ADbasic

Real-Time Development Tool for *ADwin* Systems

ADbasic Version 5.00

March 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 56819
E-Mail:	info@ADwin.de
Internet	www.ADwin.de

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

Table of contents

Dear Reader,

ADbasic 5 is the programming tool for your *ADwin* system that allows you to create special measurement, open-loop, or closed-loop control application. The purpose of this manual is to: introduce you to the basics of programming real-time processes for the *ADwin* system; and act as a reference manual.

The development environment *ADbasic* 5 has been completely re-worked and now provides a lot of comfort for easy handling and editing (see also "News in ADbasic 5" on page 5).

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

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

chapter 3 describes the reworked 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).

message

Conventions

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

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

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

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

The light bulb icon denotes examples showing practicable solutions.

The `Courier` font-type is used for text displayed on screen, e.g in windows or menus, or input via the keyboard. The names of menus and submenus are shown similarly: `Menu` ▸ `submenu`.
File names and path names are additionally emphasized as follows `<path\xx.ext>`.

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

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

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

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

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

– Hexadecimal notation: `11h`

– Binary notation: `10001b`

1 Introduction

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

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

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

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

An *ADbasic* with only a few lines can:

– Acquire measurement parameters up to sampling rates of 800 kHz

– Develop fast digital controllers with sampling rates of up to 400 kHz

– Simultaneously generate *and* measure analog signals, e.g. for dynamic measurement of a test stand characteristic

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

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

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

ADbasic is no longer needed once the real-time processes are running properly.
A user interface running on the computer transfers the generated binary code to the system, starts, controls and stops the processes, and controls and monitors the processes and process data of the *ADwin* system.

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

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

Under Windows, a DLL or ActiveX-interface enable access to the *ADwin* system from several programs simultaneously.
Based on this, drivers for .NET as well as for many development environments are available which help in creating a user interface, e.g. Delphi, Visual-Basic, C#.NET, Visual-C++. Optionally, measurement packages such as TestPoint, LabVIEW, Diadem, HP-VEE, Intouch and Matlab can be used.
Finally, there are also drivers for the platforms Linux, MacIntosh and Java.

2 News in *ADbasic 5*

You run *ADbasic 5* just as usual but with more comfort and a new design.

A lot of new tools hide under the new surface, which often appear at second glance only. You will soon discover the new functions to make programming considearbly easier.
Give it a try!

Easier programming

- Autocomplete for instruction or variable using CTRL-SPACE (page 30).
- Inserting code snippets with short-cuts (page 31).
- Displaying declaration of instruction or variable (page 32).
- Displaying declarations of a file (page 32).
- ToDo List to manage uncompleted tasks (page 63).
- New short-cuts (see chapter A.1).

Enhanced source code display

- Indenting text lines automatically (page 18).
- Line numbers at the left margin.
- Syntax highlighting enhanced and with new color palette (page 18).

  Please note: An instruction or a variable which is not highlighted, is a good hint for a wrongly written keyword or a missing include file.
- Colored bars at the left margin for edited lines.
- Folding text ranges (page 19).

Changes and news in the user interface

- New Editor bar, which provides a bunch of new editing functions (page 17).

– New *ADtools* bar to start the handy tools directly (page 70).
– Tool bar:
  - New buttons for Managing Projects (page 35).
  - The device no. has moved to the status bar.
– Project, parameter and process window are combined as Toolbox (page 57).
– The Project Window (page 57) displays files sorted into groups of source code files and include files.
– The source code window has a tab at the top for each open file.
– The Status Bar displays some of the current settings (page 61).

  A double click on a setting opens the appropriate dialog box.

Quicker search and find

– Finding and replacing text across several files (page 21); even Regular expression are available (page 26).
– Using bookmarks (page 28).
– Jump to a program line (page 29).
– Jumping to declaration of instruction or variable (page 29).

Miscellaneous

– Source code files are saved with a new format.

  To use files with *ADbasic 4* furthermore, you can save them using `Save as` with the file type `ADbasic4 Bas-File`.

Also library source codes have their own file format ((file type `LibFile` with file extension `*.bas`). Only this file format allows to create a library binary file.

– You may compile all files of a project–and create both binary files and library files–with a single click:
Menu `Build`, menu entry `Make All Bin files of Project`.

– Before compiling, all changed files are automatically saved (with *ADbasic 5* format, see above)

– You can use relative paths for Include or Library files.

The base directory is–if the the source code is member of the project–the directory of the project file, otherwise the directory of the source code file.

– Command Line Calling has been reworked completely and enhanced (see chapter A.4, see annex page 7). Because of the changes, command line calls are not compatible to *ADbasic 4*.

3 Development Environment

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

3.1 Basic Steps

3.1.1 Starting the Development Environment

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

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

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

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

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 *ADbasic* 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 *ADbasic* license in chapter 3.1.2 on page 9.

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

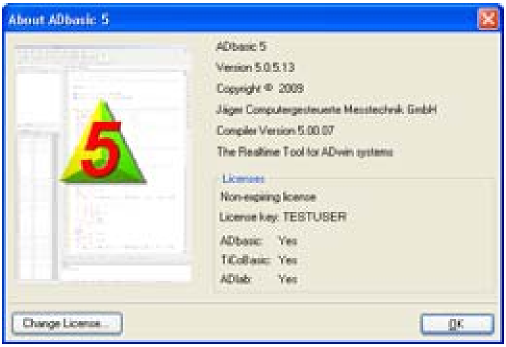
   The development environment saves the settings so that upon a new start of *ADbasic* they will not need to be entered again, unless a different *ADwin* device is used.

3.1.2 Check or change ADbasic licenses

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

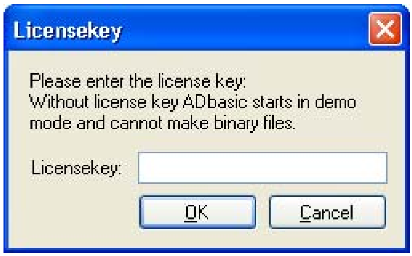
1. Select the menu entry `Help` ▸ `About`.

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



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 *ADbasic* manual.

In *ADbasic*, the following licenses are available:

– No license (demo mode)

  Without valid `License key`, *ADbasic* will operate in demo mode. In this mode the development environment 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, *ADbasic* 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 *ADbasic* are described in the License Agreement (annex see A-4).

3.1.3 Loading the ADwin Operating System

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

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

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

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

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

3.1.4 Basic Elements of the Development Environment

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

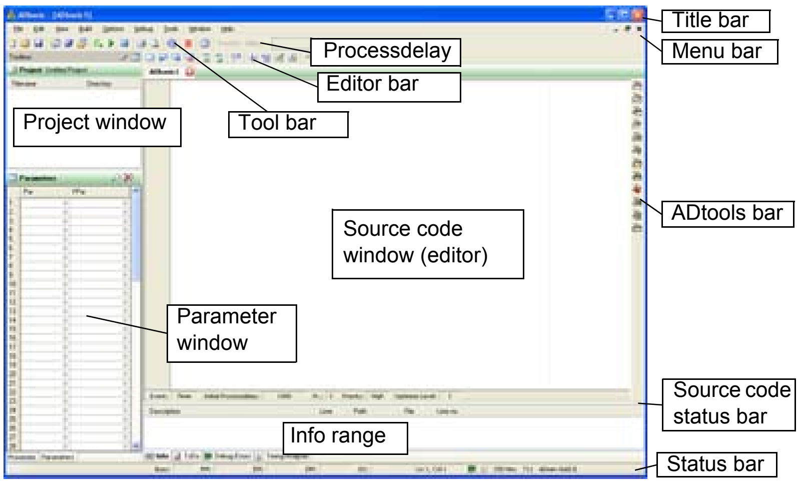


Fig. 1 – Elements of the *ADbasic* development environment

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 ADbasic (see annex).

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

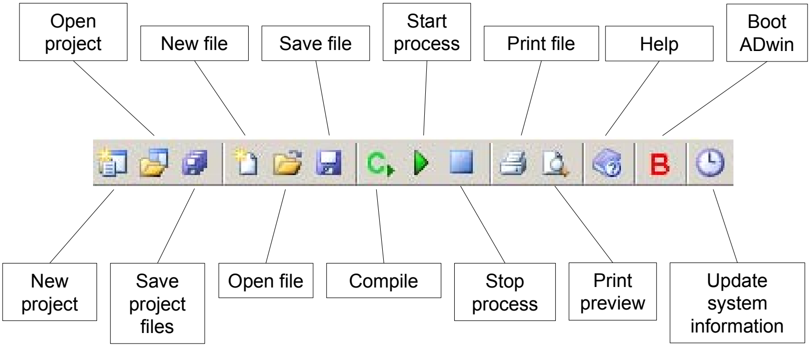


Fig. 2 – The tool bar

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 3.8.3, page 58) by clicking `Scan Global Variables` ; see Displaying used global variables and arrays on page 34.

– The info range at the bottom displays information in several windows:
  - Info window: The compiler’s error messages (highlighted red) and warnings (see chapter 3.9.1 on page 62).
  - ToDo List: A simple ToDo list from comment lines (see chapter 3.9.2 on page 63).
  - Search results from a search in all files of a project (see chapter 3.4.2 on page 21).
  - Debug information if the debug mode is enabled (see Debug mode Option, page 52).

Please note: Editing in the source code window is supported by several tools (see Creating source code on page 15).

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

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

– Processdelay (process cycle time) of the process which has the number as the currently edited source code. Displayed at the right side of the toolbar.

– The values of the global variables in the Parameter Window; a change to one of these values will immediately be transferred to the *ADwin* system.

– The status of running processes in the Process Window (page 60).

– Memory usage information in the Status Bar (see chapter 3.8.5 on page 61).

According to compiler settings, additional information is available about running processes :

– Process timing: Timing window (page 48)

– Run-time errors: Debug window (page 52)

3.2 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 35: Managing Projects).

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

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

The source code editor provides several useful tools. Call the tools via Context menu in source code window (page 16) or via Editor bar (page 17):

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

Find more editor functions here:

- Formatting source code, page 17
- Searching and replacing, page 20
- Writing programs with ease, page 29

3.2.1 Calling online help

The Help Menu (page 55) 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.

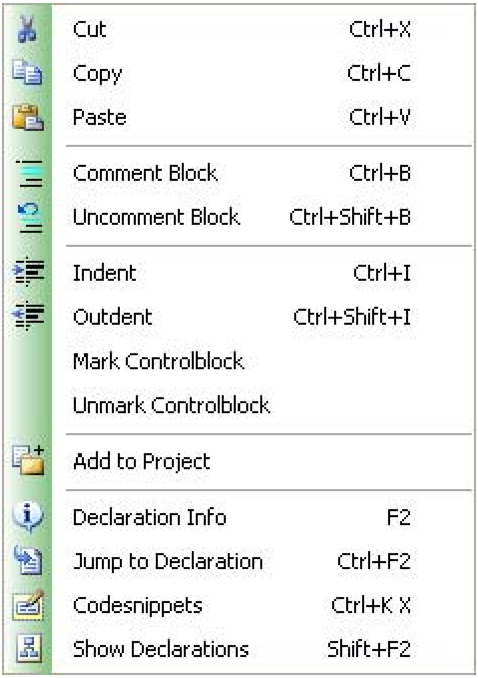
If the cursor is set upon an invalid instruction the help index shows up. Reasons may be:

- 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.

3.2.2 Context menu in source code window

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



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 19.

– `Indent`, `Outdent`: Indenting text lines, page 18.

- `Mark Control block`, `Unmark Control block`: Marking control blocks, page 28.
- `Declaration Info`: Displaying declaration of instruction or variable, page 32.
- `Jump to Declaration`: Jumping to declaration of instruction or variable, page 29.

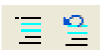
These functions are available without marking:

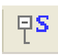
- `Add to Project`: Add a file to the project.
- `Code snippets`: Inserting code snippets, page 31.
- `Show all Declarations`: Displaying declarations of a file, page 32.

3.2.3 Editor bar

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

 Using bookmarks, page 28.

 Changing lines into comment, page 19.

 Folding text ranges, page 19.

 Displaying declaration of instruction or variable, page 32.

 Jumping to declaration of instruction or variable, page 29.

 Inserting code snippets, page 31.

 Displaying declarations of a file, page 32.

 Undo the previous editing action or redo it.

3.3 Formatting source code

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

– Syntax highlighting, page 18
– Smart formatting, page 18
– Indenting text lines, page 18
– Changing lines into comment, page 19
– Folding text ranges, page 19

Find more editor functions in the sections:

– Creating source code, page 15
– Searching and replacing, page 20
– Writing programs with ease, page 29

3.3.1 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 changedunder `Options` ▸ `Settings`, Editor - Syntax Highlight (see page 46); the window also shows an overview of syntax groups.

Syntax highlighting requires an active option `Parse Declarations` under Editor - General (see page 45).

3.3.2 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 45).

3.3.3 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.

3.3.4 Changing lines into comment

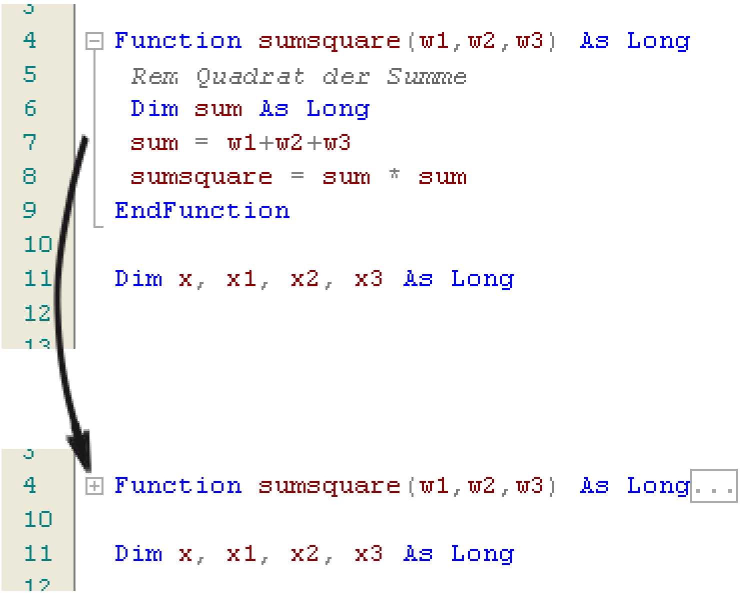
Marked lines may be changed into comment lines in one action by selecting the menu entry `Comment Block` from the source code context menu (right mouse click). The editor will then insert a comment char `'` at every of the marked lines so the compiler will skip these lines.

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

3.3.5 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 folable 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 45) is active.

3.4 Searching and replacing

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

– Finding text quickly, page 21
– Finding and replacing text, page 21
– Regular expression, page 26
– Marking control blocks, page 28
– Using bookmarks, page 28
– Jumping to declaration of instruction or variable, page 29

There are more editor functions:

– Creating source code, page 15
– Formatting source code, page 17

– Writing programs with ease, page 29

3.4.1 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.

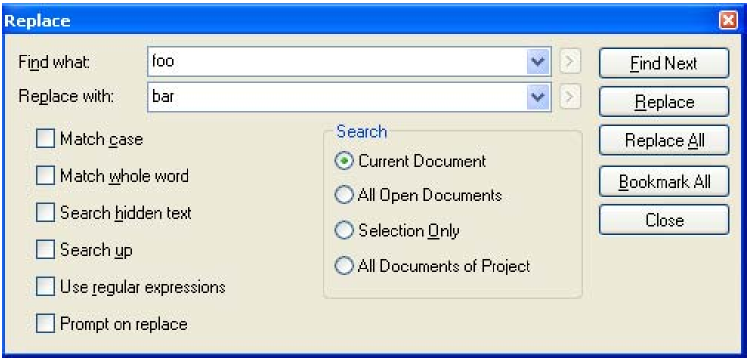
– All open documents are searched.

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

3.4.2 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 26).

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



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

Option	Description
`Match case`	Option active: Find text having the given pattern of uppercase and lowercase letters. Option inactive: Uppercase and lowercase letters are of no importance.
`Match whole word`	Option active: Find occurrences of the text as whole words. Option inactive: Find text also as part of a word.
`Search hidden text`	The option refers to Folding text ranges (see page 19). Option active: Folded text areas are searched. Option inactive: Folded atext 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 26).

Option	Description
`Prompt on replace`	Option valid with `Replace All` only. Option active: Each occurence opens a dialog box to control replacing. Option inactive: All occurences are replaced without query.

5. Set the search range.

Option	Description
`Current Document`	Start search in the current source code at cursor position. If text is selected, the cursor is positioned behind the selection.
`All open Documents`	All open documents are searched, starting with the current source code.
`Selection only`	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` ▸ `Find Next` finds the next occurence 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 all open documents, you should use `Find Next` and `Replace` to make sure you have spelled the replacement string correctly, before replacing the rest with `Replace All`.

Examples - Finding Text

Examples for finding text with Regular expressions.

– Find all spaces or tabs at the end of a line:
  `[  ]+$`

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

- Find everything on a line:
  `^.+`

  The search string finds the beginning of a line, followed by one or more of any characters, up to the end of the line.
- Find `$12.34`:
  `\$12\.34`

  Note that `.` and `$` have been escaped using the backslash `\` to hide their regular expression meanings.
- Find a string, which is valid as variable name in *ADbasic*:
  `\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`

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

Examples - Replacing Text

Examples for replacing text with Regular expressions.

- Find two numeric strings separated by one or more spaces:
  `([0-9]+) +([0-9]+)`

  and swap them around, using a colon to separate them:
  `$2:$1`
- To change simultaneously:

  from `X100000` to `X100.000`

  from `Y100123` to `Y100.123`

  from `Z600` to `Z.600`

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

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

3.4.3 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: `Ma.s` matches `Mats`, `Mars` und `Mads`, but not `Mas`.
`[ ]`	Any one of the characters 1. given explicitely in brackets, or 2. any of a range of characters separated by a hyphen (-). Examples: `h[aeiou][a-z]d` matches: `hard`, `head`, `hand` and `hold`; `[A-Za-z]` matches any single letter. The regular expression `x[0-9]` matches `x0`, `x1`, …, `x9`.
`[^]`	Any characters except for those after the caret `^`. Example: `h[^uo]t` matches `hat` and `hit`, but not `hot` or `hut`.
`^`	The start of a line (column 1). Example: The search string `^start` matches `start` only, when it is the first word on a line.
`$`	The end of a line (not the line break characters). Use this for restricting matches to characters at the end of a line, but not `\n`. Example: `end$` only matches `end` when it is the last word on a line.
`\b`	The start of a word.
`\B`	The end of a word.

Meta-zeichen:	Bedeutung:
`\n`	A new line character, for matching expressions that span line boundaries. A `\n` cannot be followed by operators `*`, `+` or `{}`. Do not use this for constraining matches to the end of a line. It's much more efficient to use "$".
`( )`	Expression in braces is stored as pattern in internal registers. The register content may be re-used in the search or replacement string. Up to 9 patterns can be stored, numbered according to their order in the regular expression. The corresponding replacement expression is `$x` and `\x` in the search string, for `x` in the range 1…9. Example: If the search string `([a-z]+) ([a-z]+)` matches `guide user`, `$2 $1` would replace it with `user guide`.
`*`	Matches zero, one or more of the preceding characters or expressions. Example: `ha*d` matches `hd`, `had` and `haad`.
`?`	Matches zero or one of the preceding characters or expressions. Example: `ha?d` matches `hd and had`, but not `haad`.
`+`	Matches one or more of the preceding characters or expressions. Example: `ha+d` matches `had and haad`, but not `hd`.
`|`	Matches either the expression to its left or its right. Example: `had|haad` matches `had`, or `haad`.
`\`	"Escapes" the special meaning of the above expressions, so that they can be matched as literal characters. Hence, to match a literal backslash `\`, you must use `\\`. Example: `^a` matches an `a` at the start of a line, but `\^a` matches the string `^a`.

3.4.4 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:**, **LOWINIT:**, **EVENT:**, **FINISH:**
- **DO** … **UNTIL**
- **FOR** … **NEXT**
- **IF** … **ENDIF**
- **SELECTCASE** … **ENDSELECT**
- **FUNCTION** … **ENDFUNCTION**
- **SUB** … **ENDSUB**
- **LIB_FUNCTION** … **LIB_ENDFUNCTION**
- **LIB_SUB** … **LIB_ENDSUB**

All control structures are also foldable text ranges (see Folding text ranges on page 19).

3.4.5 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.

3.4.6 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 nuber of the desired program line.

To show source code line numbers, the option `show linenumbers` under Editor - General (see page 45) must be enabled.

3.4.7 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 45) is active.

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

3.5 Writing programs with ease

Be at ease while programming using the following functions:

– Autocomplete for instruction or variable, page 30
– Documenting self-defined instructions and variables, page 52
– Inserting code snippets, page 31
– Displaying declaration of instruction or variable, page 32
– Displaying declarations of a file, page 32
– Displaying used global variables and arrays, page 34

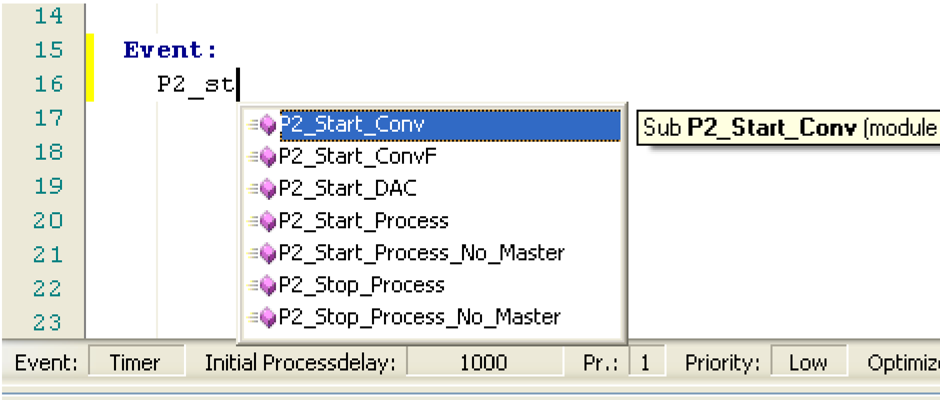
Find more editor functions here:

– Creating source code, page 15

– Formatting source code, page 17
– Searching and replacing, page 20

3.5.1 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 decalration 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.

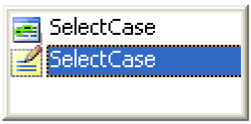
Autocomplete is only available, when the option `Parse Declarations` under Editor - General (see page 45) is active.

3.5.2 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 Select-Case 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\ADbasic\Common\` with a browser.

3.5.3 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 braes 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 45) is active.

3.5.4 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 45) is active.
– set the cursor on the keyword and press [F2].
– set the cursor on the keyword and select `Declaration Info` in the editor bar or in the context menu.

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

The display of declarations is only available, when the option `Parse Declarations` under Editor - General (see page 45) is active.

3.5.5 Displaying declarations of a file

To display all declarations, include and library files referring to a source file, set the Declarations Window to the foreground (see page 69). 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 45) is active.

3.5.6 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 58).

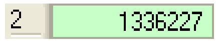
This results in two displays:

– the Global Variables Window displays all used global variables and arrays.

– in the Parameter Window the used global variables (not the arrays) are highlighted.

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

- Green: Parameter is used in the active source 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 If you change the source code the displays are not updated automatically. To do so, click the `Scan Global Variables` button again.

3.6 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.

A project allows the user:

- Displaying used global variables and arrays of a project (see page 34).
- Compile all files of project at once, using the menu entry `Build` ▶ `Make all Bin Files of Project`.
- 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 3.4.2 "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.

Project-related capabilites can be accessed via project window context menu (right mouse click, see "Project Window" on page 57) or in the menu `File`.

3.7 Menus

The menu bar contains these menus:

3.7.1 File Menu

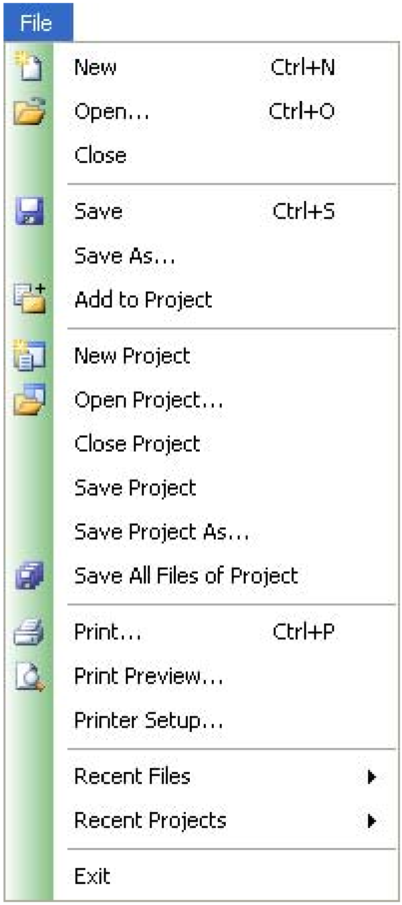
The `File` menu contains instructions for managing files and projects.

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

Projects can also be opened, saved and created in the same way as files, with the exception that no more than one project can be open at a time. More instructions are available in the project window (see chapter 3.8.2).

The print functions can also be found in the menu.

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

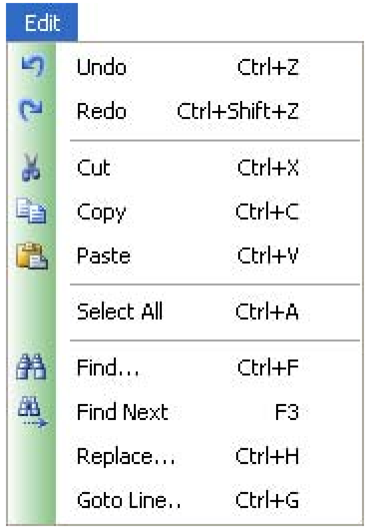


3.7.2 Edit Menu

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

Moreover the menu offers functions for searching (`Find`, `Find Next`) and replacing (`Replace`); see Finding and replacing text on page 21.

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

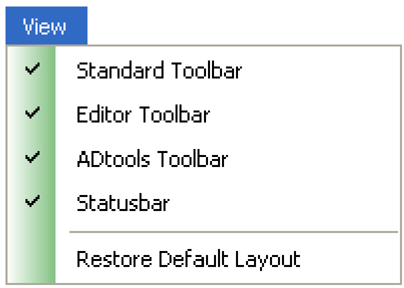


3.7.3 View Menu

In the `View` menu you may open or close

– the tool bar
– the editor bar
– the ADtools bar
– the status bar.

You find further information about the process window in chapter 3.8.4 on page 60, about the toolbar see fig. 2.

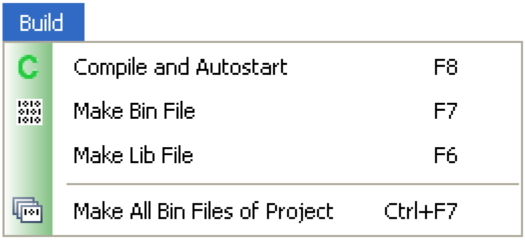


With `Restore Default Layout`, the default layout, which was active at the initial start of the *ADbasic* program, can be restored with a single mouse-click. This refers also to the Toolbox setttings (page 57).

3.7.4 Build Menu

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

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



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

A change of file may occur by automatic indenting of text lines (see chapter 3.3.3 on page 18), for example when opening a previously unformatted file.

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

The process is only started automatically if the `Autostart` option, in the `Options\Compiler` menu, is set to `Yes`. Otherwise, the process can be started with the button in the toolbar or in the process window (see page 60).

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

`Make Bin File` is only available for licensed *ADbasic* users. It compiles the active source code into a binary file and saves it automatically. The file is stored in the directory of the source code file, but with the extension `.Txn`. The `x` denotes the processor type and `n` the process number (see Options Menu, Process Options dialog box).

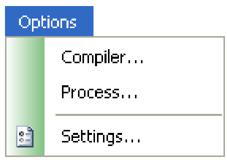
A binary file with the extension `<*.TA3>` can be transferred to an *AD-win* system equipped with a T10 processor, which administers it as process 3. Binary files can be transferred to the *ADwin* system from development environments such as C or Visual Basic (see chapter 6.3.4 on page 120).

`Make Lib File` is available for licensed *ADbasic* users only. It compiles the active source code–the file must be saved as file type `LibFile`–into a binary file and automatically saves it as library file. The library is stored in the same directory and with the same name as the source code file, but with the file extension `.LIx`. (where `x` denotes the processor type.) Afterwards the library can be included into other source codes that use their functions and subroutines (see chapter 4.5.1 on page 94).

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

3.7.5 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* systemon which the compiled source codes are to be executed as process.

