

- / -

See also

AbsI, Max_Long

Example

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, which is 20ns.

With this instruction you can delay for a necessary waiting period (e.g. after **Set_Mux**) if there is no other use of processing time.

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 (≥ 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

- / -

Not inverts the bits of an argument.

```
ret_val = Not(val)
```

val Value to be inverted (no logic expression).

LONG

`ret_val` Inverted argument.

LONG

Not runs with bits only, not with Boolean expressions. Therefore you cannot negate logic expressions (true / false) with it. Not allowed: **Not** (**Par_2** > **2**) .

And, If ... Then ... {Else ...} EndIf, Or, XOr

```
Dim val1 AS Long
Dim val2 AS Long
```

```
val1 = -3
```

$\text{Rem } -3 = 1111111111111111111111111111111101b$

```
val2 = Not(val1)           'Result: val2 = 010b = 2
```

Or

The operator `Or` combines two integer values bit wise or two Boolean expressions as a Boolean operator.

Syntax

```
ret_val = val_1 Or val_2 ...val_2    'bit wise
operator
```

```
If ((expr1 Or (expr2)) Then        'Boolean
operator
```

Parameters

`val_1, val_2` Integer value.

LONG

`expr1, expr2` Boolean expression with the value "true" or "false".

LOGIC

Notes

With `Or` you can only combine expressions of the same type (integer or Boolean) with each other, mixing them is not possible.

You can use Boolean operators only in statements such as `If ... Then ... Else` or `Do ... Until` (variables cannot have Boolean values).

If you use several Boolean operators in one line, you have to put each operation into parentheses. This is not necessary for combining of integer values.

See also

And, If ... Then ... {Else ...} EndIf, Not, XOr

Example

Bit wise operator:

```
Dim val1, val2, val3 AS Long
```

```
val1 = 0100b
```

```
val2 = 0110b
```

```
val3 = val1 Or val2
```

'Result: val3 = 0110b

Boolean operator:

```
Dim x AS Long
```

```
Dim val4 AS Long
```

Init:

```
x = 15
```

Event:

```
If ((x < 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

addr Address of the memory location to be written.

LONG

value 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 (≥ 1) of clock cycles.

LONG

Notes

In a time-controlled process the section **Event:** is called repeatedly and in fixed time intervals by the internal counter. The time interval between two cyclic calls is called process delay and is counted in clock cycles.

The time interval of the Processdelay is 20ns.

Processor	Priority	
	High	Low
T9	25ns	100µs
T10	25ns	50µs
T11	3,3ns	3,3ns = 0,003µs

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. If this delay cannot be caught up within 250ms, the communication between the *ADwin* system and the computer can be interrupted.

You may set a constant process delay by assigning a value to the variable **Processdelay** in the section **Init:**. 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 (process delay) will be changed immediately. This may be critical especially when the cycle time has been shortened: Make sure that the execution time of the program remains less than the newly set cycle time.

See also

Read_Timer

Example

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
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
```

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

Example

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 20ns.

LONG

Notes

The counter value cannot be written.

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 42.9s

See also

Processdelay

Example

```
Dim timervalue AS Long
```

Event:

```
timervalue = Read_Timer()
```

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.

```
Rem comment
instr : Rem comment
instr 'comment
```

<i>comment</i>	Any character strings.
<i>instr</i>	<i>TiCoBasic</i> instruction.

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.

```
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

DATA_n	Name of the declared DATA array (n: 1...16).	
length	Array size: Number (≥ 1) of the array elements of the type <code>Long</code> .	CONST
<mem_type>	memory, where the array elements are stored: DRAM_EXTERN : external data memory. Here the value range for length is to be set in steps of 8: $\text{length} = 8 \times a + 7; a \geq 0$ SRAM_Extern : external data memory in Pro II modules (instead of DRAM). DM_LOCAL : internal data memory (default).	LONG

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 5.3.3 on page 89.

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 **FIFO_Empty** and **FIFO_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 83, "The Data structure Ringbuffer" on page 89

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

data_no Number (1...16) of the ringbuffer array data_no. LONG

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, "The Data structure Ringbuffer" on page 89

Example

```
Rem Use global array DATA_20 as ringbuffer
Dim DATA_12[999] As Long As Ringbuffer_For_Read At
DRAM_Extern

Init:
    Rem initialize ringbuffer
    RingBuffer_Clear(12)
    Rem wait until the 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

data_no	Number (1...16) of the ringbuffer array data_no .	LONG
par_x	For data exchange <i>ADwin</i> CPU / <i>TiCo</i> only: Global variable (Par_1...Par_80), which contains a copy of the write or read pointer (handled by the <i>ADwin</i> CPU) to the ringbuffer. For any other use: Any variable, where the value may be changed.	VAR 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** only has a meaning for data exchange between *ADwin* CPU and *TiCo* processor. If so, the *ADwin* CPU sets the **par_x** value to the number of values written into or read from the ringbuffer; thus, **RingBuffer_Empty** can calculate the number of free elements or **RingBuffer_Empty** the number of used elements in the ringbuffer.

If you initialize a ringbuffer with **Ringbuffer_Clear** while exchanging data between ADwin CPU and TiCo processor, you have to do as follows:

1. Make sure, that from now on no data is written into the ringbuffer or read from it.
2. Re-initialize the data exchange in ADbasic with **TDrv_Init**.
3. Now you can resume to work with the ringbuffer.

See also

Ringbuffer (declaration), Refresh_RingBuffer, Ringbuffer_Empty, RingBuffer_Full, "The Data structure Ringbuffer" on page 89

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

data_no	Number (1...16) of the Data ringbuffer array.	LONG
par_x	For data exchange <i>ADwin</i> CPU / <i>TiCo</i> : Global variable (Par_1...Par_80), which contains a copy of the read pointer (handled by the <i>ADwin</i> CPU) to the ringbuffer.	VAR LONG
	For data exchange with external memory: 0.	
ret_val	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 186).

The global variable **par_x** only has a meaning for data exchange between *ADwin* CPU (e.g. T11) and *TiCo* processor. If so, the *ADwin* CPU reads data from the ringbuffer. After each read cycle with **Get_TiCo_RingBuffer**, the *ADwin* CPU updates its read pointer and copies the pointer value to the global variable **par_x** of the *TiCo* processor. Using the **par_x** value, **RingBuffer_Empty** calculates the number of free elements of the ringbuffer.

See also

Ringbuffer (declaration), RingBuffer_Full, Get_TiCo_Ring-Buffer (*ADbasic*), "The Data structure Ringbuffer" on page 89

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 (using
    PAR_4)
    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

data_no	Number (1...16) of the Data ringbuffer array.	LONG
par_x	For data exchange <i>ADwin</i> CPU / <i>TiCo</i> : Global variable (Par_1...Par_80), which contains a copy of the write pointer (handled by the <i>ADwin</i> CPU) to the ringbuffer.	VAR LONG
	For data exchange with external memory: 0.	
ret_val	Number of the 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 186).

The global variable **par_x** only has a meaning for data exchange between *ADwin* CPU and *TiCo* processor. If so, the *ADwin* CPU writes data into the ringbuffer. After each write cycle with **Set_TiCo_RingBuffer**, the *ADwin* CPU updates its write pointer and copies the pointer value to the global variable **par_x** of the *TiCo* processor. Using the **par_x** value, **RingBuffer_Full** calculates the number of used elements of the ringbuffer.

See also

Ringbuffer (declaration), Ringbuffer_Empty, Set_TiCo_Ring-Buffer (*ADbasic*), "The Data structure Ringbuffer" on page 89

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 (using PAR_
    3)
    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

Notes

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

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 = $[(\text{greatest constant value}) + 1] \times 4$

As an example the memory requirement with a max. `Case` constant `200` is $(200 + 1) \times 4 = 804$ Bytes; the maximum possible memory requirement is 1 KiB.