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

Fig. 3 – The `Compiler Options` dialog box

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

  The abbreviations correspond to the following full names:

Abbreviation	T11	T10	T9	T8	T5	T4	T2
Full name	ADSP TS101S	ADSP 21160	ADSP 21062	T805	T450	T400	T225

Fig. 4 – Processor Names

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

  The device number is set using the program `<ADconfig.exe>`. The default setting is `150 Hex`.
- `Do not access the Device`: If inactive, a binary file will be automatically transferred to the hardware after compilation. Thus, the ADwin hardware must be connected before compilation.

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

- `Load standard processes`: With active option the standard processes 11, 12 and 15 (see chapter 6.1.1 on page 109) are loaded into the *ADwin* system during boot process. With inactive option the loading of processes 11 and 12 is suppressed.

  This setting is only available for *ADwin-Gold* and *ADwin-light-16*.
- `Autostart`: Active option causes the binary file, generated and transferred to the *ADwin* system during compilaton, to be immediately started. With inactive option, the process requires to be started by clicking the button  in the toolbar or in the process window.
- `Remember Device No.`: Active option saves the last used Device No. (see above) on closing *ADbasic*; the next start-up will automatically use the saved number.

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

Process Options dialog box

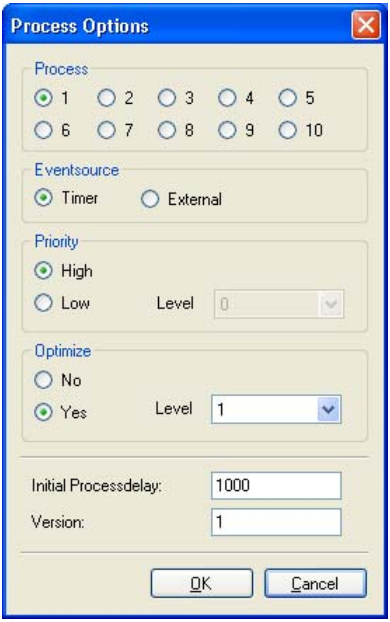
This dialog box contains the compiler options for the currently opened source code window; the properties of the process which is to be compiled from the opened source code and transferred to the *ADwin* system.

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

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

The dialog box for T4, T5 or T8 processors differs slightly from the standard dialog box and is described in the Appendix A-5.1.

Settings for source code



Settings for library

Fig. 5 – The `Process Options` dialog box

– `Process`: Process number

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

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

– `Eventsource`: Sets the event source signal which initiates the **EVENT:** section of the process.

- `Timer`
  sets the internal counter as event signal. The system variable **PROCESSDELAY** determines the delay in which the counter creates an event signal.
- `External`
  sets the (external) signal the event input of the *ADwin* system as

event signal, for instance a sensor impulse. In this case, the `Priority` option must be set to `High` anyway.
How you can use an external event input in an *ADwin-Pro* system, is explained in the *ADwin-Pro* software documentation under EventEnable.

– `Priority`: The priority of the process.

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

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

– `Optimize`: Status and level of compiler optimization.

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

  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 18.

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

`Smart format`: Format lines automatically, see "Smart formatting" on page 18.

`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 29.

`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 3.3.1 "Syntax highlighting" on page 18; complete syntax highlighting reuires an active option `Parse Declarations` under Editor - General.

You may set the highlighting individually for each syntax element (definition see liste 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:

– *ADbasic*-Syntax (`System related`):
  - `ADbasic sections`: Keywords **`INIT:`**, **`LOWINIT:`**, **`EVENT:`** and **`FINISH:`** for program sections.
  - `Compiler Directives`: Pre-compiler instructions, starting with a `#`.
  - `Reserved Keywords`: Basic instructions in *ADbasic*.
  - `Global Variables`: Global variables `Par_1` … `Par_80`, `FPar_1` … `FPar_80` and `Data_1` … `DATA_200`.
  - `External Keywords`: *ADbasic* instructions for access to inputs/outputs. 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, declared with **`#DEFINE`**.
  - `Local Variables`: Variables declared with **`DIM`**.
  - `Sub Names`: Names of user-defined modules, declared with **`SUB`** or **`LIB_SUB`**.
  - `Function Names`: Names of user-defined modules, declared with **`FUNCTION`** or **`LIB_FUNCTION`**.

- Other:
  - Numbers: Numbers in decimal, hexadecimal and binary notation.
  - Strings: Strings in "double quotes".
- Comments: Comments after **REM** or quote '.
- Standard Text: All elements which do not belong to other groups.

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 *ADbasic* files:

- BTL-Directory: The directory in which the development environment searches for the system files <*.btl>, which are transferred to the *ADwin* system during the boot process (see chapter 3.1.3).
- Include-Directory: The directory in which the compiler searches for include files <*.inc>, which can be included into the source code using **#INCLUDE** instruction (without path).
- Lib-Directory: The directory in which the compiler searches for library files <*.lib>, which can be included into the source code using **IMPORT** instruction (without path).

– `Default working directory`: The directory in which the development environment searches searches for files, if a source code file or a project is opened.

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.

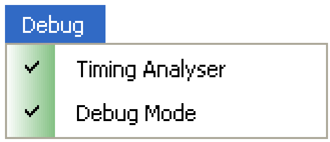
ADtools

The *ADtools* (description see chapter 3.10) can be started from the ADtools bar. If the appropriate option is active, the tool is displayed in the bar.

3.7.6 Debug Menu

The `Debug` menu offers settings which help in finding run-time or symantic errors.

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



Timing Analyzer Option

When the `Timing Analyzer` compiler option is activated, additional information about the timing characteristics of this process are available after compiling a source code. (For display of information see the Show timing information Menu Item).
The setting of this compiler option is displayed in the Status Bar, the setting of a running process in the Process Window.

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

Show timing information Menu Item

The `Show timing information` menu item opens the `Timing Information` window (with activated Timing Analyzer Option only).

For each of the processes 1…10 the window shows 7 parameters, which describe the timing characteristics of the processes since the moment it has been started. More detailed information can be found in chapter 5.3.2 "Check the Timing Characteristics (Timing Mode)".

All timing information is given in clock cycles of the processor (units see fig. 17 on page 115).

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

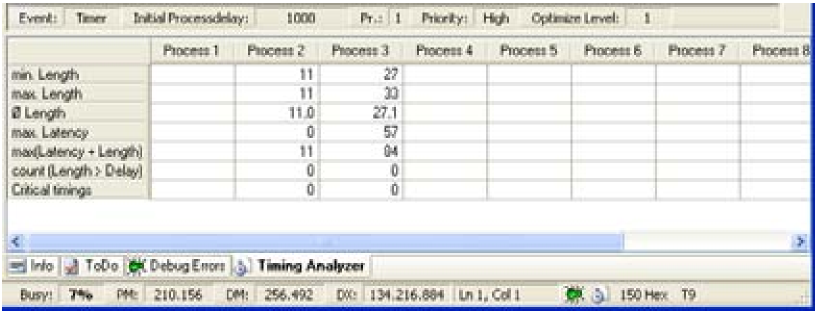
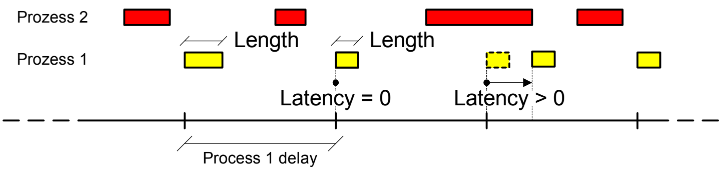


Fig. 6 – The `Timing Analyzer` window

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



The parameters in the window have the following meaning:

– `min. Length`: The minimum time measured for a process cycle
– `max. Length`: The maximum time measured for a process cycle
– ∅ `Length`: Average time of a process cycle.

The average is calculated as mean value from the previous `length` values:

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

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

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

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

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

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

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

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

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

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

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

  This parameter has a different meaning depending on the type and amount of processes (see chapter 6.2.5 "Different Operating Modes in the Operating System", page 119).

  Event signals can be lost under the following circumstances:
  - in a single time-controlled high-priority process
    (also in combination with the externally controlled process)
  - in the externally controlled process (also in combination with one or more time-controlled processes).

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

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

  A loss of an event signal is equated to the fulfilment of the condition:
  - in time-controlled processes:
    `max. latency+length` > 2 × Processdelay
  - in externally controlled processes:
    When processing the section **`EVENT:`** has just been finished, a new external event signal is already waiting. Any more event signals having arrived during this processing time will be lost.

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

Debug mode Option

The `Debug mode` compiler option, when activated, includes additional security queries into the process during the compilation of a source code (see also chapter 5.3.1 on page 104).
The setting of this compiler option is displayed in the Status Bar, the setting of a running process in the Process Window.

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

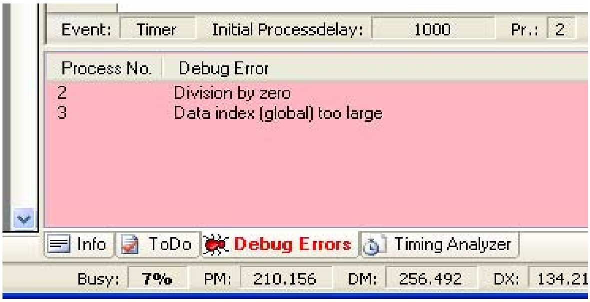


Fig. 7 – The `Debug Errors` Window

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

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

The following table shows which errors are displayed and which corrections are made. The complete list of debug error messages–including those where no corrections are made–are to be found in the annex on page A-17.

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

Run-time error	Correction
`SQRT from negative number`	The square root's result is replaced by the value `0`.

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

`Fifo index is no fifo` The array with the given index is not declared as FIFO or not declared at all.	Instruction is not executed: **FIFO_CLEAR**, **FIFO_FULL**, **FIFO_EMPTY**.
`Address of Pro II module is >15 or <1`	The process is stopped.

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

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

1. Valid for **P2_BURST_INIT**, **P2_BURST_READ**, **P2_BURST_WRITE**

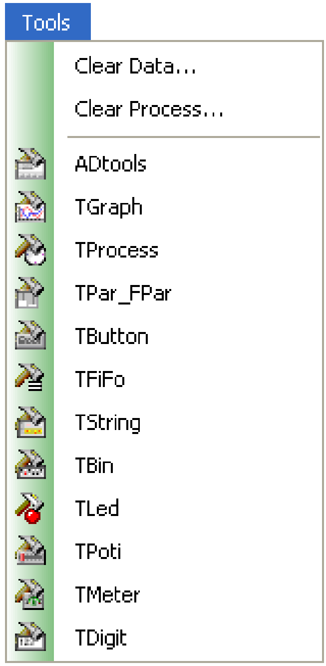
3.7.7 Tools Menu

The `Tools` menu option calls utility programs.

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

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

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



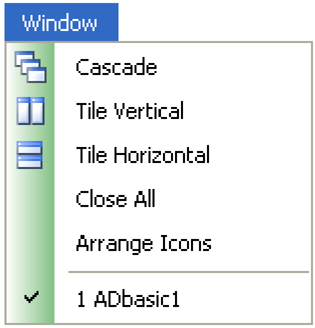
The menu entries `ADtools` and following start a tools each. Find a short description in chapter 3.10 on page 68 .

3.7.8 Window Menu

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

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

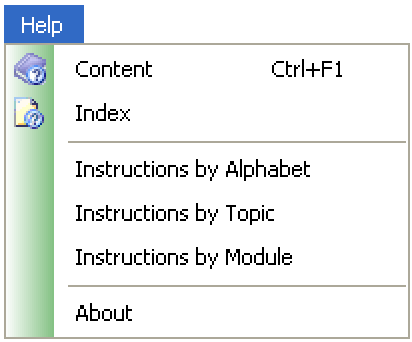
At the bottom of the menu, there is a list of open source codes; by clicking one of these menu items that source code will become the active window. The active source code is checked; in the example at right it is `ADbasic1.bas`.



3.7.9 Help Menu

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

– `Content`: Table of contents
– `Index`: Index directory
– `Instructions by ...`: Sorted lists of instructions.



The instruction lists refer to the *ADwin* system, which is set in the Compiler Options dialog box on page 40.

Altenatively, 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 entereing a valid `License key`, *ADbasic* runs in demo mode. Indemo mode, the use is only allowed for demonstration, test or evaluation purposes.

3.8 Windows

3.8.1 Toolbox

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

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

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

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

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

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

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

3.8.2 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 57).

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.

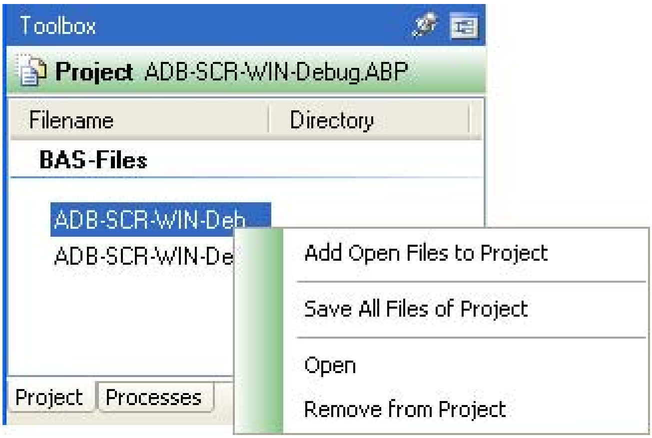


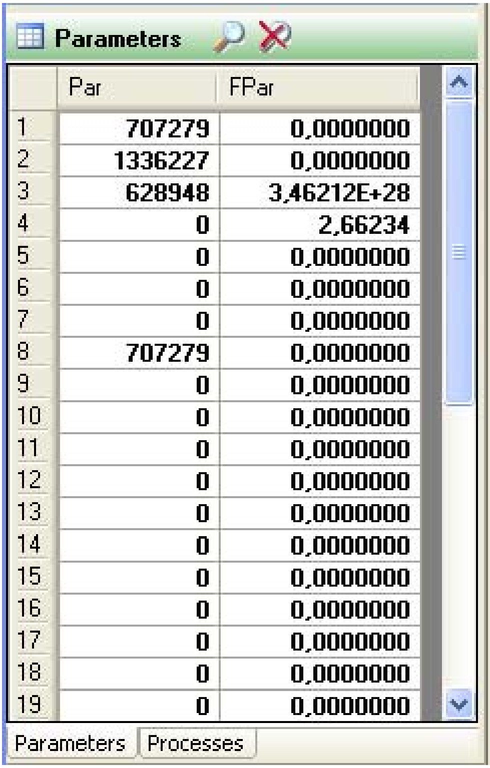
Fig. 8 – The Project Window with the Context Menu

3.8.3 Parameter Window

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

The parameter window is located in the Toolbox (see page 57).

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.



	Par	FPar
1	707279	0,0000000
2	1336227	0,0000000
3	628948	3,46212E+28
4	0	2,66234
5	0	0,0000000
6	0	0,0000000
7	0	0,0000000
8	707279	0,0000000
9	0	0,0000000
10	0	0,0000000
11	0	0,0000000
12	0	0,0000000
13	0	0,0000000
14	0	0,0000000
15	0	0,0000000
16	0	0,0000000
17	0	0,0000000
18	0	0,0000000
19	0	0,0000000

Fig. 9 – The parameter window

To change the display of a parameter's value (`Par_1…Par_80`) between decimal and hexadecimal notation (see `Par_5` in fig. 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 `Par_1…Par_80` at once.

For use of the `Scan Global Variables` button see "Displaying used global variables and arrays" on page 34.

3.8.4 Process Window

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

The process window is located in the Toolbox (see page 57). Open the process window with a click on the tab `Processes`.

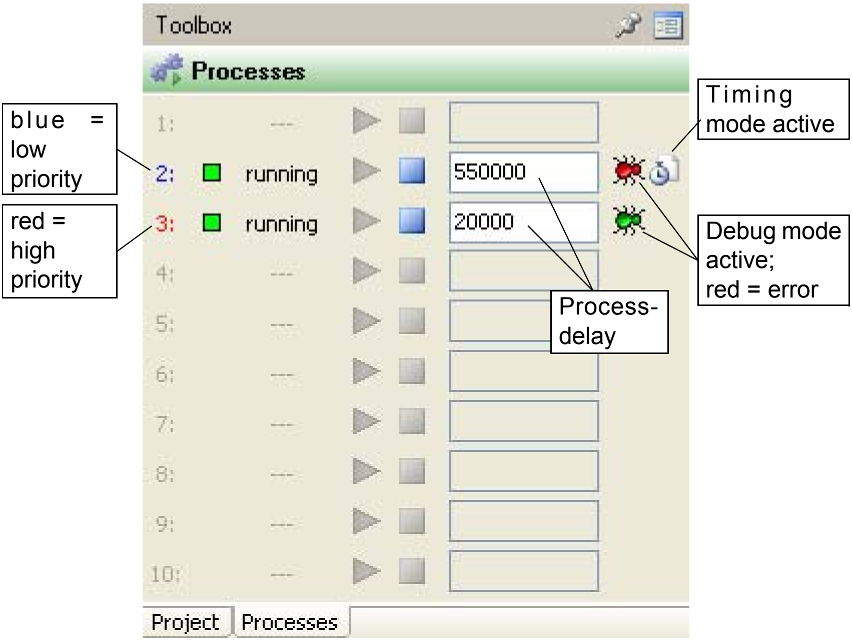


Fig. 10 – The Process Window

For each process the following information is displayed:

– Process 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 *ADwin* system. Please note to not overload the system 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 6.2.1 "Processdelay", page 115.

– Process runs in debug mode

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

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

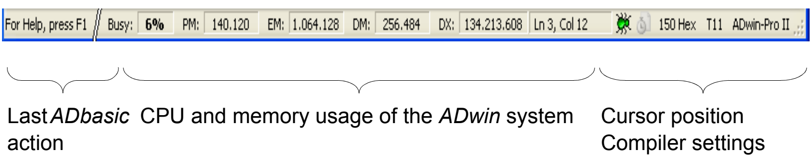
– Process runs in timing mode

  The icon is displayed if the process runs in debug mode. Find more about debug mode under Timing Analyzer Option (page 48). Timing information is displayed in the Window Timing Analyzer.

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

3.8.5 Status Bar

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



– Left side: Information about the last *ADbasic* action.

– Middle: The current CPU and memory usage of the *ADwin* system. This information is displayed, if the communication between the computer and *ADwin* system is active.

– 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`: free program memory in bytes.
– `EM`: free extra memory in bytes (T11 only).
– `DM`: free internal data memory in bytes.
– `DX` / `SX`: free external data memory in bytes.

3.9 Info range

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

– Info window
– ToDo List
– The window `Debug Errors`
– Window Timing Analyzer
– Global Variables Window
– Declarations Window

3.9.1 Info window

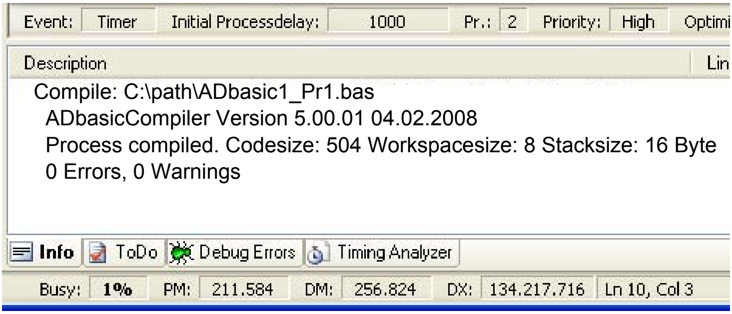
In the info window the compiler messages concerning the current source code are displayed:

– 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 occurence (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).
– `Stacksize`: Internal stack size, which is used for libraries.

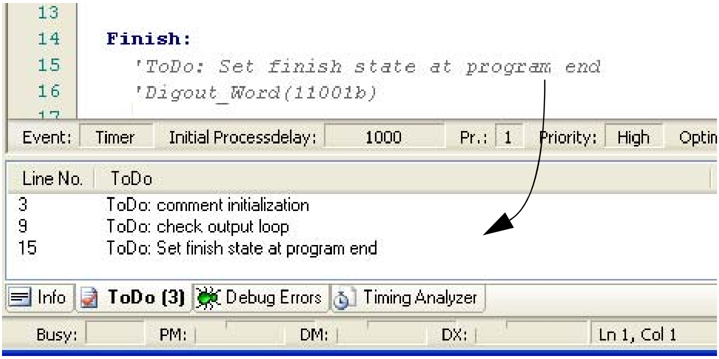
The memory size required in the external data memory (DX) will not be displayed.

3.9.2 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 62).



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

3.9.3 Timing Analyzer Window

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

The window is part of the Info range (see page 62).

All timing information is given in clock cycles of the processor (units see fig. 17 on page 115).

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

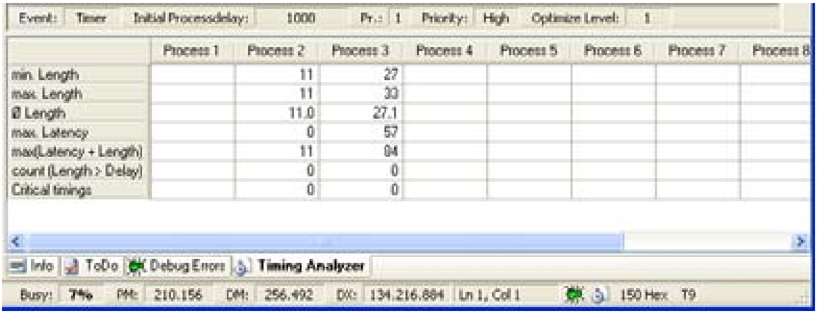
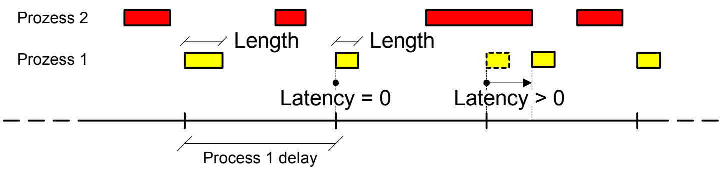


Fig. 11 – The `Timing Analyzer` window

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



The parameters in the window have the following meaning:

– `min. Length`: The minimum time measured for a process cycle
– `max. Length`: The maximum time measured for a process cycle

– ∅ `Length`: Average time of a process cycle.

The average is calculated as mean value from the previous `length` values:

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

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

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

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

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

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

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

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

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

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

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

  This parameter has a different meaning depending on the type and amount of processes (see chapter 6.2.5 "Different Operating Modes in the Operating System", page 119).

Event signals can be lost under the following circumstances:

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

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

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

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

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

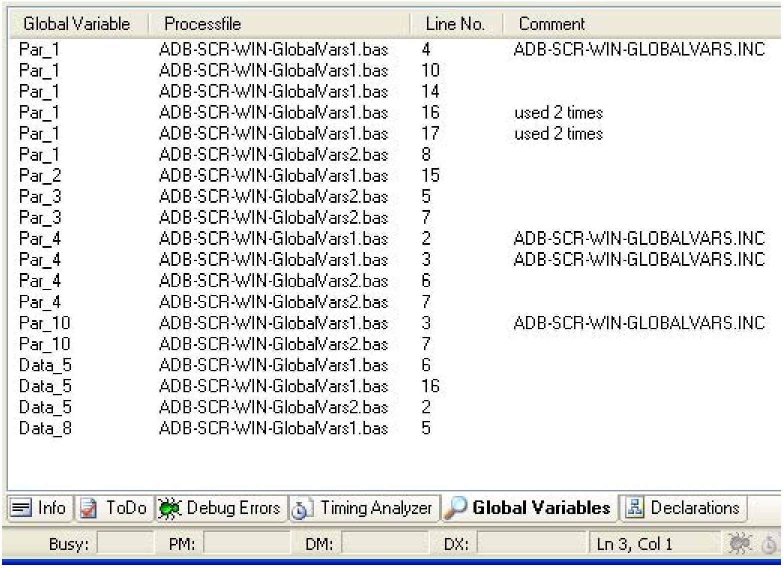
Sometimes it happens that, despite a true condition, no event is lost. Thus, you play it safe reducing the amount of true conditions as far as possible.

3.9.4 Global Variables Window

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

To start or update the display click the button `Scan Global Variables`  in the Parameter Window (see Displaying used global variables and arrays, page 34).

The window is part of the Info range (see page 62).



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	

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.

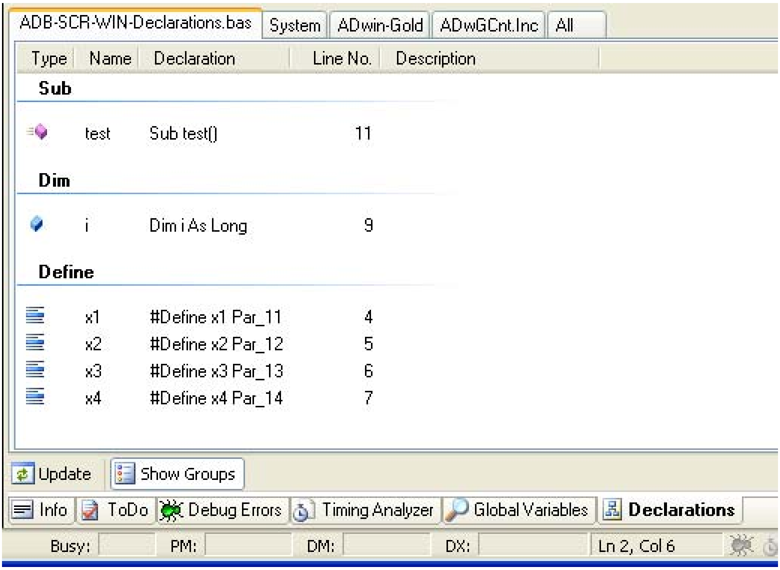
If you change the source code the window is not updated automatically. To do so, use the button `Scan Global Variables` in the parameter window.

3.9.5 Declarations Window

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 62).



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

- `[file].bas`: Declarations within the source file: local variables, arrays, instructions (**SUB**, **FUNCTION**) and symbolic names (**#DEFINE**).
- `System`: System variables and instructions being implemented in *ADbasic*, if they fit to the current compiler settings.

  Global variables `PAR` and `FPAR` are not displayed here. Please note the Global Variables Window (page 67) and the function "Displaying used global variables and arrays" (page 34).

– `ADwin-Gold`, `ADwin-light-16`: Instructions for hardware access, which are implemented in *ADbasic* 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 45) is active.

3.10 ADtools

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

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

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*.
`TGraphTiCo`	displays contents of global arrays of a TiCo processor in a graph.

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

4 Programming Processes

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

4.1 Program Design

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

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.

ADbasic 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 4 sections, which take on different tasks when executed on the *ADwin* system. fig. 10 outlines the ideal steps for an *ADbasic* program.
Each program must at a minimum, have an **EVENT:** section.

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

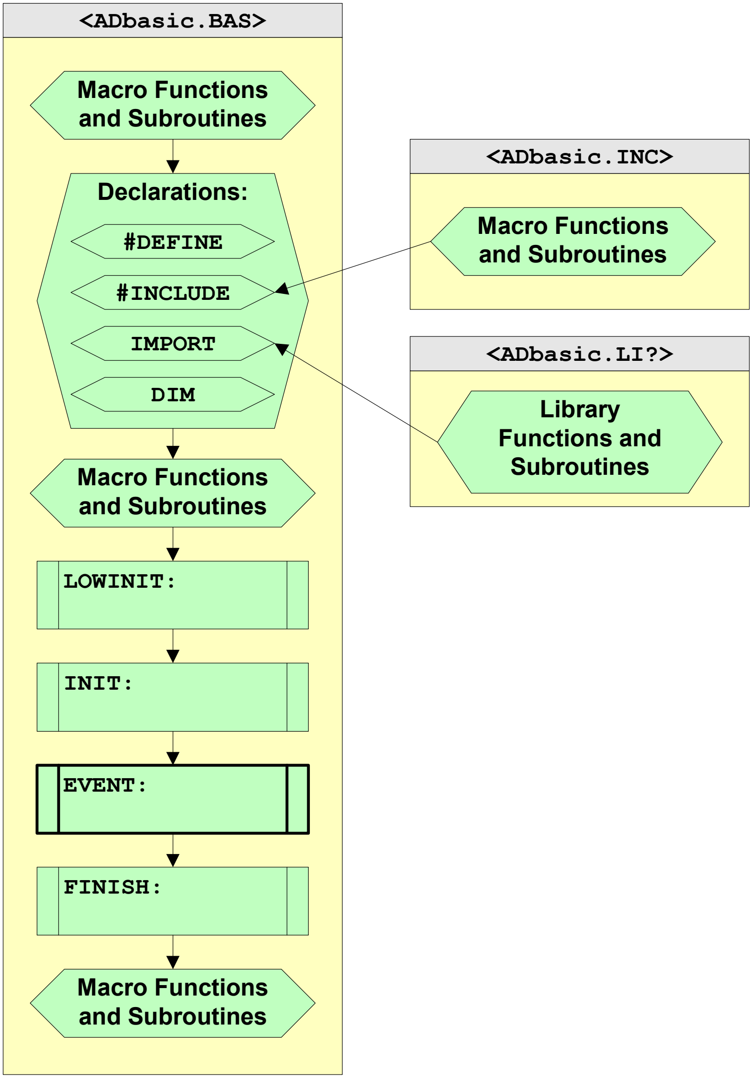


Fig. 12 – Design of an *ADbasic* program

4.1.1 The Program Sections

Each of the program sections (see fig. 12) start with a keyword, as described below.