See also

Do ... Until, For ... To ... {Step ...} Next, If ... Then ... {Else ...}
EndIf

Example**Event:**

```

Par_1=2
SelectCase Par_1
  Case 0
    Par_10 = ADC(1)
  Case 1
    Par_10 = ADC(3)
  Case 2
    Par_10 = ADC(5)
ADC(7), too

  CCase 3
    Par_11 = ADC(7)
  Case 4,5,6,7,16
16?
    Par_2 = DIGIN_WORD()
  CaseElse
    DIGOUT_WORD(Par_10)
the

EndSelect

```

```

'Evaluate Par_1
'If Par_1 = 0?
'Read out ADC(1)
'If Par_1 = 1?
'Read out ADC(3)
'If Par_1 = 2?
'read out ADC(5) and

'(by CCase)
'If Par_1 = 3?
'Read out ADC(7)
'If Par_1 = 4, 5, 6, 7 or

'read digital inputs
'Par_1: other values
'Output value of Par_10 to

'digital outputs
'End of selection

```

Shift_Left

The **Shift_Left** instruction shifts all bits of a value by a specified number of places to the left. The empty bits at the right are filled with zeroes.

Syntax

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

Parameters

val	Argument.	LONG
num	Number of places the argument is shifted (0...31).	LONG
ret_val	Argument with shifted bits or. 0 for (num<0) and for (num>31).	LONG

Notes

Shifting the bits n places to the left corresponds to the multiplication with 2^n . A possible overflow is not taken into account, which means, a set bit is lost if it is left-shifted beyond the length of an argument.

The execution time is similar to that one of a comparable multiplication operator.

See also

Shift_Right

Example

```
Dim val1, val2 AS Long
```

Event:

```
val1 = 1024  
val2 = Shift_Left(val1, 2) 'Result: val2=4096
```

Shift_Right

The **Shift_Right** instruction shifts all bits of a value by a specified number of places to the right. The empty bits at the left are filled with zeroes.

Syntax

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

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

Example

```
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 (≥ 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 μs (=25*100ns) =
settling               'time of the MUX
  START_CONV(1)        'Start conversion
```

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

`macro_` Name of the subroutine.
`name`

`val_1, val_` Name of the passed parameter;
`2` for arrays use the syntax with dimension brackets:
`array[]` or `Data_n[]`.

LONG

Notes

You will find general information about macros in chapter 5.5.1 on page 99.

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. In front of the section `Init:`
2. After the 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 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

Example

- / -

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

Example

```
Dim value AS Long
```

Event:

```
value = 0100b XOr 0110b
```

```
Rem Result: value = (4 XOr 6) = 0010b = 2
```


8.3 Gold II: *TiCo* processor

This section describes *ADbasic* instructions which allow the *ADwin* CPU (T11) to access the *TiCo* processor.

Initialize	TDrv_Init
Global variables	Get_Par, Get_Par_Block Set_Par, Set_Par_Block
Global arrays	GetData_Long, SetData_Long
Ringbuffer	Get_TiCo_RingBuffer, RingBuffer_Empty Set_TiCo_RingBuffer, RingBuffer_Full
Processdelay	TiCo_Get_Processdelay, TiCo_Set_Processdelay
Process control	TiCo_Start_Process, TiCo_Stop_Process TiCo_Stop, TiCo_Start, TiCo_Reset, TiCo_Reset_Mode
System information	Get_TiCo_Status, Process_Status, Workload
Data transfer	TiCo_Flash, TiCo_Load

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

tico_no	Number (1...2) of <i>TiCo</i> processor.	LONG
Par_no	Number (1...80) of global variable.	LONG
ret_val	Value ($-2^{31} \dots +2^{31}-1$) of global variable.	LONG

Notes

Several values are read more quickly using **Get_Par_Block**.

See also

Get_Par_Block, GetData_Long, Set_Par, Set_Par_Block, SetData_Long, TDrv_Init

Valid for

Gold II

Example

see file C:\ADwin\ADbasic\Examples\Get_Par.bas.

```
#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* processors 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...2) of <i>TiCo</i> processor.	LONG
<code>dest_array[]</code>	Destination array, into which values are transferred.	ARRAY
<code>dest_array_idx</code>	Destination start index (1...n): array element, where the first value is stored.	LONG
<code>start_idx</code>	Index (1...80) of the first global variable, that is read.	LONG
<code>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

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)
    REM read Par_20...Par_29 from TiCo and write values to
    REM Par_5...Par_14 of ADwin CPU
    Get_Par_Block(tico_no,PAR,5,20,10)

```


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

<code>tdrv_</code>	Array holding settings for data transfer, e.g. <i>TiCo</i>	ARRAY
<code>datatable</code>	number.	LONG
<code>[]</code>		
<code>src_</code>	Number (1...16) of write ringbuffer <code>Data_n</code> on the	FLOAT
<code>array_no</code>	<i>TiCo</i> processor.	
<code>dest_</code>	Destination array, into which values are to be	ARRAY
<code>array[]</code>	transferred.	LONG
<code>dest_</code>	Index (1...n) of the first element in <code>dest_</code>	LONG
<code>array_idx</code>	<code>array[]</code> to be written.	
<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 <code>Par_n</code> 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

`ret_val`

Success status of instruction:

LONG

-1: Error: The *TiCo* write ringbuffer is dimensioned wrongly.

≥0: Successful. The return value equals the number of transferred values.

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

Example

- / -

Get_TiCo_Status

Get_TiCo_Status returns, whether the *TiCo* processor is active.

Syntax

```
#Include ADwinGoldII.inc  
ret_val = Get_TiCo_Status ()
```

Parameters

ret_val	Status of <i>TiCo</i> processor: 0: Processor is stopped. 1: Processor is running. -3: Error, no <i>TiCo</i> processor available.
---------	--

LONG

Notes

- / -

See also

Process_Status, Workload

Valid for

Gold II

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 ADwinGoldII.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.	ARRAY
<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.	ARRAY
<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

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 Data_
  REM 2[20])
  REM into ADwin CPU Data_4 (starting from index 5).
  REM 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 ADwinGoldII.Inc

ret_val = Process_Status(tico_no,
    process_no_no)
```

Parameters

<code>tico_no</code>	Number (1...2) of <i>TiCo</i> processor.	LONG
<code>process_no_no</code>	Number (1) 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

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 ADwinGoldII.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.	ARRAY
<code>ret_val</code>		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

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

<code>tdrv_datatable</code>	Array holding settings for data transfer, e.g. <i>TiCo</i> number.	ARRAY
<code>[]</code>		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 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

Example

- / -

Set_Par

Set_Par sets the value of a global variable **Par_x** on a *TiCo* processor.

Syntax

```
#Include ADwinGoldII.Inc

Set_Par(tico_no, Par_no, value)
```

Parameters

tico_no	Number (1...2) of <i>TiCo</i> processor.	LONG
Par_no	Number (1...80) of global variable.	LONG
value	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 to synchronize *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

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 ADwinGoldII.Inc

Set_Par_Block(tico_no, src_array[],
              src_array_idx,
              Par_no, Par_count)
```

Parameters

<code>tico_no</code>	Number (1...2) of <i>TiCo</i> processor.	LONG
<code>src_array[]</code>	Source array, from which values are to be transferred.	ARRAY
<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 to synchronize *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

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 ADwinGoldII.Inc  
ret_val =  
    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. <i>TiCo</i>	ARRAY
<code>datatable</code>	number.	LONG
<code>[]</code>		
<code>dest_</code>	Number (1...16) of read ringbuffer <i>Data_n</i> on the	FLOAT
<code>array_no</code>	<i>TiCo</i> processor.	
<code>src_</code>	Source array, from which values are transferred.	ARRAY
<code>array[]</code>		LONG
<code>src_</code>	Index (1...n) of the first element in <code>src_array[]</code>	LONG
<code>array_idx</code>	to be read.	
<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
<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

`ret_val`

Success status of instruction:

LONG

-1: Error: The *TiCo* read ringbuffer is dimensioned wrongly.