– **LOWINIT:** can only be used within high-priority processes.

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

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

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

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

– **EVENT:** is the main program section, which is (characteristically) called in regular time intervals until it is stopped. This section is triggered by a cyclic timer event or an external event, depending on the configuration..

– ⚠ **FINISH:** is executed only once after a process has been stopped; it is, therefore, the counterpart to the initialization. This section is always executed at low-priority, level 1.

The **LOWINIT:**, **INIT:** and **FINISH:** sections are optional, while the **EVENT:** section is not and must be included in your program.

4.1.2 User defined instructions and variables

Symbolic names

The instruction **#DEFINE** defines symbolic names (see page 153). Group all of these definitions at the beginning of the file and before the start of the program sections.

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

Arrays and Local Variables

In an *ADbasic* program the local variables and all arrays must be declared with **DIM** before they can be used (see page 155). The global variables Par_n and FPar_n are already pre-defined and do not need to be declared. Variables and arrays have no defined contents after being declared, therefore they should be initialized.

Within the process all variables and arrays are available in all program sections. The global variables and arrays may also be accessed from other processes and from the PC, 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 4.5.1 on page 96). However, the macro definition cannot be done within the program sections. (see fig. 12. on page 73).

Libraries

Libraries must be included before the program sections that use them. Library functions **LIB_FUNCTION** … **LIB_ENDFUNCTION** and subroutines **LIB_SUB** … **LIB_ENDSUB**, when used more than once within a program, require less memory than similar macro functions or subroutines described above (see also chapter 4.5.3 on page 97).

4.2 Variables and Arrays

4.2.1 Overview

Data structure	Name	Data type	Notes
Global variables and arrays			
Variable (Scalar)	Par_1…Par_80	**LONG**	Pre-defined, not declarable, memory area DM
	FPar_1…FPar_80	**FLOAT**	
System variable	**PROCESSDELAY**	**LONG**	
	PROCESS_ERROR	**LONG**	
	PROZESSN_RUNNING	**LONG**	
One- or two-dimensional array (vector)	Data_1[][]… Data_200[][]	**LONG**, **FLOAT**, **STRING**, **FIFO**	Name DATA_ not changeable, only declaration of array number and dimension.
Local variables and arrays			
Variable (Scalar)	selectable	**LONG**, **FLOAT**	must be declared
One-dimensional array (vector)	selectable	**LONG**, **FLOAT**, **STRING**	must be declared

Variables are normally stored in the internal memory DM and arrays in the external memory DX (memory map, see chapter 4.3.1), if not determined explicitely.

All data types have a length of 32-bit.

4.2.2 Data Structures

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

– variables (scalars) **VAR**

Each variable can store one value only.

– arrays, one- or two-dimensional.. ARRAY

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

One-dimensional global arrays `Data_n` may also be used as FIFO (a ring buffer which works according to the principle: First in, first out, see chapter 4.3.4 on page 87).

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

The compiler differentiates

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

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

  System variables are global variables (see page 82).

– Local Variables and Arrays (see page 82):

  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` and `FPar_1` … `FPar_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...200) specified.

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

4.2.3 Data Types

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

The compiler processes the following data types:

- `LONG`: 32-bit integer values with the ranges:

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

- `FLOAT` until T10: Floating-point values (32 bit) with the ranges:

  $-3.402823 \cdot 10^{+38} \ldots -1.175494 \cdot 10^{-38}$ (negative values, 32 bit)

  $+1.175494 \cdot 10^{-38} \ldots +3.402823 \cdot 10^{+38}$ (positive values, 32 bit)

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

- `FLOAT` since T11: Floating-point values (40 bit) with the ranges:

  $-3.402823668 \cdot 10^{+38} \ldots -1.175494351 \cdot 10^{-38}$ (negative values, 40bit)

  $+1.175494351 \cdot 10^{-38} \ldots +3.402823669 \cdot 10^{+38}$ (positive values, 40 bit)

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

☞ Accuracy of 40 bit is solely restricted to:
  - Calculations inside the *ADwin* system.
  - Evaluation of constants by the compiler.

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

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

- `STRING`: ASCII character strings, in which each character is stored as a single array element (for details see chapter 4.3.5 on page 88). A single character corresponds to an integer 8-bit value in the range 0 … 255.

The obsolete data types **SHORT** and **INTEGER**–used with processors before T9–were replaced by data type **LONG**. For reasons of compatibility the compatibilität accepts these data types furthermore but automatically replaces them by **LONG**.

☞ When combining integer and floating-point values, a type conversion will occur. Under certain circumstances this may cause calculation results discrepancies from expected results. More about this is found in section "Type Conversion" on page 94.

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

4.2.4 Entering Numerical Values

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

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

1. Decimal notation:

`x = 930`	LONG
`x = 930.0`	FLOAT

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

2. Expontential notation:

`x = 93E1`	LONG
`x = 9.3E2`	FLOAT

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

3. Binary notation:

`x = 1110100010b`	LONG

4. Hexadecimal notation (an `h` is added):

`x = 3A2h`	LONG

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

4.2.5 Global Variables (Parameters)

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

The System Variables, also globally available, are described on page 82.

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

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

– `Par_1`, `Par_2`, …, `Par_80` with the **LONG** data type for 32-bit integer values.

– `FPar_1`, `FPar_2`, …, `FPar_80` with the **FLOAT** data type for floating-point values.

Example

```
Par_5 = 700               'Parameter 5 contains the value 700.
PAR_72 = ADC(1)           'The voltage at the analog input 1
                          'is measured and stored into
                          'parameter 72.
```

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

4.2.6 Global Arrays

The global arrays enable the exchange of data between the processes on the *ADwin* system or the computer (see also chapter 6.3.1 "Data Exchange between Processes"). Up to 200 arrays of the **LONG** or **FLOAT** data type are available.

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

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

`Data_1`, `Data_2`, …, `Data_200`.

Other array numbers are not allowed. However, the declaration of non-sequential array numbers is permissible, for instance `Data_5` without `Data_1` … `Data_4` is allowed. In your program the compiler differentiates the arrays by their numbers.

Example

```
REM Declare the array 5 with 20000 elements of the type LONG.
DIM Data_5[20000] AS LONG
REM Declare the array 3 with 7×5 elements of the type FLOAT.
DIM Data_3[7][5] AS FLOAT
```

There is more information about 2-dimensional arrays in chapter 4.3.3 on page 85.

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

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

Do *not* assign a value to the element 0 of an array, for instance with `Data_1[0] = ...`.

Examples

```
Rem The value of the 200th element from array 5 is assigned
Rem to the global integer variable PAR_1.
Par_1 = Data_5[200]

Rem In this program line the 345th element from the array
Rem DATA_5 gets the value 4000.
Data_5[345] = 4000

Rem This instruction assigns the value 300.1 to the 1st element
Rem of the 2 dimensional array DATA_3.
Data_3[1][1] = 300.1
```

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

```
'Here, too, as in the example above, the value 4000 is
'assigned to the 345th element of the array DATA_5.
number1 = 345
Data_5[number1] = 4000
```

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

```
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.

4.2.7 System Variables

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

PROZESSN_RUNNING

Returns the status of the process n (with n = 1…10): the process is running, just being stopped or already stopped (see page 224). The variable can only be read.

PROCESS_ERROR

Returns the number of the previous error of process n, if debug mode is active (with n = 1…16, see page 223). 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** you query and set this time, measured in clock cycles of the counter (see chapter 6.2.1 on page 115).

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

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

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

4.2.8 Local Variables and Arrays

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

Variable names can consist of any alphanumeric characters (a-z, A-Z, or 0-9) or an undersore ("_"). 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 either integer values (type **LONG**) or floating-point values (type **FLOAT**), and each are 32 bits long.

Example

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

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

Example

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

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

4.3 Variables and Arrays – Details

4.3.1 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**. With processor T11, an additional memory area is available via **AT EM_LOCAL**.

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

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

The fig. 13 shows examples of declarations, in order to store variables and arrays in the different memory areas.

Variable / Array	Memory Area	Source Code Declaration

Local Variable	Internal (DM)	`DIM var AS <VARTYPE>`
		or
		`DIM var AS … AT DM_LOCAL`

	Addit. (EM)	`DIM var AS … AT EM_LOCAL`
	External (DX)	`DIM var AS … AT DRAM_EXTERN`
Array	Internal (DM)	`DIM array[5] AS … AT DM_LOCAL`
	Addit. (EM)	`DIM array[5] AS … AT EM_LOCAL`

(global/ local)	External (DX)	`DIM array[5] AS …`
		or
		`DIM array[5] AS … AT DRAM_EXTERN`

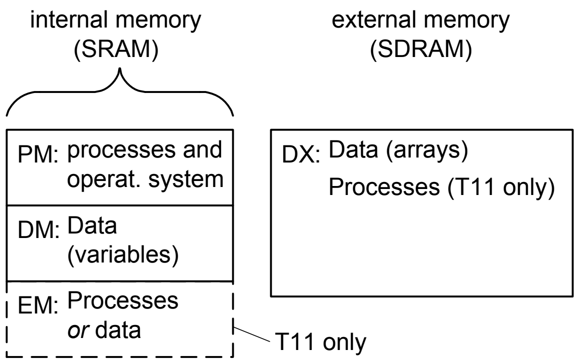
Fig. 13 – Allocation of the Memory Area with Declarations

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

4.3.2 Memory Areas

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

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



- Program memory (PM):
  Program memory occupies half of the internal SRAM and contains the operating system and processes.
- Internal data memory (DM)
  The internal data memory occupies half of the internal SRAM for storing the global and local variables.
- Additional memory (EM)

  Additional internal memory EM is available with processor T11 only. Additional memory can be used as data memory or program memory.
- External data memory (DX)
  The external data memory covers the external SDRAM and stores the global and local arrays.

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

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

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 available memory of PM, DM, EM and DX, is displayed in bytes.

4.3.3 2-dimensional Arrays

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

2-dimensional notation may simplify a problem's solution (compared to 1-dimensional arrays). At the same time it will slow down data access and require additional program memory.
The loss of access speed and the need of additional memory will increase with each access to the 2-dimensional arrays by the program.

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

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

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

– Inside of a library module (**LIB_SUB**, **LIB_FUNCTION**) which receives a 2D-array as an argument.

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

```
DIM Data_1[3][2] AS FLOAT
```

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

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

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

```
REM use in main program
Data_1[3][1] = 17
setpar1(Data_1)          'sets PAR_1 = 17

REM use in library module
LIB_SUB setpar1(BYREF array[] AS LONG)
  Par_1 = array[5]       'corresponds to DATA_1[3][1]
LIB_ENDSUB
```

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

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

$$\text{DATA\_n[i][j]} \hat{=} \text{DATA\_n}[s \cdot (i-1) + j]$$

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

4.3.4 The Data Structure FIFO

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

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

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

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

Example

```
DIM Data_5[1003] AS LONG AS FIFO
```

This instruction declares the global array with the number 5 as FIFO ring buffer with 1003 elements of the type **LONG**. Please note the special size of a FIFO with the T11 processor (see FIFO).

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

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

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

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

Example

```
DIM Data_5[1003] AS LONG AS FIFO
Data_5 = 95                'Writes the value 95 into the
                           'DATA_5 array which is declared as FIFO
Par_7 = Data_5             'Reads a value from the FIFO and
                           'stores it in the global variable
                           'PAR_7
```

To ensure that the FIFO is not full, the **FIFO_EMPTY** function should be used before writing into it. Similarly, the **FIFO_FULL** function should be used to check if there are values which have not yet been read, before reading from the FIFO.

Example

```
DIM free,used,value1 AS LONG
DIM Data_1[1003] AS LONG AS FIFO
REM Are there still elements which are not empty?
free = FIFO_EMPTY(1)
IF (free > 0) THEN
  Data_1 = value1
ENDIF
REM Are there still elements, which haven't been read?
used = FIFO_FULL(1)
IF (used > 0) THEN
  Par_7 = Data_1
ENDIF
```

4.3.5 Strings

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

The following instructions are available for string processing:

ASC	Get ASCII number of a character
CHR	Get character from an ASCII number
FLOTOSTR	Convert a float value into a string
LNGTOSTR	Convert a long value into a string
STRCOMP	Compare 2 strings to be equal
STRLEFT	Get leftbound substring from a string
STRLEN	Get length of a string

`STRMID`	Get substring from a string
`STRRIGHT`	Get rightbound substring from a string
`VALF`	Convert a string into a float value
`VALI`	Convert a string into a long value
+ String Addition	Operator to concatenate strings

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

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

```
IMPORT String.LI9
DIM text[5] AS STRING
```

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

`text[1]`	Length of the string in characters (5)
`text[2]`	Character 1 of the string
`text[3]`	Character 2 of the string
`text[4]`	Character 3 of the string
`text[5]`	Character 4 of the string
`text[6]`	Character 5 of the string
`text[7]`	The end of string character, terminating zero (`00h`)

Each element requires 4 bytes of memory. The first and last elements of the string are automatically reserved by the *ADbasic* compiler. Do not use element number 0, here `text`[0].

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

Normal Assignment

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

Example

```
text = "HELLO"
```

Element Index	Memory Contents	Meaning
`text[1]`	`05h`	Length of the string in characters (5)
`text[2]`	`48h`	ASCII value for "`H`"
`text[3]`	`45h`	ASCII value for "`E`"
`text[4]`	`4Ch`	ASCII value for "`L`"
`text[5]`	`4Ch`	ASCII value for "`L`"
`text[6]`	`4Fh`	ASCII value for "`O`"
`text[7]`	`00h`	End-of-string character

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

– single quote ('): `\x27`

– double quote ("): `\x22`

– backslash (\): `\x5C`

Character Assignment via Escape Sequence

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

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

Example

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

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

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

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

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

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

Example

```
text = "HE\x4C\x4CO"
```

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

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

String Assignments that are NOT Recommended

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

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

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

4.4 Expressions

4.4.1 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. 13).

Please note, that automatic Type Conversion may in some cases influence the evaluation of an expression (see page 94), too.

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

Fig. 14 – 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
```

4.4.2 Type Conversion

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

Do not mix up this conversion with the instructions Cast_FloatToLong or Cast_LongToFloat, which do quite a different job (see page 144).

Consider the following special features:

– Cut off decimal places

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

– Converting *all* Integers to Floats

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

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

Example

```
Par_1 = 2 / 4 * 3        'Result: PAR_1=0, because 2/4 = 0
```

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

But:

```
FPar_1 = 2 / 4 * 3       'Result: FPAR_1=1.5
Par_1 = 2 / 4.0 * 3      'Result: PAR_1=1 (cut off!)
```

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

– Prevent integers from Conversion

Even using parentheses does not prevent the automatical conversion into **FLOAT**. To absolutely make calculations in **LONG**, an individual program line must be used.

Example

```
Par_1 = 2
Par_2 = 5
'here a conversion is made:
FPar_3 = (Par_2 / Par_1) + 0.2'FPAR_3 = 2.7
'but not here:
Par_9 = Par_2 / Par_1   'PAR_9 = 2 (cut off)
FPar_4 = Par_9 + 0.2    'Result: FPAR_1 = 2.2
```

– Conversion of Arguments

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

- Each individual parameter for an instruction.
  Additionally a cut off may occur according to the parameter's data type (data type see instruction's description).
- Each argument passed to a function or subroutine.
- Each individual part of a conditional test within a Boolean expression in an **IF**…**THEN** or **DO**…**UNTIL** even if there are multiple tests linked with **AND** or **OR** .

Example

```
Par_1 = 2
FPar_2 = 5.5

'Both conditions are true,PAR_1 is not converted into
'FLOAT, therefore PAR_3 = 1.
IF ((Par_1 / 4 * 3 = 0) AND (FPar_2 * 1.1 > 5.5)) THEN
  Par_3 = 1
ENDIF

'The condition with FLOAT does not influence the
'LONG calculation, therefore PAR_3 = 0.
IF (FPar_2 * 1.1 > 5.5) THEN Par_3 = Par_1 / 4 * 3
```

4.5 Selection structures, Loops and Modules

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

- Control structures to help shorten large sections.
  - Loops for sections being frequently repeated:
    DO … **UNTIL** or
    FOR … **NEXT**.
  - Structures for case-by-case decisions:
    IF … **ENDIF** or
    SELECTCASE … **ENDSELECT**.
- Subroutine and Function Macros to define frequently used program sections as
  - Subroutine macros with **SUB** … **ENDSUB**
  - Function macros with **FUNCTION** … **ENDFUNCTION**
- Libraries of compiled subroutines and functions, which can be included into a user's source code with **IMPORT**:
  - Library subroutines with **LIB_SUB** … **LIB_ENDSUB**
  - Library functions with **LIB_FUNCTION** … **LIB_ENDFUNCTION**
- Collections of source code sections and program modules in Include-Files, which can be included into a user's source code using
  #INCLUDE `filename.Inc`

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

4.5.1 Subroutine and Function Macros

The syntax of subroutine and function macros is simple, only requiring the terms **SUB** … **ENDSUB** and **FUNCTION** … **ENDFUNCTION** around the relevant program sections, like parentheses. Contrary to subroutines, functions return a value.

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

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

You will find more information about the structure of macro modules in the instruction reference (page 176: **FUNCTION** … **ENDFUNCTION**; page 251: **SUB** … **ENDSUB**).

4.5.2 Include-Files

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

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

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

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

Include files installed with *ADbasic* contain only subroutine and function macros, defining instructions for hardware access. Thus, the appropriate position for these files to be included is the beginning of the source code (see page 73).

4.5.3 Libraries

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

The library modules are similar to the subroutine and function macros. They are created in a source code file using the **LIB_SUB** … **LIB_ENDSUB** and/or **LIB_FUNCTION** … **LIB_ENDFUNCTION** instructions. The library file is then compiled using the `Build / Make lib file` menu option.

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 4.5.1 "Subroutine and Function Macros", page 96).

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. 15), otherwise an error message will be returned by the compiler.

Recursive calls of library functions or subroutines are not allowed.

You will find more information about the structure of the library modules in the instruction reference (page 191: Lib_Function … Lib_EndFunction; page 195: Lib_Sub … Lib_EndSub).

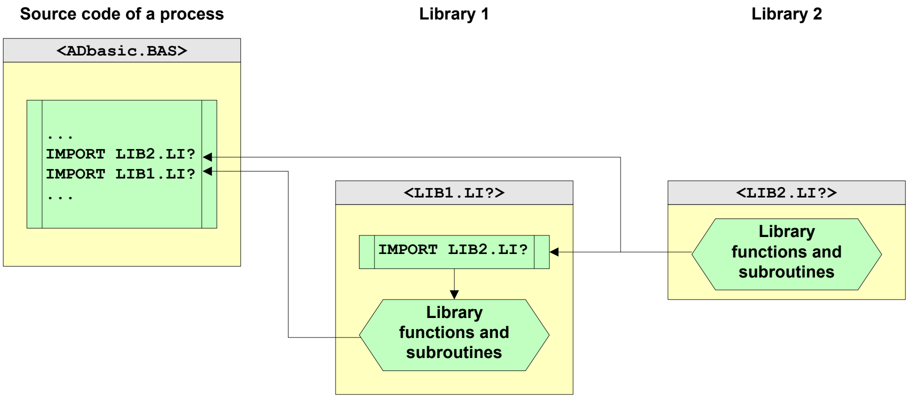


Fig. 15 – Interlaced Libraries

5 Optimizing Processes

The *ADwin* system is designed to quickly and precisely execute control and measurement tasks. Depending on the requirements it may be necessary to optimize your *ADbasic* program for a faster processing time.

The following pages illustrate steps for optimizing a program. Many factors determine the optimization process which needs to be considered with each individual case. Please refer to the "*ADbasic* Tutorial and Programming Examples" manual to find more examples for optimizing processes.

5.1 Measuring the Processing Time

For optimization it is important to measure the processing time of a process cycle or of a program section. This can be done using the internal counters of the *ADwin* system.

The processor of the *ADwin* system has two internal counters, one for high-priority processes and another for low-priority processes, each incrementing in different clock rates (see fig. 17 on page 115). The current counter value can be read using the **READ_TIMER** instruction; the counter corresponding to the running process's priority will automatically be read out.

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

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           'do NOT use float here

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 of the high-priority process requires 37 × 25 ns = 925 ns.

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

Example

```
DIM oldtime, time AS LONG

INIT:
 oldtime = READ_TIMER()

EVENT:
 time = READ_TIMER()
 Par_1 = time - oldtime
 oldtime = time
```

5.2 Useful Information

5.2.1 Accessing Hardware Addresses

Many of the *ADwin* system functions are managed by its control and data registers. These functions can quickly be executed by *directly* accessing the relevant registers with the **PEEK** and **POKE** instructions. Here, "directly" means that the functions' addresses are not calculated in the process cycle, but passed as constant values: saving computing time for the calculation.

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

5.2.2 Constants instead of Variables

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

Example

```
FPAR_1 = SQRT(PAR_2)    'with PAR_2=17
FPAR_1 = SQRT(17)
```

For the first calculation the value of the variable Par_2 must be determined during run-time. The root must then be calculated and assigned to Par_1.

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

5.2.3 Faster Measurement Function

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

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

For instance, the **ADC** instruction does not consider that the *ADwin-Gold*-system has two ADCs, which are able to convert two different channels at the same time. This is illustrated in the following example:

Example

```
REM Example for Gold
REM Set both multiplexers of the ADC to the channel 1
SET_MUX(000000b)
Rem wait for settling time
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
```

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

5.2.4 Setting Waiting Times Exactly

Using a waiting time, you can easily set an exact offset between 2 instructions, for example to bridge the multiplexer settling time between **SET_MUX** and **START_CONV**.

The instruction for setting the waiting time depends on the processor type:

– Processors T9 and T10:

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

Waiting for the multiplexer settling time of 14 µs on a Pro I module would then work like this:

```
SET_MUX(2,00000b)       'Set Mux to channel 1
REM Here a calculation may be done, which e.g. takes
REM 8µs of the free processor time.
SLEEP(60)               'wait remaining 6µs until 14µs
START_CONV(2)           'Start conversion
```

– Processor T11:

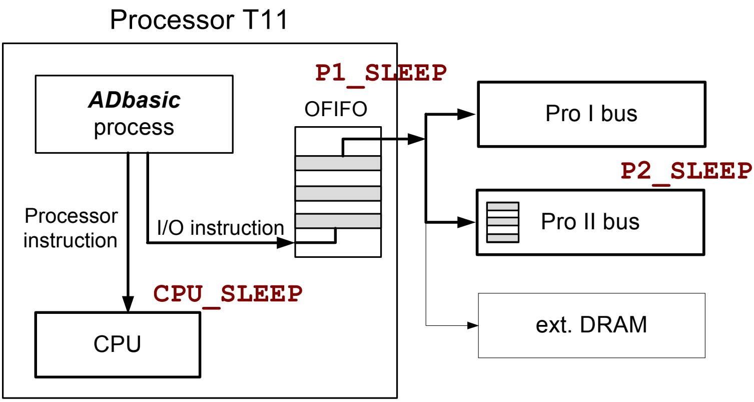
There are 3 possible instructions for the waiting time:

- **P1_SLEEP** makes the Pro I bus wait, but also Pro II bus and external DRAM.
- **P2_SLEEP** makes the Pro II bus wait.
- **CPU_SLEEP** makes the processor wait (refers to **SLEEP**).

If the waiting time gaps a delay between I/O-instructions for Pro I modules, **P1_SLEEP** is the right choice; for Pro II modules it is **P2_SLEEP**. The instruction **CPU_SLEEP** makes sense only rarely.

Waiting for the multiplexer settling time of 14 µs on a Pro I module would then work like this:

```
SET_MUX(2,00000b)       'Set Mux to channel 1
P1_SLEEP(1400)          'Make Pro I bus wait 14µs.
                        'Note the time unit.
START_CONV(2)           'Start conversion
REM The calculation follows but now; the T11 processor will
REM process it automatically in parallel with the I/O
REM instructions.
REM Attention: Within the calculation you should use variables
REM from internal memory only. Otherwise the calculation may
REM anyhow not be run until the I/O instructions are completely
REM processed.
```



Why are there different instructions for the waiting time? The processor T11 runs processor instructions and I/O instructions[1] quasi-parallel (see sketch above). This is very fast, and also leads to parallel and thus separate timing, resulting in 3 instructions for the waiting time.

The quasi-parallel processing is enabled via a 5-level buffer `OFIFO`: The operating system passes an I/O instruction into the `OFIFO` (if there is enough space) and immediately starts processing the next instruction. The example above passes the instructions **SET_MUX**, **P1_SLEEP** and **START_CONV** into the `OFIFO`; the subsequent calculation is then run in the CPU, while e.g. the Pro I bus[2] is still waiting.

☜ Please note: A calculation, that is to be processed in parallel in the CPU, may only use variables from internal memory. The operating system regards each access to the external DRAM, the common memory area for arrays, as an I/O instruction that has to walk through the `OFIFO` buffer.

5.2.5 Using Waiting Times

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

---

1. I/O instructions are those, which access external devices via the `OFIFO` buffer. External devices (as regards the CPU) are modules on the Pro I or Pro II bus and the external memory DX.
2. More precisely, the instruction **P1_SLEEP** makes the buffer `OFIFO` wait, not the Pro I bus.

The **SET_MUX** and **START_CONV** instructions require waiting time for the settling of the multiplexer and the conversion of the ADCs. During this waiting time, the processor is not busy and could be used for other tasks.

More detailed information about the required waiting times for data conversion can be found in your hardware manual.

The next example is an extension of the previous example, showing how two measurements are executed across two separate ADCs. Compared to the **ADC** instruction, this enables execution of 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 5.2.4 "Setting Waiting Times Exactly".

5.2.6 Optimization with Processor T11

This section describes how to use the specific features of the T11 processor to speed up a process, especially by optimized memory access.

If nonetheless you reach the processor's limits, further optimizations are possible, but only in connection with your specific application. Please contact our support (see address inside the manual's cover page).

Using internal memory

For time-critical sequences, use variables and arrays in the internal memory (EM or DM) as possible. While variables are declared automatically in the internal memory, arrays (both local and global) have to declared as follows:

```
DIM DataLocal[100] AS LONG AT DM_LOCAL
DIM Data_5[2000] AS FLOAT AT DM_LOCAL
```

Compared to internal memory the access of processor T11 to external memory slows down for 2 reasons. On the one hand the memory access is passed into the `OFIFO` buffer (see page 103) as I/O instruction, which can cause delays. On the other hand the administration of external memory is slower than of the internal memory.

Accessing the external memory

For the access to the external memory try to use – as fas as possible in the program – data blocks, and don't access single values. If using block-wise data transfer the processor enables an accelerated access, so e.g. transferring a block of 20 values quicker than 3 single values.

As an example, the block data transfer is quite useful, if a lot of measurement values are read in short time: At first the collected data packet is saved in quick internal memory. As soon as the measuring task reaches a non-critical stadium, the data are transferred as block into external memory using the instruction **MEMCPY**, leaving the internal memory ready for the next collected data packet.

5.3 Debugging and Analysis

Debug, timing, and trace modes are *ADbasic*'s hands-on tools for debugging and program analysis. All modes are activated via the "`Debug`" menu (see page 29) and add their helping features to those programs, which are compiled with active mode.

☞ Please note: Activating of the modes produces additional program code. Thus the program will need a longer processing time as well as additional memory – at times at considerable rate. We therefore recommend that you use these tools for developing and testing of programs only.

5.3.1 Finding Run-time Errors (Debug Mode)

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

- Division by zero
- Square root from a negative value
- Access to too large / too small element numbers of an array

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

The option "`Debug mode`" is activated from the "`Debug`" menu; do then compile the source code to be checked. On occurrence of a run-time error it is automatically displayed in the "`Debug Errors`" windows. As well, the run-time error is being corrected to maintain a stable mode of operation.

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

Details about activating and display of run-time errors are shown in section "Debug mode Option" on page 52.

5.3.2 Check the Timing Characteristics (Timing Mode)

The *ADwin* system is designed in such a manner that an arriving event signal for a high-priority process (externally generated or by an internal counter) immediatley starts the relevant process cycle. Processes with such "good" timing characteristics are deterministic and execute their tasks exactly at a predetermined period of time.

To check timing characteristics of processes requires some effort, especially when changes are to be made later, to obtain good timing characteristics. This effort is worth its price, when required higher frequencies or additional tasks put the processor workload to its limit. Another example are process cycles not start as exactly as predetermined according to the measurement task.

In the timing mode, information is generated, which can be used to check selected high-priority processes if they have "good" timing characteristics. For these processes 7 parameters are calculated, which are displayed in the Timing Analyzer Window.

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

1. An event signal does not start a process cycle immediately, but a certain (not exactly defined) time later.
2. An event signal does not start a process cycle at all, but gets "lost". Even several lost event-signals are possible.

In the first case the operating system tries to make up the delay by using available idle times in the workload of the processor, until all process cycles again start at the pre-defined period of time. In the latter case the operating system cannot make up the delay: Event signals and therefore process cycles are really lost (see chapter 6.2.5 "Different Operating Modes in the Operating System").

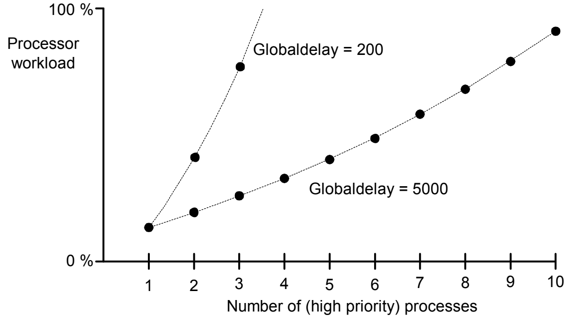
An optimal timing characteristic, especially of the high priority processes, is obtained in 2 steps by:

1. Checking Number and Priority of Processes
2. Optimal Timing Characteristics of Processes (Use Timing Mode)

Checking Number and Priority of Processes

In a high-priority process only time-critical tasks should be processed, all other tasks in one or more low-priority processes (or even processed on the PC).

If possible use only one single high-priority process. Several processes can very often be merged to a single process; if the Processdelay is identical, we highly recommend this. It's worth the effort – especially with a shorter Processdelay of the processes – because the processor workload will be essentially lower even if the the same tasks are executed. The graphic below illustrates this more clearly:



With several high-priority, time-controlled processes, process cycles cannot be prevented from starting time-delayed (except their Processdelays are integer multiples of each other).

Optimal Timing Characteristics of Processes

A high-priority process has an optimal timing characteristic under the following conditions:

- All process cycles of the process have an almost equal processing time.
- The processing time of the process cycle is as short as possible.
- The Processdelay of the process is longer than the longest processing time of all process cycles.

Nevertheless, the processor workload for high-priority processes must leave enough processor time available for the tasks of low-priority and communication processes.

To get more information about the timing characteristics of interesting processes proceed as follows:

1. Activate the timing option with `Debug` ▸ `Enable timing analyzer`.
2. Compile (and start) the *ADbasic* source code.

   For each source code which you compile with active timing option, information about timing characteristics are generated automatically. We recommend to view only a small number of processes at once, so that the timing characteristics will not be influenced too much (see below).

3. Disable the `Debug` ▶ `Enable timing analyzer` option again, so that other processes being compiled do not unnecessarily generate timing information.
4. Open the `Timing Information` window via the `Debug` ▶ `Show timing information` menu item.

Note that the timing characteristics on the *ADwin* system depend on the number and type of the processes, thus causing accordingly different parameters. One reason for this fact is the process management of the operating system (see chapter 6.2.5 "Different Operating Modes in the Operating System").

The evaluation of the information is made during run-time and needs approx. 60 clock cycles additionally (when using a T9, T10 or T11 processor) per process cycle and process. The parameters in the window are continuously updated and refer to the time passed since the last start of the processes. A short description of the parameters can be found under the Show timing information Menu Item, page 48.

The (minor) change of timing characteristics by the timing mode itself cannot be avoided and exists even if no parameters are displayed. This may result under certain circumstances in further latencies, and is also reproduced in the corresponding parameters; in short processes with a short Processdelay, a processor workload of more than 100% can be reached sometimes, so that the communication to the PC is interrupted.

Please note that during compiling high-priority processes using the timing option, a low-priority process can be considerably delayed.

6 Processes in the ADwin System

An *ADwin* system has the capability to control complex test stands while rapidly executing measurements. Programs using one or more *ADbasic* processes are used to provide this capability. Within these processes you can specify how analog and digital data is processed within the *ADwin* system and how it is exchanged with external devices and PC.

After starting the process the program[1] in the *ADwin* system is (characteristically) restarted and processed in regular time intervals. This calling of a process cycle is triggered by one of the following start signals, called events:

1. Timer event: A pulse of the internal counter. You determine for each process separately in which time interval (processdelay) a new event is triggered.
2. External event: An external signal, which arrives at the event input of the *ADwin* system. This could be for instance the pulse of an incremental encoder.

Only one of the 10 possible processes can be controlled by an external event, all other processes have to run time-controlled.

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

– The initialization in the sections **`LOWINIT:`** and/or **`INIT:`**.

– The actual function of the process cycle in the central **`EVENT:`** section (event loop).

– The final processing in the **`FINISH:`** section.

It is possible to control the processes from the computer, that is the processes are started, stopped or their processdelays changed. You can do this with *ADbasic* as well as with other development environments such as C++ or Visual Basic.

With the bootloader option, it is also possible to have processes start automatically on power-up of the *ADwin* hardware. For programming the bootloader, see manual "*ADwin* bootloader".

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

6.1 Process Management

6.1.1 Types of Processes

Within the *ADwin* system several processes can run simultaneously. The operating system is responsible for calling the process cycles according to specified rules, and for their being processed by the CPU without blocking each other.

When referring to a "process" in this manual, we mean one of the processes 1...10, that you have programmed.

You assign a priority to each process and thus determine the interaction and timing of the processes. There are the priorities:

– Processes with High-Priority

– Processes with Low-Priority

  Low-priority processes are further divided into the levels -10 (low) up to +10 (high).

The process priority is set via the menu `Options \ Process Options`.

Process	Function	Priority[a]
1…10	User-defined processes with functions and priorities you can freely define	low level *n* / high
11, 12	Predefined input / output processes	high
15	Process for controlling the flashing LED in *ADwin-Pro* and *ADwin-Gold* systems	low, level 1
Communication	Communication between the *ADwin* system and the computer: Instruction and data exchange	medium

a. The meaning of the priorities is described in the following sections

Fig. 16 – Overview of all processes

The standard processes, processes 11 and 12, are only necessary when using the drivers for the Labview and Testpoint environments. These processes can be loaded during the boot process along with the operating system, either from a developer environment (for more details, see the *ADwin* developer manual), or from *ADbasic*. To do this, set the option `Load Standard processes` to `Yes` in the *ADbasic* menu `Options` / `Compiler`.

If you are not using one of these applications you can stop the transfer of the standard processes during booting (setting `No`).

The communication process (see page 113) is part of the operating system. It receives commands of the computer and exchanges data between the *ADwin* system and computer only when the computer requests them.

⚠ If you transfer more than one process with the same process number to the system, only the last process transferred is executed, because the earlier transferred processes are overwritten.

6.1.2 Processes with High-Priority

Processes with "high" priority get preferential treatment from the operating system:

– The maximum latency from when a high priority process is called by an event to when execution of the process begins is 300 ns.

– A high-priority process cycle cannot be interrupted and is always completely processed. During this time all process cycles with low-priority are blocked.

  Neither another high-priority process cycle nor a stop instruction can interrupt a running, high-priority process cycle. In both cases the system will complete the current high priority process cycle before proceeding.

In time-controlled high-priority processes the cycle time (processdelay) can be set in intervals of 25 ns.

⚠ The software should be written so that time-critical measurement processes run with high-priority and all others run with low-priority, so that the processor can process the time-critical process cycles without any interference from other operations.

⚠ The sections **`LOWINIT:`** and **`FINISH:`** of a process – if there are any – are always executed with low-priority, priority level 1, even if the process is set to run with high-priority.

6.1.3 Processes with Low-Priority

Process cycles with low-priority are immediately interrupted when a process cycle with a higher priority is called and will stay interrupted until that higher priority process cycle has finished.

Low-priority processes are further divided into the priority levels -10 (low) up to +10 (high). Process cycles with a low level can be interrupted by those with a higher level at any time. The processor T11 keeps strictly to the priority levels for process management (see chapter 6.2.3 on page 117).

Low-priority processes of the same priority level participate in time slicing. Here the operating system apportions the computing time to the process cycles alternating and in equal time slices. One time slice takes 2 ms (processor T9) or 1 ms (processors T10, T11) on average.

Low-priority processes must always be time-controlled. The cycle time (processdelay) can be set in discrete intervals; interval size see fig. 17 on page 115.

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

6.1.4 Communication Process

The communication process has a priority level between the priorities "high" and "low". Therefore it can interrupt low-priority process cycles any time and can be interrupted by high-priority process cycles.

If the computer requests information from the *ADwin* system, the communication process must respond within 250 ms or a time-out will occur, the communication between the computer and the *ADwin* system may be interrupted. In this case the message `The ADwin system does not respond` will be displayed and the system will have to be reinitialized by rebooting the *ADwin* system. The time-out is independent of the communications interface, either USB or Ethernet.

The cause of an interruption in the communication is that the communication process does not have enough processor time allocated to it. This can be caused by the following facts:

– the processdelay of the high-priority processes is too short or

– the processing time of a high-priority process cycle is too long.

More about this subject can be found in chapter 6.3.2 on page 121.

6.1.5 Memory fragmentation

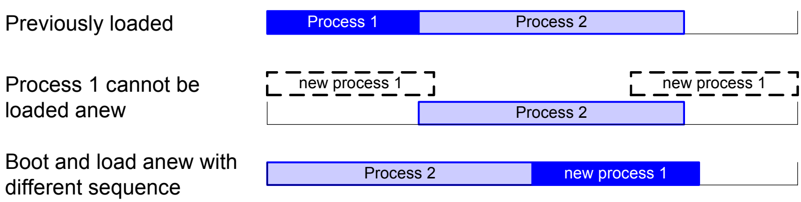
The operating system of the *ADwin* system cares for storing processes, arrays and variables at an adequate memory position and using them correctly. Therefore the user normally has no problems with memory management which thus would need no explanation.

Under certain circumstances, the error message „`Not enough memory or memory access error. Please reboot the ADwin system.`“[1] occurs. Often, the reason is an external memory fragmentation, which arises from processes or data arrays being loaded into memory multiple times and with increasing size; a typical action e.g. for the development of new processes. A simple solution is to boot the *ADwin* system and load the data anew.

A memory fragmentation is defined as free storage being dispersed between allocated regions. If now a new data block like a process or an array is loaded into memory–where it can only be stored as complete unit–it may happen, that the data block does not fit into any of the free memory fragments. You receive the above error message and have to reorganize the memory in order to obtain free memory of sufficient size.
Booting and loading anew is useful here, since the data blocks are stored consequently without a gap and the free memory remains as a unit.

The result of memory fragmentation can be a memory which cannot store any more data–regardless of being a process or a data array. According to the processor type an error message pops up or the *ADwin* system shows 100% workload. A simple solution is to boot the *ADwin* system and load the data anew.

Example: Two (quite large) processes are already loaded to memory. Process 1 is to be replaced by new code with increased size, but the data block does not fit into memory neither before nor behind process 2 and you receive the mentioned error message. After booting and loading in different sequence, process 1 can be loaded any time without the risk of memory fragmentation.



As an alternative, you may also delete process 2 manually and load both processes anew. The advantage is to retain the values of global variables and arrays; for a global array this is only true, if the array size remains unchanged. The difficulty in manual deletion, especially with increasing number of processes, is to keep the overview of the order in which processes are stored in memory.

1. With processors T9 and T10 there is no error message, but the *ADwin* system has a workload of 100%.

Alike with process memory, memory fragmentation may also occur in data memory multiple dimensioning of data arrays with changing size, e.g. during development of a process. If so, loading the process will release the allocated memory of the (newly dimensioned) arrays and for each array a new memory range has to be found, leading to memory fragmentation. The simple solution is to boot the *ADwin* system and load the data anew, too.

Generally, global arrays may be deleted individually in *ADbasic* using `Clear Data` (see chapter 3.7.7 on page 54), in order to obtain free memory of sufficient size. But if a fragmentation occurs, most times you don't know the order in which arrays are stored in data memory, so booting is normally to be preferred.

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

6.2 Time Characteristics of Processes

6.2.1 Processdelay

The time interval, in which time-controlled process cycles are called by the counter, which is the cycle time of the event section of the process. It is usually measured in clock cycles of the system clock and called *Processdelay*, (in earlier *ADbasic* versions: Globaldelay). The processdelay of each process is specified by setting the value of the system variable **PROCESSDELAY**.

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

Processor	Priority	
	High	Low
T9	25 ns	100 µs
T10	25 ns	50 µs
T11	$3.\overline{3}$ ns	$3.\overline{3}$ ns = $0.00\overline{3}$ µs

Fig. 17 – The time resolution of the system clock (units of the processdelay)

For instance, a processdelay with the value 1000 means that for a high-priority process on a processor T9 it is called in time intervals of 1000 × 25 ns = 25000 ns = 25 µs, while for a low-priority process in a time interval of

$1000 \times 100\,\mu s = 100000\,\mu s = 100\,ms$. You can specify this event interval in the program line:

```
PROCESSDELAY = 1000
```

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

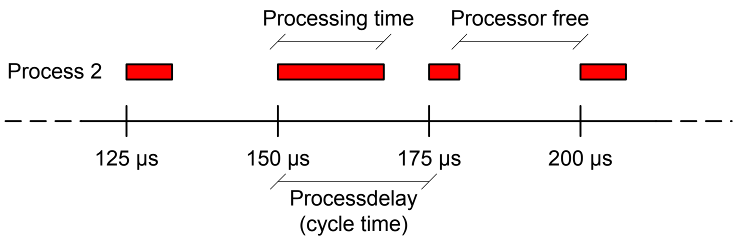


Fig. 18 – Processdelay and processing time in high-priority process cycles

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.

6.2.2 Precise Timing of Process Cycles

If you have (as shown in fig. 18) only one high-priority process, it will be called and processed exactly in its time schedule.

Make sure that the processing time of a high-priority process cycle never exceeds its cycle time (in the example below: 25 µs). This process cycle cannot be interrupted, thus other process cycles can only be partially processed or not at all, for instance the important communication process.

If there are several high-priority processes, the actually running process cycle can influence the time schedule of the remaining process cycles. In fig. 19 for instance, process 1 has to start with a delay when the processing of the active process 2 has finished.

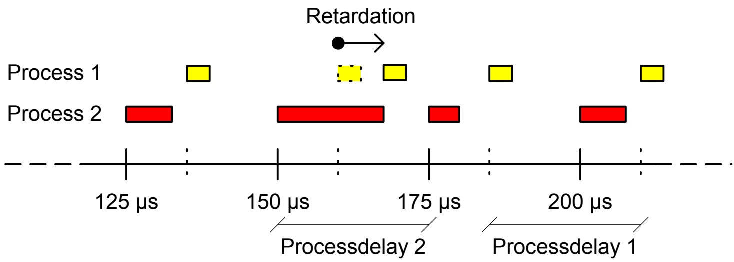


Fig. 19 – Delay of a high-priority process cycle

Keep the execution time of high-priority process cycles as short as possible. Have event loops, which require long processing time, or calculations whose result cannot be immediately be processed, always run in process cycles with low-priority.

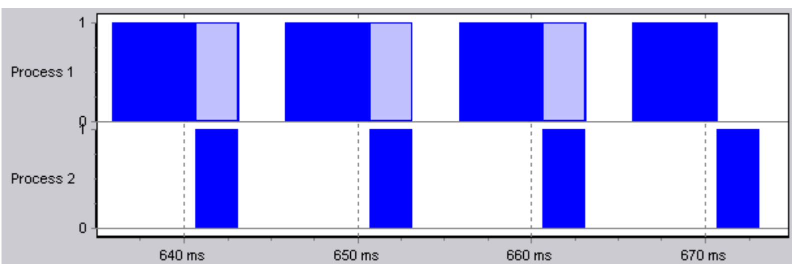
A low-priority process depends on the time characteristics of all other process cycles with the same or higher priority. Each interruption minimizes the time, a low-priority process cycle can use the computing power, and in the worst case it will not be called at all.

6.2.3 Low-Priority Processes with T11

The processor T11 manages low-priority processes strictly be their priority level. In contrast, priority levels are of little importance with T9 or T10. Nevertheless, communication process and high-priority processes still take precedence over all low-priority processes.

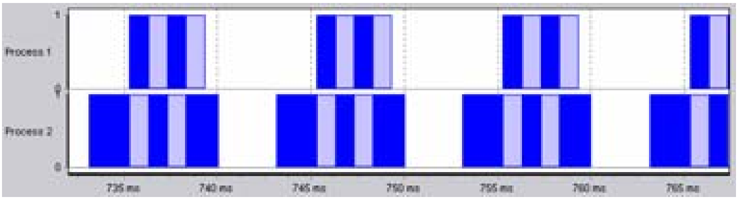
The process management of low-priority processes is different for:

– Processes of different priority levels: All processes of lower priority level are interrupted, as soon as and as long as a process of higher priority level is processed.



In this case, process 2 is of higher priority level and therefore interrupts process 1 several times.

– Processes of equal priority levels: The processes take part in time slicing, that is, within the priority level, the operating system portions out the processor's operating time to the process cycles alternating and in equal time slices (1 ms).



The example shows the changeover of the processes quite clearly. Please note the rule, that a process - process 1 in this example - immediately receives a time slice upon the call of its process cycle.

There is a rare and special case which annuls time slicing: A process receives a lot of processing time, if both it is frequently called and its process cycle takes shorter than one time slice. With each call the process interrupts other processes of the same priority level and thus "steals" their processing time.

6.2.4 Workload of the *ADwin* system

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

You can monitor the workload of the processor in the status line display `Busy` within the development environment (see chapter 3.8.5). This value gives you an indication if the processor still has enough computing time available to complete all of the required activities.

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

Please note for processor T11: Although a workload below 90% is displayed, an overload can exist, so that some process cycles might be processed with delay. In this case, the overload exists on the internal Pro I or Pro II bus, not int the processor, and can therefore not be displayed.

6.2.5 Different Operating Modes in the Operating System

The operating system differentiates between 2 operating modes for the timing characteristics in high-priority processes, depending on the fact if several time-controlled (high-priority) processes are active or only one.

If an additional externally controlled process is running, is of no importance here. The externally controlled process is managed separately by the operating system and can therefore be seen as a third operating mode.

Single Time-Controlled Process

With a single time-controlled process the operating system uses hardware components to process the event signals of the internal counter. In this case the operating system processes an incoming event signal very quickly.

The hardware components can buffer if an event signal has arrived, but not how many event signals have arrived. If an event signal has arrived, the operating system activates the next process cycle at the fixed period in time (Processdelay see chapter 6.2.1), unless a high-priority process cycle is just being processed. In this case the operating system activates the next process cycle immediately after the currently running process cycle.

If a number of event signals arrive during a high-priority process cycle, only one single process cycle is called and not the number of arrived process cycles, respectively. As a consequence all but one of those event signals are lost. Therefore we recommend the process cycles absolutely be shorter than the cycle time (Processdelay) of the process.

Several Time-Controlled Processes

With several time-controlled processes, the operating system itself manages arriving event signals. This operating mode is working slower due to this management efforts, but the number of all arriving event signals are buffered for each process. Thus it is ensured, that for each event signal a process cycle is started, even if this happens later than the pre-defined instant of time.

Frequently the time schedules for starting the process cycles are the reason for the fact that event signals continuously occur during the processing of another process cycle. With other words, the Processdelay values are not integer multiples of each other. We recommend that only few processes are used; it is often possible to merge several processes to one single process (this results in a smaller processor workload, too).

Always keep in mind that the processor workload depends very much on the number of processes running. Thus a task performed by 2 (or even more) processes will always take more workload than the same task within a single process. This is the more of importance the shorter a Processdelay is (see also chapter 5.3.2 on page 106).
Example: Processes 1 and 2 with a very short Processdelay running as a single process each generate 10% workload; both processes together have a workload of 55%.

Externally Controlled Process

The operating mode for the externally controlled processes is, independent of time-controlled processes, always the same. The operating system manages the external process as a single time-controlled process (see above), that is, arriving event-signals are processed very quickly, but event signals can also be lost.

An external event signal is a rather important information–in particular, because it cannot be predefined by the *ADwin* system–and must not get lost (finding lost events, see page 48). Therefore note to have short process cycles in this process (in the section `EVENT:`).

6.3 Communication

6.3.1 Data Exchange between Processes

Data can be exchanged between different processes via global variables (`Par_n`, `FPar_n`) or global arrays (`Data_n`). Data can be exchanged with programs running on the PC using these variables and arrays as well.

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

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

Example

Process 1 is a function generator and Process 2 is a controller. The function generator regularly writes the generated value into the global variable `Par_10`. At every event loop the controller process reads out the global variable `Par_10` and uses its contents as 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.

6.3.2 Communication between PC and *ADwin* System

From PC applications and development environments, you can control the processes on the *ADwin* system, as well as request data from or send data to the system. An *ADwin* system cannot communicate with the computer on its own, but instead responds to requests coming from the computer.

All data exchange is made via global variables (`Par_n`, `FPar_n`) or global arrays (`Data_n`). This refers also to the Data Exchange between Processes (see above).

The communication to the *ADwin* system is managed under Windows with the `ADwin32.dll` (dynamic-link library). In the *ADwin* system the communication process is responsible for this task (page 113).

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

The `ADwin32.dll` has the following tasks:

- Communication with the connected *ADwin* system via the specified communication interface: USB, Ethernet (TCP/IP).
- Recognizing and handling of communication errors.
- Blocking several computer applications if they want to access the same system at the same time.

  With the blocking mechanism several applications can simultaneously access one or more *ADwin* systems independent of each other.

If a computer application starts the communication to a system, it transfers a device number in addition to the specified instruction. The `ADwin32.dll` uses this "Device Number" to differentiate between the various *ADwin* systems and assign the corresponding configurations.

–

6.3.3 The Device Number

Each *ADwin* system connected to a computer is accessed via a unique device number (unique to the PC).

You set the device number with the program *ADconfig*: .

In *ADconfig* you link a Device Number with the communication parameters, which define how a system can be accessed (USB, Ethernet). This is the information the `ADwin32.dll` needs in order to being able to communicate with the system.

6.3.4 Communication with Development Environments

You access the *ADwin* system from the PC with the help of a user interface. You may generate this user interface with one of the conventional development environments such as ActiveX, Java, Visual Basic, C++, Delphi or C#.NET, or you may use a ready-made user interface such as TestPoint, DIAdem or MATLAB.

For each of these an appropriate driver software, which enables you to access the *ADwin* system is provided. If you have a special request, please contact us. We can also provide turnkey measurement data evaluation programs.

Under Windows a DLL or ActiveX interface can establish the communication with the system simultaneously from several programs (see also "Communication between PC and ADwin System" on page 121). The special instructions for your user interface are described more detailed in the relevant *ADwin* developer software.

From your user interface you can:

– transfer compiled programs (binary files) into the *ADwin* system. Compile the program in *ADbasic* with `Build` ▸ `Make Bin File` (see chapter 3.7.4 on page 39).

– start, control and stop processes in the *ADwin* system.

– request data from the *ADwin* system or send data to the system.

Although the *ADwin* system works independently, you can access global variables and arrays from the user interface any time, without delaying time-critical processes. This way all processes can quickly exchange data with the computer (or with each other).

7 Instruction Reference

Below, the available *ADbasic* instructions for *ADwin* processors are listed. 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 7.3 and chapter 7.4 the *ADbasic* instructions are listed for the use of the FFT Library as well as Mathematics Instructions.

7.1 Instruction Syntax

Please note:

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

CONST	constant numbers such as `35` or `3.14159`, and expressions without variables. Character constants (strings) are enclosed in quotes such as `"this text"`.
VAR	variable or array element.
ARRAY	array, also identified in the command syntax by its brackets [ ] after the array name.
FIFO	fifo array (`DATA_n` declared as fifo).

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

LONG	integer number
FLOAT	floating point number
STRING	character string
LOGIC	logic expression in a condition

  If the argument has a different data type than expected, you will get a type conversion of the argument (chapter 4.4.2 on page 94).

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

We assume that the necessary library or include file is located in the directory, which is set under the `Options` ▸ `Settings` menu, `Directory` item, (see also the instructions **#INCLUDE** or **IMPORT**).

7.2 Instructions for L16, Gold, Pro

The instructions in this section are valid for the processors of all *ADwin* systems.

+ Addition

The "+" operator adds two values (see also "+ String Addition").

Syntax

```
ret_val = val_1 + val_2
```

Parameters

val_1	Addend 1.	FLOAT LONG
val_2	Addend 2.	FLOAT LONG

Notes

Please note that combining different variable types with the "+" operator will cause a type conversion. During conversion from the type LONG into the type FLOAT rounding differences can occur which influence the result.

See also

- Subtraction, * Multiplication, / Division, ^ Power

Example

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

+ String Addition

The "+" operator concatenates two strings (see also "+ Addition").

Syntax

```
IMPORT String.LI*        '*.LI9 for T9, *.LIA for T10,
                         '*.LIB for T11

val = val_1 + val_2
```

Parameters

`val_1`	character string1.	STRING
`val_2`	character string 2.	STRING

Notes

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

See also

String "", Asc, Chr, FloToStr, Flo40ToStr, LngToStr, StrComp, StrLeft, StrLen, StrMid, StrRight, ValF, ValI

Example

```
IMPORT String.li9

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

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

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

- Subtraction

The "-" operator subtracts one value from another.

Syntax

```
val = val_1 - val_2
```

Parameters

`val_1`	Minuend.	FLOAT LONG
`val_2`	Subtrahend.	FLOAT LONG

Notes

Please note that combining different variable types with the "-" operator will cause a type conversion. During conversion from the type LONG into the type FLOAT rounding differences can occur which influence the result.

If you use "-" as a sign of a variable (unary operator), you may in some cases get unexpected results, which can be avoided by using brackets (see also chapter 4.4.1 on page 92).

See also

+ Addition, * Multiplication, / Division, ^ Power

Example

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

* Multiplication

The "*" operator mulitplies two values.

Syntax

```
val = val_1 * val_2
```

Parameters

`val_1`	Multiplicator 1.	FLOAT LONG
`val_2`	Multiplicator 2.	FLOAT LONG

Notes

Please note that combining different variable types with the "*"operator will cause a type conversion. During conversion from the type `LONG` into the type `FLOAT` rounding differences can occur which influence the result.

See also

+ 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.	FLOAT LONG
val_2	Divisor.	FLOAT LONG

Notes

Please note that combining different variable types with the "/"operator will cause a type conversion (see chapter 4.4.2 on page 94). During conversion from the type LONG into the type FLOAT rounding differences can occur which influence the result.

If the divisor is a variable with a negative sign, you should use braces to ensure you get the expected result (see also chapter 4.4.1 "Evaluation of Operators" on page 92).

See also

+ Addition, - Subtraction, * Multiplication, ^ Power, Mod

Example

```
PAR_1 = 36 / 4          'PAR_1 = 9
PAR_2 = 2 / 4 * 5       'PAR_2 = 0 -> integer calculation
PAR_3 = 27 / (-PAR_1)   'PAR_3 = -3
Rem Please note the braces in the last line
```

^ Power

The "^" operator calculates the value of a number raised to a power.

Syntax

```
val = val_1 ^ val_2
```

Parameters

val_1	Basis.	FLOAT LONG
val_2	Exponent.	FLOAT LONG

Notes

Please note that combining different variable types with the power operator will cause a type conversion. During conversion from the type LONG into the type FLOAT rounding differences can occur which influence the result.

If basis and exponent are variables with even value (but not constants), the power is nevertheless calculated using Float arithmetic. Large results therefore show the typical Float inaccuracy with large numbers.

Example:

```
PAR_2 = 31              ' variable
PAR_1 = 2^PAR_2         ' = 7FFFFFE2h
```

If the basis and/or the exponent are a variable with a negative sign, you should use braces to ensure the sign will be considered upon exponentiation (see also chapter 4.4.1 "Evaluation of Operators" on page 92). 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, Exp, LN, Log

Example

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

#…, Preprocessor Statement

An *ADbasic* 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 186).

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 FLOAT LONG STRING
expr	Expression.	FLOAT LONG STRING

Notes

If the data format of the expression is not similar to the data format of the destination variable or the array, it is converted into the appropriate data format or the assignment is rejected as illegal. During the conversion rounding differences can occur which influence the result.

Example

```
DIM val_1, val_2 AS LONG'Declaration

INIT:
  val_1 = 69            'Assignment of a constant

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

< = > Comparison

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

Syntax

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

Parameters

val_1	Operand.	FLOAT LONG
val_2	Operand.	FLOAT LONG

Notes

The following comparisons are possible:

Operator	Meaning
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
=	equal to
<>	not equal to

See also

If … Then … {Else …} EndIf, #If … Then … {#Else … } #EndIf

Example

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

AbsF

ABSF provides the absolute value of a Float variable.