≥0: Successful. The return value equals the number of transferred values.

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

Example

- / -

SetData_Long

SetData_Long reads values from an array and writes them into a global array of a *TiCo* processor.

Syntax

```
#Include ADwinGoldII.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.	ARRAY 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.	ARRAY 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

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 Data_
REM 2[1])
REM into ADwin CPU Data_1 (starting from Data_1[5]).
REM 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 ADwinGoldII.Inc

REM define settings array for TiCo x
Dim tdrv_datatable[150] As Long

TDrv_Init(tico_no, tdrv_datatable[])
```

Parameters

tico_no	Number (1...2) of <i>TiCo</i> processor.	LONG
tdrv_datatable	Array holding settings for data transfer, e.g. <i>TiCo</i> number x.	ARRAY
[]		LONG

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

Example

```
#Include ADwinGoldII.Inc
#Define tico_a 1           'TiCo no.
#Define tico_b 2           'TiCo no.
Dim Data_4[200] As Long
Dim Data_5[1000] As Long
Dim tset_1[150] As Long
Dim tset_2[150] As Long
```

Init:

```
REM initialize data transfer ADwin CPU <-> TiCo
TDrv_Init(tico_a,tset_1)
TDrv_Init(tico_b,tset_2)
```

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). flowrate is high
GetData_Long(tset_1,2,20,120,Data_4,5,3)
REM read 800 values from TiCo Data_7 (starting from
REM Data_5[9]) into ADwin CPU Data_5 (starting from
REM index 1). flowrate is high
GetData_Long(tset_2,7,9,800,Data_5,1,3)
```

TiCo_Flash

TiCo_Flash transfers a *TiCoBasic* binary file from an array into the flash memory of a *TiCo* processor.

Syntax

```
#INCLUDE ADwinGoldII.inc

ret_val = TiCo_Flash(tico_no,array[])
```

Parameters

<code>tico_no</code>	Number (1...2) of <i>TiCo</i> processor.	LONG
<code>array[]</code>	Array which holds the <i>TiCoBasic</i> binary file to be transferred.	ARRAY
<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 **Data2File** which is provided in several programming languages.
- Transfer the data of the global **array[]** into the flash memory using **TiCo_Flash**.

If **array[]** does not contain a *TiCoBasic* binary file the return value is invalid.

See also

TiCo_Load

Valid for

DIO-32-TiCo Rev. E, Cnt-D Rev.E, Cnt-T Rev. E, Cnt-I Rev. E,
RSxxx-2 Rev. E, RSxxx-4 Rev. E, CAN-2 Rev. E

Example

- / -

TiCo_Get_Processdelay

TiCo_Get_Processdelay returns the **Processdelay** (cycle time) of a process on a *TiCo* processor.

Syntax

```
#Include ADwinGoldII.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.	ARRAY LONG
<code>process_no</code>	Number (1) 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

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_Load

TiCo_Load transfers a *TiCoBasic* binary file from an array into the memory of a *TiCo* processor.

Syntax

```
#INCLUDE ADwinGoldII.inc

ret_val = TiCo_Load(tico_no, array[])
```

Parameters

<code>tico_no</code>	Number (1...2) of <i>TiCo</i> processor.	LONG
<code>array[]</code>	Array which holds the <i>TiCoBasic</i> binary file to be transferred.	ARRAY 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 **Data2File** which is provided in several programming languages.
- Transfer the data of the global **array[]** into the memory using **TiCo_Load**.

If **array[]** does not contain a *TiCoBasic* binary file the return value is invalid.

See also

TiCo_Flash

Valid for

DIO-32-TiCo Rev. E, Cnt-D Rev.E, Cnt-T Rev. E, Cnt-I Rev. E,
RSxxx-2 Rev. E, RSxxx-4 Rev. E, CAN-2 Rev. E

Example

- / -

TiCo_Reset

TiCo_Reset stops all *TiCo* processors and restarts them afterwards.

Syntax

```
#Include ADwinGoldII.Inc  
  
TiCo_Reset()
```

Parameters

- / -

Notes

Stopping immediately interrupts any running process as well as counters on the *TiCo* processors. Data are not changed by stopping.

The *TiCo* processors are being started at the same time. Thus, the instruction can be used to synchronize the *TiCo* processors.

See also

TiCo_Stop, TiCo_Start

Valid for

Gold II

Example

- / -

TiCo_Reset_Mode

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

mode	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_Reset, TiCo_Stop, TiCo_Start

Valid for

Gold II

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 ADwinGoldII.Inc

TiCo_Set_Processdelay (tdrv_
    datatable[], process_no,
    value)
```

Parameters

<code>tdrv_</code>	Array holding settings for data transfer, e.g. <i>TiCo</i>	ARRAY
<code>datatable</code>	number.	LONG
<code>process_</code>	Number (1) of <i>TiCo</i> process.	LONG
<code>no</code>		
<code>value</code>	Value to be set for Processdelay .	LONG

Notes

- / -

See also

TiCo_Get_Processdelay, Process_Status, TDrv_Init

Valid for

Gold II

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 all *TiCo* processors.

Syntax

```
#Include ADwinGoldII.Inc  
TiCo_Start()
```

Parameters

- / -

Notes

The *TiCo* processors are being started at the same time. Thus, the instruction can be used to synchronize the *TiCo* processors.

See also

TiCo_Stop, TiCo_Reset

Valid for

Gold II

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

<p>tdrv_ datatable []</p>	<p>Array holding settings for data transfer, e.g. <i>TiCo</i> number.</p>	<div style="background-color: #006400; color: white; padding: 2px 5px; font-weight: bold;">ARRAY</div> <div style="border: 1px solid #006400; padding: 2px 5px; width: fit-content; margin-top: 5px;">LONG</div>
<p>process_ no</p>	<p>Number (1) of <i>TiCo</i> process.</p>	<div style="border: 1px solid #006400; padding: 2px 5px; width: fit-content; margin-top: 5px;">LONG</div>

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

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 all *TiCo* processors.

Syntax

```
#Include ADwinGoldII.Inc  
  
TiCo_Stop()
```

Parameters

- / -

Notes

Stopping immediately interrupts any running process as well as counters on the *TiCo* processors. Data are not changed by stopping.

See also

TiCo_Start, TiCo_Reset

Valid for

Gold II

Example

- / -

TiCo_Stop_Process

TiCo_Stop_Process stops a process on a *TiCo* processor.

Syntax

```
#Include ADwinGoldII.Inc  
  
TiCo_Stop_Process(tdrv_datatable[],  
                  process_no)
```

Parameters

<code>tdrv_</code>	Array holding settings for data transfer, e.g. <i>TiCo</i>	ARRAY
<code>datatable</code>	number.	LONG
<code>[]</code>		
<code>process_</code>	Number (1) of <i>TiCo</i> process.	LONG
<code>no</code>		

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

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)
```


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.	ARRAY 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 der 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

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)
```


8.4 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
Process control	P2_TiCo_Start_Process, P2_TiCo_Stop_Process P2_TiCo_Stop, P2_TiCo_Start, P2_TiCo_Reset
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

module	Specified module address (1...15).	LONG
tico_no	Number (1...2) of <i>TiCo</i> processor on the module.	LONG
Par_no	Number (1...80) of the global variable.	LONG
ret_val	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

CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E,
DIO-32-TiCo Rev. E, RSxxx-2 Rev. E, RSxxx-4 Rev. E

Example

see file C:\ADwin\ADbasic\Examples\P2_GET_PAR.bas.

```
#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* processors 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

<code>module</code>	Specified module address (1...15).	LONG
<code>tico_no</code>	Number (1...2) of <i>TiCo</i> processor on the module.	LONG
<code>dest_array[]</code>	Destination array, into which values are transferred.	ARRAY
<code>dest_array_idx</code>	Destination start index (1...n): Array element, from which values are stored.	LONG
<code>Par_no</code>	Index (1...80) of the first global variable to be read.	LONG
<code>Par_count</code>	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

CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E,
DIO-32-TiCo Rev. E, RSxxx-2 Rev. E, RSxxx-4 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

<code>module</code>	Specified module address (1...15).	LONG
<code>ret_val</code>	<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

CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E,
DIO-32-TiCo Rev. E, RSxxx-2 Rev. E, RSxxx-4 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.	ARRAY
<code>datatable</code>	module address and <i>TiCo</i> number.	LONG
<code>[]</code>		
<code>src_</code>	Number (1...16) of write ringbuffer <code>Data_n</code> on the	FLOAT
<code>array_no</code>	<i>TiCo</i> processor.	
<code>dest_</code>	Destination array, into which values are to be	ARRAY
<code>array[]</code>	transferred.	LONG
<code>dest_</code>	Index (1...n) of the first element in <code>dest_</code>	LONG
<code>array_idx</code>	<code>array[]</code> to be written.	
<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 <code>Par_n</code> 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

`ret_val`

Success status of instruction:

LONG

-1: Error: The *TiCo* write ringbuffer is dimensioned wrongly.

≥0: Successful. The return value equals the number of transferred values.

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_datatable[]`. 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

CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E,
DIO-32-TiCo Rev. E, RSxxx-2 Rev. E, RSxxx-4 Rev. E

Example

- / -

P2_Get_TiCo_Status

P2_Get_TiCo_Status returns, whether *TiCo* processors are active on the the specified module.

Syntax

```
#Include ADwinPRO_ALL.inc  
  
ret_val = P2_Get_TiCo_Status (module)
```

Parameters

<code>module</code>	Specified module address (1...15).	LONG
<code>ret_val</code>	Status of <i>TiCo</i> processors. -3: Error, no <i>TiCo</i> processors available. -2: Error, not a <i>Pro II</i> module. ≥0: Bit pattern of the state of operation of processors: Bit = 0: Processor is stopped. Bit = 1: Processor is running.	LONG

Bits in <code>ret_val</code>	31:01	00
<i>TiCo</i> processor	–	1

Notes

- / -

See also

P2_Process_Status, P2_Workload

Valid for

CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E,
DIO-32-TiCo Rev. E, RSxxx-2 Rev. E, RSxxx-4 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.	ARRAY
<code>datatable</code>	module address and <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, 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_array[]</code>	Destination array where values are stored.	ARRAY
<code>dest_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 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

P2_Get_Par, P2_Set_Par, P2_SetData_Long, P2_TDrv_Init

Valid for

CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E,
DIO-32-TiCo Rev. E, RSxxx-2 Rev. E, RSxxx-4 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

<code>module</code>	Specified module address (1...15).	LONG
<code>tico_no</code>	Number (1...2) of <i>TiCo</i> processor on the module.	LONG
<code>process_no</code>	Number (1) of <i>TiCo</i> process.	LONG
<code>ret_val</code>	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

CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E,
DIO-32-TiCo Rev. E, RSxxx-2 Rev. E, RSxxx-4 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

tdrv_	Array holding settings for data transfer, e.g.	ARRAY
datatable	module address and <i>TiCo</i> number.	LONG
[]		
Data_no	Number (1...16) of write ringbuffer Data_n on the <i>TiCo</i> processor.	LONG
ret_val	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

CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E,
DIO-32-TiCo Rev. E, RSxxx-2 Rev. E, RSxxx-4 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

<code>tdrv_</code>	Array holding settings for data transfer, e.g.	ARRAY
<code>datatable</code>	module address and <i>TiCo</i> number.	LONG
<code>[]</code>		
<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 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

CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E,
DIO-32-TiCo Rev. E, RSxxx-2 Rev. E, RSxxx-4 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

module	Specified module address (1...15).	LONG
tico_no	Number (1...2) of <i>TiCo</i> processor on the module.	LONG
Par_no	Number (1...80) of the global variable.	LONG
value	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 to synchronize *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

CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, DIO-32-TiCo Rev. E, RSxxx-2 Rev. E, RSxxx-4 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

module	Specified module address (1...15).	LONG
tico_no	Number (1...2) of <i>TiCo</i> processor on the module.	LONG
src_array[]	Source array, from which values are to be transferred.	ARRAY
src_array_idx	Index of the array element, starting from which values are read from array[] .	LONG
Par_no	Index (1...80) of the first global <i>TiCo</i> variable to be written.	LONG
Par_count	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 to synchronize *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, P2_SetData_Long, P2_TDrv_Init

Valid for

CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, DIO-32-TiCo Rev. E, RSxxx-2 Rev. E, RSxxx-4 Rev. E

Example

see Datei C:\ADwin\ADbasic\Examples\P2_Set_Par_Block.bas.

```
#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.	ARRAY
<code>datatable</code>	module address and <i>TiCo</i> number.	LONG
<code>[]</code>		
<code>dest_</code>	Number (1...16) of read ringbuffer <code>Data_n</code> on the	FLOAT
<code>array_no</code>	<i>TiCo</i> processor.	
<code>src_</code>	Source array, from which values are transferred.	ARRAY
<code>array[]</code>		LONG
<code>src_</code>	Index (1...n) of the first element in <code>src_array[]</code>	LONG
<code>array_idx</code>	to be read.	
<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

`ret_val`

Success status of instruction:

LONG

-1: Error: The *TiCo* read ringbuffer is dimensioned wrongly.

≥0: Successful. The return value equals the number of transferred values.

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_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

P2_Get_TiCo_RingBuffer, P2_TDrv_Init

Valid for

CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, DIO-32-TiCo Rev. E, RSxxx-2 Rev. E, RSxxx-4 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_</code>	Array holding settings for data transfer, e.g.	ARRAY
<code>datatable</code>	module address and <i>TiCo</i> number.	LONG
<code>dest_</code> <code>array_no</code>	Number (1...16) of global array <code>Data_x</code> on the <i>TiCo</i> processor.	LONG
<code>dest_</code> <code>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_</code> <code>array[]</code>	Source array, from where values are read.	ARRAY
<code>src_</code> <code>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

CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E,
DIO-32-TiCo Rev. E, RSxxx-2 Rev. E, RSxxx-4 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 read 120 values from TiCo Data_2 (starting from
REM Data_2[1])
REM into ADwin CPU Data_1 (starting from Data_1[5]).
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

module	Specified module address (1...15).	LONG
tico_no	Number (1...2) of <i>TiCo</i> processor on the module.	LONG
tdrv_datatable	Array holding settings for data transfer, e.g. module address and <i>TiCo</i> number.	ARRAY
[]		LONG

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

CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, DIO-32-TiCo Rev. E, RSxxx-2 Rev. E, RSxxx-4 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

tdrv_	Array holding settings for data transfer, e.g.	ARRAY
datatable	module address and <i>TiCo</i> number.	LONG
[]		
process_	Number (1) of <i>TiCo</i> process.	LONG
no		
ret_val	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

CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, DIO-32-TiCo Rev. E, RSxxx-2 Rev. E, RSxxx-4 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_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

<code>module</code>	Specified module address (1...15).	LONG
<code>tico_no</code>	Number (1...2) of <i>TiCo</i> processor on the module.	LONG
<code>array[]</code>	Array holding the data to be transferred.	ARRAY
<code>ret_val</code>	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> proces- sor.	LONG

Notes

Use **P2_TiCo_Flash** only in low priority processes.

The instruction **P2_TiCo_Flash** is used to transfer a *TiCo-Basic* 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 **Data2File** which is provided in several programming languages.
- Transfer the data of the global **array[]** into the flash memory using **P2_TiCo_Flash**.

If **array[]** does not contain a *TiCoBasic* binary file the return value is invalid.

See also

P2_TiCo_Load

Valid for

CAN-2 Rev. E, CNT-D Rev. E, Cnt-I Rev. E, Cnt-T Rev. E, DIO-32-TiCo Rev. E, RSxxx-2 Rev. E, RSxxx-4 Rev. E

Example

- / -

P2_TiCo_Load

P2_TiCo_Load transfers a *TiCoBasic* binary file from an array into the memory of a *TiCo* processor.

Syntax