Syntax

```
ret_val = ABSF(value)
```

Parameters

value	Argument.	FLOAT
ret_val	Absolute value of the argument.	FLOAT

Notes

The execution time of the function takes 150 ns with a T9, 75 ns with a T10, 17 ns with a T11.

See also

AbsI

Example

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

AbsI

ABSI provides the absolute value of a long variable.

Syntax

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

Parameters

value	Argument: $-(2^{31}-1)$ … $+2^{31}-1$.	LONG
ret_val	Absolute value of the argument (0 … $+2^{-31}-1$).	LONG

Notes

The execution time of the function takes 75 ns with a T9, 50 ns with a T10, 17 ns with a T11.

The smallest negative integer value -2^{31} has no positive counterpart in *ADbasic*; the absolute value of -2^{31} is therefore undefined.

See also

AbsF, Mod

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

causes the processor to wait for several processor cyclesNot, 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 fval_1 AS FLOAT
DIM val4 AS LONG
fval_1 = 3.14

Rem Boolean operation: (true And true) = true
IF ((fval_1 < 9.1) AND (fval_1 > 3.1)) THEN
 val4 = 1
ELSE
 val4 = 0
ENDIF                   'Result: val4 = 1
```

ArcCos

ARCCOS provides the arc cosine of the argument.

Syntax

```
ret_val = ARCCOS(val)
```

Parameters

val	Argument (-1 … +1).	FLOAT
ret_val	Arc cosine of the argument in radians (0…π).	FLOAT

Notes

For val < -1 the value π (3.14159...) is returned, for val > 1 the value 0 (zero).

The execution time of the function takes 2.9 µs with a T9, 1.45 µs with a T10, 0.68 µs with a T11.

See also

Sin, Cos, Tan, ArcSin, ArcTan

Example

```
DIM val_1, val_2 AS FLOAT

EVENT:
 val_1 = 0.5
 val_2 = ARCCOS(val_1)
Rem Result: val_2 = 1.0472
```

ArcSin

ARCSIN provides the arc sine of the argument.

Syntax

```
ret_val = ARCSIN (val)
```

Parameters

val	Argument (-1 … +1).	FLOAT
ret_val	Arc sine of the arguments in radians (-π/2 … +π/2).	FLOAT

Notes

The execution time of the function takes 2.8 µs with a T9, 1.4 µs with a T10, 0.67 µs with a T11.

See also

Sin, Cos, Tan, ArcCos, ArcTan

Example

```
DIM val_1, val_2 AS FLOAT

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

ArcTan

ARCTAN provides the arc tangent of the argument.

Syntax

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

Parameters

val_1	Argument (whole range of values, see "Entering Numerical Values" on page 79).	FLOAT
ret_val	Arc tangent of the argument in radians (-π/2…π/2).	FLOAT

Notes

The execution time of the function takes 1.8 µs with a T9, 0.9 µs with a T10, 0.42 µs with a T11.

See also

Sin, Cos, Tan, ArcSin, ArcCos

Example

```
DIM val_1, val_2 AS FLOAT

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

Asc

ASC determines the corresponding decimal value for a single ASCII character or for the first character of a character string.

Syntax

```
ret_val = ASC(STRING)
```

Parameters

`String`	Character string .	STRING
`ret_val`	ASCII number (0…255) of the (first) character.	LONG

See also

String "", + String Addition, Chr, FloToStr, Flo40ToStr, LngToStr, StrComp, StrLeft, StrLen, StrMid, StrRight, ValF, ValI

Example

```
DIM text[10] AS STRING

INIT:
 text="Hello"

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

Cast_FloatToLong

CAST_FLOATTOLONG changes the data type of the argument from Float into Long.

Syntax

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

Parameters

var	Bit pattern with data type long.	FLOAT
ret_val	Identical bit pattern with data type Float.	LONG

Notes

This function does **not** execute a standard type conversion of a number (see chapter 4.4.2 "Type Conversion", page 94). Use the operator "=" for the assignment of a Float value to an integer variable.

This instruction is to be reasonably used in combination with the inverse function **CAST_LONGTOFLOAT**, if there is a bit pattern representing a Float value but given with data type **LONG**. Contrary to the data type the bit pattern will remain unchanged, so it will again be interpreted as the correct Float value (see also chapter 4.2.3 on page 77).

An example of practice appears with data transfer: A CAN- or RSxxx-bus only transfers 8-bit data packages of data type integer. Therefore, a 32-bit Float value has to be changed into data type **LONG** with **CAST_FLOATTOLONG** and then divided into 4 separate 8-bit packages. The receiver has to reassemble the packages again and restore the data type Float with **CAST_LONGTOFLOAT**.

See also

Cast_LongToFloat

Cast_LongToFloat

CAST_LONGTOFLOAT changes the data type of the argument from Long into Float.

Syntax

```
ret_val = CAST_LONGTOFLOAT(val)
```

Parameters

val	Bit pattern with data type Float.	LONG
ret_val	Identical bit pattern with data type long.	FLOAT

Notes

This function does **not** execute a standard type conversion of a number (see chapter 4.4.2 "Type Conversion", page 94). Use the operator "=" for the assignment of a Float value to an integer variable.

This instruction is to be reasonably used, if there is a bit pattern representing a Float value but given with data type **LONG**. Contrary to the data type the bit pattern will remain unchanged, so it will again be interpreted as the correct Float value (see also chapter 4.2.3 on page 77).

An example of practice appears with data transfer: A CAN- or RSxxx-bus only transfers 8-bit data packages of data type integer. Therefore, a 32-bit Float value has to be changed into data type **LONG** with **CAST_FLOATTOLONG** and then divided into 4 separate 8-bit packages. The receiver has to reassemble the packages again and restore the data type Float with **CAST_LONGTOFLOAT**.

See also

Cast_FloatToLong

Chr

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

Syntax

```
IMPORT String.LI*        '*.LI9 for T9, *.LIA for T10,
                           '*.LIB for T11

CHR(vascii,dest_text)
```

Parameters

vascii	Decimal number (0…255) of the desired ASCII character.	LONG
dest_text	String variable to which the character is assigned.	STRING

Notes

If a string variable has more than one character (or element), **CHR** assigns the ASCII character only to the first element of the string.

See also

String "", + String Addition, Asc, FloToStr, Flo40ToStr, LngToStr, StrComp, StrLeft, StrLen, StrMid, StrRight, ValF, ValI

Example

```
IMPORT String.LI9

DIM text_a[1], text_b[1] AS STRING

EVENT:
  CHR(13, text_a)          'Carriage Return
  CHR(10, text_b)          'Line Feed
```

Cos

COS provides the cosine of an angle.

Syntax

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

Parameters

angle	Angle in radians ($-\pi...\pi$).	FLOAT
ret_val	Cosine of the angle (-1…1).	FLOAT

Notes

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

The execution time of the function takes 1.3 µs with a T9, 0.7 µs with a T10, 0.31 µs with a T11.

See also

Sin, Tan, ArcCos, ArcSin, ArcTan

Example

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

CPU_Sleep

Processor T11 only: **CPU_SLEEP** causes the processor to wait for a certain time.

Syntax

```
CPU_SLEEP(val)
```

Parameters

val	Number (9…715827879) of time units to wait in 10ns.	LONG

Notes

Alternatively there are the instructions **P1_SLEEP** and **P2_SLEEP** (see also chapter 5.2.4 "Setting Waiting Times Exactly"). For processors up to T10 use **SLEEP**.

The waiting time should always be smaller than the cycle time set with **PROCESSDELAY**.

⚠ In a high-priority process **CPU_SLEEP** cannot be interrupted. Thus, very high values in high-priority processes can cause an interruption in the communication to the PC.

If possible, use a constant as argument. If the argument val requires a calculation, it requires additional time; this time interval is constant and takes a few clock cycles.
The following conditions require a calculation:

- The argument is an expression with variables or array elements.
- The variable in the argument is declared in the memory area **DRAM_EXTERN**. The time interval may vary because it depends on several conditions.
- The argument is an array.
- The argument is a floating point value.

See also

IO_Sleep, NOP, P1_Sleep, P2_Sleep, Sleep

Example

```
EVENT:
  Rem Wait to start a subsequent measurement exactly 100 µs
  Rem after the external Event signal.
  CPU_SLEEP(10000)
  Rem ...
```

DATA_n

The **DIM** DATA_n[…] **AS** … instruction dimensions a global **DATA** array.
More information about dimensing see page 155.

Syntax

DIM DATA_n[dim1] {, DATA_n[dim2]} **AS** **<ARR_TYPE>** {**AT** **<MEM_TYPE>**}

DIM DATA_n[dim1]{[dim2]} **AS** **<ARR_TYPE>** {**AT** **<MEM_TYPE>**}

Parameters

DATA_n	Name of the declared DATA array (n: 1…200).
<ARR_TYPE>	Data type: **FLOAT**, **LONG**, **STRING**.
dim1, dim2	Array size: Number (≥1)of the array elements of the type **ARR_TYPE**. (CONST, LONG)
<MEM_TYPE>	memory, where the array elements are stored: **DRAM_EXTERN**: external data memory (default). **DM_LOCAL**: internal data memory (default). available for T11 only: **EM_LOCAL**: extended program or data memory.

Notes

You can access the array elements 1…Dim. The array element [0] must not be used since it is used for internal purpose.

The maximum array size depends on the available physical memory size of the *ADwin* system.

A global array may be declared 2-dimensional. The specifics are described in chapter 4.3.3 on page 85.

See also

Dim, FIFO, "Global Arrays" on page 80, "Variables and Arrays in the Data Memory" on page 83

Example

```
Rem Dimension the global array DATA_15 with
Rem 1000 long elements
DIM DATA_15[1000] AS LONG

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

Dec

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, *without* quotation marks. Special chars are not allowed, only alphanumeric characters (a…z, A…Z, 0…9) and the underscore (_).	CONST STRING
`expression`	Expression for the symbolic name, *without* quotation marks. All characters are allowed.	CONST STRING

Notes

Place this instruction at the beginning of a source code.

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

The first string up to a blank is interpreted as symbolic name, the following text until the carriage return is interpreted as an expression to be inserted[1]. 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).

See also

#Include

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

Example

```
#DEFINE setpoint PAR_1 'Comments like this are ignored
#DEFINE measured DATA_1
#DEFINE pi 3.141592654
```

With these instructions you can use the names `setpoint`, `measured` and `pi` in the source code instead of `PAR_1`, `DATA_1` and `3.141592654`.

```
#DEFINE setpoint (13 + 4^3)
PAR_1 = 2 * setpoint    '= 2 * (13 + 4^3)
```

Without the parentheses in the `#DEFINE` expression you would get the value "90" instead of the expected "154".

Dim

`DIM` declares one or more

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

Information about variables and data types can be found in chapter 4.2.3, information about FIFO arrays under the heading FIFO on page 163..

Syntax

```
DIM var1 {, var2, …} AS <VAR_TYPE>

DIM array1[dim1] {, array2[dim2]} AS <VAR_TYPE>
{AT <MEM_TYPE>}

DIM DATA_n[dim1] {, DATA_n[dim2]} AS <VAR_TYPE>
{AS FIFO} {AT <MEM_TYPE>}

DIM DATA_n[dim1][dim2] AS <VAR_TYPE> {AT <MEM_TYPE>}
```

Parameters

`var1`, `var2`	Names of the declared variables.
`array1`, `array2`, `DATA_n`	Names of the declared arrays. For `DATA_n` you can select `n` from 1…200.
<VAR_TYPE>	Data type: **FLOAT**, **LONG**. for arrays also: **STRING**.
`dim1`, `dim2`	Array size: Number (≥1) of the array elements of the type **VAR_TYPE**.
<MEM_TYPE>	Memory where the variables are stored: **DRAM_EXTERN**: external memory (default for arrays). **DM_LOCAL**:local memory (default for variables). available for T11 only: **EM_LOCAL**: extended program or data memory.

CONST
LONG

Notes

The global variables `PAR_n` and `FPAR_n` must not be declared, because they are predefined.

If you want to access data from the computer or from several processes, you can only do this by using *global* variables and arrays.

In an array you can access the elements 1…`Dim`. The array element [0] must not be used, because it is used for internal purposes.
The maximum array size depends on the physical memory on the *ADwin* system.

String variables are *local* arrays of type `STRING` (see "Strings" on page 88). They cannot be declared as FIFO.

See also

DATA_n, Event:, FIFO, Finish:, Init:, LowInit:, String "", "2-dimensional Arrays" on page 85, "Variables and Arrays in the Data Memory" on page 83

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 1003 Long elements as Fifo
DIM DATA_20[1003] AS LONG AS FIFO

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

Do … Until

DO…UNTIL repeatedly executes a block of instructions until the Exit condition evaluates to "true". The block is executed at least one time.

Syntax

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

Parameters

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

See also

< = > Comparison, And, Or, For … To … {Step …} Next, SelectCase

Notes

You can nest **DO…UNTIL** loops repeatedly; only the available memory size will limit the number of nested loops.

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

Example

```
DIM count AS LONG
DIM DATA_1[103] AS LONG AS FIFO

INIT:
  count = 1

EVENT:
  DO                    'Start loop
    DATA_1 = ADC(1,4)   'Read out measurement value
    INC count           'Increase count variable
  UNTIL (count > 103)   'Are 100 measurements being made?
```

End

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

Syntax

END

Notes

END stops the processing of an **EVENT:** section immediately and starts processing the section **FINISH:** (if existing). Any instructions in the **EVENT:** section following the **END** instruction are not processed.

In the other program sections you should use the **EXIT** instruction instead of **END**.

See also

Exit, ProcessN_Running, Restart_Process, Start_Process, Start_Process_Delayed, Stop_Process

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: {AT <MEM_TYPE>}
```

Parameters

<MEM_TYPE>	T11 only since Rev. E04: memory area, where the program section **EVENT:** is stored. **PM_LOCAL**: internal program memory (default). **EM_LOCAL**: extended internal program or data memory. **DRAM_EXTERN**: external data memory.

Notes

See also overview of program sections in chapter 4.1.1 on page 74.

The program section **EVENT:** is the central functional section, which in a process is called in (typically) regular intervals, until it is stopped. Depending on the settings the call is triggered by a cyclic timer Event signal or by an external Event signal. See more in chapter 6 "Processes in the ADwin System".

The processor type T11 can store each program section in a different memory area (see chapter 4.3.2 "Memory Areas"). The huge, but slow memory area **DRAM_EXTERN** should be used for none-time-critical program sections; mostly these are the sections **LOWINIT:**, **INIT:**, **FINISH:**.

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

See also

Dim, LowInit:, Init:, Finish:

Example

```
DIM val_1 AS FLOAT

EVENT:
  val_1 = -5.3
```

Exit

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

Syntax

```
EXIT
```

Notes

EXIT stops the processing of the process and the current program section immediately; the following program lines in the same section will not be executed. Even the section **FINISH** will not be processed.

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

See also

End, ProcessN_Running, Reset_Event, Restart_Process, Start_Process, Start_Process_Delayed, Stop_Process

Example

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

Exp

EXP calculates the power to the base e of the argument.

Syntax

```
ret_val = EXP(val)
```

Parameters

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

Notes

The execution time of the function takes 1.3 µs with a T9, 0.7 µs with a T10, 0.31 µs with a T11.

See also

LN, Log

Example

```
DIM val_1, val_2 AS FLOAT

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

FIFO

The **DIM** DATA_n **AS FIFO** instruction defines a global DATA array as a ring buffer.

Syntax

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

Parameters

DATA_n	Name of the declared DATA-field (n: 1…200).
<ARR_TYPE>	Defined variable type: **FLOAT**, **LONG**.
Dim	Array size: Number of elements of type **ARR_TYPE** in the array. With processor T11 the range for Dim it be set in steps of 4 only: Dim = 4 × a + 3; a ≥ 0.

Notes

Once a DATA array is defined as FIFO ring buffer (see also chapter 4.3.3 on page 85), it cannot be used as a "normal" array.

FIFO arrays (first in, first out) are managed by data pointers. After dimensioning the array you should initialize these data pointers with **FIFO_CLEAR**, in the section **LOWINIT:** or **INIT:**. The data in the FIFO are not changed neither by dimensioning the array nor by initializing.

If you write data into a FIFO array faster than you read it, older stored data will be overwritten and are lost. To avoid this you can use the instructions **FIFO_EMPTY** and **FIFO_FULL** to determine the amount of space in the array.

If (with processor T11 only) the array size is set to a non-VaIId array size Dim, the FIFO array is automatically dimensioned using the next greater and VaIId array size. As an example the compiler will change an array size [1000] automatically to [1003].

See also

Dim, DATA_n, FIFO_Clear, FIFO_Empty, FIFO_Full

Example

```
Rem Dimension the global array DATA_20 with
Rem 1003 Long elements as fifo ringbuffer
DIM DATA_20[1003] AS LONG AS FIFO
```

FIFO_Clear

FIFO_CLEAR initializes the write and read pointers of a FIFO array.

Syntax

```
FIFO_CLEAR(arraynum)
```

Parameters

`arraynum`	Number of the DATA-FIFO array (1…200).	LONG

Notes

Initalization of the write and read pointers does not change the data in the the array.

The FIFO pointers are not initialized upon dimensioning. You should initialize the pointers in the sections **LOWINIT:** or **INIT:** with **FIFO_CLEAR**.

Initializing the FIFO pointers during program run is useful, if you want to clear all data of the array (because of a measurement error for instance).

See also

FIFO, FIFO_Empty, FIFO_Full

Example

```
DIM DATA_1[20003] AS LONG AS FIFO 'Declaration
DIM reinit_fifo_flag AS LONG

INIT:
  FIFO_CLEAR(1)          'Initialize the FIFO pointer

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

FIFO_Empty

FIFO_EMPTY determines the number of empty elements in a FIFO array.

Syntax

```
ret_val = FIFO_EMPTY(arraynum)
```

Parameters

arraynum	Number of the DATA-FIFO-array (1…200).	LONG
ret_val	Number of the empty array elements.	LONG

Notes

If you want to write data into a FIFO array, you can use this instruction, to determine if the FIFO still has enough empty elements.

With processor T11, please note dimensioning in steps of 4 (see page 163).

See also

FIFO, FIFO_Clear, FIFO_Full

Example

```
DIM DATA_1[20003] AS LONG AS FIFO'Declaration

INIT:
  FIFO_CLEAR(1)          'Initialize the FIFO pointer

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

FIFO_Full

FIFO_FULL determines the number of elements used in the FIFO array.

Syntax

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

Parameters

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

Notes

Before reading out or using data from the FIFO array, you should use this instruction, to check if there is data in the FIFO. If there is no data an undefined value is returned from the FIFO array.

With processor T11, please note dimensioning in steps of 4 (see page 163).

See also

FIFO, FIFO_Clear, FIFO_Empty

Example

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

INIT:
  FIFO_CLEAR(1)          'Initialize the FIFO pointer

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

Finish:

The key word **FINISH:** marks the start of the finishing program section. The program section always has low-priority, level 1.

Syntax

```
FINISH: {AT MEM_TYPE}
```

Parameters

<MEM_TYPE> T11 only since Rev. E04: memory area, where the program section **EVENT:** is stored.
PM_LOCAL: internal program memory (default).
EM_LOCAL: extended internal program or data memory.**DRAM_EXTERN**: external data memory.

Notes

See also overview of program sections in chapter 4.1.1 on page 74.

The program section **FINISH:** is run once as soon as the process is stopped.

After having processed the last instruction in the **FINISH:** section, there will be a certain delay until the process status "stopped" is valid.

The processor type T11 can store each program section in a different memory area (see chapter 4.3.2 "Memory Areas"). The huge, but slow memory area **DRAM_EXTERN** should be used for none-time-critical program sections; mostly these are the sections **LOWINIT:**, **INIT:**, **FINISH:**.

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

See also

Dim, LowInit:, Init:, Event:, ProcessN_Running

Example

```
DIM val_1 AS FLOAT

FINISH:
  val_1 = -5.3
```

FloToStr

FLOTOSTR converts a floating point value into a character string.

Syntax

```
IMPORT String.LI*        '*.LI9 for T9, *.LIA for T10,
                          '*.LIB for T11

FLOTOSTR(val, String[])
```

Parameters

`val`	Value to be converted.	FLOAT
`String[]`	String in the format: `{-}#.######E{-}##.`	ARRAY STRING

Notes

The length of the returned string varies from 11 to 13 characters, depending on the sign of mantissa and exponent.

See also

Asc, Chr, Flo40ToStr, LngToStr, String "", StrComp, StrLeft, StrLen, StrMid, StrRight, ValF, ValI

Example

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

DIM text[13] AS STRING
DIM pi, number AS FLOAT

INIT:
  pi = 3.141592654
  FPAR_1 = -pi^-20

EVENT:
  Rem Convert a floating point number into a string
  FLOTOSTR(FPAR_1, text)
  PAR_1 = text[1]       'String length = 13
  PAR_2 = text[2]       'ASCII character 2Dh = "-"
  PAR_3 = text[3]       'ASCII character 31h = "1"
  PAR_4 = text[4]       'ASCII character 2Eh = "."
  PAR_5 = text[5]       'ASCII character 31h = "1"
  PAR_6 = text[6]       'ASCII character 34h = "4"
  PAR_7 = text[7]       'ASCII character 30h = "0"
  PAR_8 = text[8]       'ASCII character 32h = "2"
  PAR_9 = text[9]       'ASCII character 35h = "5"
  PAR_10 = text[10]     'ASCII character 35h = "5"
  PAR_11 = text[11]     'ASCII character 45h = "E"
  PAR_12 = text[12]     'ASCII character 2Dh = "-"
  PAR_13 = text[13]     'ASCII character 31h = "1"
  PAR_14 = text[14]     'ASCII character 30h = "0"
  PAR_15 = text[15]     'String end character = 0
```

Flo40ToStr

Processor T11 only: **FLO40TOSTR** converts a floating point value into a character string.

Syntax

```
IMPORT String.LI*        '*.LIB for T11
FLO40TOSTR(val, String[])
```

Parameters

`val`	Value to be converted.	FLOAT
`String[]`	String in the format: `{-}#.########E{-}##.`	ARRAY STRING

Notes

The length of the returned string varies from 13 to 15 characters, depending on the sign of mantissa and exponent.

See also

Asc, Chr, FloToStr, LngToStr, String "", StrComp, StrLeft, StrLen, StrMid, StrRight, ValF, ValI

Example

```
IMPORT String.LIB       'String library for T11

DIM text[15] AS STRING
DIM pi, number AS FLOAT

INIT:
  pi = 3.141592654
  FPAR_1 = -pi^-20

EVENT:
  Rem Convert a floating point number into a string
  FLO40TOSTR(FPAR_1, text)
  PAR_1 = text[1]        'String length = 13
  PAR_2 = text[2]        'ASCII character 2Dh = "-"
  PAR_3 = text[3]        'ASCII character 31h = "1"
  PAR_4 = text[4]        'ASCII character 2Eh = "."
  PAR_5 = text[5]        'ASCII character 31h = "1"
  PAR_6 = text[6]        'ASCII character 34h = "4"
  PAR_7 = text[7]        'ASCII character 30h = "0"
  PAR_8 = text[8]        'ASCII character 32h = "2"
  PAR_9 = text[9]        'ASCII character 35h = "5"
  PAR_10 = text[10]      'ASCII character 35h = "6"
  PAR_11 = text[11]      'ASCII character 35h = "4"
  PAR_12 = text[12]      'ASCII character 35h = "7"
  PAR_13 = text[13]      'ASCII character 45h = "E"
  PAR_14 = text[14]      'ASCII character 2Dh = "-"
  PAR_15 = text[15]      'ASCII character 31h = "1"
  PAR_16 = text[16]      'ASCII character 30h = "0"
  PAR_17 = text[17]      'String end character = 0
```

For … To … {Step …} Next

The **FOR**...**NEXT** instruction creates a program loop which executes a specified number of times.

Syntax

```
FOR i = X TO Y {STEP Z}
  …                  'instruction block
NEXT i
```

Parameters

i	Count variable.	LONG
X	Start value of the run variable.	LONG
Y	End value of the run variable.	LONG
Z	Step length (≥1) of the run variable; default: 1.	LONG

Notes

The instruction block is executed at least once, even if the start value X is greater than the end value Y.

Declare the count variable as **LONG** variable.

⚠ A high priority process cannot be interrupted by another process, which is also true while executing a time intensive **FOR...NEXT** loop. Since the *ADwin* system cannot respond to other events in this time, it is important to keep the number of loops small for high priority processes.

See also

Do … Until, If … Then … {Else …} EndIf, SelectCase

Example

```
DIM index AS LONG
DIM sinus[360] AS FLOAT'Array for sine values
DIM pi AS FLOAT

INIT:
  pi = 3.14159
  Rem Calculate the sine values in degrees (0° to 359°)
  FOR index = 1 TO 360
    sinus[index] = (2047*SIN((index - 1) * 2*pi/360))
  NEXT index
  index = 1               'Initialize the count index

EVENT:
  DAC(1, sinus[index])   'Output the amplitude value
  INC index               'Increase the count index
  Rem From 360 degrees onward, restart at 0
  IF (index > 360) THEN index = 1
```

Function … EndFunction

FUNCTION…ENDFUNCTION is used to define a function macro with passed and returned values.

Syntax

```
FUNCTION macro_name({val_1, val_2, …}) AS <VAR_TYPE>
{DIM var AS <VAR_TYPE>}
  …                'instruction block
  macro_name = … 'assign return value
ENDFUNCTION
```

Parameters

`macro_name`	Name of the function and of the return value, data type **<VAR_TYPE>**.
`val_1`, `val_2`	Names of passed parameters; for arrays use the syntax with dimension brackets: `array[]` or `DATA_n[]`.
<VAR_TYPE>	Data type of the function and the return parameter: **FLOAT** or **LONG**, but not **STRING**.

FLOAT
LONG
STRING

Notes

You will find general information about macros in chapter 4.5.1 on page 96.

This instruction defines a function macro, which means that the whole instruction block between **FUNCTION** and **ENDFUNCTION** is inserted any place where the macro is called.

Functions help to make your source code more clearly-structured. Please note that each function call will increase the size of the compiled file.

You may insert functions at the following 3 locations:

1. Before the section **INIT:**/**LOWINIT:**
2. After the section **FINISH:**
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 **LOWINIT:**, **INIT:**, **EVENT:**, or **FINISH:** can be defined.
- local variables can be defined at the beginning, which are only available in the function and for the processing period.
  This is true even when a variable has the same name as a variable outside of the function.
- a value should be assigned to the function name, which will be the returned value for the function in the source code.

A function is called with its name and with the arguments you have defined; the function must be used as argument in the calling program line, e.g. in an assignment (see example). All expression types (including one- and two-dimensional arrays) are allowed as arguments, as long as they have the appropriate data type.
If you don't define arguments you nevertheless have to use the (empty) braces for the function's call: `name()`.

If an array is used as a passed parameter the syntax is different for call and definition:

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

Values are assigned to elements of passed arrays as usual:
`array_def[2] = value`

If a value is assigned to a passed parameter `x` within the function, the function's call must not use a constant `x`, but a variable or a single array element. If so, a passed parameter can be used to hold a return value.

If a passed parameter is part of an expression inside a function the parameter should be set in braces. This avoids problems with the order of operator evaluation.

See also

#Include, Sub … EndSub, Lib_Function … Lib_EndFunction, Lib_Sub … Lib_EndSub

Example

```
FUNCTION average(w1, w2, w3) AS FLOAT
Rem The function calculates the mean of the values
Rem w1, w2 und w3
  DIM sum AS FLOAT
  sum = w1 + w2 + w3
  average = sum/3
ENDFUNCTION
```

Calling the function e.g. is done by the following program lines:

```
  x = average(x1, x2, x3)
  DAC(1,average(x1, x2, x3))
```

The same function with an array as passed parameter:

```
FUNCTION average_array(array[]) AS FLOAT
  average_array=(array[1] + array[2] + array[3])/3
ENDFUNCTION
```

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

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

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

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** and **OR**. If the condition is "true" the instructions after **THEN** are executed.
instr	Instruction (corresponds to an instruction line).