```
#Include ADwinPRO_ALL.inc

ret_val = P2_TiCo_Load(module,tico_no,array[])
```

Parameters

<code>module</code>	Specified module address (1...15).	LONG
<code>tico_no</code>	Number (1...2) of <i>TiCo</i> processor on the module.	LONG
<code>array[]</code>	Array holding the data to be transferred.	ARRAY
<code>ret_val</code>	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` which is provided in several programming languages.
- Transfer the data of the global `array[]` into the memory using **P2_TiCo_Load**.

See also

P2_TiCo_Flash

Valid for

-RSxxx4 Rev. E, CAN-2 Rev. E, CNT-D Rev. E, Cnt-I Rev. E, Cnt-T Rev. E, DIO-32-TiCo Rev. E, RSxxx-2 Rev. E

Example

- / -

P2_TiCo_Reset

P2_TiCo_Reset stops the *TiCo* processors on the specified modules and restarts them afterwards.

Syntax

```
#Include ADwinPro_All.Inc  
  
P2_TiCo_Reset(module_pattern)
```

Parameters

module_ Bit pattern to address the modules, where *TiCo* LONG
pattern processors are to be reset:
Bit = 0: Ignore module.
Bit = 1: Reset *TiCo* processors on the module.

Bits in module_pattern	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

CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E,
DIO-32-TiCo Rev. E, RSxxx-2 Rev. E, RSxxx-4 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

tdrv_	Array holding settings for data transfer, e.g.	ARRAY
datatable	module address and <i>TiCo</i> number.	LONG
process_	Number (1) of <i>TiCo</i> process.	LONG
no		
value	Value to be set for Processdelay .	LONG

Notes

- / -

See also

P2_TiCo_Get_Processdelay, P2_Process_Status, P2_TDrv_Init

Valid for

CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E,
DIO-32-TiCo Rev. E, RSxxx-2 Rev. E, RSxxx-4 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_ Bit pattern to address the modules, where *TiCo* LONG
pattern processors are to be started:
Bit = 0: Ignore module.
Bit = 1: Start *TiCo* processors on the module.

Bits in module_pattern	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_Reset

Valid for

CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E,
DIO-32-TiCo Rev. E, RSxxx-2 Rev. E, RSxxx-4 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

<code>tdrv_</code>	Array holding settings for data transfer, e.g.	ARRAY
<code>datatable</code>	module address and <i>TiCo</i> number.	LONG
<code>[]</code>		
<code>process_</code>	Number (1) 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_TiCo_Stop_Process, P2_Workload

Valid for

CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E,
DIO-32-TiCo Rev. E, RSxxx-2 Rev. E, RSxxx-4 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* processors on the module.

Bits in module_pattern	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_Reset

Valid for

CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E,
DIO-32-TiCo Rev. E, RSxxx-2 Rev. E, RSxxx-4 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.	ARRAY
<code>datatable</code>	module address and <i>TiCo</i> number.	LONG
<code>process_</code>	Number (1) 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

CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, DIO-32-TiCo Rev. E, RSxxx-2 Rev. E, RSxxx-4 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

<p>tdrv_ datatable []</p>	<p>Array holding settings for data transfer, e.g. module address and <i>TiCo</i> number.</p>	<div style="border: 1px solid black; padding: 2px; display: inline-block;">ARRAY</div> <div style="border: 1px solid black; padding: 2px; display: inline-block;">LONG</div>
<p>ret_val</p>	<p>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.</p>	<div style="border: 1px solid black; padding: 2px; display: inline-block;">FLOAT</div>

Notes

The return value is the average processor workload in der 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

CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E,
DIO-32-TiCo Rev. E, RSxxx-2 Rev. E, RSxxx-4 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)
```

9 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-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.	
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

NUL 00h 0	SOH 01h 1	STX 02h 2	ETX 03h 3	EOT 04h 4	ENQ 05h 5	ACK 06h 6	BEL 07h 7
BS ¹ 08h 8	TAB ² 09h 9	LF ³ 0Ah 10	VT 0Bh 11	FF 0Ch 12	CR ⁴ 0Dh 13	SO 0Eh 14	SI 0Fh 15
DLE 10h 16	DC1 11h 17	DC2 12h 18	DC3 13h 19	DC4 14h 20	NAK 15h 21	SYN 16h 22	ETB 17h 23
CAN 18h 24	EM 19h 25	SUB 1Ah 26	ESC 1Bh 27	FS 1Ch 28	GS 1Dh 29	RS 1Eh 30	US 1Fh 31
SPC ⁵ 20h 32	! 21h 33	" 22h 34	# 23h 35	\$ 24h 36	% 25h 37	& 26h 38	' 27h 39
(28h 40) 29h 41	* 2Ah 42	+ 2Bh 43	, 2Ch 44	- 2Dh 45	. 2Eh 46	/ 2Fh 47
0 30h 48	1 31h 49	2 32h 50	3 33h 51	4 34h 52	5 35h 53	6 36h 54	7 37h 55
8 38h 56	9 39h 57	: 3Ah 58	; 3Bh 59	< 3Ch 60	= 3Dh 61	> 3Eh 62	? 3Fh 63
@ 40h 64	A 41h 65	B 42h 66	C 43h 67	D 44h 68	E 45h 69	F 46h 70	G 47h 71
H 48h 72	I 49h 73	J 4Ah 74	K 4Bh 75	L 4Ch 76	M 4Dh 77	N 4Eh 78	O 4Fh 79
P 50h 80	Q 51h 81	R 52h 82	S 53h 83	T 54h 84	U 55h 85	V 56h 86	W 57h 87
X 58h 88	Y 59h 89	Z 5Ah 90	[5Bh 91	\ 5Ch 92] 5Dh 93	^ 5Eh 94	_ 5Fh 95
` 60h 96	a 61h 97	b 62h 98	c 63h 99	d 64h 100	e 65h 101	f 66h 102	g 67h 103
h 68h 104	i 69h 105	j 6Ah 106	k 6Bh 107	l 6Ch 108	m 6Dh 109	n 6Eh 110	o 6Fh 111
p 70h 112	q 71h 113	r 72h 114	s 73h 115	t 74h 116	u 75h 117	v 76h 118	w 77h 119
x 78h 120	y 79h 121	z 7Ah 122	{ 7Bh 123	 7Ch 124	} 7Dh 125	~ 7Eh 126	□ 7Fh 127

¹ Backspace, ² Tabulator, ³ Linefeed,

⁴ Carriage Return, ⁵ Space

A.3 License Agreement

Between the buyer of *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 to type all options to avoid ambiguities¹.

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  
    /EEMx /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
```

Optional settings are given in brackets []. The character | separates options, which are mutually exclusive.

File names can be written without, with relative or with absolute path names. The base directory for a file name without or with relative path name is the working directory, from which the command line is called.

Main Options

/M	Generate a binary file with the extension <code>.TIn</code> . n Process number; see option /PNx.
/L	Generate a library file with the extension <code>.TLx</code> . x Processor type; see option /Px.
/MAKE	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
/H	Display a short help text.
/VER	Display compiler version number.

Options

src.bas	File name of the source code to be compiled; type with suffix <code>.bas</code> . With main option /M, specify after /PROCESS. Compiler warnings are written into the file <code>src.wrn</code> , error messages into the file <code>src.err</code> .
/A"dest"	[Path and] name of the binary or library file <dest> which is to be generated, without suffix. The default is the file name <code>src</code> . The file suffix <code>.TIn</code> (binary file) or <code>.TLx</code> (library file) is attached automatically.
/IP"path"	Directory, where include files are searched. This setting overwrites the <i>TiCoBasic</i> standard directory and should thus be used with caution.
/LP"path"	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 //SPI Pro II; Default I
/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 chapter 7.1.1 on page 110. Default. Excludes /EE and /EN.
/EE	Create externally triggered process, see also chapter 7.1.2 on page 111; requires options /EEA, /EEM, /EEV, /EEO. Schließt /ET and /EN aus.
/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	Comparing 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 chapter 7.1.3 on page 112. Excludes /EE and /ET.

/PNx	Number x (1...4) of the process. Default: 1.
/PH	Create process with high priority. Default. See also chapter 7.1 on page 110.
/PL	Create process with low priority (time triggered process only). See also chapter 7.1 on page 110.
/PDx	Set cycle time (Processdelay) of the process to x. Default: 3000. See also chapter 7.2.1 on page 113.
/Ox	Set optimize level x (0, 1, 2) of the compiler, see also Process Options dialog box (page 50).
/O0	Optimize level 0 (=don't optimize)
/O1	Optimize level 1 (Default)
/O2	Optimize level 2
/Vx	Set process version x, see Process Options dialog box (page 50). 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 4.10.1).

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 you delete the files containing the warnings and error messages before compilation, so that you can very easily check if the compilation has proceeded without any errors.

A.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.TL9> 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  
- bas_dmo6f.bas /ET /PN3 /PH /O1
```

Compiles the demo file <bas_dmo6f.bas> into a binary file for a *Gold II* system with *TiCo* processor. It is a timer triggered 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 timer triggered process with number 3 and high priority .

A.4.4 Command line calls in Windows

The term and functionality "command line call" come from DOS, where commands to the operating system DOS had to be entered in command lines. Entering such command lines is still possible under Windows.

There are several ways to enter commands under Windows:

- Open a Command Prompt window (from Windows start menu, directory `Programs / Accessories`).

The compiler call needs the Windows environment anyway. Thus, the call works only from the Command Prompt window, not from original DOS-mode.



- Select `Run` in the start menu and enter a command line in the input window.
- For frequently needed command lines create an icon on the desktop. When you generate an icon enter the command line directly.

One or more command lines can be combined in one batch file `<*.bat>`, for example in order to compile several source code files of a project with only one call.

When you call a command line you have to transfer the relevant options and parameters.

A.5 Index

Symbols

- · 122
- # · 126
- #Define · 135
- #Else · 151
- #EndIf · 151
- #If · 151
- #Include · 158
- * · 123
- + · 121
- / · 124
- : · 127
- < = > · 129
- = · 128
- ^ · 125
- ' (Rem) · 185

Numerics

150h, *see* device no.

A

- AbsI · 130
- absolute value
 - integer number · 130
- ActiveX
 - communication to ADwin system · 117
- ADconfig · 118
- Add Open Files to Project · 61
- Add to Project
 - context menu · 17
 - project window · 61
- addition · 121
- ADtools · 74
- ADtools, set bar · 58
- ADWIN_GOLDII · 151
- ADWIN_PROII · 151

ADWIN_SYSTEM · 151

And · 131

arithmetic functions

- · 122
- * · 123
- + · 121
- / · 124
- ^ · 125
- Dec · 134
- Inc · 157

arrays

- allocate memory area · 86
- DATA_n · 133
- global · 83
 - first element · 84
- initialize · 78
- local · 85
 - first element · 86
- overview · 80

(Dim) AS · 137

ASCII-character set · 4

assign a value · 82

assignment (=) · 128

(Dim ...) AT · 137

autocomplete, instruction or variable · 36

autoindent · 54

AutoSave · 47

autostart · 47

B

bar, menu · 43

binary file

see also library

create · 47

from command line · 9

from TiCoBasic · 47

transfer to TiCo processor · 41

binary notation · 82

- bit shifting
 - left · 199
 - right · 200
- bookmark · 34
- Bootloader
 - menu entry · 58
- bootloader
 - programming · 41
- break, *see* stop process
- BTL file
 - directory settings · 57
- busy display · 67
- bypass waiting time · 173

C

- case sensitivity · 16
- Case, CCase, CaseElse (Select-Case ...) · 196
- change license key · 10
- change to TiCoBasic · 5
- clear parameter scan · 40
- code size · 68
- code snippets · 37
- color settings · 55
- command line
 - call · 9
 - line length
 - standard · 76
 - with #Include · 158
 - upper case / lower case · 76
- Comment Block · 23
- comment, *see* remarks
- communication
 - between ADwin CPU and TiCo processor · 118
 - between processes · 116
 - with the TiCo processor · 116
- comparison
 - < = > · 129

- compiler
 - AutoSave · 47
 - call · 47
 - command line call · 9
 - compiler message, error / status · 68
 - preprocessor statement · 126
 - set options · 48
- compiler instructions
 - #Define · 135
 - #If ... Then · 151
 - #Include · 158
- conditional jump
 - If ... Then · 149
 - SelectCase · 196
- constant · 78
- context menu
 - project window · 61
 - source code window · 17
- control block
 - context menu · 17
 - mark · 33
- control structures · 99
 - toggle folding · 23
- counter
 - internal, clock cycle · 113
 - read · 184
- cursor position · 67
- cycle time · 113

D

- data exchange
 - between processes · 116
 - with the TiCo processor · 116
- data loss
 - on initialize · 12
 - on reset · 12
 - ringbuffer · 90

- data memory
 - see also memory
 - allocate · 86
 - overview, internal, external · 87
 - data structures
 - global arrays · 83
 - global variables · 82
 - local variables and arrays · 85
 - overview · 80
 - Ringbuffer · 89
 - data types
 - overview · 82
 - data word, numbering of bits · 2
 - Data_n · 83
 - dimensioning · 137
 - overview · 133
 - Dec · 134
 - decimal notation · 82
 - decimal separator · 82
 - declaration
 - jump to · 35
 - see dimensioning
 - show all · 39
 - show single info · 38
 - declarations
 - display all · 72
 - decrement · 134
 - Define, see #Define
 - definition of macros, position in the
 - program · 79
 - demo mode · 10
 - design of an TiCoBasic
 - program · 76
 - development environment
 - bars and windows · 13
 - directory settings · 57
 - short-cuts · 1
 - start · 9
 - device no.
 - definition · 118
 - set · 49
 - Dim · 137
 - dimensioning
 - instruction Dim · 137
 - memory area · 86
 - position in the program · 78
 - directory settings · 57
 - Disable Trace · 17
 - display
 - all declarations · 39
 - current information · 15
 - memory usage: CPU, PM, DM, DX · 67
 - passed parameters · 38
 - single declaration info · 38
 - syntax highlighting · 21
 - display declarations · 72
 - division
 - by 2 · 200
 - simple · 124
 - DM, see memory
 - DM_LOCAL
 - Dim · 137
 - Do ... Until · 140
 - DRAM_EXTERN
 - Dim · 137
 - DX, see memory
- ## E
- editor
 - general · 54
 - print settings · 56
 - syntax highlighting · 55
 - editor bar · 19
 - Else (If ... Then) · 149
 - Enable Trace · 17
 - End · 142
 - EndFunction · 146
 - EndIf (If ... Then) · 149
 - EndSelect (SelectCase ...) · 196
 - EndSub · 202
 - enter license key · 10