LOGIC

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, int ehe **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) of the form: LOGIC

`<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_CARD ADWIN_GOLD ADWIN_GOLDII ADWIN_L16 ADWIN_PRO ADWIN_PROII	"`System`" setting in the window "`Compiler Options`".
PROCESSOR	T9 T10 T11	"`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-7) the system parameters refer to the command line options `/Sx` and `/Px`.

See also

< = > Comparison, If … Then … {Else …} EndIf

Example

```
Rem set low priority Processdelay to 800µs
#IF PROCESSOR = T11 THEN 'If CPU = T11
  Rem T11: 800µs = 240000 x 3,3ns
  PROCESSDELAY = 240000
#ELSE
  #IF PROCESSOR = T10 THEN 'If CPU = T10
    Rem T10: 800µs = 16 x 50µs
    PROCESSDELAY = 16
  #ELSE                   'other CPU, here: CPU = T9
    Rem T9: 800µs = 8 x 100µs (also other CPUs)
    PROCESSDELAY = 8
  #ENDIF
#ENDIF
```

Import

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

Syntax

```
IMPORT {path}file
```

Parameters

`file`	File name of the library file *without* quotes. The file extension is `.LI9` for T9, `.LIA` for T10, `.LIB` for T11.	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 4.5.2 on page 97.

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 47). 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, otherweise the directory of the source code file.

The following library files are delivered with *ADbasic*:

`String.li9`, `String.liA`, `String.liB`	String instructions for T9, T10 and T11 processors.

`FFT.li9`, `FFT.liA`, `FFT.liB`	FFT instructions for T9, T10 and T11 processors.

See also

#Include, Lib_Function … Lib_EndFunction, Lib_Sub … Lib_EndSub

Example

```
Rem import the string library for the T9 processor
IMPORT String.LI9
Rem import a user library for the T10 processor
IMPORT C:\MyFiles\ADwinLibs\dig2volt.LIA
```

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

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

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

Notes

You find general information about include files in chapter 4.5.2 on page 97.

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 47). 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, otherweise the directory of the source code file.

To include any of the include files delivered with *ADbasic*–the files contain instruction to access hardware I/Os–you enter the first characters of the instruction **#INCLUDE**, press [CTRL][SPACE] and select the required include file from the list. Alternatively use one of the code snippets from the "Hardware" group.

☞ Please note: A program line with an **#INCLUDE** instruction should not exceed 136 characters (maximum length for other lines see page 133). 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: {AT <MEM_TYPE>}
```

Parameters

<MEM_TYPE> T11 only since Rev. E04: memory area, where the program section **EVENT:** is stored.
PM_LOCAL: internal program memory (default).
EM_LOCAL: extended internal program or data memory.**DRAM_EXTERN**: external data memory.

Notes

See also overview of program sections in chapter 4.1.1 on page 74.

The program section **INIT:** is run once as soon as the process is started and (if existing) the program section **LOWINIT:** is finished. The delay between having processed the last instruction of the **INIT:** section and starting the **EVENT:** section is about 1 × **PROCESSDELAY**.

The program section has the priority as set for the process (menu entry "`Options / Process`"). With high priority the section cannot be interrupted and should then be as short as possible.

The processor type T11 can store each program section in a different memory area (see chapter 4.3.2 "Memory Areas"). The huge, but slow memory area **DRAM_EXTERN** should be used for none-time-critical program sections; mostly these are the sections **LOWINIT:**, **INIT:**, **FINISH:**.

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

See also

Dim, LowInit:, Event:, Finish:

Example

```
DIM val_1 AS FLOAT
INIT:
  val_1 = -5.3
```

IO_Sleep

IO_SLEEP causes instructions for access to inputs and outputs of a Gold II system to wait for a certain time.

Syntax

```
IO_SLEEP(val)
```

Parameters

val	Number (12, 14, ...n) of time units to wait in 10ns. Only even numbers are valid. An invalid number will automatically be decreased by 1.	LONG

Notes

Alternatively, there is the instruction **CPU_SLEEP** (see also chapter 5.2.4 "Setting Waiting Times Exactly").

The instruction **IO_SLEEP** is used to wait a defined time between 2 accesses to inputs/outputs. The total waiting time is the sum of the processing time for the I/O access and the waiting time by **IO_SLEEP**.

The waiting time should always be smaller than the cycle time set with **PROCESSDELAY**.

⚠ In high-priority processes, improper values can cause an interruption in the communication to the PC:

- Make sure, that the argument always has a value greater than 12; else very long waiting times can arise.
- Use very high values with care, because the communication to the PC is interrupted for a long time (danger of timeout).

If possible, use a constant as argument. If the argument val requires a calculation, it requires additional time; this time interval is constant and takes a few clock cycles.
The following conditions require a calculation:

- The argument is an expression with variables or array elements.
- The variable in the argument is declared in the memory area **DRAM_EXTERN**. The time interval may vary because it depends on several conditions.
- The argument is an array.
- The argument is a floating point value.

See also

CPU_Sleep, NOP

Example

```
#INCLUDE ADwinGoldII.inc

EVENT:
  SET_MUX1(11010b)       'Mux 1: Set channel and gain
  IO_SLEEP(200)          'wait 2 µs (=200*10ns)
                         '= max. MUX settling time
  START_CONV(1)          'start conversion of ADC1
  Rem ...
```

Lib_Function … Lib_EndFunction

With **LIB_FUNCTION**…**LIB_ENDFUNCTION** a function with passed and return parameters is defined in a library file.

Syntax

```
LIB_FUNCTION lib_name(<LIB_PAR1> {, <LIB_PAR2>, …} )
 AS <FCT_TYPE>
  {DIM var AS <VAR_TYPE>}
  {#DEFINE name expression}
  …                       'Instruction block
  name = …
LIB_ENDFUNCTION
```

Syntax of passed parameters **<LIB_PAR>**::
<BY_TYPE> var_name **AS** **<VAR_TYPE>** {**AT** **<MEM_TYPE>**}

Parameters

lib_name	Name of the library function and of the return value; data type **<FCT_TYPE>**.
<FCT_TYPE>	Data type: **FLOAT**, **LONG**.
var_name	Name of a passed parameter inside of library function; for arrays use the syntax with dimension brackets: array[] or DATA_n[].
<BY_TYPE>	Methods for the transfer of parameters: **BYREF**: pass reference (pointer) to variable or array. **BYVAL**: pass value only.
<VAR_TYPE>	Data type: **FLOAT**, **LONG**, **STRING**.
<MEM_TYPE>	Useful for processor T10 only: Type of memory, where the passed parameters are stored; to be used only with arrays: **DRAM_EXTERN**: external memory. **DM_LOCAL**:local memory.

Notes

You will find general information about library files in chapter 4.5.3 on page 97.

Generate library functions (and library subroutines) in a separate source code file. The compilation with "Build/Make lib file" creates the library file. With **IMPORT** those library modules are included into a process which are being called in the process.

In a library function you

- can declare and use local variables and arrays (only one-dimensional).
  Declare variables always at the beginning of the subroutine, but never outside.
- can use global variables and arrays which are passed as parameters.
- can process one-dimensional arrays only.
  You can pass two-dimensional arrays as parameters, but they will

be considered as one-dimensional arrays in the function (see also chapter 4.3.3 on page 85).

- should assign a value to the function name, which will be the value returned for the function in the source code.
- cannot define process sections such as **LOWINIT:**, **INIT:**, **EVENT:**, or **FINISH:**.
- cannot call a library function or subroutine from the same library file.
  If necessary you have to put the function, which is to be called, into a new library file and Import it from there.
- cannot use **SELECTCASE**.

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

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

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

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

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

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

See also

Lib_Sub … Lib_EndSub, Import, 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.LI9
#DEFINE cnt 10              'Number of the samples
#DEFINE samples DATA_1 'Number of measm. values
#DEFINE filtered DATA_2'Number of filtered measm.
                            'values
#DEFINE length 1000     'Length of the array
DIM samples[length] AS LONG'Source array
DIM filtered[length] AS LONG'Destination array
DIM i AS LONG               'Count variable

INIT:
  i = 1                     'Initialize the count variable
  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>** {**AT <MEM_TYPE>**}

Parameters

`lib_name`	Name of the library subroutine.
`var_name`	Name of a passed parameter inside of library Sub; for arrays use the syntax with dimension brackets: `array[]` or `DATA_n[]`.
<BY_TYPE>	Methods for the transfer of parameters: **BYREF**: pass reference (pointer) to variable and array. **BYVAL**: pass value only.
<VAR_TYPE>	Data types: **FLOAT**, **LONG**, **STRING**.
<MEM_TYPE>	Useful for processor T10 only: Type of memory, where the passed parameters are stored; to be used only with arrays: **DRAM_EXTERN**: external memory. **DM_LOCAL**:local memory.

Notes

You will find general information about library files in chapter 4.5.3 on page 97.

Generate library subroutines (and library functions) in a separate source code file. The compilation with "`Build/Make lib file`" creates the library file. With **IMPORT** those library modules are included into a process which are being called in the process.

In a library subroutine you can

- declare and use local variables and arrays (only one-dimensional). Declare variables always at the beginning of the subroutine, but never outside.
- use global variables and arrays which are passed as parameters.
- process one-dimensional arrays only.
  You can pass two-dimensional arrays as parameters, but they will be considered as one-dimensional arrays in the function (see also chapter 4.3.3 on page 85).
- cannot define process sections such as **LOWINIT:**, **INIT:**, **EVENT:**, or **FINISH:**.
- cannot call a library function or subroutine from the same library file.
  If necessary you have to put the function, which is to be called, into a new library file and Import it from there.
- cannot use **SELECTCASE**.

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

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

☞ Refers to processor T10 only: Passed parameters should always be declared **AT <MEM_TYPE>**, to save valuable processor time (**<MEM_TYPE>** must fit with the declaration of the passed parameters in the calling program, see **DIM**). If not, the library subroutine has to detect the parameter's memory type at run time.

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

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

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

See also

Lib_Function … Lib_EndFunction, Import, Function … EndFunction, Sub … EndSub

Example:

```
Rem Measurement value conversion from Digits(0…65535)
Rem to Volt(±10V)
LIB_SUB dig2volt(BYREF digit[] AS LONG, BYVAL ptr AS LONG,
   BYVAL cnt AS LONG, BYVAL gain AS LONG,
   BYREF volt[] AS FLOAT)
  DIM i AS LONG
  FOR i = ptr TO (ptr + cnt)
   volt[i] = ((digit[i] * 20 / 65536) - 10) / gain
  NEXT i
LIB_ENDSUB
```

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

```
Rem The library 'DIG2VOLT' is imported
IMPORT C:\MyFiles\ADwinLibs\DIG2VOLT.LI9

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

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

INIT:
  i = 1                   'Initialize the count variable
  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
```

LN

LN provides the natural logarithm (to base e) of an argument.

Syntax

```
ret_val = LN(val)
```

Parameters

val	Argument.	FLOAT
ret_val	Natural logarithm of the argument.	FLOAT

Notes

The execution time of the function takes 1.45 µs with a T9, 0.7 µs with a T10, 0.37 µs with a T11.

See also

Log, Exp

Example

```
DIM val1, val2 AS FLOAT

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

LngToStr

LNGTOSTR converts an integer value into a string.

Syntax

```
IMPORT String.LI*        '*.LI9 for T9, *.LIA for T10,
                          '*.LIB for T11

LNGTOSTR(value, STRING)
```

Parameters

value	Value to be converted.	LONG
String	Result String.	ARRAY STRING

Notes

The length of the generated string depends on the character which is to be converted and on the sign. String lengths of 1 to 11 characters are possible.

You will find information about the string structure in chapter 4.3.5 on page 88.

See also

String "", + String Addition, Asc, Chr, FloToStr, Flo40ToStr, StrComp, StrLeft, StrLen, StrMid, StrRight, ValF, ValI

Example

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

INIT:
  a = -1234567890

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

Log

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

Syntax

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

Parameters

val	Argument.	FLOAT
ret_val	Decimal logarithm of the argument.	FLOAT

Notes

The execution time of the function takes 1.5µs with a T9, 0.75µs with a T10, 0.38µs with a T11.

See also

LN, Exp

Example

```
DIM val1, val2 AS FLOAT

EVENT:
  val1 = 5.3
  val2 = LOG(val1)        'Result: val2 = 0.724…
```

LowInit:

The key word **LOWINIT:** marks the start of an initializing program section. The program section always has low-priority, level 1.

Syntax

```
LOWINIT: {AT MEM_TYPE}
```

Parameters

<MEM_TYPE> T11 only since Rev. E04: memory area, where the program section **EVENT:** is stored.
PM_LOCAL: internal program memory (default).
EM_LOCAL: extended internal program or data memory.**DRAM_EXTERN**: external data memory.

Notes

See also overview of program sections in chapter 4.1.1 on page 74.

The program section **LOWINIT:** is run once as soon as the process is started. The section serves to initialize, e.g. variables or data connections. **LOWINIT:** is always run before the **INIT:** section (if existing).

The section **LOWINIT:** is suitable for huge non-time-critical initialization sequences since it can be interrupted (due to low priority).

The processor type T11 can store each program section in a different memory area (see chapter 4.3.2 "Memory Areas"). The huge, but slow memory area **DRAM_EXTERN** should be used for none-time-critical program sections; mostly these are the sections **LOWINIT:**, **INIT:**, **FINISH:**.

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

See also

Dim, Init:, Event:, Finish:

Example

```
DIM val_1 AS FLOAT

LOWINIT:
  val_1 = -5.3
```

Max_Float

MAX_FLOAT returns the greater of 2 Float values.

Syntax

```
ret_val = MAX_FLOAT(val1, val2)
```

Parameters

`val_1`	Compared value 1	FLOAT
`val_2`	Compared value 2	FLOAT
`ret_val`	The greater of both values.	FLOAT

Notes

- / -

See also

AbsF, Max_Long, Min_Long, ValF

Example

```
EVENT:
  FPAR_10 = MAX_FLOAT(FPAR_1,FPAR_2)
```

Min_Float

MIN_FLOAT returns the smaller of 2 Float values.

Syntax

```
ret_val = MIN_FLOAT(val1, val2)
```

Parameters

val_1	Compared value 1	FLOAT
val_2	Compared value 2	FLOAT
ret_val	The smaller of both values.	FLOAT

Notes

- / -

See also

AbsF, Max_Long, Min_Long, ValF

Example

```
EVENT:
  FPAR_10 = MIN_FLOAT(FPAR_1,FPAR_2)
```

Max_Long

MAX_LONG returns the greater of 2 integer values.

Syntax

```
ret_val = MAX_LONG(val1, val2)
```

Parameters

val_1	Compared value 1	LONG
val_2	Compared value 2	LONG
ret_val	The greater of both values.	LONG

Notes

- / -

See also

Absl, Max_Float, Min_Long, Vall

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

val_1	Compared value 1	LONG
val_2	Compared value 2	LONG
ret_val	The smaller of both values.	LONG

Notes

- / -

See also

Absl, Max_Long, Min_Float, Vall

Example

```
EVENT:
  PAR_10 = MIN_LONG(PAR_1,PAR_2)
```

MemCpy

Processor T11 only: **MEMCPY** copies a specified amount of array elements from a source array to a destination array.

Syntax

```
MEMCPY(array1[i1], array2[i2], count)
```

Parameters

`array1[]`	Name of the source array.	LONG FLOAT STRING
`i1`	Index (≥1) of the first copied array element.	LONG
`array2[]`	Name of the destination array.	LONG FLOAT STRING
`i2`	Index (≥1) of the first array element to be written.	LONG
`count`	Number (≥1) of array elements to be copied.	LONG

Notes

MEMCPY is the simple and much faster alternative to copying data in a **FOR**…**NEXT**-loop.

The instruction may be used neither with FIFO arrays nor with local variables.

☞ Please note: The data types of source and destination array must be identical and the destination array must be declared large enough to hold all copied data.

The access to indexes out of bounds can be monitored in debug mode for the destination array (see Debug mode Option on page 52). The source array cannot be monitored.

See also

Dim

Example

```
DIM DATA_1[75], DATA_2[100] AS FLOAT

EVENT:
 Rem Copy 70 array elements from DATA_1 to DATA_2
 MEMCPY (DATA_1[5], DATA_2[30], 70)
```

NOP

NOP (No OPeration) causes the processor to wait for one processor cycle.

Syntax

```
NOP
```

Notes

The execution time of the instruction normally is one processor cycle:

T9	25ns
T10	25ns
T11	$3,\bar{3}$ns

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

CPU_Sleep, P1_Sleep, P2_Sleep, Sleep

causes the processor to wait for several processor cycles**Not**

NOT inverts the bits of an argument.

Syntax

```
ret_val = NOT(val)
```

Parameters

`val`	Value to be inverted (no logic expression).	LONG
`ret_val`	Inverted argument.	LONG

Notes

If possible, use this function only with integer values (of the type **LONG**). Floating point values (of the type **FLOAT**) are converted into integer values before they are inserted: The decimal places are truncated and the value rounded if necessary before the **NOT** operation.

NOT runs with bits only, not with Boolean expressions. Therefore you cannot negate logic expressions (true / false) with it. Not allowed: **NOT**(PAR_2 > 2).

See also

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

Example

```
DIM val1 AS LONG
DIM val2 AS LONG

val1 = -3              '-3 =
                       ' 11111111111111111111111111111101b
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, causes the processor to wait for several processor cyclesNot, 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
```

P1_Sleep

Processor T11 only: **P1_SLEEP** causes the Pro I bus to wait for a certain time.

Syntax

```
P1_SLEEP(val)
```

Parameters

val	Number of the time units to wait in 10ns: with constants: 7…715827879. with variables: 9…715827879.	LONG

Notes

Alternatively there are the instructions **CPU_SLEEP** and **P2_SLEEP** (see also chapter 5.2.4 "Setting Waiting Times Exactly"). For processors up to T10 use **SLEEP**.

P1_SLEEP is used to wait a defined time between 2 accesses to modules on the Pro I bus.

The waiting time should always be smaller than the cycle time set with **PROCESSDELAY**.

⚠ In a high-priority process **P1_SLEEP** cannot be interrupted. Thus, very high values in high-priority processes can cause an interruption in the communication to the PC.
Do not use values lower than the minimum value given.

If possible, use a constant as argument. If the argument `val` requires a calculation, it requires additional time; this time interval is constant and takes a few clock cycles.
The following conditions require a calculation:

- The argument is an expression with variables or array elements.
- The variable in the argument is declared in the memory area **DRAM_EXTERN**. The time interval may vary because it depends on several conditions.
- The argument is an array.
- The argument is a floating point value.

See also

CPU_Sleep, NOP, P2_Sleep, Sleep

Example

```
EVENT:
  SET_MUX(1,0)           'Set multiplexer on module 1
  P1_SLEEP(250)          'wait 2.5 µs (=250*10ns)
                         '= Mux settling time
  START_CONV(1)          'Start conversion
  Rem ...
```

P2_Sleep

Processor T11 only: **P2_SLEEP** causes the Pro II bus to wait for a certain time.

Syntax

`P2_SLEEP(val)`

Parameters

`val`	Even number (14…715827878) of the time units to wait in 10ns. An odd number is not allowed.	LONG

Notes

Alternatively there are the instructions **CPU_SLEEP** and **P1_SLEEP** (see also chapter 5.2.4 "Setting Waiting Times Exactly"). For processors up to T10 use **SLEEP**.

P2_SLEEP is used to wait a defined time between 2 accesses to modules on the Pro II bus.

The waiting time should always be smaller than the cycle time set with **PROCESSDELAY**.

⚠ In a high-priority process **P2_SLEEP** cannot be interrupted. Thus, very high values in high-priority processes can cause an interruption in the communication to the PC.

If possible, use a constant as argument. If the argument `val` requires a calculation, it requires additional time; this time interval is constant and takes a few clock cycles.
The following conditions require a calculation:

- The argument is an expression with variables or array elements.
- The variable in the argument is declared in the memory area **DRAM_EXTERN**. The time interval may vary because it depends on several conditions.
- The argument is an array.
- The argument is a floating point value.

See also

CPU_Sleep, NOP, P1_Sleep, Sleep

Example

```
EVENT:
  P2_SET_MUX(0)           'Set multiplexer
  P2_SLEEP(210)           'wait 2.1 µs (=210*10ns)
                          '= Mux settling time
  P2_START_CONV(1)        'start conversion
  Rem ...
```

Peek

PEEK reads the contents of a specified memory location of the *ADwin* system.

Syntax

```
ret_val = PEEK(addr)
```

Parameters

addr	Address of the memory location to be read out.	LONG
ret_val	Contents of the memory location.	LONG

Notes

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

See also

Poke, Read_Timer

Example

The instruction below reads the value of the memory address 30h, which is the data register of the ADC1 on the *ADwin-Gold* system and contains the converted analog value.

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

Poke

POKE writes a value into a specified memory location of the *ADwin* system.

Syntax

```
POKE(addr, value)
```

Parameters

addr	Address of the memory location into which values are written.	LONG
value	Value to be written.	LONG

Notes

With **POKE** you are overwriting the specified memory address. Information stored there will be lost.

Do not write to memory addresses whose functions you do not know. If you do, it is possible that important data, processes or even the operating system will be destroyed.
If this should happen, existing measurement data is lost. To recover, you must reboot the *ADwin* system and reload the processes.

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

See also

Peek, Read_Timer

Example

```
'Change memory locations of an ADwin-Gold system
'Write into DAC register 1: 3072 (=+5V in the range ±10V)
POKE(20400050h, 3072)
POKE(20400010h, 011b)  'Start output on all DACs
POKE(204000C0h, 111100b)'Set outputs DIO18…DIO21 to High
```

Processdelay

The system variable **PROCESSDELAY** defines the process delay (cycle time) of a process.

PROCESSDELAY replaces the system variable **GLOBALDELAY** which is still valid for reasons of compatibility.

Syntax

`ret_val = PROCESSDELAY`

or

`PROCESSDELAY = expr`

Parameters

`ret_val`	Current cycle time in clock cycles.	`LONG`
expr	Cycle time to be set: Number (≥1) of clock cycles.	`LONG`

Notes

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

The time interval of the Processdelay depends on the process priority and the processor type:

Processor	Priority	
	High	Low
T9	25 ns	100 µs
T10	25 ns	50 µs
T11	$3{,}\overline{3}$ ns	$3{,}\overline{3}$ ns = $0{,}00\overline{3}$ µs

With high-priority processes select a sufficiently large process delay to avoid overloading the *ADwin* system (see also chapter 6.1.4 on page 113). As a rule of thumb the processor workload (display field: "`Busy x%`" in the status bar) should be under 90 percent and must not exceed 100 percent.
If the time needed for processing the section **EVENT:** is larger than the process delay, the next counter call and following will be delayed. If this

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

You may set a constant process delay by assigning a value to the variable **PROCESSDELAY** in the section **INIT:** / **LOWINIT:**. You will then overwrite the default value you have set in the dialog window "`Options / Process`" under "`Initial Processdelay`".

You can set the variable only once in a section.

If the parameter **PROCESSDELAY** is changed in a process cycle in the section **EVENT:**, the cycle time (processs 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
  PROCESSDELAY = 40000
  Rem For T9 and T10, high priority: 1 ms
  Rem For T11, high+low priority: 0.133 ms
  Rem ...
```

If you need a longer cycle time than may be set with **PROCESSDELAY** you can use an auxiliary variable:

```
INIT:
  Rem Set max. cycle time
  PROCESSDELAY = 2147483647
  Rem For T9 und T10, high priority: 53.7s
  Rem For T11, high+low priority: 7.2s
  Rem initalize auxiliary variable
  PAR_1 = 0

EVENT:
  INC PAR_1
  Rem use 100fold cycle time
  Rem For T9 und T10, high priority: 89.5 min
  Rem For T11, high+low priority: 12min
  IF (PAR_1 = 100) THEN
    PAR_1 = 0
    Rem run program
  ENDIF
```

Process_Error

PROCESS_ERROR returns the previously occurred error of the current process.

Syntax

```
ret_val = PROCESS_ERROR
```

Parameters

`ret_val` Number of the previously occurred error in the process: `LONG`

0: no error
1: Division by zero
2: Square root from negative value
10: Accessing a too high element number of a global array.
11: Accessing a too small element number (≤ 0) of a global array.
12: Accessing a too high element number of a local array.
13: Accessing a too small element number (≤ 0) of a local array.
30: FIFO index is not a FIFO.

Notes

The return value is defined only if debug mode is enabled (see Debug mode Option, page 52). The variable is read-only.

See also

ProcessN_Running, Start_Process, Stop_Process

Example

```
EVENT:
  PAR_10 = SQRT(PAR_12)
  Rem read previous error in the process
  PAR_2 = PROCESS_ERROR
```

ProcessN_Running

The system variable **PROCESS**n**_RUNNING** returns the current status of the specified process.

Syntax

```
ret_val = PROCESSn_RUNNING
```

Parameters

n	Number of the requested process (0…12, 15).	CONST LONG
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, Exit, Restart_Process, Start_Process, Start_Process_Delayed, Stop_Process

Example

```
EVENT:
  Rem Get the status of process 2
  PAR_2 = PROCESS2_RUNNING
```

Read_Timer

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

Syntax

```
ret_val = READ_TIMER()
```

Parameters

`ret_val` Current counter value. LONG

Notes

The counter value cannot be written.

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

process priority	T9	T10	T11
high	25 ns	25 ns	$3,\overline{3}$ ns
low	100 µs	50 µs	$3,\overline{3}$ ns

You may determine a time interval from the difference of 2 timer values. Please note that any read timer value will be reached again after a certain time interval, which depends on the units of time given above:

process priority	T9	T10	T11
high	107.4 s	107.4 s	14.3 s
low	119.3 h	59.7 h	14.3 s

See also

Processdelay

Example

```
DIM timervalue AS LONG

EVENT:
  timervalue = READ_TIMER()
```

Rem, '

The compiler instructions `Rem` or "`'`" make it possible to insert comments into the source code for a program. Any text in a program line following the instruction is ignored by the compiler.

Syntax

```
Rem comment
```

```
instr : Rem comment
```

```
instr 'comment
```

Parameters

`comment`	Any character strings.
`instr`	*ADbasic* instruction.

Notes

The instruction only applies to the line in which it is used. If a comment requires more than one text line, then you must begin each line with the instructions `Rem` or "`'`".

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

Example

```
Rem This is a comment that needs more than
Rem one text line
'This is a comment line, too
DIM min AS LONG: Rem comment after an instruction
DIM max AS LONG        'Also a comment after an instruction
```

Reset_Event

RESET_EVENT deletes all external Event signals, which are to be processed.

Syntax

RESET_EVENT

Notes

The instruction is only ValId for externally controlled processes and in the **INIT:** section.

We recommend to run the instruction at the end of the **INIT:** section. This prevents a too early Event signal (coming up during initialization) from starting the main program (**EVENT:** section) too early.

More about the operating mode of the opreating system for externally controlled processes see section "Externally Controlled Process" on page 120.

See also

End, Exit, ProcessN_Running, Start_Process, Stop_Process

Example

```
INIT:
  Rem Initialization
  Rem ...
  RESET_EVENT              'Reset former Event signals

EVENT:
  Rem Any Event signal starts the main program
  Rem ...
```

Restart_Process

Processor T11 only: **RESTART_PROCESS** starts the same process again.

Syntax

RESTART_PROCESS

Notes

The instruction is ValId in the program section **FINISH:** only.

All lines of the program section after **RESTART_PROCESS** will be executed, before the process starts anew. For better readability we recommend put the instruction at the end of the program section.

☞ The instruction may cause an endless loop. Prevent an endless loop by using **RESTART_PROCESS** inside of a conditional block.

See also

End, Exit, If … Then … {Else …} EndIf, Start_Process, Start_Process_Delayed, Stop_Process

Example

```
EVENT:
  Rem ...

FINISH:
  Rem ...
  IF (cond = 2) THEN
    Rem If condition is true, the process is started anew
    RESTART_PROCESS
  ENDIF
```

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

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

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 = [(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            'Evaluate PAR_1
    CASE 0                    'If PAR_1 = 0?
      PAR_10 = ADC(1)         'Read out ADC(1)
    CASE 1                    'If PAR_1 = 1?
      PAR_10 = ADC(3)         'Read out ADC(3)
    CASE 2                    'If PAR_1 = 2?
      PAR_10 = ADC(5)         'read out ADC(5) and ADC(7), too
                              '(by CCase)
    CCASE 3                   'If PAR_1 = 3?
      PAR_11 = ADC(7)         'Read out ADC(7)
    CASE 4,5,6,7,16           'If PAR_1 = 4, 5, 6, 7 or 16?
      PAR_2 = DIGIN_WORD()'read digital inputs
    CASEELSE                  'PAR_1: other values
      DIGOUT_WORD(PAR_10)'Output value of PAR_10 to the
                              'digital outputs
  ENDSELECT                   'End of selection
```

Shift_Left

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

Syntax

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

Parameters

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

Notes

Use only integer values for the argument if possible. Floating point values (of the type **FLOAT**) are converted into integer values before shifting them. The decimal places are truncated and the value is rounded if necessary.