- equal to = · 129
 - error
 - forced by Cut&Paste · 46
 - try lower optimization level · 53
 - error message, compiler · 68
 - Ethernet · 117
 - evaluate
 - operators · 97
 - Event
 - program section · 143
 - set signal source · 52
 - event
 - external signal · 108
 - externally controlled · 111
 - lost signal
 - externally controlled process · 115
 - timer-controlled process · 115
 - measure time difference · 103
 - timer controlled
 - 110
 - without trigger · 112
 - exclusive Or operation · 204
 - exponential notation · 82
 - expressions
 - evaluate · 97
 - symbolic names · 78
 - external data memory (DX) · 88
 - external data memory (SX) · 88
 - external event signal · 108
 - external memory (SDRAM) · 87
- F**
- F1: call help · 16
 - Felder
 - Ringbuffer · 186
 - FIFO · 89
 - file name
 - binary file · 47
 - library · 48
- find**
- declaration of instruction/variable · 35
 - examples · 29
 - regular expressions · 31
 - text · 26
 - text quickly · 25
- Finish: · 144
- fold text ranges · 23
- font settings · 55
- For ... Next · 145
- format, smart · 21
- Function · 146
 - library
 - definition · 161
 - macro · 146
 - position in the program · 79
- function
 - general features · 99
 - library
 - general · 100
- G**
- Get_Par · 207
 - Get_Par_Block · 209
 - Get_TiCo_RingBuffer · 211
 - Get_TiCo_Status · 214
 - GetData_Long · 215
 - global arrays, *see* arrays, global
 - global variables · 71
 - global variables, *see* variables, global
 - Globaldelay · 180
 - Gold2cess_Status · 218
 - goto line · 35
 - greater than >, >= · 129
- H**
- halt
 - TiCo processor · 12

halt, *see* stop process

Hardware access

read · 155

write · 179

Header · 56

help

call selected · 16

F1 · 16

hexadecimal notation · 82

I

If · 149

see also #If · 151

Import · 153

In · 155

Inc · 157

Include · 158

directory settings · 57

Include a file: #Include · 158

Include a library: Import · 153

include

include-file, general · 100

increment · 157

indent

lines · 23

TiCoBasic sections · 54

info range · 68

info window · 68

Init: · 160

initialize · 12, 78

input license key · 10

insert code snippets · 37

instruction

autocomplete · 36

declaration info · 38

display passed parameters · 38

jump to declaration · 35

measure processing time · 102

separator (:) · 127

instruction reference · 119

integer numbers

value range · 82

internal counter

clock cycle · 113

internal memory

data (DM) · 88

SRAM · 87

interrupt, *see* stop process

J

jump to declaration · 35

jump to program line · 35

jump, conditional

If ... Then · 149

SelectCase · 196

K

keyboard, settings display · 67

L

language · 57

less than <, <= · 129

Lib_EndFunction · 161

Lib_EndSub · 166

Lib_Function · 161

Lib_Sub · 166

library

create

from command line · 9

from TiCoBasic · 48

directory settings · 57

function · 161

general · 100

Import · 153

position in the program · 79

subroutine · 166

toggle folding · 23

library file

create · 47

license agreement · 5

- license key · 10
- line length, max.
 - standard · 76
 - with #Include · 158
- lines
 - change to comment · 23
 - indenting · 23
 - jump to · 35
 - numbering · 54
 - smart format · 21
- Load Bin File · 58
- logic functions
 - And · 131
 - Not · 176
 - Or · 177
 - Shift_Left · 199
 - Shift_Right · 200
 - XOr · 204
- long, *see* integer numbers

M

- macro
 - function · 146
 - general features · 99
 - position in the program · 79
 - toggle folding · 23
- Make Bin File, Make Lib File · 47
- manual indenting · 23
- Mark Control block · 33
- Max_Long · 171
- Maximum
 - integer values · 171
- maximum line length
 - standard · 76
 - with #Include · 158
- measure processing time · 102
- measurement graph · 74

- memory
 - see also* data memory
 - allocate · 86
 - areas (PM, DM, DX) · 87
 - calculate need of · 68
 - workload · 67
- menu
 - bar · 43
 - build · 47
 - edit · 46
 - file · 45
 - help · 59
 - options · 48
 - select · 14
 - tools · 58
 - view · 46
 - window · 59
- Min_Long · 172
- Minimum
 - integer values · 172
- multiplication
 - by 2 · 199
 - simple · 123

N

- names, local variables · 85
- negative sign · 98
- new in TiCoBasic · 5
- Next (For ...) · 145
- none: without event trigger · 112
- NOP · 173
- NOPs · 174
- Not · 176
- not equal to <> · 129
- notation of numbers · 82
- notes, *see* remarks
- number, *see* device no.
- numerical values, notation · 82

O

- operating system
 - directory settings · 57
 - load, see initialize
- operators
 - And · 131
 - evaluate · 97
 - negative sign · 98
 - Or · 177
 - priority · 97
 - XOr · 204
- Optimierung
 - Speicherzugriff · 107
- optimize
 - calculate polynoms quickly · 125
 - constants instead of variables · 104
 - general · 102
 - measure faster · 104
 - measure processing time · 102
 - register access · 103
 - setting waiting time · 105
 - use waiting times · 105
- options setting
 - ADtools · 58
 - compiler · 48
 - directory · 57
 - editor · 54
 - general · 54
 - language · 57
 - print · 56
 - process · 50
 - syntax highlight · 55
- Or · 177
- Or operation · 177
- Out · 179
- outdent lines · 23

P

- P2_Get_Par · 255

- P2_Get_Par_Block · 257
- P2_Get_TiCo_Bootloader_Status · 259
- P2_Get_TiCo_RingBuffer · 260
- P2_Get_TiCo_Status · 263
- P2_GetData_Long · 265
- P2_Process_Status · 268
- P2_Ringbuffer_Empty · 270
- P2_Ringbuffer_Full · 271
- P2_Set_Par · 272
- P2_Set_Par_Block · 274
- P2_Set_TiCo_RingBuffer · 276
- P2_SetData_Long · 279
- P2_TDrv_Init · 282
- P2_TiCo_Flash · 286
- P2_TiCo_Get_Processdelay · 284
- P2_TiCo_Load · 288
- P2_TiCo_Reset · 290
- P2_TiCo_Set_Processdelay · 292
- P2_TiCo_Start · 294
- P2_TiCo_Start_Process · 295
- P2_TiCo_Stop · 297
- P2_TiCo_Stop_Process · 298
- P2_Workload · 300
- Par_n · 82
- parameter scan · 40
- parameter window · 63
- parameters, see variables, global
- parse and indent · 54
- passed parameters, display · 38
- PM, see memory
- polynoms, calculate quickly · 125
- power · 125
 - replace in polynomial · 125
- pre-processor
 - overview instructions · 126
- pre-processor instructions
 - #Define · 135
 - #If ... Then · 151
 - #Include · 158
- Print layout · 56

- print settings · 56
- priority
 - operators · 97
- problems
 - slow editor · 54
- process
 - autostart · 47
 - communication · 116
 - externally controlled · 111
 - operating modes for timing · 114
 - options, show · 14
 - processing time · 114
 - query status · 183
 - setting options · 50
 - stop, see stop process
 - time characteristic · 113
 - timer controlled
 - low priority · 110
 - high priority · 110
 - without event trigger · 112
- process control
 - End · 142
 - ProcessN_Running · 183
- process cycle
 - call
 - by event · 108
 - time interval · 113
- process optimization, see optimize
- Process_Status · 218
- Processdelay · 113
 - syntax · 180
 - time resolutions · 113
- Processn_Running · 183
- Processor · 151
- program architecture
 - jump
 - If ... Then · 149
 - SelectCase · 196
 - library
 - function · 161
 - Lib_Sub · 166
 - loop
 - Do ... Until · 140
 - For ... Next · 145
 - modules
 - function · 146
 - subroutine Sub · 202
 - remarks Rem · 185
- program design · 76
- program improvement, see optimize
- program line, jump to · 35
- program memory · 88
- program section
 - event: · 78
 - Finish: · 78
 - Init: · 78
 - overview · 78
- program structure
 - overview · 99
 - include-file · 100
 - library · 100
 - module (macro) · 99
 - toggle folding · 23
- project
 - general · 42
 - highlight used parameters · 40
 - window · 61
- Prozessn_Running · 183