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

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

See also

Shift_Right

Example

```
DIM val1, val2 AS LONG

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

Shift_Right

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

Syntax

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

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

Use only integer values for the argument if possible. Floating point values (of the type **FLOAT**) are converted into integer values before shifting them. The decimal places are truncated and the value is rounded.

If the argument val is a positive number, shifting it num places to the right corresponds to a division by 2^n. A possible division remainder is not taken into account, which means, a set bit is lost if it is right-shifted beyond the length of an argument.

The execution time is shorter than the execution time of a comparable division. For instance `val_2 = SHIFT_RIGHT(val_1,3)` is faster than `val_2 = val_1 / 8`.

See also:

Shift_Left

Example

```
DIM val1, val2 AS LONG

EVENT:
  val1 = 1024
  val2 = SHIFT_RIGHT(val1, 3)'Result: val2=128
```

Sin

SIN provides the sine of an angle.

Syntax

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

Parameters

angle	Arc angle (-π…+π).	FLOAT
ret_val	Sine of the angle (-1…1).	FLOAT

Notes

If you use input values which are not in the range of -π…+π, the calculation error grows with the increasing value.

The execution time of the function takes 1.25 µs with a T9, 0.63 µs with a T10, 0.28 µs with a T11.

See also

Cos, Tan, ArcSin, ArcCos, ArcTan

Example

```
DIM val1, val2 AS FLOAT

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

Sleep

Processors until T10 only: **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

For processor T11, **SLEEP** must be replaced by one of the instructions **CPU_SLEEP**, **P1_SLEEP** or **P2_SLEEP** (see also chapter 5.2.4 "Setting Waiting Times Exactly"); mostly **P1_SLEEP** is best.

Since **SLEEP** is executed as a count loop, it cannot be interrupted in high-priority process.

⚠ Please make sure (especially when using variables) that the argument does not have a value less than 1, otherwise the *TiCo* processor *ADwin* system will become unstable. And please consider that very high values in high-priority processes can cause an interruption in the communication to the PC.

If possible, use a constant as argument. If the argument val requires a calculation, it requires additional time; this time interval is constant and takes a few clock cycles.
The following conditions require a calculation:

- The argument is an expression with variables or array elements.
- The variable in the argument is declared in the memory area **DRAM_EXTERN**.
- The argument is an array.
- The argument is a floating point value.

See also

CPU_Sleep, NOP, P1_Sleep, P2_Sleep

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
  Rem ...
```

Sqrt

SQRT returns the square root of a value.

Syntax

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

Parameters

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

Notes

The execution time of the function takes 0.9µs with a T9, 0.45µs with a T10, 0.26µs with a T11.

Example

```
DIM val_1, val_2 AS FLOAT

EVENT:
  val_1 = 16
  val_2 = SQRT(val1)     'Result: val_2 = 4
```

Start_Process

START_PROCESS starts a specified process.

Syntax

```
START_PROCESS(processnum)
```

Parameters

processnum	Number of the process to be started (1…12, 15).	LONG

Notes

Please assure, that the process is transferred to the *ADwin* system before you start it.

The instruction has no effect, if you indicate the number of a process, which

- is already running or
- has the same number as the calling process.

You can start a process with **START_PROCESS** from another process only (except for **RESTART_PROCESS**). It is not possible that a process starts itself, for instance in the section **FINISH:**.

See also

End, Exit, Restart_Process, Start_Process_Delayed, Stop_Process

Example

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

Start_Process_Delayed

Processor T11 only: **START_PROCESS_DELAYED** starts a specified process (section **EVENT:**) with the defined delay.

Syntax

```
START_PROCESS_DELAYED(processnum, delay)
```

Parameters

processnum	Number of the process to be started (1…10).	LONG
delay	Delay time (>30) in clock cycles of the timer. With T11 one clock cycle takes 3,3 ns.	LONG

Notes

Please assure, that the process is transferred to the *ADwin* system before you start it.

The instruction may only start a time-controlled process with high priority; it has no effect, if you indicate the number of a process, where one of the following is true:

- The process is externally controlled.
- The process has low priority.
- The process is running already.
- The process has the same number as the calling process.

You may start a process with **START_PROCESS_DELAYED** from a different process only (except for **RESTART_PROCESS**).

A delayed started process always begins with the **EVENT:** section, the sections **INIT:** and **LOWINIT:** will not be executed.

These items apply to the wanted starting time:

- The delay until starting time starts being counted with processing **START_PROCESS_DELAYED**; the processing time of the instruction is 30 clock cycles.
- From a high-priority program section the starting time can only be maintained, if the delay time delay is greater than the remaining processing time for the rest of the section.
  Any subsequent lines of the section must be processed, before the selected process can start. The starting time therefore is additionally delayed by a too long remaining processing time.

See also

Restart_Process, Start_Process, Stop_Process

Example

```
EVENT:
 Rem ...
 IF (cond = 2) THEN
  Rem If condition is true, process 2 is started
  Rem with a delay of 100 clock cycles.
  START_PROCESS_DELAYED(2,100)
 ENDIF
 Rem There are NO MORE program lines here to surely maintain
 Rem the wanted starting time.
```

Stop_Process

STOP_PROCESS stops a specified process from another running process.

Syntax

```
STOP_PROCESS(processnum)
```

Parameters

processnum	Number of the process to be stopped (1…12,15).	LONG

Notes

The instruction has no effect, if you indicate the number of a process, which

- has already been stopped,
- has not yet been loaded to the *ADwin* system.

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

- First the specified process gets the status "process is being stopped" (see **PROCESSn_RUNNING**); with low priority processes this will take some time (time-out).
- If the **EVENT:** section is being processed when the stop signal arrives, the execution of the **EVENT:** section is yet completed.
- Normally the **EVENT:** section is called and processed once again.
- If existing, the **FINISH:** section is processed (always at low-priority).
- When **STOP_PROCESS** has completed, the specified process is inactive, but can be started at any time.

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

See also

End, Exit, ProcessN_Running, Restart_Process, Start_Process, Start_Process_Delayed

Example

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

String ""

Strings are put into quotes " ".

Syntax

```
IMPORT String.LI*        '*.LI9 for T9, *.LIA for T10,
                         '*.LIB for T11

DIM text[length] AS STRING

text = "ADwin"
```

Parameters

`text[]`	Name of the text variable.	ARRAY STRING
`length`	Length of the text variable.	CONST LONG

Notes

Dimension text variables with **DIM** … **AS STRING** (see page 155). A string you want to assign to a variable is put in quotes.

More information about text variables and the structure of strings can be found under "Strings" on page 88.

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

See also

+ String Addition, Dim, Asc, Chr, FloToStr, Flo40ToStr, LngToStr, StrComp, StrLeft, StrLen, StrMid, StrRight, ValF, ValI

Example

```
IMPORT String.LI9

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

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

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

  Rem Conversion into upper Case:
  Rem Lower Case: a, b, c, ..., x, y, z?
  PAR_6 = ASC(char)
  IF (PAR_6>96 AND PAR_6<133) THEN
    Rem Subtract 32 in order to convert into upper cases
    CHR(PAR_6-32,char)
  ENDIF
```

StrComp

STRCOMP checks two strings to determine if they are identical.

Syntax

```
IMPORT STRING.LI*        '*.LI9 for T9, *.LIA for T10,
                          '*.LIB for T11

ret_val = STRCOMP(string1[], string2[])
```

Parameters

`string1[]`, `string2[]`	String.	ARRAY STRING ~~CONST~~
`ret_val`	0: Strings are identical. -1: Strings are different.	LONG

Notes

If the strings do not have the same lengths, a negative value is returned, even if the shorter string is included in the longer one.

See also

String "", + String Addition, Asc, Chr, FloToStr, Flo40ToStr, LngToStr, StrLeft, StrLen, StrMid, StrRight, ValF, ValI

Example

```
IMPORT STRING.LI9

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

INIT:
  text1 = "ADBASIC"     'ADbasic correct writing
  text2 = "ADBASCI"     'ADbasic wrong writing
  text3 = "ADBASICA"    'ADbasic wrong writing

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

StrLeft

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

Syntax

```
IMPORT String.LI*        '*.LI9 for T9, *.LIA for T10,
                          '*.LIB for T11

STRLEFT(string1[], length, string2[])
```

Parameters

string1[]	String, from which is copied.	ARRAY STRING
length	Number of characters to be copied.	LONG
string2[]	String, into which is copied.	ARRAY STRING

See also

String "", + String Addition, Asc, Chr, FloToStr, Flo40ToStr, LngToStr, StrComp, StrLen, StrMid, StrRight, ValF, ValI

Example

```
IMPORT String.LI9

Rem Dimension the source and destination strings
DIM text1[32], text2[14] AS STRING

INIT:
  Rem Define source string
  text1 = "MEGA real-time with ADwin systems"

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

StrLen

STRLEN returns the number of characters in a string.

Syntax

```
IMPORT String.LI*       '*.LI9 for T9, *.LIA for T10,
                         '*.LIB for T11

ret_val = STRLEN(String[])
```

Parameters

String[]	String whose length is determined .	ARRAY STRING
ret_val	Number of characters in the string.	LONG

See also

String "", + String Addition, Asc, Chr, FloToStr, Flo40ToStr, LngToStr, StrComp, StrLeft, StrMid, StrRight, ValF, ValI

Example

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

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

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

StrMid

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

Syntax

```
IMPORT String.LI*        '*.LI9 for T9, *.LIA for T10,
                          '*.LIB for T11

STRMID(string1[], start, length, string2[])
```

Parameters

`string1[]`	String from which is copied.	ARRAY STRING
`start`	Position of the first character which is copied.	LONG
`length`	Number of characters to be copied.	LONG
`string2[]`	String into which is copied.	ARRAY STRING

See also

String "", + String Addition, Asc, Chr, FloToStr, Flo40ToStr, LngToStr, StrComp, StrLeft, StrLen, StrRight, ValF, ValI

Example

```
IMPORT String.LI9

Rem Dimension source and destination strings:
DIM text1[32], text2[20] AS STRING

INIT:
  Rem Define source string
  text1 = "MEGA real-time with ADwin systems"

EVENT:
  Rem Copy 20 characters beginning at the 6. character from
  Rem the string text1
  STRMID(text1,6,18,text2)
  PAR_1 = text2[1]       'String-length = 20 characters
  PAR_2 = text2[2]       'ASCII-character 72h = "r"
  PAR_3 = text2[3]       'ASCII-character 65h = "e"
  PAR_4 = text2[4]       'ASCII-character 61h = "a"
  PAR_5 = text2[5]       'ASCII-character 6Ch = "l"
  PAR_6 = text2[6]       'ASCII-character 2Dh = "-"
  PAR_7 = text2[7]       'ASCII-character 74h = "t"
  PAR_8 = text2[8]       'ASCII-character 69h = "i"
  PAR_9 = text2[9]       'ASCII-character 6Dh = "m"
  PAR_10 = text2[10]     'ASCII-character 65h = "e"
  PAR_11 = text2[11]     'ASCII-character 20h = " "
  PAR_12 = text2[12]     'ASCII-character 77h = "w"
  PAR_13 = text2[13]     'ASCII-character 69h = "i"
  PAR_14 = text2[14]     'ASCII-character 74h = "t"
  PAR_15 = text2[15]     'ASCII-character 68h = "h"
  PAR_16 = text2[16]     'ASCII-character 20h = " "
  PAR_17 = text2[17]     'ASCII-character 41h = "A"
  PAR_18 = text2[18]     'ASCII-character 44h = "D"
  PAR_19 = text2[19]     'ASCII-character 77h = "w"
  PAR_20 = text2[20]     'ASCII-character 69h = "i"
  PAR_21 = text2[21]     'ASCII-character 6Eh = "n"
  PAR_22 = text2[22]     'End of string sign = 0
```

StrRight

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

Syntax

```
IMPORT String.LI*       '*.LI9 for T9, *.LIA for T10,
                         '*.LIB for T11

STRRIGHT(string1[], length, string2[])
```

Parameters

`string1[]`	String from which it is copied.	ARRAY STRING
`length`	Number of the characters to copy.	LONG
`string2[]`	String into which it is copied.	ARRAY STRING

See also

String "", + String Addition, Asc, Chr, FloToStr, Flo40ToStr, LngToStr, StrComp, StrLeft, StrLen, StrMid, ValF, ValI

Example

```
IMPORT String.LI9

Rem Dimension the source and destination string:
DIM text1[32], text2[13] AS STRING

INIT:
  Rem Define the source string
  text1 = "MEGA real-time and ADwin systems"

EVENT:
  Rem Get 13 characters from the string text1,
  Rem starting at right
  STRRIGHT(text1,13,text2)
  PAR_1 = text2[1]       'String-length = 13 characters
  PAR_2 = text2[2]       'ASCII-character 41h = "A"
  PAR_3 = text2[3]       'ASCII-character 44h = "D"
  PAR_4 = text2[4]       'ASCII-character 77h = "w"
  PAR_5 = text2[5]       'ASCII-character 69h = "i"
  PAR_6 = text2[6]       'ASCII-character 6Eh = "n"
  PAR_7 = text2[7]       'ASCII-character 2Dh = "-"
  PAR_8 = text2[8]       'ASCII-character 53h = "S"
  PAR_9 = text2[9]       'ASCII-character 79h = "y"
  PAR_10 = text2[10]     'ASCII-character 73h = "s"
  PAR_11 = text2[11]     'ASCII-character 74h = "t"
  PAR_12 = text2[12]     'ASCII-character 65h = "e"
  PAR_13 = text2[13]     'ASCII-character 6Dh = "m"
  PAR_14 = text2[14]     'ASCII-character 73h = "s"
  PAR_15 = text2[15]     'End of string sign = 0
```

Sub … EndSub

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

Syntax

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

Parameters

macro_name	Name of the subroutine.	
val_1, val_2	Name of the passed parameter; for arrays use the syntax with dimension brackets: array[] or DATA_n[].	FLOAT LONG

Notes

You will find general information about macros in chapter 4.5.1 on page 96.

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:/LOWINIT:
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 LOWINIT:, INIT:, EVENT:, or FINISH: can be defined,
- local variables can be defined at the beginning, which are only available in the function and for the processing period.

This is true even when a variable has the same name as a variable outside the function.

If a passed parameter is part of an expression inside a subroutine the parameter should be set in braces. This avoids problems with precedence rules (e.g. BODMAS).

A subroutine is called with its name and with all its arguments 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, Lib_Sub … Lib_EndSub, Lib_Function … Lib_EndFunction

Example

```
SUB Fast_Dac1(val1)
 Rem Outputs val1 on the analog output 1 of an ADwin-Gold
 POKE(20400050h, (val1))'Write value into the
                        'output register
 POKE(20400010h, 11011b) 'Start conversion
ENDSUB
```

Calling the subroutine `Fast_Dac1` is made with the program line:

```
 Fast_Dac1(NewValue)
```

The same subroutine with an array as passed parameter:

```
SUB Fast_Dac1(array[])
  Rem Outputs element 3 of the array on the
  Rem analog output 1 of an ADwin-Gold
  POKE(20400050h, (array[3]))'Write value to output
  POKE(20400010h, 11011b) 'Start conversion
ENDSUB
```

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

```
  Fast_Dac1(array)
```

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

Tan

TAN returns the tangent of an argument.

Syntax

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

Parameters

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

Notes

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

The execution time of the function takes 1.33 µs with a T9, 0.67 µs with a T10, 0.31 µs with a T11.

See also

Sin, Cos, ArcSin, ArcCos, ArcTan

Example

```
DIM val1, val2 AS FLOAT

EVENT:
  val1 = 5.3
  val2 = TAN(val1)      'Result: val2 = -1.50...
```

Trace_Mode_Pause

TRACE_MODE_PAUSE disables the trace mode.

Syntax

TRACE_MODE_PAUSE

Notes

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

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

See also

Trace_Mode_Resume

Example

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

    'For this program section the trace
    'mode is continously activated

    TRACE_MODE_PAUSE     'Trace mode disabled
  ENDIF
```

Trace_Mode_Resume

TRACE_MODE_RESUME activates the trace mode beginning in the next program line.

Syntax

TRACE_MODE_RESUME

Notes

TRACE_MODE_RESUME enables the trace mode in an *ADbasic* program again after it has been disabled with **TRACE_MODE_PAUSE**. The disabling/enabling concerns trace-active program lines only, which are marked with a `?` (question mark).

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

See also

Trace_Mode_Pause

Example

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

    'For this program section the trace
    'is continously activated

    TRACE_MODE_PAUSE     'Trace mode disabled
  ENDIF
```

ValF

VALF converts a string into a floating point number.

Syntax

```
IMPORT String.LI*        '*.LI9 for T9, *.LIA for T10,
                          '*.LIB for T11

ret_val = VALF(String[])
```

Parameters

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

Mantissa (max. 10 characters)				Exponent (0…99)		
{+} –	vvvvv	. ,	nnnnn	e E	{+} –	nn

`ret_val` Generated floating point value. FLOAT

Notes

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

The character "E" divides mantissa from exponent. With T9 and T10, in the mantissa only a maximum of 7 characters (pre-decimal *and* decimal places) are evaluated, with T11 a maximum of 10 characters. If you have more characters the last of them will be lost. As decimal separator either the dot or the comma are allowed.

Please note the value range for float values in chapter 4.2.3 on page 77. Values outside the value range are interpreted as "infinite" or zero.

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

See also

String "", + String Addition, Asc, Chr, FloToStr, Flo40ToStr, LngToStr, StrComp, StrLeft, StrLen, StrMid, StrRight, ValI

Example

```
IMPORT String.LI9

DIM text[20] AS STRING

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

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

Vall

VALI converts a string into an integer number (**LONG**).

Syntax

```
IMPORT String.LI*        '*.LI9 for T9, *.LIA for T10,
                           '*.LIB for T11

ret_val = VALI(String[])
```

Parameters

String[]	String to be converted in the format: Sign: + (optional) or -. Pre-decimal places: max. 10 characters.	ARRAY STRING

{+} -.	vvvvvvvvvv

ret_val	Generated long value.	LONG

Notes

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

Please note the value range for long values:
-2147483648 to +2147483647
Values outside this range are interpreted as zero.

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

See also

String "", + String Addition, Asc, Chr, FloToStr, Flo40ToStr, LngToStr, StrComp, StrLeft, StrLen, StrMid, StrRight, ValF

Example

```
IMPORT String.LI9

DIM text[20] AS STRING

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

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

XOr

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

Syntax

```
… val_1 XOR val_2 …
```

Parameters

val_1, val_2 Integer value. LONG

See also

And, causes the processor to wait for several processor cyclesNot, Or

Example

```
DIM value AS LONG
EVENT:
  value = 0100b XOR 0110b
  Rem Result: value = (4 XOr 6) = 0010b = 2
```

7.3 FFT Library

The FFT library contains *ADbasic* instructions for Fast Fourier Transformation. The library runs with processor type T9 or later.

Notes for the use of the library

⚠ If arrays are declared in the internal memory (**AT DM_LOCAL**), the processing time is clearly smaller. Thus, a calculation of an FFT with 1024 values takes about 23ms in spite of 35ms (using a T9 processor).

⚠ Only use the instructions of the FFT library in a process of low priority or in a process section **LOWINIT:** or **INIT:**. If the calculation of an FFT in a high priority process takes very long, the PC assumes an error and aborts the communication to the *ADwin* system with an appropriate error message.

The folder `<C:\ADwin\ADbasic\lib\FFT_doc+demo>` contains all examples for the library instructions.

Fast-Fourier Transformations

The Fast Fourier Transformation (FFT) is an algorithm for fast calculation of a discrete Fourier transformation. The FFT is applicable for a lot of tasks in signal processing, e.g. to

- Calculate a signal's frequency spectrum.
- Get the frequency response from an impulse response
- Derive an FIR-filter kernel from the frequency response.
- digital filters.
- Convert a time based signal in vibration technology into a frequency based state.
- Approximate identification of frequencies in a sampled signal.

Table of contents

Name	Function	
FFT_MAG	`FFT_MAG` returns the magnitudes (modulus) of complex data.	268
FFT_SCALE	`FFT_SCALE` scales the result of an FFT calculation to the size of the components of the source data.	266
FFT_PHASE	`FFT_PHASE` returns the phase of complex data.	270
FFT_MAG_ SCALE	`FFT_MAG_SCALE` returns the scaled magnitudes (modulus) of complex data.	272
FFT_INIT	`FFT_INIT` initializes 2 auxiliary arrays for the calculation of Fast Fourier Transformations.	273
FFT_CALC	`FFT_CALC` calculates a Fast Fourier Transformation after previous initialization.	274
FFT_CALC_DM	`FFT_CALC_DM` calculates a Fast Fourier Transformation after previous initialization and is optimized for processor T10.	276
FFT_CALC_DX	`FFT_CALC_DX` calculates a Fast Fourier Transformation after previous initialization and is optimized for processor T10.	278

FFT

FFT performs a complex Fast Fourier Transformation with complex input and output data.

Syntax

```
IMPORT FFT.LI*        '*.LI9 for T9, *.LIA for T10,
                      '*.LIB for T11

FFT(real[], img[], z_real[], z_img[],
     array1[], array2[], count)
```

Parameters

`real[]`	Real part of source data.	FLOAT ARRAY
`img[]`	Imaginary part of source data.	FLOAT ARRAY
`z_real[]`	Result: Real parts (index 1…`count`/2) of the transformed data. Array size: 4 × `count`.	FLOAT ARRAY
`z_img[]`	Result: Imaginary parts (index 1…`count`/2) of the transformed data. Array size: 4 × `count`.	FLOAT ARRAY
`array1[]`, `array2[]`	Arrays for internal calculations. Array size: 4 × `count`.	FLOAT ARRAY
`count`	Number (≥ 2) of source data points. The number of points must be a power of 2.	LONG

Notes

The Fourier transformation returns a correct result, if the frequency components f_i of the source data remain inside the following range (referring to the sampling frequency f_{sample}):

$$0 \le f_i \text{ and } f_i < f_{sample}/2$$

The transformed data, the complex amplitudes of the frequency spectrum, is returned in the elements 1…`count`/2 of the arrays `z_real` and `z_img`. The surplus array elements (up to 4 × `count`) are required for internal calculations and hold intermediate results.

The result of the transformation is not scaled to the size of the components of the source data. If scaling is required the transformed data can be scaled with **FFT_SCALE**.

The following table shows how the calculated frequency spectrum refers to the element index of the arrays `z_real` and `z_img` (normalization of the frequency axis), with t_{total} as total sampling time.
The example below has a sampling time t_{total} = 0.1 s; thus, the element index [1024] refers to the frequency (1024-1) / 0.1 s = 10230 Hz.

Element index	[1]	[2]	···	[i]	···	[count/2]
Frequency [Hz]	0	$\frac{1}{t_{total}}$	···	$\frac{i-1}{t_{total}}$	···	$\frac{count/2-1}{t_{total}}$

If you need to calculate several FFTs with the *same* number of source data, the processing time can be reduced: Instead of **FFT**, call **FFT_INIT** first and then several times **FFT_CALC**.

See also

FFT, FFT_Mag, FFT_Scale, FFT_Phase, FFT_Mag_Scale, FFT_Init, FFT_Calc, FFT_Calc_DM, FFT_Calc_DX

Example

The Example program (for *ADwin-Gold* and *ADwin-light-16*)

`<C:\ADwin\ADbasic\lib\FFT_doc+demo\FFT_demo.bas>`

reads the analog signal at input 1 (2048 samples in 0.1 s) and calculates an FFT from it. If for example the signal is a sine of 1000 Hz, the maximum values are stored in `data_3[101]` (real part) and `data_4[101]` (imaginary part).

FFT_Scale

FFT_SCALE scales the result of an FFT calculation to the size of the components of the source data.

Syntax

```
IMPORT FFT.LI*          '*.LI9 for T9, *.LIA for T10,
                        '*.LIB for T11

FFT_SCALE(unscaled[], scaled[], count)
```

Parameters

unscaled[]	Unscaled data from an FFT calculation.	FLOAT ARRAY
scaled[]	Result: Scaled data.	FLOAT ARRAY
count	Number of data.	LONG

Notes

The instruction runs according to the formula:

$$\text{scaled[i]} = \begin{cases} i \neq 1: & \text{scaled[i]} = \text{unscaled[i]}/n \\ i = 1: & \text{scaled[i]} = \text{unscaled[i]}/(n \cdot 2) \end{cases}$$

If **FFT_SCALE** uses the resulting arrays of **FFT**, you have to set count = count / 2 (with count is a parameter of **FFT**).

FFT_SCALE scales the result of an FFT calculation to the size of the components of the source data. It does *not* scale the frequency axis of the spectrum (see the notes of **FFT**).

See also

FFT, FFT_Mag, FFT_Phase, FFT_Mag_Scale

Example

The example program (for all *ADwin* systems)

<C:\ADwin\ADbasic\lib\FFT_doc+demo\FFT_scale_demo.bas>

creates a signal from some sine signals, samples the signal, calculates the FFT, the magnitude and scales the magnitude.

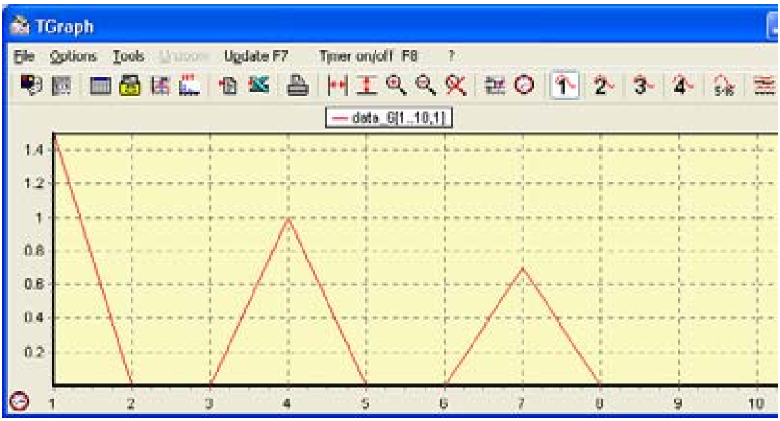
The source signal results from:

- a sine signal of 60 Hz and the amplitude 0.7
- a sine signal of 30 Hz and the amplitude 1.0
- a DC signal with the amplitude 1.5

The amplitudes of the scaled frequency spectrum (see graphic below, created with `TGraph.exe`) exactly show the size of the superposed source signals:

```
data_6[7] = 1          Index 7: 60 Hz
data_6[4] = 0.7        Index 4: 30 Hz
data_6[1] = 1.5        Index 1: DC signal
```

All other amplitudes have the value 0 or close to 0 caused by round-off noise.



FFT_Mag

FFT_MAG returns the magnitudes (modulus) of complex data.

Syntax

```
IMPORT FFT.LI*        '*.LI9 for T9, *.LIA for T10,
                      '*.LIB for T11

FFT_MAG(real[], img[], magnitude[], count)
```

Parameters

real[]	Real part of the complex data.	FLOAT ARRAY
img[]	Imaginary part of the complex data.	FLOAT ARRAY
magnitude[]	Result: Magnitudes of the complex data.	FLOAT ARRAY
count	Number of complex data.	LONG

Notes

The magnitude of a complex value is calculated with the formula:

$$\text{magnitude[i]} = \sqrt{\text{real[i]}^2 + \text{img[i]}^2}$$

FFT calculates the amplitudes of a frequency spectrum as complex values. The instructions **FFT_MAG** and **FFT_PHASE** convert the complex amplitudes into magnitude and phase.

If **FFT_MAG** uses the resulting arrays of **FFT**, you have to set `count` = `count` / 2 (with `count` is a parameter of **FFT**).

See also

FFT, FFT_Phase, FFT_Mag_Scale

Example

The example program (for *ADwin-Gold* oder *ADwin-light-16*)

<C:\ADwin\ADbasic\lib\FFT_doc+demo\FFT_mag_demo.bas>

samples the analog signal at input 1 (2048 samples in 0.1 s), calculates the FFT and the magnitudes. If for example the signal is a sine of 1500 Hz, the maximum absoute value is stored in `Data_5[151]`.

FFT_Phase

FFT_PHASE returns the phase of complex data.

Syntax

```
IMPORT FFT.LI*        '*.LI9 for T9, *.LIA for T10,
                      '*.LIB for T11

FFT_PHASE(real[], img[], phase[], count)
```

Parameters

`real[]`	Real part of the complex data.	FLOAT ARRAY
`img[]`	Imaginary part of the complex data.	FLOAT ARRAY
`phase[]`	Result: Phase of the complex data.	FLOAT ARRAY
`count`	Number of complex data.	LONG

Notes

The phase of a complex value is calculated with the formula (see also `<math.Inc>`):

$$\text{phase[i]} = \begin{cases} \text{real[i]} \neq 0: & \text{phase[i]} = \text{atan}(\text{img[i]}/\text{real[i]}) \\ \text{real[i]} = 0: & \text{phase[i]} = \text{sgn}(\text{img[i]}) \cdot \pi/2 \end{cases}$$

FFT calculates the amplitudes of a frequency spectrum as complex values. The instructions **FFT_MAG** and **FFT_PHASE** convert the complex amplitudes into magnitude and phase.

If **FFT_PHASE** uses the resulting arrays of **FFT**, you have to set `count` = `count` / 2 (with `count` is a parameter of **FFT**).

See also

FFT, FFT_Mag, FFT_Mag_Scale

Example

The example program (for all *ADwin* systems)

`<C:\ADwin\ADbasic\lib\FFT_doc+demo\FFT_phase_demo.bas>`

creates 2 phase-delayed sine signals (by π/2), samples the signals, calulates the FFT, the scaled magnitudes and the phase values.

The calculated frequency spectrum has the following values:

```
data_6[4] = 1           Index 4: 30 Hz
data_7[4] = -0.018410   Phase about 0

data_26[4] = 1          Index 4: 30 Hz
data_27[4] = 1.552389   Phase about π/2
```

All other amplitudes have the value 0 and the referring phase values are undefined.

FFT_Mag_Scale

FFT_MAG_SCALE returns the scaled magnitudes (modulus) of complex data.

Syntax

```
IMPORT FFT.LI*        '*.LI9 for T9, *.LIA for T10,
                      '*.LIB for T11

FFT_MAG_SCALE(real[], img[], mag_scal[],
    count)
```

Parameters

`real[]`	Real part of the complex data.	FLOAT ARRAY
`img[]`	Imaginary part of the complex data.	FLOAT ARRAY
`mag_scal[]`	Result: Scaled magnitudes of the complex data.	FLOAT ARRAY
`count`	Number of complex data.	LONG

Notes

FFT_MAG_SCALE returns the same result as the call of **FFT_MAG** and **FFT_SCALE**, but it is processed faster.

If **FFT_MAG_SCALE** uses the resulting arrays of **FFT**, you have to set `count` = `count` / 2 (with `count` is a parameter of **FFT**).

See also

FFT, FFT_Mag, FFT_Scale

Example

The example program `<FFT_scale_demo_opt.bas>` (for all *ADwin* systems) is similar to the example `<FFT_scale_demo.bas>` (see page 266), but uses **FFT_MAG_SCALE** instead.

FFT_Init

FFT_INIT initializes 2 auxiliary arrays for the calculation of Fast Fourier Transformations.

Syntax

```
IMPORT FFT.LI*        '*.LI9 for T9, *.LIA for T10,
                      '*.LIB for T11

FFT_INIT(array1[], array2[], count)
```

Parameters

`array1[]`, `array2[]`	Result: Auxiliary values for internal calculations. Array size: 4 × `count`.	FLOAT ARRAY
`count`	Number (≥ 2) of source data points. The number of points must be a power of 2.	LONG

Notes

FFT_INIT is only required and useful, if one of the instructions **FFT_CALC**, **FFT_CALC_DM** or **FFT_CALC_DX** is called next.

If you need to calculate several FFT with the *same* number of source data, the processing time can be reduced: Instead of **FFT**, call **FFT_INIT** first and then several times **FFT_CALC**.

See also

FFT, FFT_Calc, FFT_Calc_DM, FFT_Calc_DX

Example

See example program <FFT_scale_demo_opt.bas> (for all *ADwin* systems) in folder <C:\ADwin\ADbasic\lib\FFT_doc+demo>.

FFT_Calc

FFT_CALC calculates a Fast Fourier Transformation after previous initialization.

Syntax

```
IMPORT FFT.LI*        '*.LI9 for T9, *.LIA for T10,
                      '*.LIB for T11

FFT_CALC(real[], img[], z_real[], z_img[],
     array1[], array2[], count)
```

Parameters

real[]	Real part of source data.	FLOAT ARRAY
img[]	Imaginary part of source data.	FLOAT ARRAY
z_real[]	Result: Real parts (index 1…count/2) of transformed data. Array size: 4 × count.	FLOAT ARRAY
z_img[]	Result: Imaginary parts (index 1…count/2) of transformed data. Array size: 4 × count.	FLOAT ARRAY
array1[], array2[]	Arrays for internal calculations. Array size: 4 × count.	FLOAT ARRAY
count	Number (≥ 2) of source data points. The number of points must be a power of 2.	LONG

Notes

The instruction is useful only, if **FFT_INIT** was called before.

If you need to calculate several FFT with the *same* number of source data, the processing time can be reduced: Instead of **FFT**, call **FFT_INIT** first and then several times **FFT_CALC**.

Prczessor T10 only: Instead of **FFT_CALC**, **FFT_CALC_DM** or **FFT_CALC_DX** may be used to calculate an FFT in shorter time.

See also

FFT, FFT_Init, FFT_Calc_DM, FFT_Calc_DX

Example

See example program <FFT_scale_demo_opt.bas> (for all *ADwin* systems) in folder <C:\ADwin\ADbasic\lib\FFT_doc+demo>.

FFT_Calc_DM

FFT_CALC_DM calculates a Fast Fourier Transformation after previous initialization and is optimized for processor T10.

Syntax

```
IMPORT FFT.LIA
FFT_CALC_DM(real[], img[], z_real[], z_img[],
     array1[], array2[], count)
```

Parameters

`real[]`	Real part of source data. The array must be declared **AT DM_LOCAL**.	FLOAT ARRAY
`img[]`	Imaginary part of source data. The array must be declared **AT DM_LOCAL**.	FLOAT ARRAY
`z_real[]`	Result: Real parts (index 1…`count`/2) of transformed data. The array must be declared **AT DM_LOCAL** with array size: 4 × `count`.	FLOAT ARRAY
`z_img[]`	Result: Imaginary parts (Index 1…`count`/2) of transformed data. The array must be declared **AT DM_LOCAL** with array size: 4 × `count`.	FLOAT ARRAY
`array1[]`, `array2[]`	Arrays for internal calculations. The arrays must be declared **AT DM_LOCAL** with array size: 4 × `count`.	FLOAT ARRAY
`count`	Number (≥ 2) of source data points. The number of points must be a power of 2.	LONG

Notes

The instruction is useful only, if **FFT_INIT** was called before.

FFT_CALC_DM has the same function as **FFT_CALC** (and **FFT_CALC_DX**), but calculates an FFT faster when using the processor T10. This optimization is not possible for processors T9 or T11.

FFT_CALC_DM may only be used, if the arrays are declared in the internal memory.
Using the processor T10, the calculation of an FFT with 1024 samples

takes about 11 ms instead of 14 ms with **FFT_CALC**. Both timing values were determined with arrays in the internal memory **DM_LOCAL**.

See also

FFT, FFT_Init, FFT_Calc, FFT_Calc_DX

Example

See example program <FFT_scale_demo_opt.bas> (for all *ADwin* systems) in folder <C:\ADwin\ADbasic\lib\FFT_doc+demo>.

FFT_Calc_DX

FFT_CALC_DX calculates a Fast Fourier Transformation after previous initialization and is optimized for processor T10.

Syntax

```
IMPORT FFT.LIA
FFT_CALC_DX(real[], img[], z_real[], z_img[],
     array1[], array2[], count)
```

Parameters

`real[]`	Real part of source data. The array should be declared **AT DRAM_EXTERN**.	FLOAT ARRAY
`img[]`	Imaginary part of source data. The array should be declared **AT DRAM_EXTERN**.	FLOAT ARRAY
`z_real[]`	Result: Real parts (index 1…`count`/2) of transformed data. The array should be declared **AT DRAM_EXTERN** with array size 4 × `count`.	FLOAT ARRAY
`z_img[]`	Result: Imaginary parts (index 1…`count`/2) of transformed data. The array should be declared **AT DRAM_EXTERN** with array size 4 × `count`.	FLOAT ARRAY
`array1[]`, `array2[]`	Arrays for internal calculations. The arrays should be declared **AT DRAM_EXTERN** with array size 4 × `count`.	FLOAT ARRAY
`count`	Number (≥ 2) of source data points. The number of points must be a power of 2.	LONG

Notes

The instruction is useful only, if **FFT_INIT** was called before.

FFT_CALC_DX has the same function as **FFT_CALC** (and **FFT_CALC_DM**), but calculates an FFT faster when using the processor T10. This optimization is not possible for processors T9 or T11.

FFT_CALC_DX may only be used, if the arrays are declared in the external memory.
Using the processor T10, the calculation of an FFT with 1024 samples

takes about 49ms instead of 53ms with **FFT_CALC**. Both timing values were determined with arrays in the external memory **DRAM_EXTERN**.

See also

FFT, FFT_Init, FFT_Calc, FFT_Calc_DM

Example

See example program <FFT_scale_demo_opt.bas> (for all *ADwin* systems) in folder <C:\ADwin\ADbasic\lib\FFT_doc+demo>.

7.4 Mathematics Instructions

The include file `math.inc` contains additional mathematics instructions, which are not part of the instruction set of the *ADbasic* compiler.

The instructions are available for processors since type T9.

Mathematics instructions

Name	Function	
MOD	`MOD` returns the integer remainder of an integer division.	282

Mod

MOD returns the integer remainder of an integer division.

Syntax

```
INCLUDE Math.inc
val = MOD(x_param, y_param)
```

Parameters

`x_param`	Dividend.	LONG
`y_param`	Divisor.	LONG
`val`	Remainder of the division `x_param` / `y_param`.	LONG

Notes

The remainder calculation performs the truncated division, where the quotient is defined by truncation. With this definition the quotient is rounded towards zero and the remainder has the same sign as the dividend.

The integer remainder of a division by zero equals the dividend: `MOD(x,0) = x`.

The execution time of the modulo function takes up to 3.5 µs with a T9, up to 1.67 µs with a T10, and 0.44 µs with a T11 (high priority).

See also

/ Division, AbsI

Example

```
PAR_1 = MOD(17, 3)      'PAR_1 = 2
PAR_2 = MOD(-9, 5)      'PAR_2 = -4
PAR_3 = MOD(72, PAR_2) 'PAR_3 = 3
```

8 How to Solve Problems?

If problems already occur during installation, please refer to the documentation for your *ADwin* system. Make sure all settings have been carried out properly and completely. Also check if the base address, the processor type, etc. are set correctly in the menu `Options\Compiler`. If your problems still persist, please give your local technical support office a call.

If you need help of a more substantial nature, you can contact us directly; you find the address inside the manual's cover page.

Appendix

A.1 Short-Cuts in ADbasic

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

Short cut key	Function	Matching menu item
F1	Show help topic for marked instruction.	
CTRL-F1	Show online help 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	Boot *ADwin* system.	
F6	Create library.	`Build` ▸ `Make Lib File`
F7	Create binary file.	`Build` ▸ `Make Bin File`
CTRL-F7	Create binary files of the project.	`Build` ▸ `Make All Bin Files`
F8	Compile source code.	`Build` ▸ `Compile`
CTRL-F8	Start process.	
F9	Stop process.	
CTRL-SPACE	Insert or complete a declaration.	
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`

Short cut key	Function	Matching menu item
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 *ADbasic* – termed the Licensee –

and Jäger Computergesteuerte Messtechnik GmbH, Rheinstraße 2 - 4, 64653 Lorsch – termed hereinafter Jäger Messtechnik GmbH – the following license agreement is concluded:

1. OBJECT OF THE LICENSE AGREEMENT

1.1 Object of the license agreement is the software of the compiler and the development system *ADbasic* (hereinafter termed *ADbasic* software) as well as the printed user manual "*ADbasic*: The Real-Time Development Tool for *ADwin* Systems" (hereinafter termed "printed materials").

1.2 The company Jaeger Messtechnik GmbH draws your attention to the fact that it is not possible according to the state of the art to develop computer software in such a way that no errors occur in all applications and combinations. Only a computer software which is basically practicable according to the user documentation is object of the license agreement.

2. EXTENT OF USAGE

2.1 Jaeger Messtechnik GmbH grants the Licensee a single, non-exclusive and individual right of use. This means that you may use the enclosed copy of the *ADbasic* software only on a single computer and only in one single location. The Licensee may transfer the *ADbasic* software in physical form (that is stored on a storage device) from one computer to another computer, provided that it is only used individually on one single computer at any time. A usage other than these restrictions is not permitted.

2.2 Programs generated by the Licensee with the *ADbasic* software, may be distributed and used without restriction.

3. SPECIAL RESTRICTIONS

The Licensee is not permitted to

a) pass or otherwise give to any third party access to the *ADbasic* software without prior written consent of Jaeger Messtechnik GmbH,

b) electronically transfer the *ADbasic* software from one computer to another over a network or a data transfer channel,

c) change or modify, translate, reverse engineer, decompile or disassemble the *ADbasic* software without prior written consent of Jaeger Messtechnik GmbH.

4. OWNERSHIP

4.1 Upon purchasing the product, only title to the physical storage device, where the *ADbasic* software has been stored, is passed to the Licensee. No title to the rights of the *ADbasic* software itself is passed to the Licensee.

4.2 Jaeger Messtechnik GmbH reserves all rights for publication, copying, processing and commercialization of the *ADbasic* software.

5. COPYRIGHTS

5.1 The *ADbasic* software and the printed materials are protected by copyright.

For backup purposes the Licensee may generate a single copy of the *ADbasic* software. He must reproduce the copyright notice of Jaeger Messtechnik GmbH on the copy. The copyright notice on the *ADbasic* software must not be removed.

5.2 It is expressly not permitted to fully or partially copy or reproduce the *ADbasic* software as well as the printed materials in its original or modified form or merged or included in other software.

6. GRANT OF LICENSE

6.1 The right to use the *ADbasic* software can only be granted to a third party with prior written consent of Jaeger Messtechnik GmbH. The Licensee must then completely delete the software which he has installed and pass it to the third party. (The transfer has to include the original data carrier with the documentation, backup version included). The license may furthermore only be transferred to a third party, if the latter agrees for the benefit of Jaeger Messtechnik GmbH to the terms and conditions of this License Agreement and to the General Conditions of the company Jaeger Messtechnik GmbH.

6.2 You must not rent, lease or lend the *ADbasic* software.

7. PERIOD OF AGREEMENT

7.1 The period of the License Agreement is unlimited.

7.2 The right of the Licensee for using the *ADbasic* software voids automatically without notice of termination, if he violates a condition of this

License Agreement. Upon termination of the license, the Licensee must destroy the original data medium and all copies of the *ADbasic* software, possible modified copies included, as well as the printed materials.

8. CLAIM FOR DAMAGES AND PENALTY UPON VIOLATION OF THE CONTRACT

8.1 If the Licensee violates conditions of this License Agreement he must pay damages.

8.2 Notwithstanding, Jaeger Messtechnik GmbH will charge a penalty of 20,000.00 EURO for violation of the copyright, unauthorized usage of the software, and unauthorized distribution of the software to third parties.

8.3 The title to omission on completion of the contract is not influenced by the claim for damages and the penalties.

9. MODIFICATIONS AND UPDATES

Jaeger Messtechnik GmbH is entitled to update the *ADbasic* software upon its own discretion. Jaeger Messtechnik GmbH is not obliged to have updates of the *ADbasic* software available for the Licensee.

For extensive updates Jaeger Messtechnik GmbH reserves the right to charge an additional fee.

10. WARRANTY AND LIABILITY OF JAEGER MESSTECHNIK GMBH

a) Jaeger Messtechnik GmbH assumes warranty to the Licensee that at the moment of delivery the data medium, on which the *ADbasic* software is stored, is error-free in accordance with the accompanying materials, when applied under normal operating conditions and under normal maintenance conditions.

b) If the data medium is faulty, the Licensee is granted a replacement within the warranty period of 6 months from the date of delivery. He must return the data medium as well as a copy of the invoice to Jaeger Messtechnik GmbH or to the distributor from whom he has purchased the product.

c) If a fault as described in Section 10 b) is not eliminated within an adequate period of time by replacement of the product, the Licensee may choose between either allowance (price reduction) or conversion (rescission of the License Agreement). The Licensee is not entitled to any further claims.

d) For the reasons mentioned in Section 1.2 Jaeger Messtechnik GmbH does not assume liability for the absence of defects with regards to the *ADbasic* software. In particular Jaeger Messtechnik GmbH does not assume warranty for the fact that the *ADbasic* software meets the requirements and purposes of the Licensee or is compatible to other programs he is working with. The Licensee is responsible for the correct choice and the consequences of using the *ADbasic* software, as well as for the results he intends to obtain or has obtained. The same applies for the printed materials which are delivered with the *ADbasic* software.

e) Jaeger Messtechnik does not assume liability for damages, unless Jäger Messtechnik GmbH has caused damages by intention or by gross negligence. Liability because of properties assured by Jaeger Messtechnik GmbH remains unaffected. Liability is excluded for consequential damages, which are not part of the assurance given above.

f) Jaeger Messtechnik GmbH does not assume liability for damages caused by viruses, which are passed on by the data medium. The Licensee is hold responsible for checking the data medium for viruses, before installing the *ADbasic* software on his computer.

11. FINAL CONDITIONS

The invalidity of some individual conditions does not affect the validity of the License Agreement.

In addition to the conditions of this License Agreement the General Terms and Conditions of Jaeger Messtechnik GmbH apply.

A.4 Command Line Calling

The *ADbasic* compiler cannot only be activated through the user interface, but it can also be directly called in Windows or DOS (with a so-called "command line call"). The compiler works the same in both cases, it can compile a source code file and generate a binary or library file.

The compiler will only be called after you have entered your license key in *ADbasic*.

The command line call has changed since *ADbasic 4*. Thus, you have to check the syntax of previously written calls.

Please note the general hints about Command line calls in Windows on page 12.

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 ambiguities[1].

As an alternative, options of a single call may be written into a makefile and the compiler called with main option /MAKE.

At last there are the main options /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.

Syntax

```
ADbasicCompiler /M src.bas
[/A"dest"] [/IP"path"] [/LP"path"] [/Lx] [/Sx] [/Px]
[/ET | /EE] [/PNx][/PH | /PL | /PLx] [/PDx] [/Ox]
[/Vx]
```

```
ADbasicCompiler /L src.bas
[/A"dest"] [/IP"path"] /LP"path"] [/Lx] [/Sx] [/Px]
[/Ox]
```

```
ADbasicCompiler /MAKE"makefile"
```

```
ADbasic /H
```

```
ADbasic /VER
```

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.

1. As an example, a call with all options given remains correct, even when a default setting is being changed.

Main Options

`/M`	Generate a binary file with the extension `.Txn`. `x` Processor type; see option `/Px`. `n` Process number; see option `/PNx`.
`/L`	Generate a library file with the extension `.LIx` . `x` Processor type; see option `/Px`.
`/MAKE`	Read main option, file name and other options of a single call from the `makefile`. The text in the `makefile` may be written using several lines. Options outside the `makefile` are not permitted
`/H`	Display a short help text.
`/VER`	Display compiler version number.

Options

`src.bas`	File name of the source code to be compiled; type with suffix `.bas`. Compiler warnings are written into the file `src.wrn`, error messages into the file `src.err`.
/A"dest"	[Path and] name of the binary or library file `<dest>` which is to be generated, without suffix. The default is the file name `src`. The file suffix `.Txn` (binary file) or `.LIx` (library file) is attached automatically.
/IP"path"	Directory, where include files are searched. This setting overwrites the *ADbasic* standard directory and should thus be used with caution.
/LP"path"	Directory, where library files are searched. This setting overwrites the *ADbasic* 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: `/SC` Cards `/SL` Light-16 `/SG` Gold; Default `/SGII` Gold II `/SP` Pro `/SPII` Pro II
`/Px`	Processor type, for which the file is compiled: `/P2` Processor T2 `/P4` Processor T4 `/P5` Processor T5 `/P8` Processor T8 `/P9` Processor T9; Default `/P10` Processor T10 `/P11` Processor T11
`/ET`	Create timer triggered process, see also chapter 6 on page 110. Default. Excludes `/EE`.
`/EE`	Create externally triggered process, see also chapter 6 on page 110. Schließt `/ET` aus.
`/PNx`	Number `x` (1…10) of the process. Default: 1.
`/PH`	Create process with high priority. Default. See also chapter 6.1.2 on page 112.
`/PL`	Create process with low priority and priority level 1 (time triggered process only). See also chapter 6.1.3 on page 112.
`/PLx`	Create process with low priority and priority level `x` (-10…10).
`/PDx`	Set cycle time (Processdelay) of the process to `x`. Default: 1000, T11: 3000. See also chapter 6.2.1 on page 115.
`/Ox`	Set optimize level `x` (0, 1, 2) of the compiler, see also Process Options dialog box (page 42). `/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 42). 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 *ADbasic* displays in the info window (see chapter 3.9.1 on page 62).
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\ADbasic\ADbasiccompiler.exe /L
Z:\Myfiles\test.bas
```

This command line compiles the source code <test.bas> and generates the library file <test.LI9> 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-Gold*.
- Processor: T9.
- Optimize level: 1.

If you do the call from the directory <C:\ADwin\ADbasic>, you can shorten this line to:

```
ADbasicCompiler.exe /L Z:\Myfiles\test.bas
```

The shortest version is when the source code is stored in the directory <C:\ADwin\ADbasic> (here without file name extension):

```
ADbasicCompiler /L test.bas
```

Anyway, we recommend the complete version–at least for automation of the call:

```
ADbasiccompiler /L test.bas /A"test" /LE /SG /P9 /O1
```

```
ADbasiccompiler /L Z:\Myfiles\String.bas /SP /O1
```

This command line compiles the source code <string.bas> into a library file for a *Pro* system with processor T9. It is a timer triggered process with number 1 and high priority.
The same call, for processor T10 only, is as follows:
```
ADbasiccompiler /L Z:\Myfiles\String.bas /P10 /SL /O1
```

```
ADbasicCompiler /M C:\ADwin\ADbasic\samples_ADwin\bas_
dmo6f.bas /LE /SG /P9 /ET /PN3 /PH /O1
```

Compiles the demo file <bas_dmo6f.bas> into a binary file for a *Gold* system with T9 processor. It is a timer triggered process with number 3 and high priority.

```
ADbasiccompiler /M C:\ADwin\ADbasic\samples_ADwin\bas_
dmo6 /LE /P8 /SL /O1
```

Compiles the demo file <bas_dmo6.bas> into a binary file for a *Light-16* card with processor T8, without optimization. It is a timer triggered process with number 2 and low priority

```
C:\ADwin\ADbasic\ADbasic /M C:\user\my_file.bas /LE /P4
/SC /A"your_file" /O1
```

This instruction compiles the file <my_file.bas> for an *ADwin*-Card with processor T4. It is an externally triggered process with number 5 and low priority. The generated binary file has the name <your_file.T45> and can be found in the same directory where the source code is saved: <C:\user>.

```
ADbasicCompiler /M C:\user\my_file.bas /LE /SG /P9
/A"Y:\somewhere\your_file" /ET /PN3 /PH /O1
```

The binary file now is saved as <Y:\somewhere\your_file.T93>; It is a timer triggered process with number 3 and high priority .

A.4.4 Command line calls in Windows

The term and functionality "command line call" 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 Obsolete Program Parts

For compatibility reasons the development environment also offers settings for *ADwin* systems with transputer processors (T4, T5, T8).

Dialog Window `Process Options`

In this dialog window you set compiler options for the currently open source code window, that is you set the properties of the process, which is compiled from the current source code and transferred to the *ADwin* system.
You must make the necessary settings separately for each of the source code windows by opening the dialog window again (unless you want to use the default settings).

If you have set the processor types T4, T5 or T8 in the dialog window Compiler Options, the dialog window shown in fig. 1 is opened.

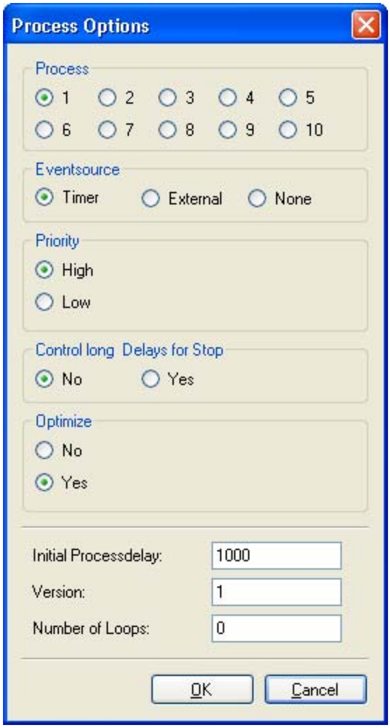


Fig. 1 – The Dialog Window `Process Options` for processors T4 … T8

– `Event`: Here you set which event signal is to start the section **`EVENT:`** of your process.

With the setting `Timer` you define the number of counts of the internal counter as the event signal. In this case you use the system variable **`PROCESSDELAY`** to define time intervals which triggers an event signal.

With `Extern` you determine that a signal at the event input of your *ADwin* hardware starts the process. This could be for instance an impulse of a sensor. Such a process must run at high-priority. In this case set the option `Priority` to `High`.

How to use an external event input with an *ADwin-Pro* system, is described in the software documentation under **`EVENTENABLE`**.

With the setting `None` the process starts immediately after it has been transferred to the system. The section **EVENT:** is – independent of any event signals – it is restarted immediately after the execution (infinite loop).
In a high-priority process you have to assure that the process also provides computing time for other tasks (e.g. communication with the computer).

– `Process`: Set the number (1…10), with which the transferred process is accessed on the system.

  If several processes are running simultaneously on the *ADwin* system, you must assign a separate number to each of the processes.

– `Number of Loops`: If you like, you can set here the number of times the program cycles through the event loop before it stops. When this number is reached, the process stops automatically. A setting you have changed will be active upon the next start of the process (not in the currently running process), you needn't recompile your program.

  If you enter the value "0", the program is repeated until you stop the process with:
  - the instruction **END**,
  - the instruction **STOP_PROCESS** or
  - the stop icon in the development environment.

– `Version`: Here you enter an integer value, in order to differentiate between different versions of your program.

– `Priority`: Set here the priority of the process. You will find more information about this subject in chapter 6.1 "Process Management". The setting `Level` does not exist for the transputer processor type.

– `Control long Delays for Stop`: This setting is only available when you use the processors T2 ... T8.

  The stopping of a process is delayed, if it is not called frequently (cycle time interval > 5 milliseconds). We recommend you use the option in this case, because this option will speed up the stop procedure.

– `Optimize`: The optional optimization shortens the process execution time of up to 20 percent. A higher setting under `Level` leads to shorter execution times.

If unexpected compiler or run-time errors occur, you can sometimes avoid them by setting a lower `Level` for the optimization.

– `Delay`: Set here the processdelay (cycle time), before the process is to begin.

A.6 List of Debug Error messages

The following error messages can be displayed, if the option `Debug mode` is enabled in *ADbasic*; see Debug mode Option, page 52.

Run-time error
`Division by zero`
`SQRT from negative number`

`Data n: Index is too large / Data n: Index is less than 1`
`Array index is too large / Array index is less than 1`
Access to local or global array elements which are not declared, i.e. with indices that are too large or too small.
A trailing `(inc)` in the error message is an additional hint for our support where the error has been detected.

`Fifo index is no fifo` The array with the given index is not declared as FIFO or not declared at all.
`Address of Pro II module is >15 or <1`
`P2_Burst_xxx`[1]`: "startadr" is not divisable by 4`
`P2_Burst_xxx`[1]`: Number of values is not divisable by 4`
`P2_Burst_INIT: Number of  values is not divisable by 4` / `by 8`
`P2_Burst_Read_Unpacked1: Number of values is not divisable by 8`
`P2_Burst_Read_Unpacked2: Number of values is not divisable by 4`
`P2_Burst_Read_Unpacked8: Number of values is not divisable by 2`
`P2_Burst_Read: Number of values smaller than 1` / `than 4`
`P2_GetData/SetData_Long: TiCo DATA does not exist`
`P2_GetData/SetData_Long: TiCo DATA has wrong datatype`
`P2_GetData/SetData_Long: TiCo DATA index too large`
`P2_GetData/SetData_Long: TiCo DATA index < 1`
`P2_Digout_FIFO_Write: timestamp difference <` 2

Run-time error
`Media_Read` / `Media_Write`: `start_block` + `count_blocks128` > `num_blocks` `start_block` < 0 Access to an invalid range of the storage media, with a block number that is too large or too small

1. Valid for **P2_BURST_INIT**, **P2_BURST_READ**, **P2_BURST_WRITE**

A.7 Index

H

I

J

K

L

P

U

V