R

- Read_Timer · 184
- Refresh_RingBuffer · 188
- register access · 103
- regular expressions · 31
- Rem · 185

- remarks · 185
- replace
 - examples · 30
 - regular expressions · 31
 - text · 26
- reset
 - TiCo processor · 12
- Ringbuffer
 - dimensioning · 137
 - overview · 186
- ringbuffer
 - check number of elements · 91
 - data loss · 90
 - design of data structure · 89
- RingBuffer_Clear · 190
- RingBuffer_Empty · 192, 220
- Ringbuffer_For_Read · 186
- Ringbuffer_For_Write · 186
- RingBuffer_Full · 194, 221

S

- Save All Files of Project · 61
- SDRAM, *see* memory
- search · 26
 - declaration of
 - instruction/variable · 35
 - examples · 29
 - regular expressions · 31
- SelectCase · 196
- separator : · 127
- Set_Par · 222
- Set_Par_Block · 224
- Set_TiCo_RingBuffer · 226
- SetData_Long · 229
- settings
 - print · 56
- Shift_Left · 199
- Shift_Right · 200
- (bit) shifting
 - left · 199
 - right · 200

- Short-cuts · 1
- show
 - declarations · 39
 - line numbers · 54
 - process options window · 14
- show declarations · 72
- Sleep · 201
- smart format · 21
- snippets · 37
- source code
 - creating · 16
 - formatting · 21
 - information · 14
 - status bar · 67
 - structured display · 21
 - to do's · 70
 - use in a project · 61
- source code status bar · 14
- special char, find · 31
- SRAM, *see* memory
- SRAM_EXTERN
 - Dim · 137
- stack size · 68
- starting TiCoBasic · 9
- status bar · 67
- status bar of source code
 - window · 14
- status message, compiler · 68
- Step (For ...) · 145
- stop
 - TiCo processor · 12
- stop process
 - itself
 - in Event: · 142
- structure
 - Coloured display of source
 - code · 21
 - indent lines · 23
 - program sections · 99
 - toggle folding · 23
- Sub · 202

- subroutine
 - general features · 99
 - library
 - definition (Lib_Sub) · 166
 - general · 100
 - macro · 202
 - position in the program · 79
- subtraction · 122
- SX, *see* memory
- symbolic names · 78
- syntax
 - highlighting · 21, 55
- system variable
 - overview · 85
 - Processdelay · 180
 - ProcessN_Running · 183

T

- tab
 - size · 54
- TCP/IP
 - see* Ethernet
- TDrv_Init · 232
- terminate, *see* stop process
- text
 - find And replace · 26
 - find quickly · 25
 - fold ranges · 23
 - indenting · 23
 - smart format · 21
- Then (If ... Then) · 149
- TiCo bootloader
 - menu entry · 58
- TiCo bootloader, programming · 41
- TiCo processor
 - reset / stop · 12
- TiCo_Flash · 234
- TiCo_Get_Processdelay · 236
- TiCo_Load · 238
- TiCo_Reset · 240
- TiCo_Reset_Mode · 242

- TiCo_Set_Processdelay · 243
- TiCo_Start · 245
- TiCo_Start_Process · 246
- TiCo_Stop · 248
- TiCo_Stop_Process · 249
- TiCo-11 · 151
- TiCoBasic
 - demo mode · 10
 - license agreement · 5
 - start · 9
- TiCoBasic: differences to ADbasic · 5
- TiCoBasicCompiler, command line · 9
- time
 - cycle time · 113
- time saving
 - constants instead of variables · 104
 - measure faster · 104
 - register access · 103
 - setting waiting time · 105
 - use waiting times · 105
- timer event · 108
- timer, *see* counter
- timing
 - see* optimize
 - operating modes
 - externally controlled process · 115
 - general · 114
 - timer-controlled process · 115
- To (For ...) · 145
- to do list · 70
- toggle folding · 23
- tool bar · 14
- toolbox · 61
- Tools
 - bootloader · 58
 - load binary file · 58
 - TGraphTiCo · 58

tools

- TBin · 74
- TButton · 74
- TDigit · 74
- TFifo · 74
- TGraph · 74
- TLed · 74
- TMeter · 74
- TPar_FPar · 74
- TPoti · 74
- TProcess · 74

U

- Uncomment Block · 23
- Unmark Control block · 33
- Until (Do ...) · 140
- upper / lower case letters · 16
- USB · 117
- user defined instructions and variables · 78
- user surface · 13
- utility programs, *see ADtools*

V

- value range · 82

variables

- autocomplete · 36
- declaration info · 38
- display · 63
- global · 82
 - highlight used · 40
 - name · 80
- initialize · 12, 78
- jump to declaration · 35
- local · 85
 - allocate memory area · 86
 - name length · 85
- overview · 80
- switch hex/decimal display · 64
- symbolic names · 78
- see also* system variable

view

- to do list · 70

W

- wait
 - NOP · 173
 - setting waiting time exactly · 105
 - Sleep · 201
- Warten
 - NOPs · 174
- Window
 - source code status bar · 14

window

- compiler options · 48
- declarations · 72
- global variables · 71
- info range · 68
- info window · 68
- overview · 13
- parameter · 63
- process Options · 50
- project · 61
- source code information · 14
- status bar · 67
- to do list · 70
- toolbox · 61

- without event trigger · 112

Workload · 251

workload

- definition · 114
- display · 67
- workspace size · 68

X

XOr · 204

Symbols		FFT_Calc	312	Not	235
< = > (comparison)	151	FFT_Calc_DM	314	Or	236
+ (addition)	140	FFT_Calc_DX	316	P	
+ (String addition)	141	FFT_Init	311	P1_Sleep	238
- (subtraction)	143	FFT_Mag	305	P2_Sleep	240
* (multiplication)	144	FFT_Mag_Scale	309	Peek	242
/ (division)	145	FFT_Phase	307	Poke	243
^ (power)	146	FFT_Scale	303	Processdelay	245
= (assignment)	150	FIFO	182	Processn_Running	249
: colon	149	FIFO_Clear	184	Process_Error	248
" " (String)	271	FIFO_Empty	186	R	
#Define	170	FIFO_Full	187	Read_Timer	250
#If ... Then ... {#Else ...}		Finish:	188	Rem	252
#EndIf	201	Flo40ToStr	192	Reset_Event	253
#Include	206	FloToStr	190	Restart_Process	254
#..., preprocessor state-		For ... To ... {Step ...}		S	
ment	148	Next	194	SelectCase	255
A-B		Function ... EndFunction		Shift_Left	258
AbsF	153	196		Shift_Right	260
AbsI	154	G-J		Sin	262
And	155	If ... Then ... {Else ...} En-		Sleep	263
ArcCos	157	dIf	199	Sqrt	265
ArcSin	158	Import	203	Start_Process	266
ArcTan	159	Inc	205	Stop_Process	269
Asc	160	Init:	208	" " (String)	271
C		IO_Sleep	210	StrComp	273
Cast_FloatToLong	161	K-L		StrLeft	275
Cast_LongToFloat	162	Lib_Function ... Lib_End-		StrLen	278
Chr	163	Function	212	StrMid	279
Cos	164	Lib_Sub ... Lib_EndSub		StrRight	282
CPU_Sleep	165	217		Sub ... EndSub	285
D		LN	222	T-Z	
DATA_n	167	LngToStr	223	Tan	288
Dec	169	Log	225	Trace_Mode_Pause	289
Dim	172	LowInit:	226	Trace_Mode_Resume	290
Do ... Until	175	M-O		ValF	291
E-F		Max_Float	228	Vall	294
End	177	Max_Long	230	XOr	296
Event:	178	Min_Float	229		
Exit	180	Min_Long	231		
Exp	181	Mod	320		
FFT	300	NOP	234		

